How Developers Diagnose Potential Security Vulnerabilities with a Static Analysis Tool

Justin Smith, Brittany Johnson, Emerson Murphy-Hill, Bill Chu, and Heather Richter Lipford

Abstract—While using security tools to resolve security defects, software developers must apply considerable effort. Success depends on a developer's ability to interact with tools, ask the right questions, and make strategic decisions. To build better security tools and subsequently help developers resolve defects more accurately and efficiently, we studied the defect resolution process — from the questions developers ask to their strategies for answering them. In this paper, we report on an exploratory study with novice and experienced software developers. We equipped them with Find Security Bugs, a security-oriented static analysis tool, and observed their interactions with security vulnerabilities in an open-source system that they had previously contributed to. We found that they asked questions not only about security vulnerabilities, associated attacks, and fixes, but also questions about the software itself, the social ecosystem that built the software, and related resources and tools. We describe the strategic successes and failures we observed and how future tools can leverage our findings to encourage better strategies.

Index Terms—Software engineering, Human factors, Security, Software tools, Programming environments.

1 Introduction

Software developers are a critical part of making software secure, a particularly important task considering security vulnerabilities are likely to cause incidents that affect company profits as well as end users [7]. When software systems contain security defects, developers are responsible for fixing them. In fact, a recent survey by Christakis and Bird found that developers cared more about security issues than other reliability issues [8].

Static analysis tools, like Find Security Bugs (FSB) [59] and many others [55]–[57], [63], [65], promise to help developers remove security defects early in the development life cycle. These tools locate and report on potential software security vulnerabilities, such as SQL injection and cross-site scripting even before the code executes.

Unfortunately, these tools are confusing for developers to use. Researchers cite several related reasons why these tools do not help developers resolve defects, for instance, the tools: "may not give enough information" [24]; produce "bad warning messages" [8]; and "miscommunicate" with developers [23].

Our work investigates this tool understanding problem by advancing our knowledge of how developers use a security-focused static analysis tool to resolve security defects, including developers' information needs, defect resolution strategies, and assumptions. This is the first study to investigate these facets of security tool usage.

To that end, we conducted an exploratory think-aloud study with ten developers who had contributed to iTrust [61], a security-critical medical records software system written in Java. We observed each developer as they assessed potential security vulnerabilities identified by FSB. We operationalized developers' information needs by measuring questions — the verbal manifestations of information needs. Using a card sort methodology, two authors sorted 559 questions into 17 categories. We report the questions participants asked throughout our study, discuss the strategies participants used to answer questions in each category, and also describe the assumptions participants made.

In non-security domains, work that identifies information needs has helped toolsmiths both evaluate the effectiveness of existing tools [1], and improve the state of program analysis tools [29], [45], [54]. Similarly, we expect that categorizing developers' information needs while using security-focused static analysis tools will help researchers evaluate and toolsmiths improve those tools.

An earlier version of this work appeared as a conference paper [50]. The contribution we presented in that paper was a categorization of questions developers asked while resolving security defects. While that contribution explains developers' information needs, it did not explain how developers *answer* those questions. In this work, we explore how developers acquire the information they need either actively (through strategies) or passively (through assumptions).

More specifically, this paper elaborates on the earlier work with the following two additional contributions:

- A catalog of developers defect resolution strategies, organized by the information need each strategy addresses. For example, to understand how to implement a fix, participants strategically surveyed multiple sources of information, including the web, the tool's notification, and other modules in the code.
- A description of the common assumptions that undermined those strategies. For example, during some tasks, participants incorrectly assumed input validation had been handled securely.

- *J. Smith is with North Carolina State University, Raleigh, NC 27603. E-mail: jssmit11@ncsu.edu*
- *B. Johnson is with University of Massachusetts Amherst, Amherst, MA 01003. E-mail: bjohnson@cs.umass.edu*
- *E. Murphy-Hill is with North Carolina State University, Raleigh, NC 27603. E-mail: emerson@csc.ncsu.edu*
- *B. Chu is with University of North Carolina at Charlotte, Charlotte, NC 28223. E-mail: billchu@uncc.edu*
- *H. Richter Lipford is with University of North Carolina at Charlotte, Charlotte, NC 28223. E-mail: heater.lipford@uncc.edu*

2 Methodology

We conducted an exploratory study with ten software developers. In our analysis, we extracted and categorized the questions developers asked during each study session. Section 2.1 outlines the research questions we sought to answer. Section 2.2 details how the study was designed and Sections 2.3, 2.4, and 2.5 describe how we performed our three phases of data analysis. Study materials can be found online [58] and in the appendices.

2.1 Research Question

We answer the following research questions:

- **RQ1**: What information do developers need while using static analysis tools to diagnose potential security vulnerabilities?
- **RQ2**: What strategies do developers use to acquire the information they need?
- **RQ3**: What assumptions do developers make while executing these strategies?

We measured developers' information needs (RQ1) by examining the questions they asked. The questions that we identified are all available online [58] and in the appendices. We list several exemplary questions throughout Section 3 alongside the strategies developers used (RQ2) and assumptions developers made (RQ3) while answering those questions. Where possible, we also describe how participants' strategies and assumptions led to information seeking successes or failures.

2.2 Study Design

To ensure all participants were familiar with the study environment and Find Security Bugs (FSB), each in-person session started with a five-minute briefing section. The briefing section included a demonstration of FSB's features and time for questions about the development environment's configuration. During the briefing section, we informed participants of the importance of security to the application and that the software may contain security vulnerabilities.

Additionally, we asked participants to use a think-aloud protocol, which encourages participants to verbalize their thought process as they complete a task or activity [40]. 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 study artifacts for data analysis.

Following the briefing period, participants progressed through encounters with four vulnerabilities. Figure 1 depicts the configuration of the Eclipse for Java 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.

2.2.1 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 [60]. The code base comprises over 50,000 lines of code, including the test packages. Participants were equipped with FSB, an extended version of FindBugs.

TABLE 1
Participant Demographics

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

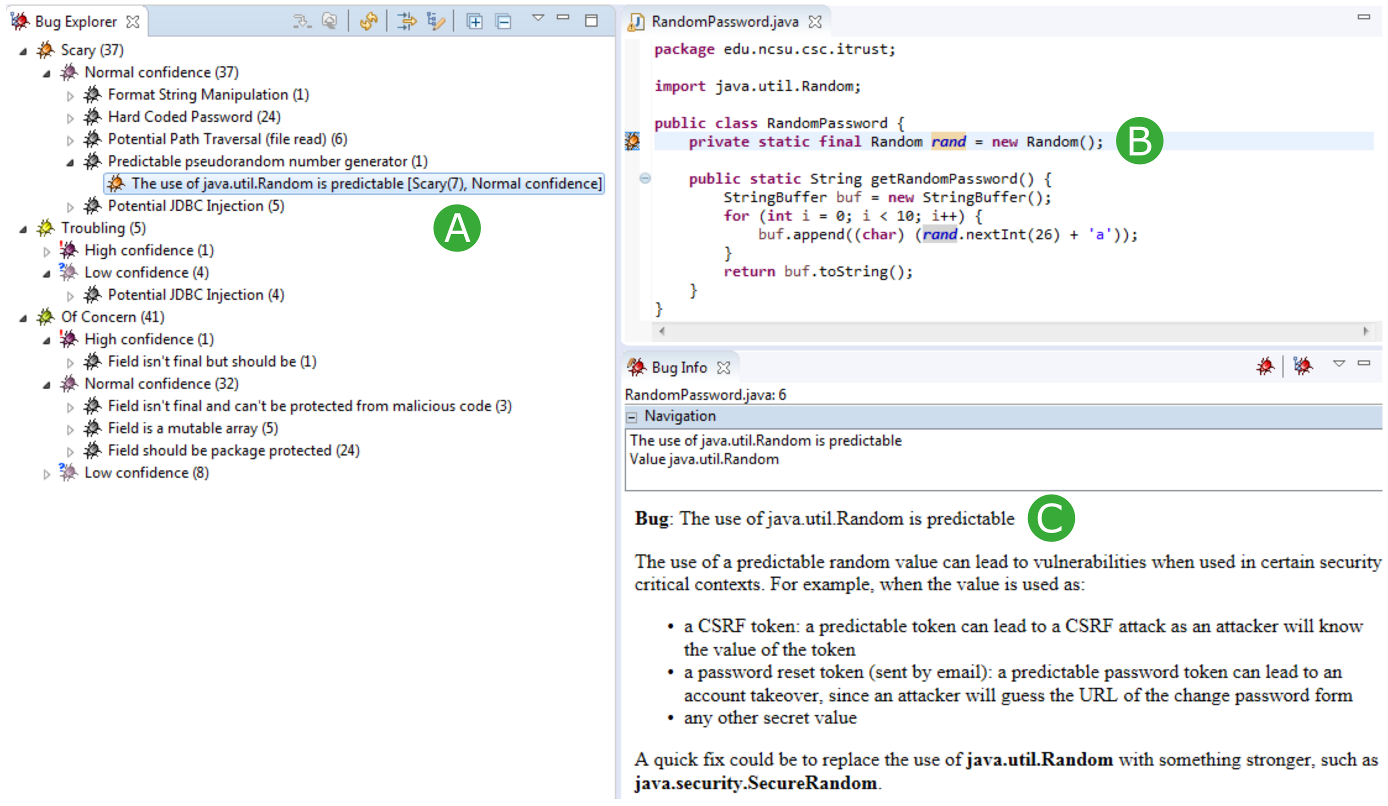
We chose FSB because it detects security defects and compares to other program analysis tools, such as those listed by NIST, [55] OWASP, [63] and WASC [65]. Some of the listed tools may include more or less advanced bug detection features. However, FSB is representative of static analysis security tools with respect to its user interface, specifically in how it communicates with its users. FSB provides visual code annotations and textual notifications that contain vulnerability-specific information. It summarizes all the vulnerabilities it detects in a project and allows users to prioritize potential vulnerabilities based on several metrics such as bug type or severity.

2.2.2 Participants

For our study, we recruited ten software developers, five students and five professionals. We recruited both students and professionals to diversify the sample; our analysis does not otherwise discriminate between these two groups. Table 1 gives additional demographic information on each of the ten participants. Asterisks denote previous use of security-oriented tools. Participants ranged in programming experience from 4 to 10 years, averaging 6.3 years. Participants also self-reported their familiarity with security vulnerabilities on a 5 point Likert scale, with a median of 3. Although we report on experiential and demographic information, the focus of this work is to identify questions that span experience levels. In the remainder of this paper, we will refer to participants by the abbreviations found in the participant column of the table.

We faced the potential confound of measuring participants' questions about a new code base rather than measuring their questions about vulnerabilities. To mitigate this confound, we required participants to be familiar with iTrust; all participants either served as teaching assistants for, or completed a semester-long software engineering course that focused on developing iTrust. This requirement also ensured that participants had prior experience using static analysis tools. All participants had prior experience with FindBugs, the tool that FSB extends, which facilitated the introduction of FSB.

However, this requirement restricted the size of our potential participant population. Accordingly, we used a nonprobabilistic, purposive sampling approach [19], which typically yields fewer participants, but gives deeper insights into the observed phenomena. To identify eligible participants, we recruited via personal contacts, class rosters, and asked participants at the end of the study to recommend other qualified participants. Although our study involved only ten participants, we reached saturation [14] rather quickly; no new question categories were introduced after the fourth participant.



Scary (37)
  Normal confidence (37)
    Format String Manipulation (1)
    Hard Coded Password (24)
    Potential Path Traversal (file read) (6)
    Predictable pseudorandom number generator (1)
      The use of java.util.Random is predictable [Scary(7), Normal confidence]
    Potential JDBC Injection (5)
Troubling (5)
  High confidence (1)
  Low confidence (4)
    Potential JDBC Injection (4)
Of Concern (41)
  High confidence (1)
    Field isn't final but should be (1)
  Normal confidence (32)
    Field isn't final and can't be protected from malicious code (3)
    Field is a mutable array (5)
    Field should be package protected (24)
  Low confidence (8)

```
package edu.ncsu.csc.itrust;

import java.util.Random;

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

    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();
    }
}
```

RandomPassword.java: 6
Navigation
The use of java.util.Random is predictable
Value java.util.Random

Bug: The use of java.util.Random is predictable

The use of a predictable random value can lead to vulnerabilities when used in certain security critical contexts. For example, when the value is used as:

- a CSRF token: a predictable token can lead to a CSRF attack as an attacker will know the value of the token
- a password reset token (sent by email): a predictable password token can lead to an account takeover, since an attacker will guess the URL of the change password form
- any other secret value

A quick fix could be to replace the use of **java.util.Random** with something stronger, such as **java.security.SecureRandom**.

Fig. 1. Study environment, including: short vulnerability description in the bug explorer (A); vulnerable code (B); long vulnerability description (C).

2.2.3 *Tasks*

First we conducted a preliminary pilot study ($n = 4$), in which participants spent approximately 10 to 15 minutes with each task and showed signs of fatigue after about an hour. To reduce the effects of fatigue, we asked each participant to assess just four vulnerabilities. We do not report on data collected from this pilot study.

The tasks we chose encompass a subset of the vulnerability remediation activities in the wild. In a talk given at an RSA conference, Dan Cornell describes the activities involved in vulnerability remediation tasks in industry [9]. He illustrates how the tasks performed in our study compare to tasks outside the lab. By his account, vulnerability remediation includes activities such as planning, setting up development environments, performing functional testing, and deployment. Because our study focuses on how developers diagnose vulnerabilities, we do not ask participants to perform these auxiliary tasks.

When selecting tasks, we ran FSB on iTrust and identified 118 potential security vulnerabilities across three topics. To increase the diversity of responses, we selected tasks from mutually exclusive topics, as categorized by FSB. For the fourth task, we added a SQL injection vulnerability to iTrust by making minimal alterations to one of the database access objects. Our alterations preserved the functionality of the original code and were based on examples of SQL injection found on OWASP [64] and in open-source projects. We chose to add a SQL injection vulnerability, because among all security vulnerabilities, OWASP ranks injection vulnerabilities as the most critical web application security risk.

For each task, participants were asked to assess code that "may contain security vulnerabilities" and "justify any proposed code changes." Table 2 summarizes each of the four tasks and the remainder of this section provides more detail about each task, including excerpts from the lines FSB initially flagged. All participants progressed through the tasks in a fixed order. The tasks were not otherwise related to each other, aside from the fact that they were all located in iTrust. Although we are not specifically studying task completion time, we report the mean completion time for each task.

Task 1
The method associated with Task 1, `parseSQLFile`, opens a file, reads its contents, and executes the contents of the file as SQL queries against the database. Before opening the file, the method does not escape the filepath, potentially allowing arbitrary SQL files to be executed. However, the method is only ever executed as a utility from within the unit test framework. Therefore, the method is only ever passed a predefined set of filepaths that cannot be maliciously manipulated.

To complete this task, participants needed to recognize that `parseSQLFile` was only used in tests and that the filepaths were essentially hard-coded, which could be accomplished by examining all of `parseSQLFile`'s call locations. The mean completion time for this task was 14 minutes and 49 seconds.

```
private List<String> parseSQLFile(String path)
{
 FileReader r =
  new FileReader(new File(path));
 ...
}
```

Task 2
The method associated with Task 2 is used to generate random

passwords when a new application user is created. FSB warns `Random` should not be used in secure contexts (such as password generation) and instead suggests using `SecureRandom`, a more secure alternative. Using `SecureRandom` does impose a slight performance trade-off, however participants were not explicitly instructed that performance was a concern. Correct fixes for this vulnerability replace `Random` with `SecureRandom`, ensure the number generator is securely seeded, and appropriate API calls are used. The mean completion time for this task was 8 minutes and 52 seconds.

```
public class RandomPassword
{
 private static final Random r = new Random();
 ...
}
```

Task 3
The method associated with Task 3 reads several improperly validated string values from a form. Entering an apostrophe (') into any of the fields modifies the underlying Java server page (JSP) and permits form manipulation. Additional modification of the form fields can produce more unexpected behavior. The values entered into the form are eventually redisplayed on the web page exposing the application to a cross site scripting attack. Correct fixes for this task either modify the JSP to escape output or modify the validation methods to reject apostrophes. Either of these fixes require participants to navigate away from the file containing the FSB warning. The mean completion time for this task was 13 minutes and 19 seconds.

```
protected void doPost(HttpServletRequest req,
    HttpServletResponse response)
{
 currentMID = req.getParameter("currentMID");
 recPhone = req.getParameter("recPhone");
 recEmail = req.getParameter("recEmail");
 ...
}
```

Task 4
In the method associated with Task 4, a SQL statement object is created using string interpolation, which is potentially vulnerable to SQL injection. FSB recommends using `PreparedStatements` instead. In this case, although the inputs had likely already been sanitized elsewhere in the application, the cost of switching to `PreparedStatements` is negligible. Furthermore, the standard as implemented by other database access objects (DAOs) in iTrust is to use the more secure `PreparedStatement` class. Therefore, a correct fix would be to convert the method to use `PreparedStatements`. The mean completion time for this task was 8 minutes.

```
public void addApptRequest(ApptBean bean)
{
 Statement stmt;
 ...
 String query = String.format(
 "INSERT INTO appointmentrequests
 (appt_type, patient_id, doctor_id,
 sched_date, comment, pending, accepted)
 VALUES ('%s', %d, %d, '%s', '%s', %d, %d);",
 bean.getApptType(),
 bean.getPatient(),
 bean.getHcp(),
 bean.getDate(),
 bean.getComment(),
 ...
 stmt.executeUpdate(query);
 ...
}
```

2.3 Data Analysis — Questions

To analyze the data, we first transcribed all the audio-video files using oTranscribe [62]. Each transcript, along with the associated recording, was analyzed by two of the authors for 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.

2.3.1 Question Criteria

Drawing from previous work on utterance interpretation [34], we developed five criteria to assist in the uniform classification of participant statements. A statement was coded as a question only if it met one of the following criteria:

- **The participant explicitly asks a question.**
  Ex: *Why aren't they using `PreparedStatements`?*
- **The participant makes a statement and explores the validity of that statement.**
  Ex: *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."**
  Ex: *I don't know that it's a problem yet.*
- **The participant clearly expresses uncertainty over a statement.**
  Ex: *Well, it's private to this object, right?*
- **The participant clearly expresses an information need by describing plans to acquire information.**
  Ex: *I would figure out where it is being called.*

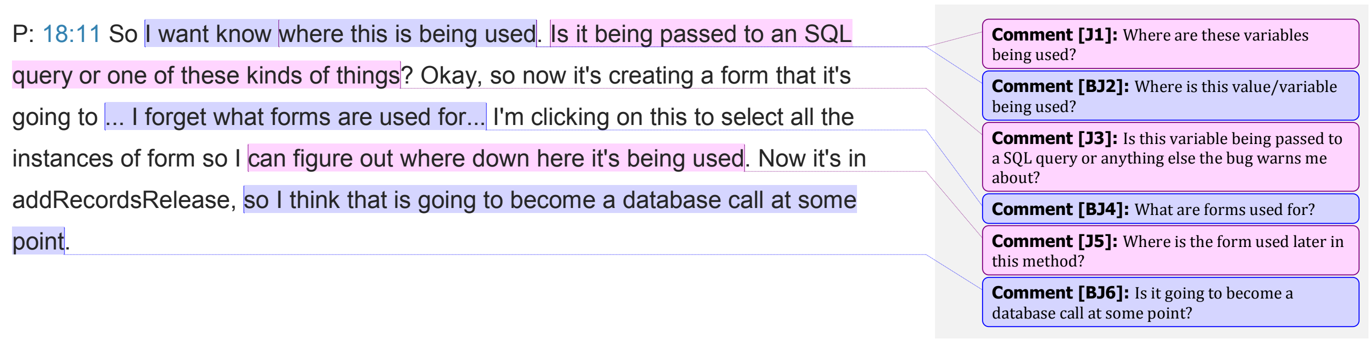
2.3.2 Question Extraction

To make sure our extraction was exhaustive, the first two authors independently coded each transcript using the criteria outlined in the previous section. When we identified a statement that satisfied one or more of the above criteria, we marked the transcript, highlighted the participant's original statement, and clarified the question being asked. Question clarification typically entailed re-wording the question to best reflect the information the participant was trying to acquire. From the ten sessions, the first coder extracted 421 question statements; the other coder extracted 389.

It was sometimes difficult to determine what statements should be extracted as questions; the criteria helped ensure both authors only highlighted the statements that reflected actual questions. Figure 2 depicts a section of the questions extracted by both authors from P8 prior to review.

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"



P: 18:11 So I want know where this is being used. Is it being passed to an SQL query or one of these kinds of things? Okay, so now it's creating a form that it's going to ... I forget what forms are used for... I'm clicking on this to select all the instances of form so I can figure out where down here it's being used. Now it's in addRecordsRelease, so I think that is going to become a database call at some point.

Comment [J1]: Where are these variables being used?

Comment [BJ2]: Where is this value/variable being used?

Comment [J3]: Is this variable being passed to a SQL query or anything else the bug warns me about?

Comment [BJ4]: What are forms used for?

Comment [J5]: Where is the form used later in this method?

Comment [BJ6]: Is it going to become a database call at some point?

Fig. 2. Question merging process

2.3.3 Question Review

To remove duplicates and ensure the validity of all the questions, each transcript was reviewed jointly by the two authors who initially coded it. During this second pass, the two reviewers examined each question statement, discussing its justification based on the previously stated criteria. The two reviewers merged duplicate questions, favoring the wording that was most strongly grounded in the study artifacts. Including questions that were only identified by one reviewer, 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.

It is also important to note that participants' questions related to several topics in addition to security. We discuss the questions that are most closely connected to security in Sections 3.2.1, 3.3.6, and 3.5.3. Although our primary focus is security, we are also interested in the other questions that participants posed, as those questions often have security implications and require special considerations in the context of security. For example, researchers have observed that developers ask questions about data flow, like *What is the original source of this data*, even outside security [32]. A developer concerned with security might need to consider that attackers are one potential source of data. Data originating from an attacker could be crafted to disguise its malicious nature or to do more damage to the system. In such a context, the implications are somewhat unique to security, because, for example, potentially insecure data sources often require special handling to prevent attacks.

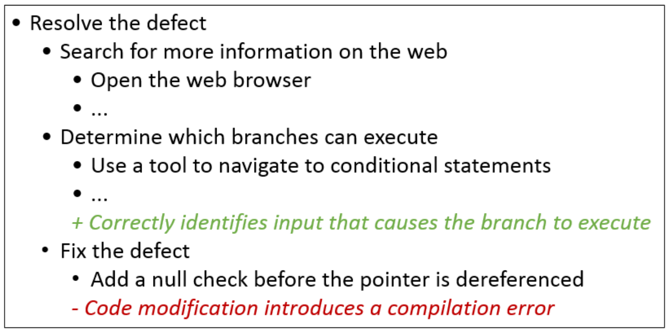
2.3.4 Question Sorting

To organize our questions and facilitate discussion, we performed an *open* card sort [22]. 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 needs. 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 154 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 [30].



- Resolve the defect
  - Search for more information on the web
    - Open the web browser
    - ...
  - Determine which branches can execute
    - Use a tool to navigate to conditional statements
    - ...

    + Correctly identifies input that causes the branch to execute
  - Fix the defect
    - Add a null check before the pointer is dereferenced

    - Code modification introduces a compilation error

Fig. 3. Example strategy tree for a null dereference defect.

2.4 Data Analysis — Strategies

2.4.1 Strategy Definitions

We define a defect resolution strategy as all the actions a developer takes to validate and resolve a defect. Such actions include invoking tools, modifying the source code, and searching the web, among many other things. We adapt Bhavani and John's related definition of a strategy, "a method of task decomposition that is non-obligatory and goal directed," to determine which actions to include in a strategy [4].

Consider the following example which illustrates this notion of non-obligatory actions and how we use it as a stopping criteria for our strategy analysis. Say a developer's goal is to get more information about the code. One strategy would be to read the code itself, another would be to read the documentation. These are both considered strategies because they help the developer achieve a goal, and the developer is not obligated to choose either one of these approaches. Now consider a negative example where a developer's goal is to read the code. We wouldn't consider the developer's act of fixating his gaze on the screen a strategy, because it is obligatory (assuming the developer does not also use a screen reader or other similar technology).

There are many ways to represent strategies. Our screen and audio recordings represent strategies with high fidelity because they contain everything that happened during the study. However, this format is difficult to work with and does not summarize the essence of a strategy. Another format we considered was to synthesize each participant's strategies into unstructured textual summaries. This format has the advantage over raw recordings of being easy to search and consume, but it does not represent the relationships between strategies.

Ultimately, we created a new notation for representing strategies, called *strategy trees*. Strategy trees are based on the idea of attack trees [37]. Attack trees encode the actions an attacker could take to exploit a system, including the idea that he may combine multiple actions to achieve a higher-level goal. Much like attack trees, our representation organizes actions hierarchically. Whereas attack trees describe an attacker's actions, we use strategy trees to represent the hierarchical set of actions a developer takes to resolve a security defect. Figure 3 depicts an example strategy tree.

2.4.2 Strategy Extraction

To answer RQ2 the first author performed two additional passes through the transcripts and recordings. This analysis took place after the questions analysis had been completed and was conducted separately. In the first pass, we considered participants' screen recordings and think-aloud verbalizations to identify their defect resolution strategy for each task. We watched the videos, pausing approximately every minute to take notes. Using the definitions of strategies and stopping criteria described in the previous section (Section 2.4.1) we decided what to include in the strategy trees. This process resulted in 40 strategy trees, which we finally referenced against each other to ensure the use of consistent terminology.

During the second pass we added annotations to each strategy tree. As depicted in Figure 3 (lines prefixed with $+/-$), we annotated the trees whenever an action led to a success or a failure. To best understand which strategies contributed to success, we measured success/failure as granularly as possible. Rather than annotating the entire tree as successful or not, we annotated the sub-strategies that compose the tree. Participants could succeed with some strategies while failing with others. For example, participants could succeed in locating relevant information, but fail to interpret it correctly. Some sub-strategies were not observably successful/failure-inducing and were, therefore, not annotated.

The following criteria, though not complete, guided us when in determining when strategies were successful or unsuccessful. Whenever we observed an outcome that contributed to a participant's understanding of the problem or demonstrated progress toward a solution, it was marked as a success. Whenever we observed the participant making a verifiably incorrect statement, abandoning a strategy without achieving the desired result, or overlooking information they sought, it was marked as a failure.

2.4.3 Strategy Review

A second researcher who had not been involved in the original strategy tree extraction, the second author, assessed the completeness and validity of the strategy tree analysis. Rather than duplicate the strategy tree extraction process, the researcher reviewed the existing trees. Extracting the strategy trees from scratch consumed a considerable amount of time, approximately 50 hours. Roughly, the process involved watching each of the 40 task videos, pausing every minute to take notes, and iteratively cross-checking all the trees for consistent terminology. In comparison, reviewing the existing trees took 10 hours.

To assess the completeness of the strategy trees, the reviewer was instructed to, "consider whether each strategy tree is missing any important elements." The reviewer commented on 3 of the 40 tasks (7.5%). These discrepancies were resolved after they were discussed by the first and second authors. Relatively few discrepancies between evaluators suggests the trees capture most strategies participants executed.

To validate the extracted strategy trees, the second researcher reviewed each tree while watching the screen/audio recording corresponding to that tree. The reviewer was instructed to confirm the presence of each strategy tree element in each video and placed a check mark next to tree elements as they were observed. This process resulted in the reviewer checking off all of the tree elements, which suggests our strategy trees accurately reflect actions that actually occurred.

2.5 Data Analysis — Assumptions

Like strategies, assumptions can satisfy developers information needs. Unlike strategies, assumptions do not comprise any actions and instead represent an accepted truth without proof. To draw a programming analogy, one can think of assumptions as *null* strategies — satisfying an information need without investigation.

To answer RQ3, we identified participants' assumptions and analyzed each assumption to determine if it was correct. For the purposes of our study, assumptions are defined rather narrowly so that they can be identified unambiguously. Specifically, we only consider the assumptions participants explicitly state using the word "assume" or related words. This choice enables us to accurately identify assumptions, but underapproximates the assumptions made during our study.

Participants likely made many assumptions implicitly that we did not capture. Because such implicit assumptions are pervasive [11], it would be intractable to identify them all. For example, participants may have implicitly assumed the Eclipse environment was configured correctly and all the relevant source files were accessible. Instead of speculating about all of these assumptions, we only consider the assumptions participants explicitly stated.

To identify the explicit assumptions participants made, we searched the transcripts for keywords. To derive the search set, we started with the following base words: *assume*, *presume*, *reckon*, *believe*, and *guess*. Next, we applied lemmatisation to identify all the inflected forms of these words, such as *hypothesize*, *speculate*, *suppose*, etc. Finally, we searched the transcripts for the stems of all the words identified by lemmatisation.

This search process returned some false-positives. For example, one participant referred to the interviewer's assumptions rather than making an assumption of his own asking, "Am I *assumed* to have some prior knowledge?" To filter only the assumptions participants made about the task or the code, the first author inspected each search result.

After determining the result was an assumption, we evaluated its correctness. For example, one participant (P4) assumed an external library was implemented securely. We determined this assumption was incorrect by locating a vulnerability in the library using the common vulnerability and exposures database.[1] We discuss this particular assumption in Section 3.3.4 and the rest of the assumptions throughout Section 3.

3 Results

3.1 Interpreting the Results

In the next four sections, we discuss our study's results using the categories we described in Section 2.3. Due to their large number, we grouped the categories to organize and facilitate discussion about our findings. Table 3 provides an overview of these groupings. The table also describes which tasks each question category occurred in (e.g., we observed questions from the Understanding Concepts category in all four tasks).

For each category, we selected several questions to discuss. A full categorization of questions can be found online and in the appendix along with the full descriptions of participants strategies and assumptions [58]. 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 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.

1. cve.mitre.org

The structure of most results categories consists of five parts: an overview of the category, several of the questions we selected, a discussion of those questions (RQ1), a discussion relating the category to questions from previous studies, and a discussion of how participants answer the questions in that category (RQ2 and RQ3). Some sections contain less discussion than others either because participants' intentions in asking questions were unclear, or participants asked questions without following up or attempting to answer them at all.

To answer RQ2, we describe developers' defect resolution strategies. These strategies for answering individual questions often overlap, especially within a given category. Accordingly, to present a coherent discussion and avoid duplication, we discuss strategies for each category rather than for each question.

In this section we also describe participants' correct and incorrect assumptions (RQ3) as well as elaborate on how those assumptions contributed to defect resolution. While performing their tasks, participants made different types of assumptions. Overall, we observed 73 total assumptions — 27 incorrect and 46 correct. We identified assumptions across all participants, except for P3, who may have made assumptions without stating them explicitly. Additionally, participants stated at least one assumption during each task.

3.2 Vulnerabilities, Attacks, and Fixes

3.2.1 Preventing and Understanding 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 (Table 2) 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)

Participants sought information about the types 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 read the notification about the Path Traversal vulnerability and stated, "I guess I'm thinking about different types of attacks." Before reasoning about how a specific attack could be executed, he wanted to determine which attacks were relevant to the notification.

Participants also sought information about specific attacks from the notification, asking how particular attacks could exploit a given vulnerability. Participants hypothesized about specific attack vectors, how to execute those attacks, and how to prevent those attacks now and in the future. Seven participants, concerned about false positives, asked the question, *Is this a real vulnerability?* To answer that question, participants searched for hints that an attacker could successfully 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

TABLE 3
Organizational Groups and Emergent Categories

Group	Category	Tasks	Clusters	Location in Paper
Vulnerabilities, Attacks, and Fixes	Preventing and Understanding Potential Attacks	T1 – T4	11	Section 3.2.1
	Understanding Approaches and Fixes	T1 – T4	10	Section 3.2.2
	Assessing the Application of the Fix	T1 – T4	9	Section 3.2.3
	Relationship Between Vulnerabilities	T2 – T4	3	Section 3.2.4
Code and the Application	Locating Information	T1 – T4	11	Section 3.3.1
	Control Flow and Call Information	T1 – T4	13	Section 3.3.2
	Data Storage and Flow	T1, T3, T4	11	Section 3.3.3
	Code Background and Functionality	T1 – T4	17	Section 3.3.4
	Application Context/Usage	T1 – T4	9	Section 3.3.5
	End-User Interaction	T1, T3, T4	3	Section 3.3.6
Individuals	Developer Planning and Self-Reflection	T1 – T4	14	Section 3.4.1
	Understanding Concepts	T1 – T4	6	Section 3.4.2
	Confirming Expectations	T1 – T4	1	Section 3.4.3
Problem Solving Support	Resources and Documentation	T1 – T4	10	Section 3.5.1
	Understanding and Interacting with Tools	T1 – T4	9	Section 3.5.2
	Vulnerability Severity and Rank	T1 – T4	4	Section 3.5.3
	Notification Text	T1 – T4	3	Section 3.5.4
	Uncategorized	T1 – T4	10	

random seed and use that information to determine other users' passwords.

Previous studies have identified related information needs pertaining to preventing and understanding bugs, problems, defects, and failures. For example, several questions from the prior literature appear similar to the questions we identified: "Is this a problem?" [27]; "What does the failure look like?" [27]; "How do I debug in this environment?" [32]. However, the difference in terminology between these prior findings and ours (problems/failures/bugs vs. attacks) reflects the novelty of our contribution. As we discussed in Section 2.3, attacks require special consideration because they originate from a malicious agents.

Strategies and Assumptions: Participants used various strategies to answer questions about attacks. When available, participants read FSB's vulnerability information. When FSB did not provide sufficient information, participants turned to the web, searching on sites like Google and StackOverflow.

These strategies for getting information about different types of attacks were prone to two types of failures. First, because web search engines were not fully aware of participants' programming contexts, they returned information about a superset of the relevant attacks. For example, P2 searched for attacks that exploit unvalidated input vulnerabilities. Searching the web returns results about cross site scripting attacks, injection attacks, and buffer overflow attacks. However, due to Java's automatic array bounds checking, buffer overflow attacks are not feasible in the vast majority of Java programs, including iTrust. Devoting more effort to buffer overflow attacks would have distracted P2 from the other more relevant attacks.

Secondly, by executing these strategies some participants erroneously considered only a subset of the possible attacks. This failure was especially evident for the Servlet Parameter vulnerability, where a cross site scripting attack was feasible, but a SQL injection attack was not. Some participants correctly determined the data was sanitized before reaching the database, dismissed SQL injection, and prematurely concluded the vulnerability was a false positive. By failing to consider cross site scripting attacks, participants overlooked the program path that exposed a true attack.

3.2.2 Understanding Approaches and Fixes (8){10}

When resolving security vulnerabilities, participants explored alternative ways to achieve the same functionality more securely. For example, while evaluating the potential 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)
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)

Some notifications, including those for the SQL Injection and Predictable Random vulnerabilities, explicitly offered fix suggestions. In other cases, participants turned to a variety of sources, such as StackOverflow, official documentation, and personal blogs for alternative approaches.

Three participants specifically cited StackOverflow as a source for alternative approaches and fixes. P7 preferred StackOverflow as a resource, because it included real-world examples of broken code and elaborated on why the example was broken. Despite the useful information some participants found, often the candidate alternative did not readily provide meta-information about the process of applying it to the code. 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.

While attempting to assess the Servlet Parameter vulnerability, P8 decided to explore some resources on the web and came across a resource that appeared to be affiliated with OWASP [64]. Because he recognized OWASP as "the authority on security," he clicked the link and used it to make his final decision regarding the vulnerability. It seemed important to P8 that recommended approaches came from trustworthy sources.

Previous studies have similarly observed that developers ask questions like those in this category. For instance, Ko and colleagues report developers ask, "What data structures or functions can be used to implement this behavior?" while considering the 'space of existing reusable code' [27]. Additionally, it has similarly been reported that developers ask, "Which function or object should I pick?" [32] and need information about alternative upstream frameworks ("Comparison with similar upstreams") [20]. These previous findings pertain to general programming tasks or implementations of new behavior, which is the primary difference compared with our results. Our results reveal the salience of these questions to developers working on *defective* code.

Strategies and Assumptions: Participants' strategies for acquiring information about fixes centered around three information sources — FSB, the web, and other modules in the code. Even when FSB informed participants about a specific alternative, participants sought supplementary information from the other two external sources.

For example, during Task 4, FSB suggested using `PreparedStatements`. In search of information about `PreparedStatements'` syntax, participants navigated to the project's related code modules they suspected would contain `PreparedStatements`. Participants largely found examples from these related modules helpful, because they were easy to translate back to the original method.

For Task 2, participants similarly sought examples of `SecureRandom`, but were ultimately less successful. Unaware that the project contained examples of `SecureRandom`, in this case, participants turned to online resources. Although some participants understood the relevant online API documentation, others struggled to compare the online documentation to the original method.

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

Once participants had identified an approach for fixing a security vulnerability (Section 3.2.2), they asked questions about applying the fix to the code. For example, while considering the use of `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 Approaches and Fixes*** (Section 3.2.2). These questions focus on the process of applying and reasoning about a given fix, rather than identifying and understanding possible fixes.

Will the notification 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)

When searching for approaches to resolve vulnerabilities, participants gravitated toward fix suggestions provided by the notification. As noted above, 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 `SecureRandom` without considering any alternative solutions, even though the use of that suggested fix reduces performance. It seems that providing suggestions without discussing the associated trade-offs appeared to reduce participants' willingness to think broadly about other possible solutions.

Based on the identification of these questions, assessing the application of fixes is important in a security context, but perhaps not unique to security. Prior studies have identified some questions that are similar to those asked by our participants: "Did I make any mistakes in my new code?" [27]; "What are the implications of this change for API clients, security, concurrency, performance, platforms, tests, or obfuscation?" [32]; "Will this completely solve the problem or provide the enhancement?" [47].

Strategies and Assumptions: Although participants were not asked to make any modifications to the code, they did describe their strategies for applying and assessing fixes. Many participants looked for ways to apply fixes automatically. However, none of the FSB notifications included quick fixes. Still, some participants unsuccessfully attempted to apply quick fixes by clicking on various interface elements.

To assess whether the fix had been applied correctly, participants commonly described one particular heuristic. P1, for instance, would make the change and simply "see if the bug [icon] goes away." This strategy typically ensures the original vulnerability gets fixed, but fails when the fix introduces new defects.

3.2.4 Relationship Between Vulnerabilities (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 speculating about the similarities between the vulnerabilities they inspected.

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

Previous studies report questions about relationships between code elements, but not defects [12], [32]. For instance, "Which other code that I worked on uses this code pattern / utility function?" [12]. Ko and colleagues briefly discuss the relationships between defects as it relates to the question: "Is this a legitimate problem?" Sometimes developers answered this question by determining if the same problem had been reported by others as a duplicate [27].

Strategies and Assumptions: Participants asked a few passing questions about the relationships between vulnerabilities, but were not explicitly asked to relate the vulnerabilities to each other across tasks. As a result, we did not observe the emergence of any predominant strategies or assumptions in this category.

3.3 Code and the Application

3.3.1 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 3.3.2 and 3.3.3, questions in this category more generally refer 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)

All ten participants wanted to locate where defective code and tainted values were in 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 and whether it was employed to generate sensitive data like passwords.

In other cases, while fixing one method, four participants wanted to find other methods that implemented similar functionality. They hypothesized that other code modules 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.

Asking 'Where' questions and locating information has been broadly discussed in the prior literature, though the types of information sought depend on the study. Some developers asked questions about locating recent changes made by teammates such as "Where has code been changing [this week]?" and "Where have changes been made related to you?" [12]. Others ask questions about the functionality of the code, more resembling the questions we identified: "Where is this functionality implemented?" [32]; "Where is this defined?" [32]; "Where is this variable or data structure being accessed?" [47]; "Where are instances of this class created?" [47]. None of Ko and colleagues' questions refer to locating information, but they report, as we will discuss subsequently, that participants used search tools to find information. [27].

Strategies and Assumptions: For the majority of participants, scrolling through the source file was a key component of their strategies for locating information. As Robillard and colleagues observed, such unstructured navigation may signal ineffectiveness [44]. Eclipse provides many tools designed to help users locate specific types of information. For example, OPEN DECLARATION locates method declarations, FIND REFERENCES locates references. More generally Eclipse provides a customizable SEARCH tool for locating other information. Despite the availability of such dedicated tools from the start, many participants first scrolled through the package explorer and open files, failed to find the information they needed, then switched to using tools. We hypothesize that developers did not use search tools because of a lack of familiarity and that knowledge of tools improves developers' effectiveness in resolving security defects.

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

Participants sought information about the callers and callees of potentially vulnerable methods.

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)

Participants asked some of these questions while exploring the Path Traversal vulnerability. While exploring this vulnerability, many participants eventually hypothesized that all the calls originated from the same test class, therefore were not user-facing, and thus would not be called with tainted values. Three participants explicitly asked, *Are all calls coming from the same class?* In fact, in this case, participants' hypotheses were partially correct. Tracing up the call chains, the method containing the vulnerability was called from multiple classes, however those classes were all contained within a test package. Even though all participants did not form this same hypothesis, all ten participants wanted call information for the Path Traversal Vulnerability, often asking the question, *Where is this method being called?*

Call information is not a security-specific information need. For instance, prior studies have identified the following related questions: "What's statically related to this code?" [27]; "When during the execution is this method called?" [47]; "Where is this method called or type referenced?" [47]. Further, Latoza and Myers identified a similar category of 'Hard-to-answer questions' in their study. [32] Many questions they report overlap with those that we report. However, they do not report questions about the distribution of call sites, in other words, whether calls all originate from the same source or multiple possible call sites. As we discussed above, this type of question had security implications, because it helped our participants assess whether a vulnerability could be exploited.

Strategies and Assumptions: Participants used various strategies to obtain information about control flow. 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 Eclipse's MARK OCCURRENCES tool (code highlighting near (B) in Figure 1), which, to a lesser extent, was error-prone for the same reason. Further, it only highlighted calls within the current file.

Participants additionally employed Eclipse's FIND 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 identified references to a single method. Eclipse's CALL HIERARCHY tool enabled users to locate calls and traverse the project's entire call structure. That said, it only identified explicit calls made from within the system. If the potentially vulnerable code was called from external frameworks, CALL HIERARCHY would not alert the user.

We hypothesize that developers first default to the tools and techniques, like scrolling or using MARK OCCURRENCES, that are easiest for them to invoke. Which tools are easier to use may depend on an individual developer's familiarity.

3.3.3 Data Storage and Flow (10){11}

Participants often wanted to better understand data being collected and stored: where it originated and where it was going. For example, participants wanted to determine whether data was generated by the application or passed in by the user. Participants also wanted to know if the data touched sensitive resources like a database. Questions in this category focus on the application's data — how it is created, modified, or used — unlike the questions in Section 3.3.2 that revolve around call information, essentially the paths through which the data can travel.

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)

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

In their study, Sillito and colleagues identified two questions that relate to the questions in this category: "What data can we access from this object?" and "What data is being modified in this code?" [47]. Additional questions arose in the study by Latoza and Myers, who identified a similarly named category [32]. Unlike these prior studies, we observed some unique questions with security implications. In our study, for example, participants asked whether data reached sensitive contexts (e.g., databases or web displays) and also whether all data passed through sanitization/validation before reaching those contexts. This suggests that developers might require dedicated tool support — beyond what is provided by general navigation tools — to answer these questions.

Strategies and Assumptions: While exploring this vulnerability, participants adapted tools such as the CALL HIERARCHY tool to also explore the program's data flow. As we discussed in *Control Flow and Call Information*, the CALL HIERARCHY tool helped participants identify methods' callers and callees. Specifically, some participants used the CALL HIERARCHY tool to locate methods that were generating or modifying data. Once participants identified which methods were manipulating data, they manually searched within the method for the specific statements that could modify or create data. They relied on manual searching, because the tool they were using to navigate the program's flow, CALL HIERARCHY, did not provide information about which statements were modifying and creating data.

Some participants failed because they used aggregation strategies, rather than tracing individual data elements. This was particularly evident for Task 3, where 12 variables were aggregated in a form object. By tracing individual variables, some participants uncovered the incorrect validation function. Others, like P6, grew tired of tracing each variable and instead considered the form as a whole. As a result, he overlooked the validation function and incorrectly concluded the vulnerability was a false positive.

Rather than investigate, participants also made assumptions about where data comes from, whether it be from a user or hard-coded test data. For Task 1, P1 incorrectly assumed data was passed in from a user. He followed up on this weakly-held assumption by further investigating the origin of the data. His successful investigation strategy, which included him using the CALL HIERARCHY tool, led him to correct his initial assumption and resolve the defect correctly.

3.3.4 Code Background and Functionality (9){17}

Participants asked questions concerning the background and the intended function of the code being analyzed. The questions in this category differ from those in Section 3.3.5 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)

Participants were interested in what the code did as well as the history of the code. For example, P2 asked about the amount of effort put into the code to determine whether he trusted that the code was written securely. He explained, "People were rushed and crunched for time, so I'm not surprised to see an issue in the servlets." Knowing whether the code was thrown together haphazardly versus reviewed and edited carefully might help developers determine if searching for vulnerabilities will likely yield true positives.

Questions about code background have been commonly reported by previous studies: "How much work [have] people done?" [12]; "Who created the API [that I'm about to change]?" [12]; "Why was this code implemented this way?" [27]; "Why was it done this way?" [32]; "Why wasn't it done this other way?" [32]; "Was this intentional, accidental, or a hack?" [32]; "When, how, by whom, and why was this code changed or inserted?" [32]. One difference in a security context is the subtext of the questions. As we discussed above, our participants asked about rushed components and components designed by novices, because those components might be more vulnerable. Answering questions about code background could help developers allocate their scarce security resources.

Strategies and Assumptions: To answer questions in this category, participants relied primarily on their prior knowledge of the system. Although they were prohibited by our study design, developers in collocated teams also seek this design rationale from the author of the code through face-to-face conversations [27]. We observed relatively few instances where participants executed additional strategies, such as looking at version control information or system documentation, to substantiate their existing knowledge.

This strategy, or lack thereof, was susceptible to failures when participants overestimated their knowledge of the system. For example, while assessing the Path Traversal vulnerability, P2 incorrectly assumed that the methods in the DBBulder module were only called during development and deployment when iTrust first starts up.

Participants also sought information about external libraries and commonly made assumptions about these external libraries, their correctness, and their functionality. Some assumptions centered around the proper usage of an external library. For instance, while skimming `SecureRandom`'s documentation during Task 2, P8 incorrectly assumed `SecureRandom` does not provide a `nextInt` method. Based on that assumption, P8 proposed a workaround that was valid, but duplicated the functionality `nextInt` already provided.

Other assumptions pertained to the security of external libraries. For example, P7 correctly assumed that `SecureRandom` handles the secure seeding of random number generators. In contrast with P8, P7's correct assumption led him to propose a more succinct solution. P4 stated that he typically uses the Django framework for web applications and assumed that using such external library frameworks meant that he was following best security practices. Though this assumption did not directly impact any of P4's tasks, it illustrates a potentially troubling trust for external libraries. Unfortunately, web framework libraries like Django are susceptible to their own vulnerabilities, many of which have been enumerated in online databases.[2] We hypothesize that developers look for shallow cues, like a familiar name (Django) or certain keywords (Secure) while assessing the trustworthiness of external libraries.

3.3.5 Application Context and Usage (9){9}

Unlike questions in Section 3.3.4, these questions refer to system-level concepts. For instance, often while 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 vulnerability/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)

Participants tried to determine if the code in the Potential Path Traversal 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.

Prior studies report that developers ask about how code is generally intended to be used: "What is the purpose of this code" [27] "What is the intent of this code?" [32]. More closely related to the questions in this category, developers ask, "How do test cases relate to packages/classes?" [12]. As we will discuss below, in a security context participants asked this type of question to determine whether to scrutinize a module.

Strategies and Assumptions: Participants were particularly interested in differentiating between two contexts — test code and application code. As P4 explained, test code does not ship with the product, so it is held to a different standard from a security perspective. To answer their questions about tests, participants sometimes used tools for traversing the call hierarchy; using these types of tools allowed them to narrow their search to only locations where the code of interest was being called.

Participants' strategies for this task were undermined by the FSB "Bug Explorer" view, because it obscured many of the usual cues participants would have used to differentiate between contexts. In iTrust, tests are organized into a test package, which is separate from the application code. By default, the "Bug Explorer" view closes the package explorer and omits package names from the notification text. Without visible package names or the package explorer, some participants were misled. For instance, P9 incorrectly inferred the code for Task 3 was contained within a test class before eventually correcting his mistake, "This one I would be more inclined to ignore, because it's a test method I believe... Wait no, it's not a test method. No, it's not." Had P9 not corrected his mistake, he would have misdiagnosed the error and ignored a true positive.

3.3.6 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)

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. While working on the same task, four participants also asked whether end-user input reached the code being analyzed.

The questions in this category are most closely related to *reachability questions*, and with some effort could be rephrased as such [31]. For instance, *Is there input coming from the user?*, asks whether there is a feasible upstream trace originating at a user input function reaching the current statement. Otherwise, prior studies that report information needs in general programming contexts say little about end-user interactions.

However, these questions do pertain to security research on attack surfaces [21] and attack surface approximation [52]. An attack trace, or the sum of all paths for untrusted data into and out of a system, describes where end-user input interacts with a system. We hypothesize that providing developers attack surface information, such as whether a program point is on the attack surface, could help them answer the questions in this category.

Strategies and Assumptions: For participants, answering their questions required manual inspection of the source code. For instance, P6 found a `Validator` method, which he manually inspected, to determine if it was doing input validation. He incorrectly concluded that the `Validator` method adequately validated the data.

P2 used CALL HIERARCHY to trace end-user input; he assumed the vulnerability was a true positive if user input reached the vulnerable line. P1 and P6 searched similarly and determined that because all the calls to the code of interest appeared to happen in methods called `testDataGenerator()`, the code was not vulnerable.

More generally, participants' strategies for answering questions about end-user interaction typically included strategies from other categories. In other words, participants gathered evidence by Locating Information (Section 3.3.1), examining Control Flow (Section 3.3.2), retracing Data Flow (Section 3.3.3) information, and recalling information about the Code Background (Section 3.3.4). Participants successfully answered questions in this category when they successfully synthesized the results of these other efforts.

3.4 Individuals

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

This category contains questions that participants asked about themselves. The questions in this category involve the participants' relationship to the problem, rather than specifics of the code or the vulnerability notification.

2. cve.mitre.org

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)

Participants most frequently asked if they understood the situation, whether it be the code, the notification, or a piece of documentation. For instance, as P6 started exploring the validity of the SQL Injection vulnerability, he wanted to know if he fully understood the notification before he started exploring, so he went back to reread the notification before investigating further. These questions occurred in all four vulnerabilities. We identified two questions in the prior literature that could be categorized here: "What am I supposed to work on?" [12] and "Is the problem worth fixing?" [27].

Strategies and Assumptions: Compared to other categories, the strategy and assumption analysis results in this category are sparse. Although we asked participants to think-aloud, they seemed to resolve questions in this category internally.

3.4.2 Understanding Concepts (7){6}

Some participants encountered unfamiliar terms and concepts in the code and vulnerability notifications, which prompted them to ask questions.

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

For some vulnerability patterns, FSB links to resources that define relevant concepts. Otherwise, information about concepts and terminology could be found online. When asked what information he would like to see added to the notification for the Servlet Parameter vulnerability, which did not include any links, P4 noted he would have liked the notification to include links defining a servlet is and how it related to client control.

Strategies and Assumptions: Participants' strategies for finding information about concepts included clicking the links provided by FSB. For example, while assessing the Potential Path Traversal vulnerability, P2, unsure of what path traversal was, clicked a link, labeled "path traversal attack," provided by FSB to get more information. However, this strategy failed when the hyperlink text poorly described the links' contents. Unable to determine the quality and type of information hidden behind FSB's links, some participants preferred to search the web for information about concepts.

If a link was not available or if its contents were unclear, participants went to the web to get more information. For instance, P7 and P8 both searched the web for the same unfamiliar term ('CSRF token') while working on the Predictable Random vulnerability — for this task, FSB did not provide any links.

We observed participants using several noteworthy sub-strategies while searching online. To find information pertaining to their particular situation, several participants copied and pasted content directly from the FSB notification into their web browser. Participants also demonstrated the simple, yet successful, strategy of iteratively refining their search terms. For example, after performing a search that failed to return relevant results, P1 added specificity to his search by adding the term 'Java'. Finally, we noticed some consistency in the strategies participants were using to find information about concepts — several different participants used the exact same search terms.

3.4.3 Confirming Expectations (4){1}

A few participants wanted to be able to confirm whether the code accomplishes what they expected. The question asked in this category was, *Is this doing what I expect it to?*

Strategies and Assumptions: We did not observe many overarching strategies or assumptions in this category, in part because participants rarely signaled to us when they were testing their expectations. When we did observe participants questioning their expectations, their strategies varied situationally.

3.5 Problem Solving Support

3.5.1 Resources and Documentation (10){10}

Many participants indicated they would use external resources and documentation to gain new perspectives on vulnerabilities. For example, while 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 prevent or resolve this? (5)
Is this a reliable/trusted resource? (3)
What type of information does this resource link me to? (2)

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 notifications, 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 for them to know what information the link was offering; others did not know if they could trust the information they found.

Some participants clicked FSB's links expecting one type of information, but finding another. For example, P2 clicked the first link, labeled "WASC: 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 attempted another web search for the same information. A few participants did not know the links existed, so they typically used other strategies, such as searching the web.

Other participants expressed interest in consulting their team members. For example, when P10 had difficulty with the Potential Path Traversal vulnerability, he stated that he would normally ask his team members to explain how the code worked. Presumably, the code's author could explain how the code was working, enabling the developer to proceed with fixing the vulnerability.

Prior information needs studies discuss the use of resources and documentation. For instance, Fritz and Murphy report that developers rely on team members for code review, "Who has knowledge to do code review?". In software ecosystems, downstream users rely on the availability of good documentation [20]. Ko and colleagues's questions do not refer directly to documentation or help resources [27]. Instead, documentation is discussed as a means of obtaining answers to questions. It seems quality help resources and documentation are valued in many contexts, including but not limited to security.

Strategies and Assumptions: Participants' primary strategy involved consulting the code (i.e. variable and class names) along with its sparse comments. Participants seemed to successfully use this information to understand basic code constructs (i.e. control structures, variable types, method calls, and which variables were being stored in which data structures). However, these strategies sometimes failed to convey security-relevant semantic information. Further, this strategy fell short when participants sought documentation for IDE tools or external APIs; such information was not readily available within the code itself.

Supplementing their primary strategy, participants gathered additional information from web resources. Additionally, participants described their plans to contact team members for help, although they could not do so within the confines of the study.

3.5.2 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 use the tools, and how to interpret their 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)
How can I annotate that these strings have been escaped and the tool should ignore the warning? (1)

Participants asked questions about accessing the tools needed to complete a certain task. Participants sometimes sought information from a tool, but could not determine how to invoke the tool or possibly did not know which tool to use. The question, *What tool do I need for this?* points to a common blocker for both novice and experienced developers, a lack of awareness [39].

Other information needs studies discuss how developers use tools to answer questions and that there are mismatches between questions asked and those that tools can answer [27], [47]. However, relatively few studies report any questions about tools themselves.

Strategies and Assumptions: We observed two types of strategies in this category, tool selection strategies and tool evaluation strategies. While selecting tools, participants often knew what functionality they wanted to achieve, but were unsure how to find a tool to achieve that functionality. For example, P2 knew he wanted to navigate to methods up the call hierarchy, but struggled to identify an appropriate tool. His tool selection strategy involved first Ctrl-hovering over the current method's name followed by right clicking the method name. This strategy failed because the tool was not available in the Ctrl-hover menu and he failed to recognize the tool in the right-click menu.

Rather than search for the ideal tool, some participants opted to opportunistically invoke tools and evaluate the appropriateness of their output. Unfortunately, these strategies were susceptible to misinterpretations. For example, one participant opened, closed, and reopened the CALL HIERARCHY tool several times, unable to determine whether it was appropriate.

When FSB failed to effectively communicate information about the location of defects to participants, they made assumptions. For Task 3, P10 incorrectly assumed FSB was indicating an issue with the database access objects (DAOs), rather than the sanitization methods. He probably made this assumption, because FSB placed its bug markers in the file containing the DAOs and not in their associated sanitization methods. Based on that assumption, he proposed adding a sanitization method to those objects. This fix is incorrect, because sanitization methods already existed in another class and just needed to be modified. The proposed fix would have resolved the vulnerability locally, but also would have introduced redundant code and violated iTrust's architectural convention of organizing all validator classes in the /validate folder. Furthermore, any component still using the faulty sanitization methods would still be vulnerable.

3.5.3 Vulnerability 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 vulnerability? (2)
How do the rankings compare? (2)
What do the vulnerability rankings mean? (2)
Are all these vulnerabilities the same severity? (1)

Most of these questions came from participants wanting to know more about the tool's method of ranking the vulnerabilities in the code. For example, after completing the first task (Potential Path Traversal), P1 discovered the tool's rank, severity, and confidence reports. He noted how helpful the rankings seemed and included them in his assessment process for the following vulnerabilities. As he began working through the final vulnerability (SQL Injection), he admitted that he did not understand the tool's metrics as well as he thought. He wasn't sure whether the rank (15) was high or low and if yellow was a "good" or "bad" color. Some participants, like P6, did not notice any of the rankings until after completing all four sessions when the investigator asked about the tool's rankings.

Participants in Ko and colleagues' study made similar inquiries about the severity of defects, asking: "Is this a legitimate problem?" [27]; "How difficult will this problem be to fix?" [27]; "Is the problem worth fixing?" [27].

Strategies and Assumptions: Severity ratings typically help developers triage vulnerabilities. Due to the limitations of our controlled study, we preselected the vulnerabilities and the order in which they would appear. As a result, working with the severity rankings was not critical for participants to complete their tasks. Therefore, we observed few assumptions and relatively shallow strategies in this category; some participants asked questions about the rankings, but most did not follow up with an investigation.

3.5.4 Notification Text (6){3}

FSB provided short and long descriptions of each vulnerability (See (A) and (C) in Figure 1, respectively). Participants read and contemplated these notifications to guide their analysis.

What does the notification text say? (5)
What is the relationship between the notification text and the code? (2)
What code caused this notification to appear (2)

Beyond asking about the content of the notification, participants also asked questions about how to relate information contained in the notification 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.

As we discussed in Section 3.5.2 prior information needs studies discuss the use of tools, but identified relatively few questions about the tools themselves. Similarly, relatively few questions about tool notifications have been reported by previous studies. Sillito and colleagues identify,"Where in the code is the text in this error message or UI element?" which is similar to our, *What is the relationship between the notification text and the code?* [47]. Ko and colleagues report that their participants struggled to make sense of tool notifications, "Three developers used static analysis tools to check for fault-prone design patterns, but could not understand the tools' recommendations" [27]. These questions likely arise in our study, because our focus was on how participants interacted with a static analysis tool.

Strategies and Assumptions: Participants' defect resolution strategies appeared to include reading portions of the notification text. The three participants (P3, P4, and P6) who reported the lowest familiarity with vulnerabilities started 11 of 12 tasks by immediately reading the notification text. Conversely, the participants (P5 and P7) who reported the most familiarity with vulnerabilities only started by reading the notification text for 2 of 8 tasks.

The distinctions between these two workflows affected how participants built hypotheses. Low-familiarity participants built their initial hypothesis from the notification and tested that hypothesis against the code. On the other hand, high-familiarity participants used the notification to refine their already-formed hypotheses. It remains an open question whether tool notifications currently support one of these workflows more than the other. The prominence of explanatory text and reference information over resolution shortcuts might suggest FSB caters to the hypothesis-building novices. Either way, the presence of two distinct workflows suggests tools should support both.

Participants' strategies also varied in terms of how much of the notification they read. Some participants read the entirety of the message before proceeding, whereas others only read the short descriptions. Several participants read the visible parts of the long description, but neglected to scroll further to reveal the rest of the message. Often, this led to problems. For instance, P3 struggled with the Predictable Random vulnerability because he initially failed to find documentation for `SecureRandom`. P3 started the task by reading only the first half of the long description. After finally finding the documentation online and completing the task, the interviewer directed his attention to the overlooked second half of the long description. P3 read the rest of the description and realized it contained the information he sought, "I didn't scroll down that far. I see it even has a link to generating strong random number. I could have clicked on this link."

As an aside, it is possible that pointing out the FSB links to P3 at this point in the study could have influenced his subsequent two tasks. However, the FSB notifications for the two subsequent tasks did not include any links, so even if P3 had been artificially influenced by our interruption, the interruption likely had little effect on his final two tasks.

4 Discussion

In this section we discuss the implications of our work.

4.1 Flow Navigation

When iTrust performed security-sensitive operations, participants wanted to determine if data originated from a malicious source by tracing program flow. Similarly, given data from the user, participants were interested in determining how it was used in the application and whether it was sanitized before being passed to a sensitive operation. Questions related to these tasks appear in four different categories (Sections 3.3.1, 3.3.2, 3.3.3, 3.3.6). We observed participants using three strategies to answer program flow questions, strategies that were useful, yet potentially error-prone.

First, when participants asked whether data comes from the user (*a user-facing source*), and thus cannot be trusted, or if untrusted data is being used in a sensitive operation, participants would navigate through chains of method invocations. When participants navigated through chains of method invocations, they were forced to choose between different tools, where each tool had specific advantages and disadvantages. Lightweight tools, such as FIND and MARK OCCURRENCES, could be easily invoked and the output easily interpreted, but they often required multiple invocations and sometimes returned partial or irrelevant information. For example, using MARK OCCURRENCES on a method declaration highlights all invocations of the method within the containing file, but it does not indicate invocations in other files. On the other hand, heavyweight tools, such as CALL HIERARCHY and FIND REFERENCES, return method invocations made from anywhere in the source code, but were slower and clumsier for participants. Moreover, even heavyweight tools do not return all invocations when looking for tainted data sources; for instance, CALL HIERARCHY does not indicate when methods are being called from outside the system by a framework.

Second, when participants asked whether a data source was user-facing, participants would make inferences based on class names. For instance, any class that started with `Test` participants assumed was as JUnit test case, and thus was not user-facing, and therefore not a potential source of tainted data. When participants made inferences based on class names, their inferences were generally correct that the class name accurately described its role. However, this strategy fails in situations where the word "Test" is overloaded; this happens in iTrust where "Test" can also refer to a medical laboratory test.

Third, a common strategy for participants was to rely on their existing knowledge of sensitive operations and data sources in the application. When participants relied on existing knowledge of sensitive operations and data sources, such reliance may be failure-prone whenever the code has been changed without their knowledge. Indeed, prior research suggests that developers are less knowledgeable about unstable code [13]. Additionally, when a developer only contributes to a portion of the system, as is often the case in the open source community [38], he may be unable to reason about the system-wide implications of a change.

Much like work that examines more general programming tasks [32], we observed that participants would have benefited from better program flow navigation tools while investigating security vulnerabilities. Although researchers have proposed enhanced tools to visualize call graphs [33] and trace control flow to its origin [3], in a security context, these tools share the same limitations as the existing heavyweight tools. Existing tools like CodeSonar [56] and Coverity provide source-to-sink notifications for analyzing security vulnerabilities, but take control away from the programmer by forcing the developer into a tool-dictated workflow.

We envision a new tool that helps developers reason about control flow and data flow simultaneously, by combining the strengths of existing heavy and lightweight tools. We imagine such a tool could use existing heavyweight program analysis techniques, but still use a lightweight user interface. For example, such a tool might use a full-program, call hierarchy analysis technique in the back end, but use a MARK OCCURRENCES-like user interface on the front end. To indicate calls from outside the current class, additional lightweight notifications would be needed. Such a tool could support both lightweight and systematic investigation of the flow of potentially tainted data. We implemented such a prototype tool, which we call Flower, as a plugin for the Eclipse IDE [49]. Our preliminary evaluation showed that Flower helped developers navigate program flow quickly and accurately, especially over program structures with relatively few branches.

4.2 Structured Vulnerability Notifications

FSB provided explanatory notifications of potential vulnerabilities. However, to completely resolve vulnerabilities, participants performed many cognitively demanding tasks beyond simply locating the vulnerability and reading the notification, as is evidenced by the breadth of questions they asked. To resolve potential vulnerabilities, we observed participants 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 documentation and other resources; weighing existing knowledge against information in the notification; and reasoning about the feasibility of all the possible attacks. Yet, these strategies were limited in three respects.

Participants used error-prone strategies even when more reliable tools and strategies were available. For example, in Section 3.5.1, we noted that participants, unaware of the relevant hyperlinks embedded within the notification text, searched for links to external resources using web search tools. The web searches often returned irrelevant results. However, when the interviewer pointed out the embedded links after the session, participants stated that they probably should have clicked them.

Second, even after choosing an effective strategy, participants were often unaware of which tools to use to execute the strategy. For example, while assessing the Servlet Parameter vulnerability, participants wanted to determine whether certain input parameters were ever validated, but were not aware of any tools to assist in this process. Previous research suggests that both novice and experienced developers face problems of tool awareness [39].

Third, regardless of the strategies and tools participants used, they had to manually track their progress on each task. For example, the Servlet Parameter vulnerability involved twelve tainted

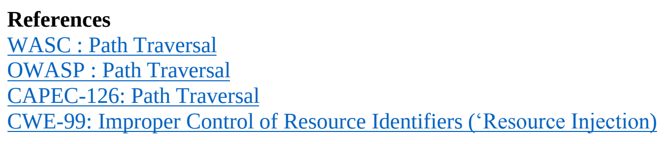

References
WASC : Path Traversal
OWASP : Path Traversal
CAPEC-126: Path Traversal
CWE-99: Improper Control of Resource Identifiers (‘Resource Injection)

Fig. 4. Reference links provided by FSB for Path Traversal vulnerability

parameters and introduced the possibility of several types of attacks. Participants had to reason about each of those attacks individually and remember which attacks they had ruled out. In a more general programming context, researchers have warned about the risks of burdening developers' memories with too many concurrent tasks — overburdening developers' attentive memories can result in *concentration failure* and *limit failure* [41].

We envision an approach that addresses these limitations by explicating developers' strategies in the form of hierarchically structured checklists. Previous research suggests that checklists can effectively guide developers [42]. We propose a structure that contains hierarchical, customizable tasks for each type of notification. For example, the structure would contain 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.” This structure would also include a checklist-like feature that allows users to save the state of their interaction with a particular notification — for example, checking off which attack vectors they have already ruled out — diminishing the risk of *concentration failure* and *limit failure*. Additionally, each task could include links to resources that relate specifically to that task and tool suggestions that could help developers complete the task. We present more details about our vision for such a tool, including a mockup, in a related publication [48].

4.3 Context-Informed Web Search

Accessing web resources was an important element of participants' strategies across several categories (Section 3.2.1, Section 3.2.2, Section 3.4.2, Section 3.5.1). To find success, participants sought information that was scattered around different sites. Depending on the task and an individual's approach, participants found critical clues hidden on StackOverflow, official documentation pages, and personal blogs, for instance.

To reach these helpful web resources, participants either followed the links that FSB had curated, or used an online search engine. Both of these approaches are failure-prone. However, by drawing inspiration from the positive aspects of each of these approaches, we will propose a new tool-assisted approach for gathering resources from the web while diagnosing a static analysis defects. We will refer to this approach as *context-informed web search*.

FSB includes links to web resources in its description of some defects under the header “References” (Figure 4). FSB's designers assumedly deem these references relevant and helpful, and, when references to the right types of information were available, participants tended to use them. Unfortunately, the links in each references section cover a limited number of topics. As in the example depicted by Figure 4, the reference material might describe the defect and how to exploit it, but not provide code examples for remedies. In practice, participants sought information and visited sites beyond what was available in the finite reference sections. To summarize, the references encapsulate the tool *designer's* expertise and knowledge of a specific defect pattern, but are not tailored to the user's needs.

Participants also used search engines to locate web resources, sometimes as a primary means, other times after having exhausted the FSB links. With this approach, participants had to sift through long lists of irrelevant results to find useful information. As described in Section 3.4.2, some participants iteratively refined their search terms to filter out as many irrelevant results as possible. Others simply tested link after link until they found something useful or gave up.

The two approaches for accessing web resources described so far (using tool links and performing a search) fail in complementary ways. The tool's list of links only cover a narrow set of topics, with each link highly relevant to the defect reported. Conversely, search engines over-approximate the set of relevant resources, but cover far more topics of interest.

We envision a new way for developers to access web resources while resolving defects with static analysis, *context-informed web search*. This approach improves on existing approaches by combining the breadth of web search with the relevance afforded by a tool's contextual awareness. Users would perform a *context-informed web search* just as they would any other web search, but enter their query into a specialized search engine, rather than a general-purpose search engine. The specialized search engine would work in tandem with the developers' static analysis tool and would start by fetching results just like a general-purpose search engine. Before returning those results to the user, the specialized search engine would query the static analysis tool for details about the developers context — information the tool has already computed in order to detect defects. Finally, the specialized search engine would only display context-relevant results.

Consider, for example, a developer who searches for 'secure random number.' A general-purpose search engine would return results targeted at Ruby, Javascript, Android, and Java developers. The specialized search engine could conclude from the static analysis tool that the developer was using Java and importing java.util.Random and filter out irrelevant results pertaining to other languages.

This information searching approach draws inspiration from several prior efforts. The Mica tool [51] analyzes search results and presents users with information about which results might be most relevant, compared with our approach, which proposes taking information from the source code and a static analysis tool. Rahman and colleagues [43] and Goldman and Miller [15] similarly describe adapted search engines, theirs based in the IDE. These previous works create search queries using information from the source code and from stack traces. Our proposed approach differs in that we also consider contextual information from a static analysis tool.

5 Related Work

We have organized the related work into three subsections. Section 5.1 outlines some of the current approaches researchers use to evaluate security tools, Section 5.2 references other studies that have explored developers' information needs, and Section 5.3 relates our work to the other work on strategies.

5.1 Evaluating Security Tools

Using a variety of metrics, many studies have assessed the effectiveness of the security tools developers use to find and remove vulnerabilities from their code [2], [35], [36].

Much research has evaluated the effectiveness of tools based on their false positive rates and how many vulnerabilities they detect [2], [10], [25]. For instance, Jovanovic and colleagues evaluate their tool, Pixy, a static analysis tool that detects cross-site scripting vulnerabilities in PHP web applications [25]. They considered Pixy 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 security-oriented static analysis, which creates static analyzers based on inputs from the user [35]. 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 manual penetration testing, exploratory manual penetration testing, static analysis, and automated penetration testing [2]. 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 vulnerability-finding techniques [10]. 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, these studies do not evaluate whether the tools address developers' information needs. Unlike existing studies, our study examines a security tool through the lens of developers' information needs and, accordingly, provides a novel framework for evaluating security tools.

5.2 Information Needs Studies

Several studies have explored developers' information needs outside the context of security. In contrast to previous studies, our study focuses specifically on the information needs of developers while performing a more specific task, assessing security vulnerabilities. Though some prior studies identify needs pertaining to debugging and defect fixing, none specifically study the process of diagnosing security defects. Unsurprisingly, some of the information needs previously identified as general programming needs and general debugging needs also occur while developers assess security vulnerabilities (e.g., Sections 3.3.2 and 3.3.3). Here we will summarize these prior works and compare to their methodologies chosen and task domains studied. In the results section (Section 3) we highlight noteworthy similarities and differences in terms of our findings.

Sillito and colleagues studied the questions asked during change tasks [47], creating a catalog of 44 questions. They conducted two observational studies with developers who were either familiar or unfamiliar with the code they were contributing to. Unlike our study, the tasks in these studies were primarily focused on changes that added features to the code base, rather than fixing some type of defect.

Fritz and Murphy conducted a series of 11 open interviews and identified 78 questions developers ask [12]. Rather than study questions that focus on the source code, their study identifies questions that require developers to integrate multiple types of information. As a result, most questions identified in this study pertain to the social and collaborative aspects of developing software, as opposed to more granular questions about code. Based on the questions they identified, Fritz and Murphy developed a model and prototype tool to assist developers with answering their questions. Our study complements this work because we identify developers' questions while observing them interacting with a code base and a static analysis tool.

Similar to Fritz and Murphy, Haenni and colleagues also investigate developers' social and collaborative information needs by surveying framework and library developers in software ecosystems [20]. They organize their findings into two main categories, upstream and downstream information needs, identifying 6 and 8 needs in each category, respectively.

In a similar approach to our study, Ko and colleagues identified 21 information needs by observing developers — however, in their study participants were members of collocated teams [27]. 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 [32]. 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 [31]. These studies give us insight into the needs of developers during general programming tasks. Our work examines information needs during a more specific programming task, diagnosing security vulnerabilities with static analysis. We identified information needs unique to security, such as needs pertaining to attacks and vulnerabilities, as well as previously known information needs with novel security implications. Throughout Section 3 we have discussed how the information needs identified in previous studies overlap with the information needs we identified.

5.3 Defect Resolution Approaches

Prior work has noted the importance of strategies. In their inspirational work, Bhavnani and John observed that knowledge of tasks and tools for complex computer applications such as CAD were insufficient for developers to be efficient; they found that users needed to learn good strategies [5]. Additionally, they argue that the identification of efficient strategies should be a key research goal. Our work furthers progress toward that goal by identifying successful strategies for resolving security vulnerabilities.

Similarly, Katz and Anderson noted the importance of good strategies. They found that a developer's choice of bug-location strategy affects his or her ability to assess the correctness of a line of code [26]. Our work extends that of Katz and Anderson by focusing on the defect resolution process after a potential bug has been located.

Several studies have explored the effects of strategies and assumptions in end-user programming environments. Grigoreanu and colleagues examined gender differences in debugging strategies for spreadsheet end-user programmers [16], [17]. Through a think-aloud study, they found that women and men employed different strategies. For example, women tended to use comprehensive information processing styles, where they overview the task before making changes; in contrast, men tended to selectively process information. Gross and colleagues conducted an exploratory study in which end-user programmers were asked to find and modify code for specific functionality with unfamiliar programs, which they classified into models [18]. Ko and colleagues categorized six types of learning barriers faced by end-user programmers [28]. They discussed how end-user programmers can overcome learning barriers by making simplifying assumptions, but incorrect assumptions often lead to errors. We are similarly interested in strategies and assumptions. In contrast to the prior work, we focus on the domain of security vulnerabilities rather than end-user programming environments.

Researchers have also studied strategies in the context of program comprehension. Program comprehension refers to the process by which developers make sense of unfamiliar code. An overly simplified summarization of program comprehension strategies broadly organizes models into two categories: bottom-up and top-down. In strategies viewed from a bottom-up [46] perspective, developers combine small pieces of information to form larger "chunks" until the entire program is understood. Alternatively, in the top-down model [6], developers maintain a mental model of the system based on their domain knowledge and update that mental model as they discover new evidence. Mayrhauser and Vans provide more thorough descriptions of six different models of program comprehensions strategies [53]. Our findings suggest that program comprehension plays an important role in defect resolution, because developers' defect resolution strategies, especially those we describe in Sections 3.3.4 and 3.3.5, are partly about about trying to understand code. Indeed, to resolve some defects, developers must explore both familiar and unfamiliar code. However, other strategies we identified, such as those pertaining to modifying code and interacting with tools, seem beyond the scope of program comprehension.

6 Threats to Validity

We faced the internal threat of recording questions that participants asked because they lacked a basic familiarity with the study environment. We took two steps to mitigate this threat. First, we required participants to have experience working on iTrust. Second, at the beginning of each session, we briefed participants on FSB and the development environment. During these briefing sessions, we gave participants the opportunity to ask questions about the environment and study setup, though we cannot say for certain that participants asked all the questions they had at that time. Thus, some of the questions we identified may reflect participants' initial unfamiliarity with the study environment and FSB. Since we are interested broadly in developers' information needs, the initial questions they ask about a new tool and environment still are an important subset to capture.

Our mitigation strategy of recruiting participants who were familiar with the iTrust code base introduced its own threat. This design decision limited our population to students and developers who studied at North Carolina State University at some point in time. The participants we studied limit generalization and may not represent the range of developers who would use security tools. For instance, we likely cannot completely generalize these results to security experts — none of our participants self-identified as such.

The fact that this study was conducted in a controlled environment rather than an industrial development setting raises a threat to the external validity of our results. Though we cannot and do not claim that we have identified a comprehensive categorization of all security-related questions all developers might ask, we have made several efforts to mitigate this threat. First, we included both students and professionals in our sample, because questions might differ based on experience. Further, participants were equipped with FSB, a representative open source static analysis tool with respect to its user interface. Finally, we chose iTrust, an industrial-scale open-source project as our subject application.

Another reason we cannot claim our categorization is comprehensive is because the questions and strategies may have been dependent on the four FSB notifications we chose and the order they were presented in. We did not study all types of security defects. In fact, there are many defects that are not even detected by FSB. Although we chose the four notifications to span different topics as categorized by FSB, there may be information needs and strategies these topics did not expose.

As we discussed in Section 2.5, participants likely made many assumptions without explicitly stating them. Participants might have also felt obliged to actively pursue strategies while being observed in a lab setting. As a result, we likely identified fewer assumptions than participants actually made. Therefore, relative to the results for RQ2 where strategies were visible, there are fewer assumptions to discuss. To mitigate this threat, we used stemming and lemmatisation to capture as many explicitly stated assumptions as possible, specifically those where participants did not just use the word "assume." Despite the scarcity of assumptions in our results, we believe they are worthwhile to include because they give us insight into how developers sometimes choose to satisfy information needs without actively executing a strategy.

Participants also may have spent an unrealistic amount of time (either too much or too little) on each task due to working outside their normal environment. To counteract this threat, we did not restrict the amount of time alloted for each task. Further, whenever a participant asked the interviewer what to do next, the interviewer provided minimal guidance, typically prompting the participant to proceed as (s)he would in her normal work environment.

An additional threat is that the think aloud protocol may have influenced participants actions. Consequently, participants may have had different information needs or strategies than they would in the wild. For example, developers may have reflected more on the vulnerabilities than they would have in their normal working environments, causing them to approach them more carefully. This is perhaps evidenced by the existence of the Developer Planning and Self-Reflection category (Section 3.4.1).

7 Conclusion

This paper reported on a study that explored how developers resolve security defects while using static analysis. During our study, we asked ten software developers to describe their thoughts as they assessed potential security vulnerabilities in iTrust, a security-critical web application. We presented the results of our study as a categorization of questions and a catalog of strategies for answering those questions. This work advocates for tools that not only detect vulnerabilities, but also help developers actually resolve those vulnerabilities. Our findings have several implications for the design of static analysis tools. Most broadly, tools should support effective strategies and provide information that aligns with the information needs we have identified. In particular, our results suggest that tools should help developers, among other things, search for relevant web resources.

Acknowledgments

We would like to thank our study participants. Special thanks to Xi Ge, Anthony Elliott, Emma Laperruque, and the Developer Liberation Front[3] for their assistance. This material is based upon work supported by the National Science Foundation under grant numbers 1318323 and DGE–0946818.

References

[1] N. Ammar and M. Abi-Antoun. Empirical evaluation of diagrams of the run-time structure for coding tasks. In *Reverse Engineering (WCRE), 2012 19th Working Conference on*, pages 367–376. IEEE, 2012.

[2] 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.

[3] 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.

[4] S. K. Bhavnani and B. E. John. From sufficient to efficient usage: An analysis of strategic knowledge. In *Proceedings of the ACM SIGCHI Conference on Human factors in computing systems*, pages 91–98. ACM, 1997.

[5] S. K. Bhavnani and B. E. John. The strategic use of complex computer systems. *Human-Computer Interaction*, 15(2):107–137, Sept. 2000.

[6] R. Brooks. Towards a theory of the cognitive processes in computer programming. *International Journal of Man-Machine Studies*, 9(6):737–751, 1977.

[7] 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.

[8] M. Christakis and C. Bird. What developers want and need from program analysis: An empirical study. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 332–343. ACM, 2016.

[9] D. Cornell. Remediation statistics: what does fixing application vulnerabilities cost. *Proceedings of the RSAConference, San Fransisco, CA, USA*, 2012.

[10] 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.

[11] N. Fairclough. *Analysing discourse: Textual analysis for social research*. Psychology Press, 2003.

[12] 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.

[13] 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.

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

[15] M. Goldman and R. C. Miller. Codetrail: Connecting source code and web resources. *Journal of Visual Languages & Computing*, 20(4):223–235, 2009.

[16] V. Grigoreanu, J. Brundage, E. Bahna, M. Burnett, P. Elrif, and J. Snover. Males' and females' script debugging strategies. In *Proceedings of the 2Nd International Symposium on End-User Development*, IS-EUD '09, pages 205–224, Berlin, Heidelberg, 2009. Springer-Verlag.

[17] V. Grigoreanu, M. Burnett, S. Wiedenbeck, J. Cao, K. Rector, and I. Kwan. End-user debugging strategies: A sensemaking perspective. *ACM Trans. Comput.-Hum. Interact.*, 19(1):5:1–5:28, May 2012.

[18] P. Gross and C. Kelleher. Non-programmers identifying functionality in unfamiliar code: Strategies and barriers. *J. Vis. Lang. Comput.*, 21(5):263–276, Dec. 2010.

[19] G. Guest, A. Bunce, and L. Johnson. How many interviews are enough? an experiment with data saturation and variability. *Field methods*, 18(1):59–82, 2006.

[20] N. Haenni, M. Lungu, N. Schwarz, and O. Nierstrasz. Categorizing developer information needs in software ecosystems. In *Proceedings of the 2013 International Workshop on Ecosystem Architectures*, pages 1–5. ACM, 2013.

3. research.csc.ncsu.edu/dlf/

[21] M. Howard, J. Pincus, and J. M. Wing. Measuring relative attack surfaces. In *Computer security in the 21st century*, pages 109–137. Springer, 2005.

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

[23] B. Johnson, R. Pandita, J. Smith, D. Ford, S. Elder, E. Murphy-Hill, S. Heckman, and C. Sadowski. A cross-tool communication study on program analysis tool notifications. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 73–84. ACM, 2016.

[24] 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.

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

[26] I. R. Katz and J. R. Anderson. Debugging: An analysis of bug-location strategies. *Hum.-Comput. Interact.*, 3(4):351–399, Dec. 1987.

[27] A. J. Ko, R. DeLine, and G. Venolia. Information needs in collocated software development teams. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 344–353, Washington, DC, USA, 2007. IEEE Computer Society.

[28] A. J. Ko, B. A. Myers, and H. H. Aung. Six learning barriers in end-user programming systems. In *Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing*, VLHCC '04, pages 199–206, Washington, DC, USA, 2004. IEEE Computer Society.

[29] 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.

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

[31] 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.

[32] 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.

[33] 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.

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

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

[36] 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.

[37] S. Mauw and M. Oostdijk. Foundations of attack trees. In D. Won and S. Kim, editors, *Information Security and Cryptology - ICISC 2005*, volume 3935 of *Lecture Notes in Computer Science*, pages 186–198. Springer Berlin Heidelberg, 2006.

[38] A. Mockus, R. T. Fielding, and J. D. Herbsleb. Two case studies of open source software development: Apache and mozilla. *ACM Trans. Softw. Eng. Methodol.*, 11(3):309–346, July 2002.

[39] 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.

[40] 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.

[41] 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.

[42] 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*, 2009.

[43] M. M. Rahman, S. Yeasmin, and C. K. Roy. Towards a context-aware ide-based meta search engine for recommendation about programming errors and exceptions. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*, pages 194–203. IEEE, 2014.

[44] M. P. Robillard, W. Coelho, and G. C. Murphy. How effective developers investigate source code: An exploratory study. *IEEE Transactions on software engineering*, 30(12):889–903, 2004.

[45] 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.

[46] B. Shneiderman and R. Mayer. Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Computer & Information Sciences*, 8(3):219–238, 1979.

[47] J. Sillito, G. C. Murphy, and K. De Volder. Asking and answering questions during a programming change task. *IEEE Transactions on Software Engineering*, 34(4):434–451, 2008.

[48] J. Smith. Identifying successful strategies for resolving static analysis notifications. In *Proceedings of the 38th International Conference on Software Engineering Companion*, pages 662–664. ACM, 2016.

[49] J. Smith, C. Brown, and E. Murphy-Hill. Flower: Navigating program flow in the ide. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'17)*, Oct. 2017.

[50] J. Smith, B. Johnson, E. Murphy-Hill, B. Chu, and H. R. Lipford. Questions developers ask while diagnosing potential security vulnerabilities with static analysis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 248–259, New York, NY, USA, 2015. ACM.

[51] J. Stylos and B. A. Myers. Mica: A web-search tool for finding api components and examples. In *Visual Languages and Human-Centric Computing, 2006. VL/HCC 2006. IEEE Symposium on*, pages 195–202. IEEE, 2006.

[52] C. Theisen, K. Herzig, P. Morrison, B. Murphy, and L. Williams. Approximating attack surfaces with stack traces. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*, pages 199–208. IEEE Press, 2015.

[53] A. Von Mayrhauser and A. M. Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, 1995.

[54] 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.

[55] Nist source code security analyzers. http://samate.nist.gov/index.php/Source_Code_Security_Analyzers.html.

[56] Codesonar. http://grammatech.com/codesonar.

[57] Coverity. http://coverity.com/.

[58] Security questions experimental materials. https://figshare.com/projects/How_Developers_Diagnose_Potential_Security_Vulnerabilities_with_Static_Analysis/24439.

[59] Find security bugs. http://h3xstream.github.io/find-sec-bugs/.

[60] Hippa statute. http://hhs.gov/ocr/privacy/.

[61] itrust software system. http://agile.csc.ncsu.edu/iTrust/wiki/doku.php?id=start.

[62] Otranscribe. http://otranscribe.com.

[63] Owasp source code analysis tools. http://owasp.org/index.php/Source_Code_Analysis_Tools.

[64] Owasp. http://owasp.org/index.php/Main_Page.

[65] Web application security consortium static code analysis tools. http://projects.webappsec.org/w/page/61622133/StaticCodeAnalysisList.

Justin Smith Justin is a PhD student at North Carolina State University. His research interests include human-computer interaction, software engineering, and security tools. Contact him at jssmit11@ncsu.edu; http://www4.ncsu.edu/~jssmit11

Brittany Johnson Brittany is a postdoctoral researcher at University of Massachusetts, Amherst. Her research interests include human-computer interaction, software tools and processes, and machine learning. She holds a Ph.D. in Computer Science from North Carolina State University. Contact her at bjohnson@cs.umass.edu; https://people.umass.edu/bijohnson

Emerson Murphy-Hill Emerson is an associate professor at North Carolina State University. His research interests include human-computer interaction and software tools. He holds a Ph.D. in Computer Science from Portland State University. Contact him at emerson@csc.ncsu.edu; https://people.engr.ncsu.edu/ermurph3/

Bill Chu Bill Chu is Professor of Software and Information Systems at University of North Carolina at Charlotte. His research interest is in software security. He received Ph.D. in Computer Science from the University of Maryland at College Park. Contact him at billchu@uncc.edu

Heather Richter Lipford Heather Richter Lipford is a Professor at the University of North Carolina at Charlotte. Her research interests are in usable security and privacy. She holds a Ph.D. from the Georgia Institute of Technology. Contact her at Heather.Lipford@uncc.edu; https://webpages.uncc.edu/richter/