SECURITY CODE REVIEW WITH STATIC ANALYSIS TECHNIQUES FOR THE DETECTION AND REMEDIATION OF SECURITY VULNERABILITIES

by

Tyler William Thomas

A dissertation submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in
Computing and Information Systems

Charlotte

2018

Approved by:

Dr. Heather Lipford

Dr. Bill Chu

Dr. Weichao Wang

Dr. Mohamed Shehab

Dr. Shenen Chen

ABSTRACT

TYLER WILLIAM THOMAS. Security Code Review with Static Analysis Techniques for the Detection and Remediation of Security Vulnerabilities. (Under the direction of DR. HEATHER LIPFORD)

Security problems are both a large and growing concern today. Many security breaches are the result of security vulnerabilities introduced during the code construction phase. These vulnerabilities sometimes occur due to poor security training of the developer, and sometimes they are simply created by accident. Static analysis, examination of the application source code with a specialty tool, is the current solution to this problem. Unfortunately, this process produces an extremely large amount of false positives. It also cannot detect application specific issues without custom rules for each application. Consequently, these tools are often used only by security experts or abandoned entirely. In this dissertation, I conduct an interview study of application security experts to gain an understanding of their workflows and the organizational, technical, and communication challenges they face today. From these findings, I introduce tool assisted security code review fed by interactive static analysis and interactive annotation as a solution to detect and remediate greater numbers of vulnerabilities. In this dissertation, I also explore the process, warnings, and collaboration between the various roles of users for this type of tool. Lastly, I provide a set of design guidelines for security code review tools.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1: INTRODUCTION

1.1 Importance of Security

Security is a very important concern today. According to a recent IBM study, the average cost to an organization for a stolen record increased by 9% to $145 in 2014 [31]. Since millions of credit card records are stolen every year, the cost can easily run into billions of dollars. Techniques to prevent security breaches generally fall into two broad categories: network security issues and application security issues. My research focuses on Application Security, defined as "The use of software, hardware, and procedural methods to protect applications from external threats [55]." This means that application security problems generally involve problems or bugs in applications themselves. These "bugs" are collectively known as security vulnerabilities, or simply vulnerabilities. These vulnerabilities can be detected and mitigated throughout the application's life cycle. Secure programming, or defensive programming, is defined as "A form of defensive design intended to ensure the continuing function of a piece of software in spite of unforeseeable usage of said software [13]." One of the main goals of secure programming is to avoid the creation of these vulnerabilities. One contribution of my dissertation is more deeply examining this process conducted in organizations.

A large number of security issues are caused by these vulnerabilities. According to

the United States Department of Homeland Security, 6,449 new software vulnerabilities were *reported* in 2016 alone [14]. Since many vulnerabilities are unknown and many more may be unreported, the number of actual vulnerabilities in the software we use is far higher than this. Security vulnerabilities can be exploited through a variety of attacks, such as cross site scripting and SQL injection, to steal private information, harming both individuals and organizations. Thus, to the extent possible, security vulnerabilities should be found and remediated in application source code.

1.2 Security Vulnerabilities

Many types of vulnerabilities exist. These include SQL Injection, Cross Site Scripting, broken access control, broken authentication, security misconfiguration, insecure deserialization, cross site request forgery, and forced browsing.

Perhaps the most common and arguably most severe of vulnerabilities is SQL Injection. Many applications interact with a database to store information. These applications may pass various SQL queries to database management systems that manage these databases. Many of these applications are connected to the web, meaning they are exposed to hackers. Often, these applications take input from the user (potential hacker) and use this to form SQL queries.

For example, suppose that a banking application exists which allows a user to login. The username and hashed password are stored in a database. The application receives the username and password from the user and makes the following SQL query

Listing 1: Normal SQL Example

```
Select firstName, lastName, balance, accountStatus from Users
```

```
where username = USERNAME and password = HASHEDPASSWORD
```

In this case, let us suppose that USERNAME and HASHEDPASSWORD are variables which hold the values that a user has submitted. If a normal user tries to login, the SQL would look similar to this:

Listing 2: Normal SQL Example With Password

```
Select firstName, lastName, balance, accountStatus from Users
where username = tthoma81 and password =
m83sned3i2k4j383kdjngb3943
```

Normally, if the username and password were correct, the user account would be found and the user would be logged in. If not, the user account would not be found and the user would not be logged in. However, since this input is coming from a possible attacker, the attacker can submit the following string as the password.

Listing 3: SQL Injection String

```
' or '1'='1;
```

This effectively changes the sql query to the following

Listing 4: SQL Injected Query

```
Select firstName, lastName, balance, accountStatus from Users
where username = tthoma81 and password = '' or '1' = '1';
```

Since 1 will always be equal to 1, the website will login the attacker without a valid password. It is also possible for the attacker to chain together many, many combi-

nations of sql queries which may be able to extract most or all of the information in the database. Additionally, the attacker may be able to insert fraudulent information into the database and destroy it entirely.

It is also worth pointing out that a similar category of vulnerability exists called 'Command Line Injection' The vulnerability is the same as SQL Injection. However, in the case of command line injection attacks, the tainted input is passed as a command to the operating system instead of a SQL query to a database management system. This may result in attackers gaining administrator level, remote access to a machine.

1.2.1 Remediation Practices

Organizations often take steps to prevent or reduce the prevalence of these application security vulnerabilities. Most of these techniques and practices are referred to as secure programming practices. This is typically done by implementing security training programs designed to teach developers how to write code securely. Organizations of sufficient size may also hire security experts who examine source code for security vulnerabilities, conduct penetration tests, and configure various types of scanners to catch and remediate these vulnerabilities.

Resolving application security vulnerabilities later in the development process is very time consuming and expensive. Therefore, it is favorable to detect and resolve security vulnerabilities as soon as possible during the development process. However, most tools and techniques that deal with these issues do so late in the development process. The most common technique for dealing with application security issues is

static analysis of application source code, otherwise known simply as static analysis. Many tools have been designed to perform static analysis, with HP Fortify being perhaps the most commonly used [6, 4, 2, 9]. Code review is another technique which is used to detect security vulnerabilities [47]. Code review relies on the inspection of code by humans to detect issues. Many tools have also been designed to assist in this process, though the process rarely focuses on detecting security issues exclusively [5, 11, 8, 10, 1, 12, 7, 3]. Static analysis works by examining source code, tracing a path from where data enters a program to where it ends up. Both static analysis and code review will be discussed in detail in this dissertation, as my work examines security code review and interactive static analysis.

1.3 Static Analysis

Static analysis tools examine source code statically (when not running), and attempt to detect bugs in the source code. Often they are used to identify security vulnerabilities specifically. They do this by examining where potentially tainted data enters a program and tracing the data through the program. Unfortunately, however, static analysis tools can generate very large amounts of false positives [24, 76]. In order for static analysis tools to be effective, complex custom rules must be written for the tool. This is often done by a security expert for every application the tool runs on. To make matters worse, communicating information about complex vulnerabilities to application developers presents a unique challenge in and of itself. If the developer does not understand why a certain line is flagged as potentially vulnerable and is not provided with detailed information, it will be far more difficult for him

or her to resolve the problem. Consequently, static analysis tools are typically used by security experts after an application has been written, rather than by developers while they are writing the code [24].

Sadowski examined why developers, specifically Google developers, did not actually use static analysis tools. She found that developers did not like to use them for a variety of reasons, primarily false positives and scalability concerns. This led her to design Tricorder which has the ability to utilize developer-written static analyzers with an emphasis on keeping false positives to a minimum. Other studies have shown that developers are willing to tolerate many more false positives when specifically addressing security concerns [61]. This suggests that a delicate balance of not annoying the developer, while at the same time actually detecting many security concerns, is required. The only problem in this approach is that it may prove difficult to keep false positives low, while keeping detection rates high. However, as with Tricorder, it may be possible to leverage developer written analysers to minimize the false positive rate, or at least make false positives more tolerable.

In an attempt to circumvent the shortcomings of static analysis, I developed a new process called interactive static analysis. This process performs static analysis as the developer writes code [71]. This involves the developer, trains the developer, and helps combat the tediousness of static analysis. My studies have demonstrated that interactive static analysis can be performed by developers who hvae little knowledge of security [71, 72, 61]. However, as results in this dissertation will demonstrate, developers still require human assistance and judgement, which are not provided by static analysis tools.

1.4 Thesis Statement and Contributions

The existing solutions for dealing with security vulnerabilities are inadequate. Static analysis is effective but carries a high cost due to the large amount of false positives. It also leaves out developers and cannot effectively detect application specific issues without many custom rules.[24, 76, 61]

To deal with the issue of application specific vulnerabilities, I created a process called interactive annotation [61]. In this process, the application flags sensitive operations and requests that the developer graphically annotate the security logic which ensures it is not improperly accessed. The developer can then be notified if a vulnerability is detected.[61]

Despite these advances, these processes still rely on one critical component, the individual developer. They are not always accurate with their mitigations and annotations [72, 61]. I also learned from my previous studies that developers often have low confidence in their solutions to interactive static analysis and interactive annotation and desire someone to check them [61]. Code review has the potential to solve these issues. However, in practice, it often does not focus on security issues. Therefore, although it is effective for finding general defects in code, it is insufficient for detecting and mitigating vulnerabilities in code [47, 46, 21, 37, 23, 29].

A new approach is needed. In this dissertation, I investigate the use of security code review fed by interactive static analysis and interactive annotation. I study the process and design resulting from the combination of these methods. I examine the collaboration, communication feedback, and roles of participants engaged in the

process.

Additionally, this work produces the following contributions:

- Evaluation of the current workflows, challenges, and needs of application security experts;
- Identification of a vulnerability creation and remediation cycle;
- Examination of how developers identify, understand, and act on security logic within code;
- A security code review tool design and prototype implementation;
- A set of design guidelines for security code review tools;
- Analysis of the perceptions of people in different roles of performing security code review;
- Examination of how code reviewers interpret and respond to different kinds of security warning and vulnerability information.

In this dissertation, I will first discuss related work in Chapter 2. I will then present a study of application security experts in Chapter 3. Next, in Chapter 4, I will describe my evaluation of developer-provided security inputs, or annotations. In Chapter 5, I describe the novel idea of security code review, along with the design of my prototype tool. I then present an evaluation of the tool with security experts in Chapter 6 and developers in Chapter 7. Finally, I summarize my results and contributions in Chapter 8.

CHAPTER 2: BACKGROUND

In order to understand how to effectively detect and remediate security vulnerabilities efficiently in application source code, it is important to understand what work has already been done to address this problem. In this chapter, I discuss the existing body of work surrounding this issue, focusing specifically on research surrounding static analysis and code review as potential solutions to the problem.

2.1 Application Security

Most work which has already been done on the detection and remediation of security vulnerabilities lies within a field known as application security. This field encompasses all techniques, processes, and procedures used to detect and remediate security vulnerabilities found within applications themselves [16]. It is distinct from network security since traditional security hardware, software, and techniques are often ineffective when the security vulnerability is located within the application source code [16, 48].

Although every organization approaches application security differently, a nonprofit professional organization known as OWASP is often considered the leading authority in the field [17]. OWASP maintains two conferences which attract security professionals from various fields and maintain a list known as "The Top Ten" which highlights the top ten most serious and widespread of application security issues within a given

year [17].

However, implementation of OWASP best practices is often difficult. Evaluations suggest that 75% of software vendors fail to comply with the OWASP top ten during initial assessments [16]. Additionally, 30% of companies never scan for vulnerabilities during code development [16]. This occurs despite that fact that the U.S. Department of Homeland Security estimates that 90% of security incidents result from exploits against defects in software [48].

2.2 Static Analysis

Much research has been conducted to determine the effectiveness of various tools which aim to detect and remove vulnerabilities. Static analysis tools are the most commonly used technique, comprising roughly half of the entire market share for security tools [30]. Static analysis techniques detect vulnerabilities in non-running or "static" source code by looking for predefined patterns based on heuristics. To do this, static analysis tools look for a source where potentially malicious data may enter a program. They refer to this entry point as a taint source. They then trace the taint through the program. Whenever the taint ends up in a dangerous area, it is said to be in a sink. Through analyzing taint sources and sinks, static analysis tools can detect security vulnerabilities and advise the developer or security expert. However, due to high false positive rates, static analysis tools are seldom used by regular developers and are instead used by security experts [24].

The traditional solution to the high false positive rates of these tools has been to use custom rules for each application on which static analysis is used. By writing static

analysis rules, the analyser can be trained to reduce the amount of false positives through understanding of the architecture and data flow of the application. For this same reason, the rules can also help static analysis tools to detect application specific issues.

Unfortunately, these rules are extremely complex and difficult to write [24, 76]. This means they must be written by a security expert specially trained in writing static analysis rules. To make matters worse, the rules need to be written for each application on which static analysis is used. Therefore, they cannot be created once by the developer of the tool. Instead, they must be created by each organization using the tool. Complicating things further, the security expert who possesses this specialty knowledge must also possess detailed knowledge of each application the tool will be run on. Otherwise, the rules will be ineffective. As can be imagined, much training and collaboration is required to perform this task. This translates into extremely high costs for organizations wishing to use static analysis. Without these custom rules, static analysis exhibits the initially discussed problems of high false positive rates and the inability to detect application specific issues.

Attempts have been made to improve the false positive rate and developer usage of these tools without custom rules. Livshits and Lam created static analyzers based on input from users. They also found their tool to be effective, which they attributed to the low false positive rate [45]. Jovanovic evaluated a more specialized static analysis tool called Pixy. Pixy was designed purely to tackle cross site scripting vulnerabilities in one language, PHP. Jonavovic ultimately concluded that it was an effective tool because it had a very low false positive rate and was also effective in detecting zero

day vulnerabilities [43].

Sadowski examined why developers, specifically Google developers, did not actually use static analysis tools. She found that developers did not like to use them for a variety of reasons, primarily false positives and scalability concerns. This led her to design Tricorder with the ability to utilize developer written static analyzers with an emphasis on keeping false positives to a minimum. Sadowski learned that developers did use static analysis tools when these developer written static analyzers were utilized. However, the false positive rate was required to be extremely low, well under 10%.

Sadowski also found that Google had experimented with a variety of static analysis tools in the past, such as Findbugs, Klocwork, and Fault Prediction. However, they very quickly abandoned their use [57]. Sadowski suspected that this was due to a variety of reasons such as failure to integrate properly into the workflow, scaling issues, and false positives. They had previously found that when developers had to use a command line interface or had to run static analysis as a distinct step, usage rates dropped off. When the tools were actually used, they had very high false positive rates. Of the static analysis tools used in Google, Findbugs was used the most, but even then it was only used by 35 developers in 2014, and only 15 of them bothered to use it more than one time [57].

Other studies have shown that developers are willing to tolerate many more false positives when specifically addressing security concerns [61]. This suggests that a delicate balance of not annoying the developer, while at the same time actually detecting all possible security concerns may be desirable. The only problem in this approach

is that it may prove difficult to keep false positives low, while keeping detection rates high. However, it may be possible to leverage developer written analyzers, like Tricorder, to minimize the false positive rate, or at least make false positives more tolerable.

2.2.1 Just In Time Static Analysis

In an effort to provide tools for use by developers during implementation, interactive, Live, or Real-Time Static Analysis has also been explored for code defects in general. Do et al. present a concept that they label as JIT or "Just In Time" static analysis [34]. Like interactive static analysis, introduced in chapter 1, this approach presents warnings in real time, as the developer codes. However, this technique attempts to compute the most relevant results quickly, with less relevant results provided much later. Do et al, also present techniques and methods for transforming a traditional static analysis scan into a JIT static analysis process through a concept known as layered execution [34].

However, one concern of this approach is the algorithm may outpace the developer or fall behind due to a misinterpretation of the code development context. According to Do. et al, this may present a "moving target" of warnings for the developer and become annoying [34]. To evaluate JIT static analysis, Do et al. created a prototype taint analysis tool called Cheetah for Android applications. They then evaluated this prototype on a variety of open source real-world applications and found that it "returned warnings quickly enough to avoid disrupting the normal workflow of developers. [34]" They then followed this study with a within-subjects user study in which

eighteen participants performed development tasks. They found that developers fixed data leaks twice as fast when using Cheetah than when using a standard batch style static analysis tool [34]. This result is particularly relevant when considering code review solutions fed by interactive static analysis because it suggests that live, or interactive, static analysis tools may have much less of an impact on the developer's workload than traditional static analysis tools.

2.2.2 Interactive Static Analysis

I have previously explored the use of *interactive* static analysis to bypass the rule writing requirement of static analysis and combat the tediousness of conducting static analysis. This was done by making the process interactive (it works as the developer types) and contained within the IDE. I designed an Eclipse plugin called Application Security in the IDE (ASIDE) to carry out this task. When ASIDE is run, it generates warning icons in the left margin of the code editing window to the left of vulnerable lines as soon as they are written. Once the icon is clicked, it presents the developer with a list of options for auto resolving the problem and provides additional information about the problem. However, this prototype is extremely limited in its ability to detect complex and application specific vulnerabilities and contains no means of verifying that solutions are actually correct. Lastly, it does not support integration with code reviews and provides no means of collaboration [71].

Xie et al. evaluated ASIDE in two comparison user studies [71]. The results of the study showed that the interactive support provided by ASIDE to developers can effectively reduce secure programming errors introduced by developers. Additionally,

Xie et al. suggested guidelines for interactive static analysis tools [71].

One major challenge of any potential security tool is being able to cater to developers who may have limited security knowledge or training. Nguyen et al. noted that Android developers frequently fail to adhere to security best practices, despite the large quantity of security resources that are available [52]. To help bridge this gap, they developed an IDE plugin called FixDroid for both hobby and professional developers. They hoped that this tool could teach developers about secure coding practices as they code without any impact to their work. They evaluated this tool through user studies with students and professional developers. They found that "Code delivered with such a tool by developers previously inexperienced in security contains significantly less security problems [52]." Although Nguyen did not empirically evaluate Fixdroid's potential impact on the quantity of code produced, no participants reported expectations of reduced code production [52].

I and others then devised a solution to help target more complex vulnerabilities using a technique called interactive annotation [61]. When this solution is used, the tool scans the source code for predefined sensitive operations. The tool then places annotation requests to the left of each sensitive operation in the coding window. The developer then clicks on the annotation request and annotates, or highlights, the check in the code which keeps the sensitive operation from being exploitable. If the developer forgets to implement a check for a sensitive operation, he/she is reminded when asked to annotate. However, if a sensitive operation appears twice but has different checks, the tool will know that one of them is incorrect and a vulnerability will be detected. Once this occurs, a vulnerability icon appears in the column of the

code editing window and the developer can take steps to resolve the problem.[61]

Chapter 4 examines the use of annotation to determine whether developers are capable of this task. This study provides further motivation of the need for application security tools which are both designed for, and usable by, developers.

2.2.3 Dynamic Analysis

Dynamic analysis tools, tools that analyze a running application, have also been developed for security purposes and may be effective at catching and remediating security vulnerabilities. Smeets investigated the adoption of these tools among web developers and found that they were very seldom used [58]. Through qualitative analysis, he determined that the primary reason they were not often used was due to the difficulty of using them. He hypothesized that this may be due to the large amount of configuration that each dynamic analysis tool requires for each application [58]. In an effort to improve the adoption of dynamic analysis tools among web developers, Smeets created the Universal Penetration Testing Robot, UPeTeR. UPeTeR functions as a class library, which web developers include with their application. It then provides an "abstraction of required configuration data" to dynamic analysis tools to reduce the complex configuration required [58]. Individual plugins for each dynamic analysis scanner are required. Smeets evaluated the UPeTeR by having it used by the Software Improvement Group, a software consultancy company. Qualitative analysis and focus groups suggested that developers were "willing to try out and work with UPeTeR. [58]" These results are relevant to the development of any security tools because they suggest that complex or perceived complex configuration can be expected to greatly

reduce developer interest in such tools.

2.3 Security Resources and API's

When considering the creation, detection, and remediation of security vulnerabilities, it is important to consider the underlying application programming interfaces, or API's, available to the software engineer as well as the security resources which are available. If a software engineer develops an application with a vulnerable API, the application will be vulnerable to attacks and traditional methods of vulnerability detection and remediation may fail. Likewise, if a software engineer is unable to locate sufficient security resources through normal channels, additional vulnerabilities may be produced in the application.

2.3.1 Insecure Application Programming Interfaces

Despite all of the efforts to teach developers about security, this is typically not the primary role of developers and as a result, "They have varying skill sets which often do not include security [70]." Since security is such a complex task, some, such as Wurster, suggest that it may not be efficient or realistic to invest the effort required to train every developer to be a security expert. Instead Wurster suggests that investing more security training in developers who develop API's may be a more prudent option [70]. Wurster suggests that if API's are designed more securely, or designed to be more easily used in secure ways, more benefit may be obtained than simply spreading out security training to all developers. After evaluating a variety of options for accomplishing this goal, Wurster suggests that data tagging and unsuppressible warnings may be the most effective tools to achieve this result

with minimal impact to the developer's code production [70]. Of particular interest is the concept of data tagging. Wurster explains that this is similar to the encapsulation of tainted data, with the exception that potentially tainted data is "tagged" with some description about why it may be tainted [70]. This study is relevant to my dissertation because it lends support to the idea of attaching human readable comments to security warnings for the application developer.

It may also make sense to minimize the amount of security that must be implemented in code and instead shift that effort into configuration. Fahl et al. investigated 1009 IOS apps to see if they correctly implemented SSL certificate validation [38]. To do this, they downloaded 1009 apps, used them, and examined their network traffic. They then attempted to present both valid, but self signed (and therefore insecure) certificates and invalid certificates to the applications. They found that 98 of the applications were vulnerable and leaked sensitive information [38]. Fahl et al. then contacted the developers of the applications, notified them of the problems, helped them resolve them, and then interviewed those who were willing. From qualitative analysis of the interviews, they determined why these issues occur. They found that a large part of these issues occurred because developers needed the program to accept self signed certificates during development or during specific production instances [38]. The authors then proposed a detailed SSL framework, namely configuring it as a service rather than interacting with it on the code level. They then asked the participants about this framework in a separate round of interviews and concluded that it was very favorable among the developers. This result suggests that developers may respond favorably to security configurations that reduce the amount of security

that must be implemented in code form. However, we must also keep in mind that work from Smeets suggests that if that configuration is too complex or time intensive, developers may reject the tool.

2.3.2 Limited Security Resources

One major concern of any potential security tool is the potential impact on code production. Acar et al surveyed 295 app developers and conducted a lab study with 54 Android developers to determine what resources they used to solve security related problems [19]. During the lab study, they broke their participants into groups and gave them timed functional but security-critical tasks to complete. Some groups could only use stack overflow to complete their tasks while others could only use the official documentation [19]. They found that those who could only use stack overflow accomplished significantly more of the functional requirements. However, they also found that those who could only use the official documentation completed code which was significantly more secure [19]. The authors then used this result to call for more secure but usable documentation. This result suggests that developers will choose whatever option accomplishes their development goals in the shortest time even though effective secure alternatives may exist. It also suggests that official API documentation may not be particularly prone to security errors. However, developers may elect to use faster, easier options that do contain security errors, such as stack overflow.

Simplifying complex security tasks to make them usable is not a new phenomenon. In an attempt to make cryptography more usable and prevent security errors, many

experimental cryptographic API's have been created [18]. However, none of them had been evaluated. Acar et al. decided to evaluate the effectiveness of these API's. They recruited 256 Python developers from Github and gave them tasks involving asymetric and symmetric encryption using five different experimental usable cryptographic API's [18]. After analyzing the results of their tasks and the self reported sentiments of their participants through an exit survey, Acar et al. concluded that simplicity of the cryptographic API's was insufficient to ensure the security of the code. They found that this was mainly due to three reasons, namely poor documentation, lack of code examples, and a lack of auxiliary features [18]. The authors call for cryptographic tools to include support and accessible documentation with "secure, easy-to-use code examples." In other words, if a tool is to be effective, it must include code examples, easy to access documentation, and human support. Another concern for application security is the availability of general security advice for developers. For most web developers today, this advice is obtained through general resources on the world wide web [20]. To perform an empirical assessment of the effectiveness of these resources, Acar et al identified and analyzed 19 online general security guidance resources [20]. They found that highly accessible resources exist that cover topics like handling user input, user privileges, secure networking, and cryptography. However, concrete examples and exercises were lacking. Other topics such as program analysis tools, logging/auditing, and data minimization were not covered well [20].

2.4 Code Review

Another lesser explored means of detecting and mitigating security vulnerabilities is through the use of a technique known as code review. This is a bug detecting technique invented by Fagan in 1976 [35]. His original version of it is often termed a "Fagan inspection." During a Fagan inspection, four people meet to review code. One of these people is the author of the code. Another is known as the reader. A third person acts as the moderator, and the remaining people act as reviewers. The process consists of several stages, including planning, overview, preparation, meeting, and rework. The participants iterate through the stages repeatedly until all of the identified issues are fixed.[35, 36, 37]

The Fagan Inspection has been shown to be very effective at finding bugs [21, 37, 23]. This is because the process is very thorough. Fagan himself ran a study in which code was developed with Fagan inspections and other code was developed without Fagan inspections. He found that many more issues existed in the code which was developed without Fagan inspections [37]. However, the process is incredibly slow and time consuming since developers must prepare for a meeting, meet with reviewers, explain each line of code, make adjustments, and repeat the process. Current research estimates that this process typically requires about 9 man hours per 200 lines of inspected code [32]. It is for this reason that Fagan inspections are rarely used today. Instead, new techniques known as "Light code review" techniques have taken their place in many organizations [57, 25]. These "light" techniques include "over the shoulder", "pair programming", "email pass around" and "modern code re-

view." Some organizations and development teams use a hybrid approach consisting of multiple "light" techniques [57, 25].

Much work has already been done on the effectiveness of code review in many domains. McIntosh has shown that formal code inspections of the past were very effective at producing fewer bugs in code but that more modern lightweight code review tools were not as effective when the code was poorly reviewed [46]. McIntosh acknowledged the considerable overhead involved in formal code inspections. However, in order to understand modern code reviews, and possibly identify gaps in the existing research, I must examine several aspects of modern code review. First, I must examine why code review is actually performed today. Second, I must identify the kinds of issues that code reviews actually fix. Third, I must discuss and evaluate the current code review tools.

2.4.1 Why Code Review is Performed

In order to understand whether or not these modern code reviews could be effective for detecting security vulnerabilities, we must first understand how they are performed today, why they are performed today, and most importantly, how effective they actually are in practice. Bacchelli et al. investigated the objectives of code reviews and contrasted them with the actual results of code reviews [25]. They wished to determine the main challenges to modern code review. To do this, Bacchelli et al observed and interviewed developers at Microsoft and issued a survey to managers and programmers [25]. Although each team performs code review differently at Microsoft, many use a tool called CodeFlow. Bacchelli obtained 165 survey answers for the first

survey, 873 responses for the second survey, 200 code review comment threads, and a list of 100 randomly selected participants who had used CodeFlow. These participants included junior developers, senior developers, managers with one team, and managers who managed several teams. They interviewed 17 of these participants to obtain additional data.[25]

Finding defects was the number 1 stated reason among both managers and developers for performing code reviews, followed closely by "code improvement"(removing dead code, adding exceptions, etc.) and "alternative solutions [25]." Some less common reasons were "knowledge transfer", "team awareness" and the benefit of feeling that the code is shared vs. individually owned. Surprisingly, developers seemed to rank code improvement as slightly more important than managers: only 31% of managers vs. 39% of developers rated it as their primary motivation. Managers and developers differed regarding alternative solutions with 17% of developers and only 2% of managers rating this as their most important motivation for code reviews. Additionally, 8% of developers, but no managers, chose knowledge transfer as their main motivating factor. Developers also rated team awareness as the number one motivation in 12% of cases [25].

By examining the actual code review threads, the authors were able to determine whether or not actual code review results aligned with the stated motivations for code reviews. They found that, surprisingly, defects in code was only the fourth most frequent issue by total number of threads [25]. Code improvement was the most frequent category, but code improvement comments were also considered the least helpful based on the interviews. With the exception of knowledge transfer, they were

unable to determine if any of the additional stated motivations actually aligned with code review threads, due to their social nature [25].

They also identified several challenges to modern code reviews. Many interviewees directly mentioned that understanding of code was their main challenge. They found that file owners, who have great understanding of the structure of a file, often gave more thorough and insightful reviews, while reviewers who had less understanding of the code often left code improvement comments [25]. Developers pursue a variety of channels to increase their understanding of code before giving reviews. Some of them read code descriptions or tried to run the code. Others would send emails or talk to the author. This prompted the authors to suggest "features such as diffing capabilities, inline commenting, or syntax highlighting, which are limited when dealing with complex code understanding "[25]. The authors also called for organizations performing code reviews to promote better understanding of the code so the reviews are more thorough. Lastly, but perhaps most importantly, the authors called for the automation of code review tasks (which can be automated). They recommended the use of static analysis tools to detect code improvement issues so that it would "free reviewers to look for deeper, more subtle defects."[25]

2.4.2 What Does Code Review Actually Fix?

Beller, et al. built upon the work done by Bacchelli in a similar study, determining what code reviews in large open source projects actually fix in practice [27]. To do this, they examined approximately 1400 changes in the files of two large open source projects. First, they found that 75% of fixed issues involved syntactical correctness,

style issues, or other maintainability concerns, while 25% of issues involved code functionality issues. This ratio is very similar to that typically encountered in academic or industry systems [27]. Second, they found that approximately 7% to 35% of requests for review are ignored, and developers initiated changes of their own accord in 10% to 22% of cases. Lastly, they found that when a change was made to fix a bug, few changes were made. But if the code had been heavily altered recently, the amount of changes to that code were much higher [27].

Bosu and Carver evaluated the use of another open source code review tool known as Review Board on open source projects. Contrary to Beller, they found that most revisions were taken up by the developers themselves without a code review [28]. They also found that a small number of reviewers tend to make most of the review requests and that developers who contributed most to the project also addressed the most review requests [28].

Albayrak and Davenport evaluated whether or not encountering certain types of code defects during code review would influence the results of the code review. To do this, they recruited 88 students and had them examine code which contained certain defects [21]. Some students reviewed code containing indentation defects. Others observed code containing naming defects, and a third group examined code containing both naming and indentation defects. Lastly, a fourth group examined code which had no defects. In cases where both types of defects were present, students reported the most false positives and found the minimum amount of defects of any group. The indentation defect group found more defects than the naming defect group, which did not detect a significant amount of defects. All groups with code containing defects

reported more false positives than the baseline group [21]. These results suggest that when code reviews are performed with code review tools, code with large amounts of defects is also likely to produce large amounts of false positives. Code which has fewer defects is likely to produce fewer false positives during code review.

2.4.3 Modern Code Review Tools

Various tools have been developed for tackling modern code reviews. Most support a set of core common features. Two of the most important features include side by side viewing of the code before and after a proposed change, as well as inline commenting inside the code by reviewers. Additionally, most code review tools integrate with one or more types of source code repositories, such as Git, Subversion, or Perforce. Most contain some sort of dashboard, and allow users to view recent reviews. They also offer a detailed history of reviews and provide the ability to mark issues as open or closed [5, 11, 8, 10, 1, 12, 7, 3].

One very common category of code review tools is light tools with web based interfaces. One of the most popular of these tools is Gerrit. Gerrit is open source, contains a web interface, and integrates with Git repositories. It can intercept Git commits, perform reviews, and then later perform the commit [5]. Review Board is another commonly used open source code review tool with a web interface. Its features are very similar to Gerrit, but it supports any kind of file, rather than just source files. It is also not as strictly tied to Git and can support different types of repositories [11]. Thongtanunam developed a tool called “Reda” which runs on top of Gerrit and provides web based visualizations to large code review datasets [63].

Some tools are designed internally by large companies and sometimes released later as open source. One example of this is Google's own internal code review tool, Mondrian. Mondrian was designed to help Google engineers review code that used many internal libraries and services, such as BigTable [7]. Although it is not open source, portions of it were converted into an open source review tool called Reitveld [10]. This tool uses Subversion as a source code repository [10, 12]. Facebook also developed an internal tool called Phabricator. Phabricator is a much heavier tool than Reitvald. It supports several different types of repositories, instant messaging with coworkers, an alarm system which notifies that certain files are modified, and an optional command line interface [8].

Other code review tools are designed by companies to be sold as commercial products. These are often "heavy" code review tools. Two notable examples include Crucible by Atlassian and Collaborator by Smart Bear. Crucible contains a web interface and is licensed to companies. It supports various types of repositories. Although it is not open source, clients can view and edit the source code for their own needs [1]. Perhaps most interesting in this category is Collaborator, which supports multiple repositories and relies on its own proprietary client software instead of a web based interface. The tool supports most features common to other code review tools. It also includes support for 11 different types of repositories, which is more than any other tool supports. In addition to common features, it includes support for electronic signatures and auditing which meets FDA, ISO, and CMMI compliance requirements [3]. Smart Bear not only developed this tool into a commercial success, but also conducted a 10 month long study with Cisco to determine what was effective

and what wasn't in code review tools. During this time, they examined 2500 reviews from 50 developers and a grand total of 3,200,000 lines of code. They found that with the introduction of Collaborator, Cisco saved approximately 6.5 staff hours per code review with no noticeable decrease in code review effectiveness [33].

Other tools are designed for a very specific purpose in mind and do not necessarily support the set of features common to most code review tools. Several tools were designed for research purposes. These include two tools used to evaluate whether refactoring could be made more effective through code review tools. Alves designed a tool called RefDistiller to deal with refactoring issues that arise when code reviews are performed. RefDistiller examines the code after manual refactoring using two separate techniques and advises the developer on how to proceed [22]. Zhang designed a similar tool called Critics. However, Critics goes a bit farther and allows developers to customize a template which can be used to determine what sort of refactoring issues are detected [74]. It can also detect edits which are inconsistent or missing altogether through this template [74].

Other code review tools built for research include a tool designed by Holzman called SCRUB. SCRUB was experimental in that it was designed for large groups of developers to use, and it utilized reports from source code analyzers. Both the reviews and reports could be viewed from a single interface [39]. Hundhausen et al. designed an online code review tool to assist with code reviews conducted as part of a course. This online environment allows students to submit code, view code submitted by other members of their team, to leave code reviews, and to alter their code based on reviews from others. Initial evaluations showed that it made peer code reviews

both more organized and efficient [40]. Lastly, Muller had the idea of extending code review to a tabletop environment and created a preliminary prototype. This design allows groups of developers to collaborate on style issues and other code smells, as well as create reports and annotate the source code [49].

2.4.4 Code Review Recommendations

Smart Bear reported many findings from their large study and used these to make recommendations for code review tools. One of their findings was that the amount of defects detected, or the "Defect Density" drops off dramatically after about 400 lines of code are reviewed by a reviewer [33]. This means that a reviewer should not review more than 400 lines of code at one time. In addition, they recommended that sessions should last no more than 90 minutes. They also found that large quantities of simple or inaccurate reviews were provided when the code author did not prepare for the review. Therefore, they recommend that the author prepare for the review and annotate the code. Additionally, they called for quantifiable goals to be established for code reviews and metrics to be used to evaluate those goals. They also claim that the use of checklists improves results for the author and the reviewer [32]. Additionally, they discussed two effects that they believed were important to consider when performing code reviews: the first is the "Big Brother" effect which essentially states that if developers feel they are being constantly watched and evaluated on their performance with code reviews, they will become disheartened and may be both hesitant to offer reviews to others and harsh in their response to reviews [32]. Smart Bear claims that this effect should be minimized in any code review tool. Secondly, Smart

Bear suggests that even if time is not available to review all relevant code, part of it should be reviewed to benefit from something known as the ego effect. This can be thought of as the notion that developers will put in extra effort when working on their code so that they do not become embarrassed when the code is reviewed by coworkers.[33, 32]

2.4.5 Code Review and Static Analysis

Although static analysis and code review are typically two independent techniques, sometimes elements of both are combined or one is performed before the other. In cases where static analysis is conducted before code review, Panichella questioned whether or not warnings produced by static analysis tools are addressed during code reviews [53]. He also questioned if certain types of warnings are addressed more than others. To do this study, he extended a previously discussed code review tool called Gerrit [53].

Panichella located 6 large open source projects on Gerrit and the history of changes and reviews to these projects [53]. The projects themselves included Eclipse CDT, JDT Core, Eclipse Platform UI, Motech, Open Daylight Controller, and Vaadin. By examining certain XML files, Panichella was able to determine that a static analysis tool called Checkstyle was used on both Open Daylight Controller and Vaadin while both Checkstyle and another static analysis tool called PMD were used on Motech [53].

To determine whether or not warnings produced during static analysis are addressed during code reviews, Panichella ran the code for the three Eclipse projects

which did not use static analysis against both Checkstyle and PMD. He determined that for two of the Eclipse projects, there was a significant density difference in the Checkstyle warnings between the first and the last patch [53]. However, since the results of the Mann Whitney test were not significant, he could not conclude that the projects that used static analysis actually did so *before* performing code reviews. Roughly 10% of warnings were removed from projects that did not use static analysis tools, but around 15% were removed from projects which did use static analysis tools. This suggests that the use of static analysis tools causes more warnings to be removed during code review. To determine whether or not certain types of warnings were addressed more than others during code review, Panichella examined 10% of the warnings manually. From this, he was able to determine that warnings in certain categories were addressed far more often in code reviews than warnings in other categories. Specifically, over 50% of warnings in categories such as imports, type resolution, and regular expression were fixed. This result, combined with the results for the first research question, caused the author to conclude by calling for static analysis tools to be configured to highlight warnings which are more relevant in the current software development context [53].

Panichella determined which categories of issues raised in static analysis tend to actually be solved by developers. However, he was only able to determine which projects had been run through a static analysis tool based on the presence of XML files. He was unable to determine whether or not the tools had in actuality been run before the code reviews were performed [53]. Also, static analysis tools are typically run on a workstation, not a collaboration platform. This means that if these tools were

used, it is likely that it was only by one developer on the team. This developer may have had no interest in addressing warnings which did not involve his or her own code. These findings are important because they show that developers pay more attention to certain types of warnings when static analysis is combined with code review. This may translate into developers paying more attention to certain types of vulnerabilities when interactive static analysis is combined with security code review. The findings also suggest that interactive static analysis would prove superior to normal static analysis when combined with code review since it automatically highlights warnings which are more relevant in the current software development context.

Balachandran attempted to determine whether or not static analysis tools would be effective for detecting problems in code when combined with code review. He designed a tool called Review Bot, which performed static analysis using Checkstyle, PMD and Findbugs [26]. However, instead of displaying the output of these tools in the workspace, the tool used the output to create code review requests for the code on a web based code review platform Review Board. Additionally, the tool recommended a reviewer for each code review request based on a novel algorithm. Unlike previous algorithms which select reviewers based on revision history of the file, this algorithm specifically focused on the "line change history" of the lines which were considered part of the review [26].

Ultimately, Balachandran hoped to 1) Assess whether static analysis could reduce the amount of human effort expended in code reviews, 2) Determine whether or not "tightly" integrating the static analysis process with code review would increase the effectiveness of code review, and 3) Assess whether or not the new reviewer selection

algorithm is more effective than the traditional technique of selecting reviewers based on file change history [26]. To find the answers to these research questions, Balachandran used review bot on some VMware projects and found that it generated 6,252 comments. Most of these issues were coding standard violations and style related issues. To evaluate whether or not the comments were effective, Balachandran recruited 7 developers at VMware and had them analyze 1,000 of the 6,252 comments. By evaluating the responses, he determined that developers only had concerns about 14.71% of the comments. Of this quantity, 12.8% could be blamed on improper priority values (when an issue was more severe or less severe than review bot determined). The developers also agreed to fix 94% of the issues which were found [26].

Balachandran took this information to suggest that static analysis techniques would be successful in reducing human effort for code reviews, although it is unclear whether or not tightly integrating static analysis with code review in this manner is superior to performing static analysis before a code review process [26]. One benefit to this approach is that the quality of code reviews could be improved, since developers would be unable to leave simple comments about coding style (the system would have already created the comments). Integrating the process with code review, from a process standpoint, would also ensure that static analysis actually gets performed. However, performing static analysis prior to, and independently from, code review, would prevent humans from having to view and act on the comments, and it could be argued that enforcement of static analyses procedures would ensure they are actually performed. These findings are very relevant to my dissertation topic, since they show that static analysis can be used to reduce human effort in code reviews. Likewise, it

is very likely that performing interactive static analysis prior to security code review could greatly decrease the amount of human effort required to perform security code review [26].

Balachandran showed that code reviews do well when combined with static analysis, but he did not address another underlying and perhaps more serious problem. Developers don't like to use static analysis tools and find the warnings to be annoying [24]. Sadowski and her team of researchers at Google examined the dev ops cycle in Google [57]. They explained that Google uses a standard, distributed build system, which can be integrated into various IDE's. Naturally, this causes them to have a very large codebase with a lot of libraries. To keep bugs from being introduced into their code base that would break other projects, they conduct tool assisted modern code reviews using a platform very similar to Gerrit.

With this information, Sadowski designed a tool called Tricorder to be used by Google engineers [57]. She hoped to design a static analysis tool which would actually be used by developers without explicit instructions to do so. She also hoped that it would scale well and integrate into the developer's workflow. Also, she hoped that it would allow developers to design their own analyzers. Sadowski believed that false positives should be kept low, despite the fact that some issues may be missed. She also considers any false positive to be defined as something a developer does not want to see, even if the issue is legitimate.[57]

To assist with this process, Sadowsi created a language agnostic service as a part of Tricorder [57]. This enabled users to create analyzers, which could then be used by everyone at Google. However, to ensure the false positives were kept low, analyzers

which produced too many false positives (more than 10%) were put on "probation" and the author would be given a chance to correct the issue. If the issue was not corrected, the analyzer would be removed from the tool. Customization of analyzers was also offered at the project level, not at the developer level. In keeping with the notion of a minimum of false positives, Sadowski did away with the common notion of "severity" of issues, common to static analysis tools. Instead, she used two classifications. For very serious problems, some of the built in analyzers would break compilation of the code. Less serious warnings would show up in Google's code review tool. To help developers deal with these issues, the comments in the code review tool were called "robocomments" to distinguish them from human comments [57].

Analyzer writers were also encouraged to provide fixes for the issues that their algorithm detects [57]. When they were provided, Tricorder would provide a link to show the fix to developers as well as a button to preview the fix and a button to apply the fix. Developers and reviewers could also click a link called "not useful" and quickly report a bug to the analyzer writer. Reviewers also had the option of clicking a link called "please fix" to indicate to the developer that it was not a false positive and should be addressed. To keep from overburdening the developer, only the robocomments which were relevant for the current code under review were displayed [57].

To determine how well Tricorder met their goals, Sadowski evaluated the usability of the tool by assessing the quantity of "not useful" click rates and the number of clicks on the other options over a one year period [57]. Sadowski found that "not useful" was clicked for approximately 5% of warnings. They found that if they removed analyzers

which were "on probation" the number would drop to 4%. They found that for 10% to 60% (based on the analyzer) of issues, analyzer writers addressed the problems which were flagged by clicking the "not useful" link. They also found that Tricorder detected an average of 93,000 issues per day across Google and "please fix" was clicked an average of 716 times per day. Tricorder itself was run an average of 31,000 times per day. Based on this data, Sadowski inferred that the tool was generally well liked and supported her hypotheses. She claimed that "We have found that code review is an excellent time to show analysis results "[57]. She reasoned that this likely derived from the fact that the feedback is received before the code is actually committed to the repository, and it is not dependent on any particular IDE. She also stated that this was helped by the peer accountability brought on by code review [57].

Previous work has also produced some guidelines for displaying general static analysis warnings [56]. Sadowski conducted an evaluation at Google and found four major criteria that a code review process must satisfy to be adopted by developers [56]. The first is that the warning must be understandable easily by any engineer. The second criteria is that the warning must be actionable and easy to fix. Guidance must be provided. However, the issue can be more in depth than compile-time warnings. Third, the tool must produce an effective false positive rate of less than 10 percent. Sadowski explains that an effective false positive is something that a developer does not want to see, regardless of whether or not it is an actual false positive. Sadowski frequently points out that the 10% false positive tolerance is much less strict than it is for compile time warnings. Lastly, for the final criteria, Sadowski states that a fix for the warning should have the potential for significant impact on code quality. In

other words, the developer must feel that the issue is a real issue and worth the time required to take action [56].

These findings are important to my dissertation topic for several reasons. First, they show that if false positive rates are kept low, developers may actually perform static analysis. Second, they show that when the results of static analysis are placed into code review, the results are well received by developers. This is important since I intend to output the mitigations from interactive static analysis into the security code review process. To summarize:

- The use of security code review after interactive static analysis may lead to fewer issues being ignored
- The human effort to address and mitigate vulnerabilities may be reduced when interactive static analysis is combined with security code review
- Developers may react positively to combining interactive static analysis with security code review
- If my solution keeps false positive rates low, developers are likely to use it

2.4.6 Security in Code Review

A few studies have been done to assess whether or not security issues can be effectively detected and mitigated through code review. However, none of these studies involved code review tools.

One such study was done by Meneely. Meneely analyzed 159,254 code reviews, 85,948 Git commits, and 667 post-release vulnerabilities which were contained within

the open source Chromium Browser, the underlying base of Google's popular Chrome Browser [47]. They used a collection of metrics and statistics and found that counter-intuitively, when files were reviewed by more developers, they were actually more likely to contain security vulnerabilities. When files were reviewed by fewer developers, they were likely to contain fewer vulnerabilities. They also found that if the developer performing the review had prior experience either fixing vulnerabilities in code reviews or leaving reviews about vulnerabilities, the file was much less likely to contain a vulnerability [47].

Bosu conducted a similar study as Meneely but took it a step farther. Bosu wondered which sorts of security issues could effectively be detected by code reviews and which could not. Bosu also sought to determine what common characteristics exist in vulnerable code changes or VCCS and what sort of developers are more likely to produce vulnerabilities [29]. To do this, they examined 267,046 code review requests in a total of 10 different open source projects. In this process, they found 413 vulnerable code changes. They concluded that, in fact, code review could be used to catch common vulnerabilities. However, they also found that greater source code changes meant a greater chance of a vulnerability being introduced. Additionally they found that greater contributors of code changes produced more vulnerabilities than smaller contributors, but that the changes of smaller contributors were 1.8 to 24 times more likely to be vulnerable. Moreover, they found that files which were modified were more likely to contain vulnerabilities than new files. These results led them to recommend the use of dedicated security review teams to perform code reviews with a dedicated security focus [29].

These studies suggest a few important points that any security code review tool should take into consideration. The first is that using large quantities of reviewers to detect vulnerabilities via code reviews is unlikely to prove successful. However, if small numbers of developers with security experience perform reviews, they are much more likely to catch security vulnerabilities in a code review. Also, inexperienced developers are much more likely to produce vulnerabilities in their code, and files which have many changes should be treated as more suspect for containing vulnerabilities than files with fewer changes.

2.5 Conclusion

This research clearly points to an open problem. How can simple and application-specific vulnerabilities be detected and mitigated effectively and efficiently, with minimal human workload and effort? Static analysis tools are very good at detecting simple security vulnerabilities, but require custom rules which are very complex and require a large amount of time from security experts [24]. They also do a very poor job of detecting application specific vulnerabilities, and provide limited means of mitigating detected vulnerabilities or training the developer [24]. Fagan inspections are very effective at detecting bugs, but take a prohibitive amount of time and do not perform as well with detecting and resolving security vulnerabilities [37, 29, 33]. Light code review techniques are not nearly as time intensive as Fagan inspections, but whether or not they are as effective as Fagan inspections at detecting bugs is debated [46]. Additionally, they have been shown to be ineffective at detecting security vulnerabilities [29]. Tool-assisted techniques show promise, but have not been shown to be

effective on their own with detecting and mitigating security vulnerabilities [26, 57].

A different solution which addresses the shortcoming of today's techniques is needed. This solution should ideally bypass the slow and expensive task of static analysis rule writing. It should also be able to detect complex vulnerabilities and application specific vulnerabilities with minimal human effort. Moreover, it should automatically mitigate simple vulnerabilities and provide assistance to developers to help mitigate complex or application specific vulnerabilities. It should provide verification that the resolutions to vulnerabilities are actually correct and should catch incorrect resolutions. Lastly, it should facilitate or encourage a process that results in training of the developer to reduce the amount of vulnerabilities in future code.

CHAPTER 3: ROLES, NEEDS, AND CHALLENGES OF SECURITY AUDITORS

3.1 Introduction

As previously detailed, I have shown the effectiveness of interactive static analysis and interactive annotation to detect security vulnerabilities. I have also detailed the body of literature supporting the use of dedicated code review to support the detection of security vulnerabilities. However, if tool assisted dedicated code reviews are to be effective, it is probable that security experts will need to be involved in the process. Unfortunately, very little research to date has been performed on security experts who review code themselves. Therefore, little is known about their day to day job tasks and their interactions with developers.

Security vulnerabilities are introduced by developers while creating software. Organizations implement a variety of processes to find and fix vulnerabilities in the software development lifecycle, Yet, developers are not heavily involved in such security processes [42, 57]. Prior research suggests that developers feel that security is the responsibility of other parties, and consequently avoid engaging in security practices when possible, delegating to other people and processes within the organization [42, 57, 73]. Additional research has examined why developers have difficulty using static analysis tools to detect security vulnerabilities, including large numbers of false positives, lack of collaboration support, and complicated tool output [42]. Other

researchers are examining guidelines and approaches for improving the usability of static analysis tools for developers [57, 60, 59].

All of this existing and ongoing work has focused on the perspectives and needs of developers. I expand upon these results by focusing on the people who currently do application security work. Application security is often the responsibility of a "Software Security Group," or SSG, within organizations. Application security experts within that group are charged with performing static and dynamic analysis to find security vulnerabilities in application source code. By focusing on this important group of users, I examine additional needs and experiences of all people involved in application security work to further inform the design of tools and processes for reducing security vulnerabilities in applications.

In this chapter, I report the results of an interview study of application security experts who *examine source code* for security issues, who I refer to as security auditors. My goals are to gain an understanding of the security processes in their organizations, their interactions with developers and other stakeholders, and their perceptions on the challenges they face and impact on application security. This knowledge will support the research and development efforts of tools aimed at these security experts such as security code review tools, and the developers they work with. The end goal is to improve the experiences and interactions of developers and security experts, thereby improving the security of applications, and reducing the likelihood and occurrences of security attacks on applications. This study was published in the 2018 Conference on Human Factors in Computing Systems [62].

3.2 Related Work

3.2.1 Security in Software Development

Organizational factors, such as organizational structure, organizational politics and managerial pressure, can impact the quality of the code that developers write [44, 51, 50]. The security of the code is similarly impacted. For example, Kroksch & Poller examined the potential of an external security consultation to lead to new security routines within an Agile development team [54]. They found that the intervention did lead to increased awareness and desire to incorporate security as an important quality of their software. Despite this interest, security was not ultimately made an integral part of the development process because such security work was discouraged by the organizational structure and organizational goals. This case study emphasizes the importance of not just increasing developers' attention to security issues, but also to the need for organizations to develop routines and reward the security work that developers do. Similarly, Jing et al. found "A disconnect between developers conceptual understanding of security and their attitudes regarding their personal responsibility and practices for software security [73]." Even with awareness of the importance of security, developers expected other people and processes to take care of those issues. Thus, simply increasing developers' knowledge of security vulnerabilities will not necessarily lead to an increase in their performing security work.

3.2.2 Developers and Security Tools

Researchers have established that developers underuse security tools, focusing in particular on static analysis tools, for a variety of reasons. Through an interview study of developers, Johnson et al. found that while developers' perceived static analysis tools to be beneficial, they did not use them due high false positive rates, and the ways in which warnings were presented to users [42]. Developers are also impacted by their social and organizational environment, and are more likely to adopt security tools when their peers already use and trust those tools [69]. Interacting with security experts can also increase developers' feelings of responsibility for securing their code [69].

Existing commercial static analysis tools may not be sufficiently usable for developers, who will have more limited security knowledge. Smith et al. examined the kinds of questions that developers ask when assessing possible security vulnerabilities found by static analysis tools [60]. In categorizing these questions, they found that developers do a lot more work to understand a vulnerability and its remediation than simply reading the warning notification. Developers need support for a wide range of code understanding, including tracing the flow of untrusted data, comparing vulnerabilities against previous examples, and finding additional documentation. Tools do not adequately help developers resolve vulnerabilities, resulting in failures to efficiently and successfully address the vulnerability warnings [59].

3.2.3 Security Experts and Challenges They Face

Little research has been done on security experts who examine code. However, some studies have examined security experts in general, the challenges they face, and their day to day job tasks [66]. Most of these studies' methodologies involved data collection through interviewing security experts with semi structured interviews and qualitative evaluation [66, 68, 64]. Some of it is particularly encouraging. Werlinger conducted 36 semi structured interviews involving security experts from 17 organizations in government, industry and academia [66]. He found 18 key challenges that they faced. Some of the key challenges that he identified include the following: Lack of security training, lack of security culture, communication issues, access control to sensitive data, vulnerabilities in systems and applications, and lack of efficient security tools [66]. This is particularly encouraging since my research plans to target these issues specifically. It is also very relevant to my dissertation topic since I intend to study security experts who examine code. It seems very likely that they may face similar challenges since they are a subset of security experts. The findings from this study have been leveraged to form the basis for the study detailed in this chapter. [66]

Another open question is what do security experts who audit code actually do on a daily basis. Again, little research has been done on the work of security experts who audit code. However, Werlinger has studied the day to day job tasks of security experts in general [68]. He found that security experts typically iterate through three stages: preparation, anomaly detection, and anomaly analysis. During the preparation phase, they gather data on what is occurring on the network, using tools.

During the anomaly detection phase, they utilize logs, tacit knowledge, and intrusion detection tools to identify odd behavior. Lastly, during the anomaly analysis phase, they attempt to study the anomaly to determine whether or not it warrants further action [68].

Studies have also been conducted which focus on tools that security experts use, particularly Intrusion Detection Systems (IDS) [67]. This research followed the same methodology (semi structured interviews) with security experts who used intrusion detection systems. Werlinger found that security experts generally preferred command line tools, unless a graphical report or visualization was needed. Velasquez conducted a similar study that focused on the tools that system administrators, including security experts, use and she arrived at the same conclusion [64]. Werlinger found that security experts also typically leveraged intrusion detection tools as a means of quickly sorting through log files and notifying them when issues occur. The main issue with the adoption of these tools was cost, which could sometimes be around $250,000. Additionally, configuration times were mentioned as a concern. They also had concerns about false positives, but were far more concerned with the tool missing actual issues. Therefore, this suggests that any tool which is made for security experts should aim to minimize missed issues, even at a risk of including more false positives. Tools for security experts should also minimize cost and configuration time if possible [67].

3.2.4 Security Expert Interactions

Perhaps what is most interesting is yet another study by Werlinger, in which he examined the interactions among security experts and the tools they use for that interaction [65]. Again, the primary means of gathering data was through the use of semi structured interviews with participants. Werlinger found that the roles of security experts differed, as did the times that they needed to interact with other security experts or stakeholders. However, a number of trends emerged. Many security experts interacted when they performed security audits. In these cases, they needed to work with other IT specialists to explain the vulnerabilities which were found. They also interacted with each other and stakeholders when designing services that incorporated security criteria. In these cases, they often served a role of consultant. Some of them would also receive notifications directly from end users and would interact with the end users to obtain more information and advise the user. Many of them were also called upon to implement access controls as well. During this task, they often had to interact with human resources to determine who should be authorized for what task. Additionally, some of them were called upon to train and educate other specialists, as well as stakeholders. Many of them also had to forward information to relevant IT specialists when new vulnerabilities were announced by system vendors [65]. Moreover, many of them interacted with various stakeholders, other security professionals, and other staff when creating security policies. Lastly, many of the security experts who were interviewed would frequently interact with project stakeholders when responding to security incidents. This study is particu-

larly interesting to my dissertation topic, since it describes the kinds of interactions that security experts in general engage in. It is possible that security experts who examine code may be involved in similar interactions. This study has also been leveraged to help craft the questions asked as a part of the study detailed in this chapter [65].

3.2.5 Security Expert Tools for Communication

Werlinger found that the tools security experts used to interact were rather basic [65]. Email was by far the most commonly used tool, though many expressed concern about it being an inefficient means of communicating. They also used text and video chat, meetings, and phone calls as well. One major concern that security experts had was that they needed to store records of most of the conversations and reports. These channels, by default, did not lend well to this task [65].

Based on this information, Werlinger made several recommendations for improving the tools that security experts use [65]. Most importantly, he recommended that tools be made to better support collaboration. He also recommended that they decrease complexity, disseminate knowledge, and provide flexible reporting. Moreover, he recommended that security tools and communication tools be integrated to better support interactions. Lastly, he recommended that tools reduce communication overhead. He suggested that this could partly be done by integrating security and communication tools [65].

Jaferian published a list of guidelines for improving tools used for security analysts based on relevant literature from other tools and studies of security experts

[41]. Although the guidelines themselves are too numerous to list, many of them are relevant to code review tools. They include the following: Combinable tools, knowledge sharing, ability to support multiple presentation formats, command line and gui interfaces, customizability, different levels of abstraction, easy to change configuration, automatic detection, flexible reporting, integration with communication media, different UI's for different stakeholders, date correlation and filtering, and the ability to share information on the state of an application with other security experts. This information is particularly useful in the design of code review tools if they are to be used by security experts [41].

3.2.6 Related Work-Summary

The existing literature, although sparse, highlights some very important points concerning security experts. First, we know that developers underuse security tools, at least partly due to organizational factors. Second, this existing work shows that security experts often do need to interact with each other and various stakeholders [65]. Additionally, many of them had to train other specialists [65]. We also know that the tools security experts use to communicate were very simple, with email as the most common tool [65]. They were not pleased with this communication channel since it did not work well for storing records and reports. As a result, existing research recommends that tools for security experts should better support collaboration [65]. It also recommends that tools they use for security tasks support collaboration themselves.

These studies are important because they serve two purposes. First, they provide

some background on what we can expect from security experts who examine code. Second, and most importantly, they help to advise the development of tools to be used by security experts. These results suggest that any code review and interactive static analysis tool which a security expert uses should support collaboration. It should also require a minimum of configuration. Additionally, it should support some form of reports and record storage to support their interactions with other stakeholders. Also, this research shows that security experts are willing to tolerate false positives, but rapidly abandon tools if they fail to show actual issues. Moreover, the tool should be very powerful, even if it requires a command line interface for this functionality. However, the tool should also make good use of visualizations. Additionally, since knowledge dissemination was listed as a primary security challenge, the tool should help to spread the knowledge of security experts to other users and among security experts themselves.

3.3 Methodology

I conducted an interview study of experts in application security, seeking participants who either examine source code or perform static or dynamic analyses. For simplicity, I refer to these experts as security auditors throughout this chapter. I recruited these experts using a snowball sampling technique, utilizing contacts from a variety of sources and then asking participants to recommend additional experts. I started with my own personal contacts, and made additional contacts at two security events, namely the CyberSecurity Symposium at the University of North Carolina at Charlotte held annually for regional security professionals as well as OWASP's 13th

Table 1: Participant Demographics

ID	Job Title	Organization Type	ID	Job Title	Organization Type
P1	Principle Application Security Engineer	Large Business Management Software	P17	Senior Security Consultant	Small Consulting
P2	Information Security Engineer	Large Financial Institution	P18	Director of Information Security	Medium Security Software
P3	Director of Security Solutions	Large Internet Service Provider	P19	Systems Engineer (Security Role)	Medium Network Security Services
P4	Senior Software Engineer (Security Focus)	Large Engineering and Consulting	P20	Senior Security Engineer	Medium Call Center Analytics
P5	Technology Manager Information Security	Large Financial Institution	P21	Cloud and Application Security Manager	Large Consulting and Training
P6	Application Security Engineer	Large Healthcare Software	P22	Director of Security Strategy	Medium Risk Management
P7	Security Lead	Large Technology Infrastructure	P23	CTO (Chief Technology Officer)	Small Security Consulting
P8	Senior Security Architect and IT Architect	Large Hardware and Software	P24	Software Developer (Security Focus)	Small Network Security
P9	Sales Engineer	Medium Security Software	P25	Director of Engineering	Small Network Security
P10	Senior Vice President (Role: Security Architect)	Large Financial Institution	P26	Senior Technology Project Coordinator	Large Security Foundation
P11	Senior Security Director	Large Technology Infrastructure	P27	Principle Security Consultant	Medium Consulting
P12	Director of Security Research and Development	Large Security Software	P28	CTO (Chief Technology Officer), CoFounder	Medium Security Software
P13	Program Manager	Medium Consulting	P29	CSO (Chief Security Officer)	Small Security Software
P14	Application Security Consultant	Medium Security Software	P30	Software Engineer	Large Network Infrastructure
P15	Vice President of Threat Research Center	Medium Security Software	P31	Application Security Consultant	Medium Cyber Risk Management
P16	Senior Security Researcher	Medium Security Software	P32	Senior Application Security Engineer	Large Entertainment

Annual AppSecUSA Security Conference.

I recruited a total of thirty-two participants from twenty-seven different organizations, see Table 1. Job titles varied greatly, and included "Security Consultant" (n=4), "Security Engineer" (n=5), and "Systems" or "Software" Engineer with a security role (n=5). As P6 stated, *"In the industry there's security analyst, there's security engineer, appsec engineer, appsec analyst, there's a lot of names. But they should all be able to do the same thing."* Several participants held senior positions, with titles containing "Manager" (n=2), "Director" (n=6), or "Vice President" (n=2). Participants reported an average of 10.72 years (SD 5.51) of professional experience as a security expert. Twenty-seven participants had previously been employed as a developer, with an average of 9.22 (SD 6.24) years of professional experience. All of my participants were male, and 25 Caucasian. While many organizations did not produce commercial software, all participants discussed working at least in part on customer-facing software.

I conducted and recorded an interview over the phone with each participant. The interviews were semi-structured, with a set of basic questions that were varied depend-

ing on the participant's background and the answers they provided. All questions are detailed in Appendix A. I asked about participants' typical workday and responsibilities for software security. I then asked about their own as well as their organization's processes and tools for finding and mitigating vulnerabilities, how they interact with developers, the biggest challenges they and their organizations face in software security, and what solutions they think could address those challenges. Interviews ranged from 30 - 45 minutes, and experts were provided a $10 gift card as a thank you for participating. The study was approved by my university's IRB.

I collected my demographic data at the end of the interviews with a set of closed, structured questions. As part of this, I asked participants to rate their general security knowledge, secure programming knowledge, and programming skills on a scale of one to ten. Participants responded with an average of 8.13 (SD 1.34) for security knowledge and 8.23 (SD 1.23) for secure programming knowledge. Additionally, participants responded with an average of 6.61 (SD 1.37) for programming skills. As expected, participants rated themselves highly on their security knowledge, with the low standard deviation demonstrating that they felt similarly about these ratings.

Interviews were transcribed. I followed an inductive coding process, looking for both common patterns of response to the questions as well as interesting topics and comments. Two researchers independently and iteratively coded five sample participants, comparing and merging their code books with discussion between them. Agreement was reached on the codebook and all codes for those 5 participants, resulting in a codebook of 39 separate codes. The two coders then coded all remaining participants independently with no further changes to the codebook. When coding

was complete, the researchers compared each individual code and discussed and resolved any disagreements. Disagreements were tracked, and inter-rater reliability was calculated at 95.15%. Codes were then grouped into higher level categories which form the subsections of the Results section below.

In addition to the coding, I examined all of the transcribed interviews to perform workflow analysis. Based on each participant's responses, I created a workflow model for each participant and their organization. Once all diagrams were complete, I analyzed all of the diagrams, looking for common patterns and trends. Following this, I iteratively formed an aggregate workflow model representing the most common elements and patterns. Lastly, I analyzed the differences between each of the workflow diagrams as a basis to discuss the variations on my common workflow model.

3.4 Results

I begin by describing the general workflow and security processes my participants described. I then focus on the communication and organizational issues that impacted vulnerability detection and remediation. Following this, I briefly discuss technical challenges and needs that they encountered. My results are all qualitative, but I report the number of participants with similar comments to highlight how prominent different views were in my sample.

3.4.1 Security Processes

Participants worked in organizations ranging from small software companies to very large financial institutions and technology companies, see Table 1. Although processes and workflows for finding and fixing vulnerabilities varied slightly between

participants, most were quite similar. Figure 1 shows an aggregate workflow model that represents my participants and their organizations.

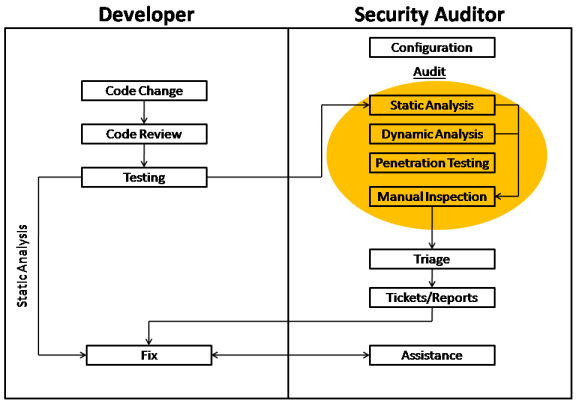

Figure 1: Aggregate Security Auditor Workflow Model

Whether the organization followed a more traditional waterfall-like process and SDLC (Systems Development LifeCycle), or an AGILE methodology, software developers would first perform their implementation tasks. My auditors (n=25) did mention that code reviews and testing were regularly performed by developers looking for all kinds of bugs and quality issues, but that the auditors were not involved in that process as security was rarely a focus of them.

My participants were all part of a separate security group operating outside of development teams. Auditors often oversaw the security of many different applications throughout the organization. Security processes were initiated after changes to the code were complete. Two timeline schedules were observed. If security processes were scheduled at regular intervals, such as annually, the security process was considered an audit. Audits continued to be performed on released code to find issues in the deployed software. Otherwise, the security processes would be scheduled based on the needs of the project and organization. Both timelines used the same processes of

static analysis, dynamic analysis, pen testing, and manual inspection to find security vulnerabilities. More details of these processes are described below.

Once a vulnerability was detected, it was then triaged. This means that the order in which security bugs were remediated was prioritized based on their severity. Security vulnerabilities were often documented alongside all other bugs using bug tracking tools, the most popular of which was JIRA[1]. In other instances, the results of all security processes would be compiled in a separate, out of band report produced for the development team. Addressing vulnerabilities then becomes part of the typical bug fixing process undertaken by the development team. In other words, all participants reported that remediation of a vulnerability was always performed by a developer.

A number of participants discussed the use of continuous integration processes within their development teams. Amazon defines Continuous Integration as "A DevOps software development practice where developers regularly merge their code changes into a central repository, after which automated builds and tests are run. [15]" When continuous integration was used, static analysis was done on the build server every time a new build was created and pushed to production. The results would be integrated into the bug tracking platform and be shown as security bugs. Depending on the severity of the issue, the bugs could fail the build. The developer would then have to fix these bugs and recommit the code. The "integrated" static analysis rules were always configured by a security auditor.

All configuration of security tools and processes was performed by security auditors themselves as needed. In addition to their primary role of finding security vulnera-

[1]https://jira.atlassian.com/

bilities, ten participants also mentioned that they participated in threat modelling or security architecture discussions as part of the overall software development process, but did not mention specifically when those activities occurred.

3.4.1.1 Finding Bugs

The primary responsibility of all of my participants was to find security vulnerabilities in application source code. As expected, all participants reported using static analysis and dynamic analysis tools, such as Fortify[2] and AppScan[3], to detect security vulnerabilities. The participants conducted all static and dynamic analyses themselves. Static and dynamic analysis were sometimes perceived differently within an organization and performed at separate times. For example, as P27 stated:

"In my experience, organizations treat them (static and dynamic analysis) as separate steps and oftentimes I feel that's the result of the way security controls inside of guidance and policy documents express those tasks. They're not treated as a holistic set of controls. They aren't done as a combined activity and I feel that's a bit of a disservice to the practice of application security." -p27

In addition to static and dynamic analysis, twelve auditors mentioned the use of penetration testing, sometimes performed by a separate security team. Finally, two participants used bug bounties to uncover problems in released code that were otherwise missed. Not surprisingly, many (n=24) participants in my sample mentioned the management of false positives and false negatives as a challenge of all of these processes.

[2]https://saas.hpe.com/en-us/software/application-security
[3]http://www-03.ibm.com/software/products/en/appscan

Thirty participants reported that they frequently examined code for vulnerabilities on their own, referring to this as "manual code inspection." This was done to understand discovered vulnerabilities, as part of a security audit, or to investigate issues such as authentication, datastore accesses, and architectural flaws that may not be found by other means. Inspections also occurred to help create customized rules for tools to reduce false positives:

"All applications end up needing custom rules written or some sort of tweak to make the (static or dynamic analysis) scanner work for that application. Manual code review is how you identify those patterns and make sure that your rules are covering them appropriately. We'd typically go through and we'd say 'Alright, let's go find the CSRF protection. Let's go find their encoding mechanism. How are they handling their database queries?"' -p17

However, code inspections were done on a limited scale due to the sheer volume of code.

Another major responsibility of my participants was performing regular audits of projects. Based on different risk designations, projects were required to undergo such audits over various intervals of time. For example, as P2 states:

"Basically there's three levels. That's high, medium, and low, and the high risk projects have to be reviewed every year. The medium ones are two years. The low ones, every three years." -p2

Static analysis, dynamic analysis, and sometimes audit meetings were conducted as a part of the audit process. My participants described the meeting as similar to a traditional code review, however it involved security auditors, project leads, and

sometimes developers.

"If a section of code is deemed very important and critical, then we do get assigned reviewers who have worked on security areas before and then it's like a peer review; they check it. It's normal code review process, but with a emphasis on... the security aspect of it." -p24

Participants would meet face to face and review code in order to identify code smells which may carry risk.

"So, the reviews are targeted typically based on two elements; one is that static analysis has told us that there is a problem or even dynamic analysis has told us there is a problem so they will look in that area. And secondly, where have code changes been made or where have new features been added to the code. So, those are high-risk spots and those are the ones that are targeted for inspection." -p7

Detected vulnerabilities were again triaged. However, verification that developers fixed the issues brought up during these audits was not performed until the next scheduled audit.

3.4.1.2 Fixing Bugs

Triage was an important aspect of my security auditors' workflow, reflecting their perceptions of risk. Serious vulnerabilities would be given a deadline for resolving. Vulnerabilities deemed critical, or higher risk, would have much shorter deadlines. The criteria by which vulnerabilities were triaged were often varied and proprietary. Generally, the probably of an exploit occurring and the impact if it did occur were the two primary factors in the triaging process.

"We're looking at things like, how it can be exploited, how many, how likely is it to be exploited, what are the assets that we're trying to protect? Because obviously if you have an application that's got only, let's say a thousand confidential records at risk, then that's a lot less of a risk to a company as if they're 100,000 confidential records, right? So those types of things are taken all into account. That is part of the risk analysis." -p2

While security auditors would follow through on the more serious vulnerabilities to ensure they were resolved, a project could still continue to move into production containing known vulnerabilities.

"And, unless that's critical or high severity in nature, a lot of lingering issues remain simply by the quantity of issues that come out" -p27

In order to assist developers, participants may provide static or dynamic analysis results, documentation, or suggestions within the bug tracking tool. However, release of the software can occur at any time, at the discretion of management. In most of my participants' organizations, it is possible for release to occur with known security vulnerabilities if management is willing to accept the risk.

"It'll be assigned to a developer. Typically we try to provide as much information in there so that they can reproduce the issue." -p21

3.4.1.3 Summary

To summarize the common security practices reported by my participants:

- Security auditors are solely responsible for detecting security vulnerabilities in code.

- Their processes are labor intensive and often manual.
- Responsibility for fixing vulnerabilities lies with development teams.

3.4.2 Developer Interaction

One of my goals was to examine the interaction that application security experts have with the developers who write the code. While my interviews confirmed that auditors are responsible for detecting bugs, fixing them required communication between auditors and developers as shown in the workflow model in Figure 1.

3.4.2.1 Communication Patterns

To understand communication patterns, I coded any time an auditor reported communication with someone else. Communication between security auditors and developers did not occur as frequently as I expected. Most communication revolved around how to remediate a vulnerability detected during static analysis. Thus, the communication involved notifying the developer of the issue and providing guidance for fixing it. Further interaction could be initiated by a security auditor with an interest in seeing a vulnerability resolved, or by a developer with questions about how to resolve a vulnerability.

Most auditors described communication primarily occurring through the bug tracking or ticketing system. Auditors would provide explanations on what the vulnerability is, and pointers on how to fix it. Developers may then make additional comments to ask questions or for more help. For more in depth conversations, developers would contact the auditor more directly. For example:

"So there is some interaction there but it's typically through the ticketing system

where we'll trade some comments back and forth. There'll definitely be like a re-test. 'Can you make sure it works for me?' They'll ask. Sometimes if, like I said, if they're new to us, if they're new to security, they might ask for a ten minute call just to make sure that they understand the issue." -p21

As P21 stated, how communication occurred was based on the team and the individual developer. Auditors stated that they tried to engage developers wherever they were most comfortable interacting, whether that be in person, over email, or simply through the comments in the bug tracking system.

In addition to the developer assigned to remediate the vulnerability, auditors also interacted with other members of the development team. Analysis results were sometimes packaged as a report and provided to the project, rather than to an individual developer. In these cases, the project lead or a designated developer with an interest in security, known as a security champion, would interact with the security auditor. My participants also interacted with developers as they sought to understand the security properties and use cases of an application during dynamic analysis, manual code inspection, or security code review.

Finally, security auditors interacted with developers during training. Many (n=23) of the security training programs were designed and taught by the auditors themselves. Training could be general application security training, or customized to the types of errors that developers are currently making.

"What we often do is periodically assess the category of vulnerabilities, what the kinds of volume and mistakes the developers are making and provide a very small part of training just on those things." -p22

While training has obvious goals of increasing the knowledge of developers, participants also commented on the importance of raising developer awareness of the auditors themselves:

"While we can argue about whether secure code training gets us anywhere, at least it raises the visibility into these issues and people come back. Now whether or not they remember, you know, all the different things that are out there, probably not. But as long as they know how to access you and they know that you're there as a resource, it is certainly very beneficial." -p21

3.4.2.2 Communication Challenges

The importance of communication to fixing bugs means that communication breakdowns can impact the security of the application when bugs are not properly addressed. One primary communication challenge was coordinating between different teams. Developers and security teams operated independently, and some organizations even had separate teams for static and dynamic analysis. Auditors also oversaw applications for many different teams of developers throughout the organization. Naturally, this leads to communication bottlenecks as auditors must navigate the different cultures and needs of these different teams to coordinate the reporting and fixing of security vulnerabilities.

To make matters worse, security auditors must sometimes coordinate with different teams of developers and project stakeholders to get approval to fix issues. Divergent interests of developers, managers, and other stakeholder means that this is sometimes very difficult. Many participants (n=10) commented that such communication issues

could cause delays in security problems being fixed.

"There's this approval process, IT has concerns, marketing has concerns, and you need that unanimous approval and that if you don't get it that vulnerability persists." -p3

Participants perceived that much of their role is in motivating and convincing other parties to implement security solutions. Security auditors in my study struggled with convincing developers or other stakeholders that a security issue was real and in need of remediation. Many (n=9) participants mentioned that developers had difficulty in seeing a harmless example exploit and understanding that the vulnerability was serious. For example, one participant mentioned that developers frequently did not understand that alert boxes delivered through cross site scripting vulnerabilities could be easily replaced with credential stealing, false login forms by an attacker.

"The biggest question they have is 'Why, why do I actually need to do this, why do I care?' They usually call me out on that, and say, 'So how do you weaponize this, right? What's the attack path here? I don't see an actual vulnerability. All I see is some alert box"' -p17

Auditors also mentioned that communicating vulnerabilities found outside of static analysis could be challenging. Static analysis tools report a particular line of code potentially containing a bug. This means that notification and subsequent remediation suggestions can be provided in the context of the source code. However, dynamic analysis is performed by using test data in a running application. Therefore, if an issue is found, the lines of code causing this issue must be first determined before they can be fixed. Several participants (n=3) mentioned that it was very difficult

for developers to interpret the results of dynamic analysis scans, locate the problem in the code, and then apply the correct fix. Similarly, two participants mentioned that developers struggled to understand how security problems in the architecture of applications were actual security issues. In these cases, auditors had to leverage knowledge of the functionality of applications to demonstrate architectural or logic flaws.

Another commonly reported issue was determining the best way to communicate and motivate different groups of people. For example, three participants commented on the need to be sensitive to the developers' feelings about their code when notifying them of security issues in that code.

"Obviously, if you're examining one of the applications that they have written, you just have to be very sensitive to the fact that... we call it 'calling someone's baby ugly.' So you have to, if you're going to point out a security flaw, you have to do it in a way that's respectful." -p11

Overall, this means that auditors must be familiar with other stakeholders' knowledge and needs in order to determine how to best communicate with them regarding security issues. P14 talked about the importance of this message delivery.

"I think a lot of what I deal with is not necessarily the technology, it's the how you communicate it effectively to developers. My boss likes to joke that we are 60 percent psychologists and 40 percent security professionals. The hard part is determining for the organization that we're talking to at that moment, what is the best way of communicating the solution and the risk associated with an app in a way that's going to resonate with developers and cause them action, and also not make them freak

out." -p14

3.4.2.3 Security Champions

One key type of developer that ten participants mentioned was someone they referred to as a security champion. The security champion was a member of the development team who was an advocate for security. This was an unofficial designation, with no formal security training needed. Instead, they only needed an interest in security.

"The security champion is typically a volunteer from the engineering team who is interested in security, and more importantly interested in reducing the incidents of security vulnerabilities. I work with him or her and discuss what the priorities should be for that particular product as we move forward towards the next release." -p7

As these participants stated, security champions served as important liaisons between the security group and development teams. They help to extend the efforts of the limited quantity of security auditors and can serve as an advocate for security on each development team. When security champions are involved, the security auditor can communicate with the security champion, and the security champion can then communicate with each of the fellow members of the development team. This requires less communication effort for the auditor, with fewer needs to motivate the champion regarding security. The security champion can then communicate as a peer to fellow developers. Consequently, six auditors mentioned that security champions were either required for the developer code review process, or frequently participated in it. Auditors (n=3) also recognized the benefit to "train the trainer" and educate

the security champion instead of all of the developers. The hope is that the security champion could then train other developers on his/her development team. The downside is that other members of the development team may not receive such training and security reports directly, resulting in decreased security awareness overall.

Participants clearly tried to cultivate the relationships they maintained with security champions in a variety of ways. For example, while these roles were informal and not tied to pay, auditors mentioned rewarding champions with trips to security conferences to enhance their training and maintain their interest in security.

3.4.2.4 Summary

Communication with development teams and other stakeholders were important aspects of a security auditor's job, with auditors needing to motivate developers to understand and fix the security problems found. Challenges and breakdowns to this communication could result in vulnerabilities remaining in the application. Security champions served as important and valued liaisons between the auditors and development teams.

3.4.3 Organizational Challenges

Auditors discussed a variety of challenges faced within their organizations that impact how application security is performed and its effectiveness.

3.4.3.1 Balancing Risk and Resources

Twenty-two of my participants mentioned that balancing risk against resource limitations is a key security challenge. The first issue was understanding the risk of the various products, which can be difficult to characterize. In addition, as several (n=4)

of the participants mentioned, not all products have the same level of risk.

"The risk level that's acceptable for product A isn't the same as it is for product B. So you don't treat everything the same, you can't just state across the board here's my watermark for risk for the organization." -p7

These risks must then be weighed against available resources, which are limited. As participants mentioned, security costs money and organizations must strike a balance between delivering a product and ensuring that the product is free of security issues. And application security is just one aspect of the security of an organization.

"If you look at percentage, for security in general, and particularly between what I would call traditional infrastructure and network security, versus application security. Those numbers are horrifically in favor of the more traditional network and operational security." -p26

These limitations mean that rarely does an organization do all possible activities for vulnerability detection and remediation.

"The reality is that most of the dev efforts will implement one or two of the things I've just described but not all of them. In each one of those areas, you get a different lens into the application. You get a different opportunity to identify and re-mediate security concerns. Without implementing them all, there's a chance, and this is what we see all the time, there's a chance a security vulnerability is being missed and making it through to the final product." -p22

One strategy participants attempted was to reduce their effort through automating tools and processes. More automation was also commonly stated (n=14) as a potential improvement to application security. Yet, participants also commented that many

of the existing processes and tools do not lend themselves well to automation. In addition, auditors were also aware of the importance of human judgement in security decisions.

"A human just has kind of this intuition about where to look and how to basically defeat the system." -p30

3.4.3.2 Limited Security Expertise

A key resource limitation is the skilled and knowledgeable security experts themselves. For example, P10 estimated that there are roughly 50 open positions to each qualified person. Another stated:

"There's just not enough people that know security. So we can't even find the resources that we would need to actually get the job done." -p21

This results in very few application security experts. As P15 states, *"If you have ten employees you actually have a fairly large security team."* This also means that the ratio of application security experts to developers is also very small. Of the participants who reported concrete numbers, they all reported a single auditor for one to two hundred developers. Rather than rely solely on application security experts, auditors emphasized the importance of training developers in security.

However, only two of my participants mentioned that secure programming training was mandatory in their organizations, and as a result, according to P15 *"The percentage of people that do the training is very, very small."* Similarly, several lamented the lack of security education. *"We have to train developers at a university level. Computer science graduates, to this day, don't get secure code in training."* -p15

3.4.3.3 Fitting in Security

As described earlier, application security experts traditionally are organized as part of a separate software security group, overseeing almost all of the detection of security vulnerabilities. This also reflects how an application's security is viewed within development teams, as a non-functional requirement that is not built in from the beginning. As P10 states:

"I would say that the big challenge there is that security and development have traditionally been disjointed and they have been separate teams. Security is the watchdogs, development does the work and all security has ever done is scan stuff. So write code, ask questions later. And we have to change that" -p10

One reason for this lack of security integration may be that security was not an important feature for small or immature applications. Yet, over time, security has grown more important. Thus, organizations may not start out with strong security processes, which means they later need to determine how to add security on top of already complex software development processes. Participants acknowledged this issue:

"Where are we going to plug in threat modeling? Where are we going to plug in static scanning? Where are we going to plug in dynamic scanning? When are we doing pen-testing? How are we doing developer training? What's the process to fix bugs? How do we prioritize those bugs? Just getting that very well integrated into what was already a well-oiled machine is doable, but it's challenging." -p20

3.4.3.4 Summary

With security as just one of many competing requirements within software development, organizing and utilizing the very limited security resources and personnel has a major impact on the resulting security of the end product. Auditors themselves are limited resources, who must understand the risks of their software and balance those against available time and efforts to be effective.

3.4.4 Technical Challenges and Needs

Participants in my sample described many challenges and needs within the tools that they used to find vulnerabilities. These issues led to a significant amount of work to run these tools and utilize the results.

3.4.4.1 Scalability and Performance

As with other studies of experts [66, 67], many (n=13) participants mentioned scalability as a key technical challenge. The specific issues for application security tools include:

- Locating, acquiring, configuring, and using tools that can adequately support all relevant languages and frameworks of an organization;
- Time and effort required to write rules for static and dynamic analyzers;
- Managing large numbers of false positives from scanning large applications or many applications; and
- Modern static analysis and dynamic analysis tools are not sufficiently robust

with large applications and have slow performance, yet it is very labor intensive to configure them to scan only pieces of an application.

Participants were most frustrated with the last issue above - that their tools may just not work at all or work too slowly to be useful. They stated that modern static and dynamic analyzers tend to crash when scanning large amounts of code or crawling large applications.

"I think all the current tools lack performance. Lack the ability to go fast. The reason why a lot of the testing tools don't fit into a traditional DevOps model is because they're really slow." -p28

"I've had a scan running for at least two or three days, I just had to kill it." -p12

3.4.4.2 Legacy and Complex Code

Participants frequently mentioned both legacy and complex code as key security challenges. In order to properly identify security vulnerabilities in code, it is often necessary for security auditors to obtain a limited understanding of a given application's functionality. If the person who wrote the code is no longer available, the security auditor must locate a suitable proxy. Unfortunately, they can be both difficult to locate and their knowledge may be incomplete. This makes the evaluation of false positives and the remediation of vulnerabilities much more difficult.

3.4.4.3 Third Party Libraries

Third party code was also very challenging (n=13) for two reasons. First, participants stated that vulnerabilities are very common in frameworks and libraries, even well-known established ones. Yet, determining the problem as well as actually

fixing it may be beyond their abilities, requiring a patch from the library vendor. Additionally, third party libraries can break static analysis and greatly contribute to false positives. This is because entry into a 3rd party library or component, when the source code is not available, constitutes a "boundary" or a "breakage." This breaks the analyses path from the source to the sink and destroys the taint signature that is used by static analysis algorithms.

"As soon as you trace that into a third party component, its kind of a dead end there." -p12

Thus, tools may report many vulnerabilities simply because the code relies on a third party library.

3.4.4.4 Automation and Smarter Tools

Many participants expressed a desire for more automation or intelligence in security tools. For example, seven participants wanted tools which are capable of assisting in triaging and remediating vulnerabilities. While participants had varied wishlists, their desires often related to tools requiring less manual configuration to run, and enabling easier data sharing and analysis across tools. In other words, auditors wanted tools to relieve more of their mundane and tedious manual labor. They acknowledged that creating smarter or more automated tools was challenging, but they were hopeful that with time, tools would improve.

3.4.5 Limitations

Security experts can be difficult to recruit for user studies, which leads to the use of convenience sampling techniques that come with limitations. My participants are

not necessarily representative of all application security experts. Given my recruiting methods, I likely have more experts from large organizations, with larger security groups and established security processes. Still, the results demonstrate the problems even mature organizations encounter surrounding security. I also have primarily U.S. based organizations, and processes and security cultures may differ in other countries. My sample is also all male, and while the security field is primarily male, I was disappointed to not have a more gender-representative sample.

3.5 Implications

As other researchers have already described, developers who write code are often not very involved in detecting security vulnerabilities in that code [42, 54, 73]. Additionally, given my auditors' complaints regarding the lack of security knowledge and training by developers, developers are likely not taking the necessary steps to prevent many of those vulnerabilities in the first place, even though they are ultimately tasked with fixing them. Placing sole responsibility for software security in the hands of very few application security experts has a profound impact, resulting in known vulnerabilities slipping into production, long delays, added expense to fix vulnerabilities, and insufficient analysis to detect potentially serious vulnerabilities. This study further supports the need to increase developer attention to preventing vulnerabilities in their code. Many of my participants directly advocated for greater developer involvement in the security process and the benefits of doing so:

"And that's the biggest challenge, getting security moving all the way to the left and addressing every single phase of the development life cycle....Then I guess 80 percent

of the kind of low hanging fruit vulnerabilities like SQL Injection that we find out in the wild would be addressed." -p17

Preventing more vulnerabilities from being committed in the first place would also free up auditors to focus on more complex or difficult to detect vulnerabilities, to more deeply monitor and analyze the risk profiles of various projects, and to provide more in-depth assistance to development teams. Involving other stakeholders may also relieve auditors of the burden of motivating and coordinating vulnerability remediation.

Participants also suggested ways to increase developer involvement in vulnerability prevention. For example:

- Several auditors (n=3) suggested that the Agile process be modified to treat security as a functional requirement instead of "technical debt." This may shift the importance of security in the eyes of developers since it would be treated at the same level as other functionality.

- Other auditors (n=6) discussed increasing the use of continuous integration, where security bugs are entered automatically into defect tracking systems with other general bugs. Security bugs are more likely to be on a level playing field with other bugs if they are continuously found and potentially prevent code from being committed. However, as mentioned earlier, performance and scalability issues make it challenging to run scans quickly.

- Many of my participants (n=19) also called for more security training for developers to provide them with a greater awareness of security and the knowledge

necessary to produce fewer security bugs.

Yet, as prior work has clearly demonstrated, the burden of security work needs to be lowered for developers and auditors alike [19, 20, 65, 70]. In addition, organizational processes need to support and reward that work rather than delegate it to one small group of security experts [54]. I believe my findings suggest three particular areas that need increased attention and future research to achieve these goals.

Security training. Auditors commented that improved training is needed for developers, such as training that is more engaging (one participant suggested gamification for example), and concrete, involving code the developer is actually working on. Auditors also felt it should be targeted towards the issues developers are actually encountering, and delivered in-situ, as developers encounter those issues in their code. Commercial tools do not currently do this. There were several concrete suggestions for training support, such as:

- (p4, p20, p31) Tools which run in the developers' IDE, that provide feedback directly to the developer about why something is wrong, and help generate code for developers;
- (p4) Tools that report problems to supervisors or others to determine the kinds of security training needed;
- (p22) Tools for collaboration around security vulnerabilities, providing contextualized communication and help beyond just the location of a bug.

Note that the first bullet is addressed by interactive static analysis [72], and I further cuscuss collaboration through code review in later chapters.

Increasing automation As shown both by existing research [42, 57] and my results, analysis tools produce many false positives, and is one reason that developers are discouraged from using such tools. My application security experts also indicated that false positives were a major challenge for them as well. Experts are required to manually create custom rules and configuration for every application to reduce these false positives, which is time consuming.

"I would love to see a static analysis tool where I could get it out of the box, run it on my application and it just worked. No tuning required...But right now I had never found a tool that allows me to just go from zero to a good or a reasonable scan result. Just doesn't exist." -P28.

False positives can be reduced with more accurate analysis algorithms, and my participants pointed out particular pain points around third-party libraries for example. However, providing more automated support for the tuning process to help users customize security scanners could also be helpful and greatly reduce the burden of using static and dynamic analysis tools.

Additionally, utilizing continuous integration processes requires detection tools to run automatically. Yet, as my auditors mentioned, many tools do not yet have sufficient performance to be run in this way, or are simply too difficult to configure to do so. Tool builders need to investigate more incremental and layered mechanisms for scanning and reporting detected vulnerabilities in order to provide faster and more localized feedback. This would also support the more interactive analysis tools proposed by myself and others to provide developers with real-time and interactive security feedback [72, 52].

Risk assessment. Assessing and acting upon security risk is a key role of auditors. Yet, this process was primarily manual. Auditors considered risk during vulnerability triage, in determining audit frequency, and in choosing which parts of the code to spend more time on. There are currently few tools that provide auditors and other stakeholders with assessments of risk to an application to support decision making. Similarly, few tools help users prioritize issues found, or are configurable based upon risk assessments. As one participant commented:

"I don't have a good tool that lets me say, 'Hey, what's, you know, what's my risk posture right now? ... am I doing as well as I should be?"' -p20

Thus, tools need to reflect such risk assessments. For example, when should static analysis results fail a build and how should it be configured to do that? If builds are broken too frequently, organizations will incur a severe cost. On the other hand, if builds are not broken frequently enough, applications will be developed with serious security issues. How much effort should developers put toward fixing different vulnerabilities? Which vulnerabilities are the most serious and which are not? If developers perform more vulnerability prevention and detection themselves, their tools will also need to reflect such risk assessments to help prioritize and direct their limited amount of time towards addressing the most important security issues.

3.5.1 Security Code Review Implications

Unfortunately, my work has also shown that the interactive static analysis process still needs interaction between software developers and security experts around security issues. These results have shown that security code reviews are not merely a

hypothetical phenomena. Some current organizations do perform security code reviews. When they are performed, they are performed separately from traditional code reviews. They are always performed as part of an audit. Participants explained that this is a separate process from normal development which is initiated repeatedly at set intervals of time for different projects. Higher risk projects with a lot of change may have more frequent audits than lower risk projects with less change. Security code reviews may be conducted as a part of these audits. They are currently very similar to traditional Fagan inspections. Surprisingly, the focus of these security code reviews is not to identify vulnerabilities, but rather, to identify risk and code smells. Once risk for specific pieces of code is determined, remediation decisions are made later. While the process may involve a developer, it is unlikely to involve each developer who contributed to the project. Unfortunately, this current process is very "heavy" for the actual detection and remediation of security vulnerabilities. Therefore, I am not studying interactions between developers and security experts in this process. Since the primary purpose of current security code reviews is to assess risk, it follows that sufficient resources are not always available to address all possible security issues. Therefore, some sort of risk assessment, or triaging of vulnerabilities is needed to determine which issues are most critical. It is logical that this process be done in a lightweight security code review tool.

It is important to point out that the new security code review process I propose later is incremental. This may help to overcome some traditional problems in non incremental security processes, such as tediousness for large projects, scalability challenges, and remediation instructions which are "out of sync" with the code devel-

opment process. However, it may introduce some new challenges. First, it adds an additional stage in the development process, which may be costly for organizations which implement it. Second, the high availability of security experts through the tool may cause them to be asked a higher volume of questions than they would otherwise be asked. This may place a strain on security expert resources.

This study also revealed that security auditors often have to take great precautions to prevent the developer from taking a defensive stance when auditing code. This may be because the developer feels that defects in their code represent mistakes they have made in their work process. Rather than view the auditor as a resource to improve their work, they may view the auditor as an obstacle. Lightweight security code review tools have the potential to combat this issue. The tool would initiate the static analysis and interaction between the security expert and developer rather than the security auditors themselves. Also, interaction between the developers and security auditors can be expected to revolve around the developer asking the security auditor questions about confusing warnings. This may cause the security auditor to be viewed more as a resource to help "pass" the system and less of an obstacle. However, since the security auditor will be evaluating each remediation, the security auditor will be acting in a judging role. This may have other, currently unperceived consequences.

These results also suggest that feeding static analysis results and warnings into security code review is much more novel than expected since static analysis results are entirely separate from the security code review process. Tools which incorporate these results should leverage these warnings for collaboration around risk rather than

the ability to remediate the issue from the tool.

3.6 Conclusion

To my knowledge, this study is the first user study to specifically focus on application security experts. My results provide further evidence that application security work is primarily performed by these experts, separately from software development teams. Separating security from development adds communication overhead and barriers. And finding and fixing security vulnerabilities so late in the software lifecycle can result in costly delays and expense to applications, and an increased likelihood that applications are not adequately secure. In particular, my results highlight the importance of triage to reflect risk assessments, the challenges of security communication that mostly occurs within bug tracking systems, the experts' role of motivating developers to fix problems, the importance of security champions, and the desires for less configuration and more automation. These results further demonstrate that improving application security will involve a combination of organizational processes to incentivize and support developers in focusing on security issues earlier, developer training to give them more concrete and actionable knowledge and motivation, and tools for both developers and experts that reduce the manual burdens of configuration and analysis. I hope these results can inform a variety of process and tool improvements to reduce the costs of vulnerability detection and remediation, and allow application security experts to focus on deeper and more complex security issues and processes.

CHAPTER 4: INTERACTIVE ANNOTATION FOR APPLICATION SPECIFIC VULNERABILITY DETECTION

1 Introduction

Although interactive static analysis has been shown to be effective for the detection and mitigation of vulnerabilities such as cross site scripting, it is not feasible for more complex issues for which code cannot be generated. For these types of issues, a different solution is needed. Unfortunately, security code review is unlikely to effective on its own for these types of problems, since general code reviews have been shown to be ineffective for complex vulnerabilities.

To combat these kinds of issues, I expanded my interactive static analysis approach discussed previously to include a novel technique I dubbed **interactive annotation**, where developers are prompted to indicate security-critical components in the code. This was done both to remind them to perform security actions and to document application-specific security information. This in turn allows a static analysis tool to reason more accurately about the code and detect more complex vulnerabilities and provide documentation of security-related decisions. I then created a prototype of this approach within ASIDE and examined access control decisions.

```
<?php
// Added security
if(!isset($_SESSION['username'])&& !isset($_SESSION['passwd'])) {
echo"<metahttp-equiv=\"Refresh\" content=\"2;url=".self_url()."/login.php\" />";
}
sql_query("DELETE FROM ".POST_TBL." WHERE post_id='$post_id'"); //SSO
?>
```

Figure 2: Access Control Vulnerability Example

I examined the performance of my tool and approach on vulnerability detection for 6 open source projects and found that I detected more vulnerabilities than existing automated approaches, with significantly less work for users than commercial tools require [75]. This approach is important to my dissertation goals because application specific issues may be very difficult to detect and mitigate. Interactive annotation, when combined with security code review, may be able to effectively detect and mitigate application specific issues without the need to write custom static analysis rules. My approach though depends on developers correctly performing the annotations, and understanding the resulting vulnerability warnings. Thus, I now examine this interaction.

I report on a user study with advanced students using ASIDE on two different open source projects. My research goals are:

- Evaluate the usability of my interface, and developers' behaviors in annotating code
- Examine how developers identify and understand security logic within code
- Examine how developers interpret and understand vulnerability warnings that result from their annotations

My results will help to improve the interface of my tool, as well as provide a deeper understanding of how such tools can communicate with developers regarding security vulnerabilities generally, and access control more specifically. They will be useful for my dissertation goals since they will provide guidance on how the interactive annotation and code review interfaces should be developed to assist in the detection and

mitigation of application specific vulnerabilities. Future commercial implementations of interactive static analysis and interactive annotation with greater usability will ultimately result in fewer security vulnerabilities. This study was published in the Symposium on Visual Languages and Human-Centric Computing in 2015 [61].

4.1 ASIDE for Interactive Annotation

In this section I describe my prototype interactive static analysis tool, named ASIDE. The current prototype focuses on access control vulnerabilities. It takes as input: (a) Abstract Syntax Trees (ASTs) of the application generated by Eclipse JDT or PDT, and (b) a set of security sensitive operations such as operations that read or modify sensitive databases. ASIDE then uses these to generate requests within the IDE for the developer to annotate security related decisions, see Figure 3. This request is indicated by a yellow highlight of the sensitive code, and a yellow question mark alongside the code. I chose a question mark to try to convey that the tool is requesting information, not indicating anything wrong with the code. Thus, in the example in Figure 3, the developer was asked to indicate where security checks are located for the function *updateAccount* that accesses sensitive database tables. Clicking on the icon or code provides a menu where the developer can access ASIDE explanations, as well as choose to enter annotation mode.

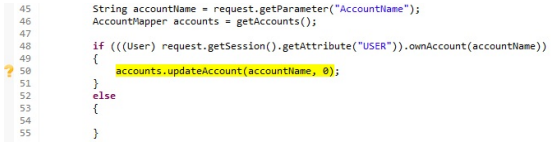

Figure 3: Annotation request in Gold Rush, shown with a yellow highlight and question mark icon.

In annotation mode, the developer would then highlight the statements performing security logic for the sensitive operation, highlighted in green in Figure 4. In doing so, the developer is reminded to add such checks, if they are not already implemented. ASIDE indicates the annotation with a green highlight and a small green diamond next to the code. The sensitive operation also turns green when an annotation is added, with the icon changing to a green check mark. Even when there are no vulnerabilities detected and no further interaction required by ASIDE, I choose to keep the green icons visible to allow developers to further modify annotations and interact with ASIDE as they continue development. Once the annotation is made, ASIDE leaves annotation mode and the developer returns to the task of coding.

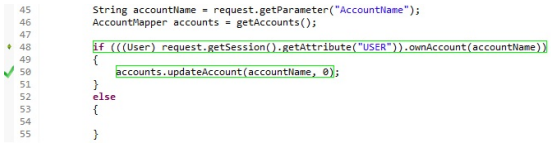

Figure 4: An annotation and annotation request that have been completed in Gold Rush. The completed request is shown with a green highlight and green checkmark icon.

With the annotations provided by the developer, static analysis is used to detect vulnerabilities. This currently utilizes a path coverage algorithm, which analyzes all paths from the web entry point of the code to the security sensitive operation, ensuring that all paths go through the annotated security checks. Paths without all of the checks are flagged as potential vulnerabilities, and presented as warnings to the developer. Alternatively, repeated sensitive operations are detected, such as other calls to the function *updateAccount*, and differing security logic is flagged as a potential vulnerability. In other words, my tool utilizes a standard static analysis technique

Figure 5: ASIDE example with sample code in Gold Rush. A warning is displayed on line 64

that assumes that the same operations should be protected by the same security logic. If the annotated security logic differs between multiple instances of the same operation, all of those instances are flagged as containing a potential vulnerability. Due to space constraints, I cannot provide all the details of how my path coverage and vulnerability detection algorithm work, but it is fully explained in my recent publication [75].

A vulnerability warning would be presented as shown in Figure 5, with the red highlighted code and red flag icon. Clicking on the warning would again provide a menu of options, brief explanations of the warning (e.g. the security logic and the other operations that are related to this warning), and access to more detailed help regarding the vulnerability. Developers can fix the warning by changing the code, and re-annotating the correct security checks. Alternatively, they can dismiss the warning if they decide there is not a vulnerability.

4.2 Methodology

I designed a user study to examine user interaction with my interactive annotation prototype. First I wanted to assess how intuitive my interface is for developers. Second, I wished to examine how developers understand and identify security logic. Third, I wanted to assess the effectiveness of my tool in assisting developers with understanding potential security vulnerabilities regarding application specific vulner-

abilities.

I recruited participants from advanced programming classes offered at the University of North Carolina at Charlotte and North Carolina State University. I chose to use students from these classes since it is often difficult to obtain a large sample of professional developers for academic research. Additionally, since most of these students are either almost finished with an undergraduate degree or pursuing a graduate degree, I feel that data obtained from them may be similar to that obtained from entry level professional developers.

Participants from the University of North Carolina at Charlotte interacted with ASIDE running on a project called Gold Rush, an internally developed Java-based banking application (99 files) to teach web application security. Participants had just completed a class assignment involving this code, and were thus familiar with it. Participants from NC State interacted with ASIDE running on a project called iTrust, an open source medical information system (1,860 files). Participants had previously fixed real bugs and added additional features to iTrust over the course of a semester in a senior-level undergraduate software engineering course at NC State. I intentionally chose code which would be somewhat familiar to students to simulate the situation of a developer working within their own projects.

Students were recruited from the selected courses, and were offered extra credit for participating (with alternative extra credit for those who chose not to participate). Sessions were conducted in the usability labs of both universities, and took about an hour. Participants were asked to sign a consent form and fill out a demographics survey. Specifically, I collected the participants' race, gender, years of programming

experience, whether or not they had been employed as a developer, class rank, major, and the quantity of both programming and security courses taken.

Once I had collected demographic data, participants were given a brief introduction to ASIDE. They were then shown and allowed to interact with a trainer example, which consisted of one annotation request. The purpose of the example was to answer any questions they had before beginning and to account for Eclipse interface quirks which could not be removed or fixed in my current prototype.

For each project, I created a set of scenarios for participants to examine. The code of both projects was modified slightly so that I could induce several application specific vulnerabilities. To simplify the tasks and time involved, I ensured that each scenario only involved code in one file, and I only showed requests and vulnerabilities for my scenarios (and suppressed any others ASIDE would normally identify).

Participants were shown the files with the requests and warnings, in the same order. For every annotation request participants were asked to annotate the security checks, if they existed. When participants made the annotation, they were asked why they chose a particular line or lines of code as the security logic. They were also asked about the meaning of warnings, and how they would fix any issues they identified. Once participants completed this task, they were asked several questions about their perceptions and use of ASIDE. I recorded the audio and screen activity during the session. Data was analyzed by transcribing the audio and creating notes based on the screen recordings. I performed open coding on the transcriptions and notes, to determine performance and look for common patterns and interesting responses.

4.3 Results

I had 28 participants – 13 (9 male, 4 female) were from UNC Charlotte and examined the Goldrush system, and 15 (14 male, 1 female) were from NC State and viewed iTrust. 21 were undergraduate students in computing, while the rest were Master's students. Most participants had taken 1 to 3 security courses and around 5 courses that included programming as a major component. Six participants reported that they had worked as a professional developer. One of the developers was a lead developer and 1 participant was a database administrator.

4.3.1 Interactive Annotation

My 28 participants encountered a total of 181 annotation requests and warnings. Participants who interacted with Gold Rush each encountered a total of 5 requests and 2 warnings while participants who interacted with iTrust each encountered a total of 4 requests and 2 warnings. Participants had no issues highlighting code to indicate an annotation. They were able to highlight code to create an annotation in 180 (99.45%) of those cases.

Users felt my interface was very intuitive and that annotations were very easy to make. When asked how easy or hard the interface was to use, 26 out of 28 participants (92.8%) gave responses indicating that the interface was easy to use, such as these participants:

"I thought it was very easy." -p6G

"I guess it's pretty easy to use and put your annotations in Eclipse." -p14T

One participant gave a neutral response due to how my menus are implemented in

Eclipse. Another participant said that the interface was difficult to use because he felt that more contextualized information was necessary to reliably make annotations.

The majority of participants were able to successfully identify security checks. By inspecting the screen recordings, I determined that participants annotated the right security logic or correctly identified reasons why an annotation was not necessary in 141 of the 181 (77.9%) annotation requests and warnings. I refer to this success rate as "accuracy." Participants annotated the wrong security checks or incorrectly identified 40 of the 181 annotation requests and warnings (22.1%).

In a previous pilot study I observed that users with more programming and security experience had an easier time using ASIDE. Thus, I wanted to identify any patterns in this larger user study. To that end, I defined participants who reported 4 or more years of programming experience as participants with "More programming experience." Consequently, I defined participants who reported fewer than 4 years of programming experience as participants with "Less programming experience." Additionally, I defined participants who had taken 2 or more security courses as participants with "More security courses." Participants who had taken fewer than 2 security courses were defined as participants with "Fewer security courses."

One might expect that participants with more security courses would be more accurate than those with fewer security courses. However, I observed no noticeable differences in the accuracy between participants with more security courses and fewer security courses. However, I should note that the quantity of security courses taken may not be a particularly good measure of a person's experience with secure programming.

Table 2: Participant accuracy by programming experience

Participant Type	Experience Level	Accuracy
Goldrush Participants	More Programming Experience	94.28%
	Less Programming Experience	50.00%
iTrust Participants	More Programming Experience	85.42%
	Less Programming Experience	90.48%
Overall	More Programming Experience	88.83%
	Less Programming Experience	68.89%

One would also expect that participants with more programming experience would be more accurate than those with less programming experience. As shown in Table 2 the Goldrush study results seem to suggest the expected pattern, in that participants with more experiences in programming were more accurate. However, the iTrust data does not support this pattern. Instead, I are encouraged by the result that participants with less programming experience seem to be able to use ASIDE effectively as well (68.89% accuracy). However, my qualitative analysis of the interviews does suggest that people with more programming experience were more confident in their responses.

In addition to programming experience, identifying security logic is also directly related to participants' familiarity of the code. In both cases, the participants did not write the security logic. Instead, they were given the code base as part of their class assignments. Therefore, some of them may have studied more and knew more about how the code is supposed to work due to their course activities.

A few struggles with annotation also provide valuable input. Two participants annotated the sensitive operation instead of the security logic. One of these two realized this halfway through the study when asked about the annotations he was

making. He explained that he was annotating the request simply to turn the yellow icon to green. After he realized the purpose of the annotations, he annotated the correct security logic for all future annotations. The other participant also said she was turning the yellow icon to green when asked why she was annotating the request. However, she did not realize this was incorrect and never corrected the behavior. Unlike the other participant, she stated she was "Not really into development"-p4G. There were a variety of other incorrect annotations, such as variable declarations, the function request.getParameter() since it is where sensitive data enters the program, or the "try" of an encapsulating try block. This likely occurred because a security error would appear to be caught by the try block and the catch block would appear to prevent the code from being vulnerable.

In order to detect vulnerabilities, my tool requires that users highlight a Boolean conditional statement or a function call that throws an exception. Thus, users would only need to highlight those small snippets of code. However, participants generally highlighted either entire lines or small blocks of code, such as the entire if code block containing the conditional statement. No participant made the minimal annotation they could have performed. This implies that my tool will either need to help users identify and highlight just the smaller snippet of security logic, or will need to parse that logic out of the larger block of code that users identify.

Interestingly, participants also felt compelled to complete an annotation even when they should not have done so. This could occur because of a false request (request did not require any security logic) or when they felt that the security logic was wrong or missing. About half of my participants (n=16) would still highlight some code

despite mentioning aloud that they thought that the annotation request was a false request, or that what they were annotating was not a sufficient or correct security logic.

Three participants expressed confusion about whether or not ASIDE could verify their annotation. My prototype does not currently verify whether the highlighted code could be a valid security logic statement (e.g. code that actually contains a Boolean conditional statement). The green color and green check that appeared after an annotation was complete seemed to indicate a false sense that the annotation was being checked by the system and was found to be correct:

"I'm gonna go ahead and annotate it, and we'll see if that makes it happy." -p2T

One participant immediately asked whether or not the system was checking the annotations after the initial training scenario. Other participants assumed that the check was correct, and only later realized that the system was not checking their annotations when I asked them if they thought their annotations were being verified. I do intend to provide such verification, and this result demonstrates that my icons lead users to expect that to occur. Thus, if I do or can not provide such verification, I may need to revisit my icon choices.

Interestingly, several participants saw the requests for annotation as requests for "security checks" in a general sense and would search for validation code. When asked why they chose to annotate certain pieces of code as the security logic for the annotation request, they would talk about how the code might be vulnerable to cross site scripting attacks or SQL injection and how the annotated code fixed that problem.

"I think maybe cross site scripting or Cross Site request forgery might do it but Im not sure exactly what the best way of describing that would be. But I think that it could be exploited by using something like cross site scripting." -p6G

"I'm thinking SQL stuff...I was going to say, I don't see any parametrized statements or anything like that." -p11G

Yet those issues could be corrected by validating untrusted input, rather than access control logic. Surprisingly, many of these participants actually still annotated the correct security logic. Thus, these participants had some generic security knowledge, and while they were confused as to why, they did equate the security logic they found in the code with somehow providing security.

4.3.2 Interpretation of Warnings

From examining the interview transcripts, twenty-seven of twenty-eight participants (96.43%) understood that red warnings were indicating a possible vulnerability. This is a very positive result, since it means that the tool was effectively able to communicate this information to participants. Participants with more security courses and those with more programming experience could often describe how to fix vulnerabilities. Thirteen out of seventeen (76.47%) participants in these categories provided at least one solution for fixing an identified vulnerability. However, many of the participants with fewer security courses or less programming experience did not seem to understand how to fix the vulnerabilities. This is expected, since this knowledge is not taught in many programming or security courses, and these participants lacked the necessary training.

For example, five participants did not give confident answers as to whether or not a sensitive operation with a warning was actually vulnerable. Four out of five (80%) of these participants fell into the category of "less programming experience". I saw an interesting pattern, where participants who expressed confusion discussed "how" vulnerable a piece of vulnerable code was." Often, they would use terms such as "probably vulnerable", or "probably not vulnerable." They would also frequently say that a code snippet was "more vulnerable than the last one" or "extremely" vulnerable instead of simply a binary of "vulnerable" or "not vulnerable." The following responses illustrate this mental model of vulnerabilities:

"I think there is a degree of vulnerability to it." -p6G

"It was saying that it was more vulnerable than before." -p2T

Some of these comments also seemed influenced by my choice in icons. Some participants also seemed to incorrectly interpret the color yellow in annotation requests as "maybe vulnerable" or "not as bad" as a warning instead of just a request for information:

"Wherever there is a question mark, its basically thats where it feels that theres a security vulnerability correct?" -p6T

However, the red in warnings was generally perceived as more severe than yellow:

"Okay so, the yellow means Hey, set this attribute if you can. The red means You need to set this attribute at this annotation because this is bad."' -p13T

Despite this confusion for some participants, I believe my approach is successful because one of the main goals of the tool is to help people identify vulnerabilities. Because participants were successful at annotating the correct security logic 77% of the

time, vulnerability detection and the resulting warnings will be reasonably accurate. Moreover, 96% of participants understood that the warnings conveyed a possible vulnerability, which means the tool would be effective in bringing vulnerabilities to the attention of developers.

However, the tool was less successful at helping developers understand why a warning occurred, and how to mitigate it. As mentioned previously, the tool determines a potential vulnerability based on annotations for the same sensitive operations across the application. Thus, determining whether a vulnerability exists, and how to fix it, should involve examining the annotations for those same sensitive operations and looking for differences between them. However, participants were seldom aware that warnings were a direct result of their annotations. This information was conveyed in the contextual help accessed when the ASIDE icons are clicked. For a warning, ASIDE provided details of the related sensitive operations and their security logic annotations. Yet, participants did not pay much attention to my messages and explanations; only four participants (14.28%) utilized the contextual help. Instead, my participants examined only part of the code to determine the validity and fixes for warnings. This could partly be attributable to the Eclipse environment. Eclipse has many windows, warnings, and other messages, such that my information may be lost in a sea of irrelevant information. This may cause information overload in participants and cause them to glaze over the help I provide in my tool. Moreover, my sample did not have an incentive structure in place to encourage the correction of these vulnerabilities. Ideally, my tool should be able to help users fix vulnerabilities, not just identify them. Future research will strive towards this goal.

4.3.3 User Perceptions and Comments

General user impressions on ASIDE were overwhelmingly positive. When each of the twenty-eight participants were asked what they thought about my tool as a whole, only one participant responded negatively. Several participants mentioned that they felt it would be a great aid to help catch vulnerabilities that they may have missed while coding their projects, implying that the process of annotation itself would be very effective at helping them catch missing security checks. One participant mentioned that the warnings would be very useful in quickly detecting vulnerabilities. When asked, no participants felt that the process was tedious. Multiple participants expressed concern that the task of interactive annotation could become tedious on large projects, but they also felt that more false positives and higher detection rates were preferred over fewer false positives and lower detection rates. Consequently, they stated that they would be willing to tolerate any tediousness on large projects, favouring the improved awareness and detection over annoyance.

"That made me stop and think. When I'm in a developer role, I'm always thinking about the functionality because that's what I get paid to deliver....It forced me to stop and think about security even if just for a few minutes. It was good." -p3G"

"Its easy for someone like me who hasnt done much security...It links up graphically- or visually where the problem is and then when you highlight a potential area thats connected to that problem." -p6T

Additionally, all participants except one stated that they would use ASIDE in the real world. The participant who stated that he would not use ASIDE in the real

world claimed that he would not use it because he did not "trust" it to catch all of the vulnerabilities. Inspection of the video recordings and transcripts showed that he identified sensitive operations which were not flagged by ASIDE, and he felt the tool was not sensitive enough. These other sensitive operations were intentionally skipped by ASIDE in this user study to control the quantity of annotation requests that study participants were asked to annotate. This is a favorable result since it echoes other participants favoring of potential detection over annoyance.

Participants offered many suggestions for future work while being interviewed, particularly relevant for this dissertation. Six participants explicitly or implicitly recommended that ASIDE be further developed into a code review tool. Participants who realized that the system did not check the validity of their annotations often desired some sort of review, but knew it would be difficult to have an automated system conduct this process. Consequently, they desired a human, perhaps a security expert, co worker, or manager, to be able to view the annotations in a code repository and provide feedback on those annotations. One participant even suggested that it would be nice to be able to mark confusing annotation requests or warnings with a note for another developer to annotate or address later.

"I like to be able to run it, I'm thinking as if I was a tech lead or a project manager, I could easily require that it be ran before submitting for code review and have the developer do all the highlighting and stuff and then it's a matter of whether I agree or not." -p3g

"Well, I imagine you could connect the information, like, to a repository so everybody would be able to look at the same annotations. If that's the case then that would

be real useful for everyone." -p2T

These comments provide further evidence of the usefulness of security code review, fed with information from tools such as ASIDE.

Other suggestions were smaller in scope, but still very practical. For example, one participant desired the ability to create multiple annotations on the same line of code, and recommended that a double diamond symbol be used in place of the single diamond symbol to indicate multiple annotations on a single line of code. Two participants suggested that it would be nice to be able to annotate security logic while coding, before the sensitive operation is written and associate the checks to the requests later. Six participants explicitly mentioned that they would like a hotkey for performing annotations, as an alternative method to clicking on a menu. All of these suggestions will be critical to scaling my tool from a basic prototype to something functional and usable on real world projects.

4.4 Discussion

Not surprisingly, graphically highlighting code in order to make an annotation was an easy task. This should ease adoption of any mechanism requiring annotations. Other annotation solutions could benefit from my results. Additionally, many improvements to the existing interface can be made based on my results [61].

Even those who did not fully understand the security vulnerabilities did a reasonable job of responding to requests and annotating the code. They also did a reasonable job of noticing when security logic was missing. This is a positive and encouraging result, as it implies that many developers will be able to accomplish this task, and

validates that my approach could be effective for a range of developers. And if annotations could be reviewed by a human with security knowledge, developers could increase their understanding of these types of vulnerabilities based on feedback from the human. Thus, code review may further strengthen the process.

However, there was significantly less understanding of how the annotations related to vulnerabilities. Moreover, very few participants were able to reason about and trace the source of a vulnerability using the tool. Those who had more programming experience and had a good understanding of the code could look at the code to do this. This meant that the tool could potentially communicate this information to these types of users if I can improve interaction with the messages I provide within ASIDE. However, ASIDE was not as helpful for users who had less programming experience and did not have a good understanding of the code.

A possible improvement could find a way to communicate this distinction clearly. In addition, my warnings are about potential vulnerabilities. A human must still determine whether or not a true vulnerability exists based on a detailed examination of the code. Thus, ideally my tool should help developers make this determination as much as they are able, and not simply imply that there is a problem.

Users also did not connect their previous annotations with subsequent warnings. A warning should imply that what is annotated may somehow be wrong. So, users should examine previous annotations for all instances of that operation when there is a warning. Tracing the cause of a vulnerability would involve comparing the security logic across those operations. Helping users to do that is a challenging task, and clearly where I need to focus on creative design solutions. The tool interface needs

to make that more prominent, and easier to perform. The explanations regarding the warning that were shown in the side dialogue of the menu were ignored. A modification to the interface that could visually show a link between the warning and relevant annotations within the code window itself may prove beneficial.

4.5 Conclusion

I believe that developers can be provided with tools to better enable them to detect and mitigate security vulnerabilities, enhancing the security of their applications. Thus I are investigating how to communicate and interact with developers regarding security vulnerabilities so that such tools are usable and effective. As I have demonstrated, developers could benefit from such tools with greater awareness of the security implications of their code and potential vulnerabilities. Even those with lesser programming and security experience were able to indicate security-related decisions in the code, thus providing valuable information to drive more complex analysis or for use in later code review. My participants were appreciative of the awareness of security that my tool provided. Providing annotations interactively through highlighting was intuitive, yet also requires more flexibility from the tool in allowing users to highlight larger chunks of code than are needed. My results provide valuable feedback into my tool design, and in particular highlight the challenge of helping developers trace and fix complex vulnerabilities. This study can serve as a baseline for additional examination of interactive annotation interfaces in security tools, and I hope to use my results to inform the design of future interfaces for interactive annotation. Specifically, I use these results in the development of a security code review and interactive

static analysis tool since they demonstrate the effectiveness of interactive annotation and also the need for verification and support. By combining this technique with security code review, developers and security experts will be able to easily collaborate together to more effectively detect and mitigate security vulnerabilities.

CHAPTER 5: DESIGN CONSIDERATIONS AND TOOL DESIGN

5.1 Introduction

As I discussed in Chapter 2, code review is very effective at detecting bugs in code and can also be used to detect security vulnerabilities in code [37, 23]. However, it has been shown to be ineffective on its own for detecting security vulnerabilities [47, 46, 29]. Researchers have called for security code review as a separate process, and in Chapter 3, participants in my study reported that this does occur [29]. Various attempts have been made to combine static analysis and standard code review for the purposes of finding defects in code [57, 26, 53]. Most have been successful at this task, but have not focused on security issues [57, 26, 53]. Previous research has shown that interactive static analysis can help mitigate the problems of static analysis and make it something developers are more willing to perform [76, 72, 71] and has also shown that it can help train the developer [76, 72, 71]. However, in the last chapter, I have shown that developers need more assurance that their solution is correct and do not always know how to resolve vulnerabilities [61]. Code review involving a security expert in a security code review could provide that assurance and catch incorrect solutions. Therefore, combining interactive static analysis with security code review may prove effective.

In this chapter, I first discuss key design considerations about the security code

review process. Moreover, I provide details of a tool design from the design considerations. I propose a new lightweight, tool assisted, security code review process. I finally discuss the security code review tool I built, an implementation of this tool design.

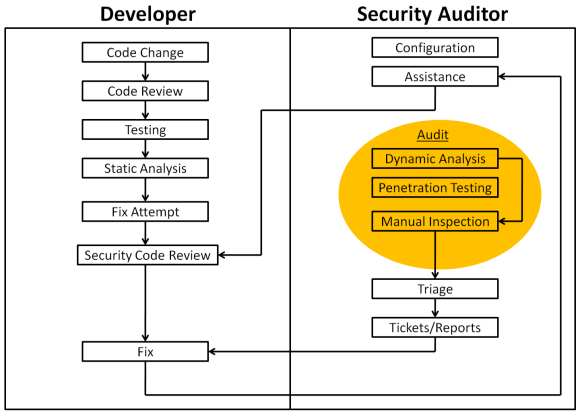

Figure 6: Security Auditor Workflow Model With Security Code Review

In Chapter 3, I discussed an aggregate security auditor workflow model based on the workflows described by my participants. If security code review fed by interactive static analysis is to be useful, it must be inserted into existing processes somehow. It seems most intuitive to place this process after functional testing, independent of audits.

In this new workflow model, developers would conduct interactive static analysis or standard static analysis, apply remediations, and feed the output into the security code review process. They would then communicate with security experts during security code reviews. Once this is done, they would attempt to fix issues based on received guidance. Audits would remain the same as before, with the removal of static analysis. Ideally audits would be conducted by security experts as an independent process. However, for organizations with limited budgets or application security

experts, audits could skip manual code inspection since the code would have already been inspected in stages. For organizations with extremely limited budgets, audits could be removed entirely. The most ideal process, with audits included, is illustrated in Figure 6.

5.2 Design Considerations

In this section, I discuss three categories of design considerations necessary to my security code review tool design. These design considerations include roles, communication and collaboration, and warnings. Moreover, I discuss why each one is important.

5.2.1 Roles

In performing security code review, three roles quickly become apparent. The first of these roles is that of the primary developer. The primary developer is considered to be the developer who wrote the code. This person requests a security code review and alters the code directly to fix the vulnerabilities. The second role is that of the reviewer or alternate developer. Alternate developers work on similar projects or code as the main developer and serve as peers. They raise issues or comment on issues raised by the IDE, as well as questions from the primary developer. Lastly, I propose that security code review involve at least one security expert to ensure the security effectiveness of the code or suggestions raised by the alternative developers. The security expert fills a similar role to the alternative developers, but is in a unique position to comment on the security effectiveness of vulnerability resolutions. Alternate developers, on the other hand, are likely to possess more knowledge of software

engineering best practices and the functionality of the underlying project code.

5.2.2 Communication and Collaboration

Security code review should involve much collaboration between the primary developer, alternate developers, and security experts. Primary developers have direct knowledge of the code being reviewed, as they have either written all of it or a large portion of it. However, they may have very little security knowledge to accurately address security warnings from static analysis and spot other security issues in code that the algorithms may miss. Security experts, on the other hand, know much about the security vulnerabilities, but are very unlikely to have much understanding of the current project code. Alternate developers are likely to have expertise somewhere between the two. Alternate developers are likely to have much knowledge of software engineering best practices and some knowledge of the project. They may also know alternative software engineering solutions to problems that the primary developer does not know. However, it is extremely unlikely that they will have the security knowledge of a true security expert. They may each provide different perspectives and different kinds of reviews and responses in this type of code review.

5.2.3 Warnings

When the primary developer interacts with a static analysis tool such as ASIDE prior to conducting a code review, they will be shown many vulnerability warnings. These include warnings for SQL injection, cross site scripting, CSRF, and access control vulnerabilities. Contextual help based on the actual underlying code will be provided when the warnings are selected. The primary developer will interact with

these warnings and attempt to resolve them. The primary developer will also perform interactive annotation for annotation requests. When the code review process occurs, the alternate developers and the security expert should both be able to observe the warnings and comment on the warnings.

5.3 Research Questions

5.3.1 Roles

- What are the activities and contributions of people in each of the three roles in security code review?
- How can a tool support these roles and activities?
- What are the perceptions of people in different roles of performing security code review?
- What type of training is required for developers to effectively perform security code review?

5.3.2 Communication and Collaboration

- How do participants in each role communicate and collaborate with each other regarding code review information?
- Which features in a tool are most effective for collaboration between people in three roles and security experts?
- How should feedback from other participants be displayed to users in each role?

5.3.3 Warnings

- How to effectively communicate static analysis warnings, mitigations, etc to code reviewers?
- How do code reviewers interpret and respond to different kinds of security warning and vulnerability information?
- How do code reviewers resolve security vulnerabilities through code review?
- How effective are they?
- What is the role of security experts in security code review?

5.4 SecView

With these design considerations in mind, I have created a tool prototype to study the security code review process. In this section, I describe the features of this design. I also discuss an example implementation of these features. Additionally, I discuss collaboration through the tool, and I provide a summary of the tool's research contributions.

5.4.1 Design Features

The SecView prototype extends Gerritt and adds an embedded web server, embedded application server, Javascript files, and a database. As a reminder, Gerritt is an open source light code review tool with a web based interface [5]. It can intercept commits in route to a git repository and postpone the commit until a code review has been completed. These features allow the tool to serve as a dedicated security code

review tool. The tool will leverage all of the existing features of Gerritt which it uses for normal code review. However, the annotations, annotation requests, and warnings are shown to other developers and security experts as a part of a security code review. Security experts are then able to login through a web interface and collaborate on the issues. The interface for the security experts is designed slightly differently than the interface for other developers. The tool enables collaboration between the primary developer, other developers, and the security experts. The details of this design are based on the results from the security auditor study, detailed in Chapter 3. The tool design includes the following features:

- Instant messaging between developers in the IDE and security experts using the web interface, handled per warning.
- Log of instant messaging conversations for each warning.
- Synchronization of annotations and warning information, made viewable to the security experts and developers.
- Ability to mark issues as correctly resolved, requiring modification, or unresolved
- Ability to push final code to repository.

5.4.1.1 Roles

When the tool is deployed, users are able to fill any of the three roles. The primary developer interacts with the tool through the IDE and initiates a code review. The

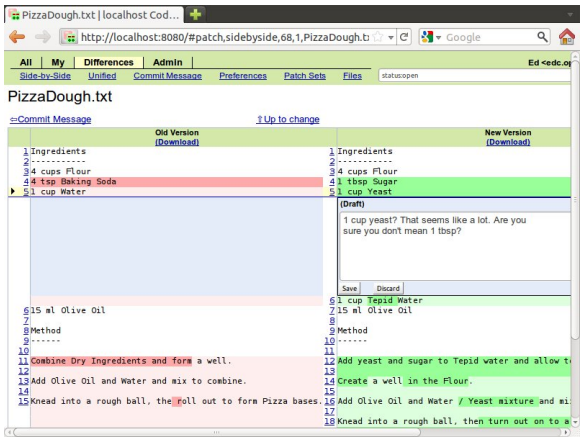

Figure 7: A screenshot of the modern lightweight code review tool, Gerrit, showing its web based interface.

security experts interact with the tool by logging in through a web interface, synced in real time. However, they do not initiate the code review and will instead receive notices of ongoing code reviews. The IDE of the primary developer is synchronized during code reviews. Additionally, other developers assigned as reviewers are able to log in and contribute to the code review through a web interface.

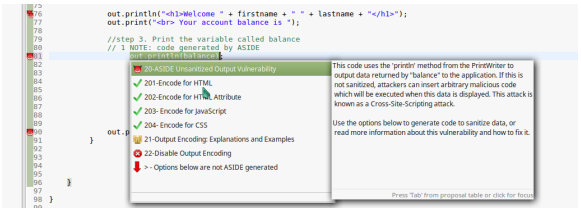

Figure 8: A screenshot showing an ASIDE Interactive Static Analysis IDE warning being resolved prior to a security code review

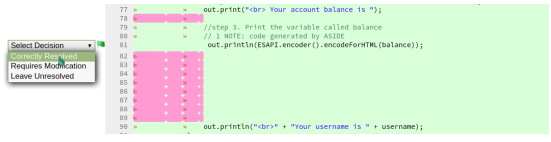

Figure 9: A security expert marking a resolved interactive static analysis warning in a security code review

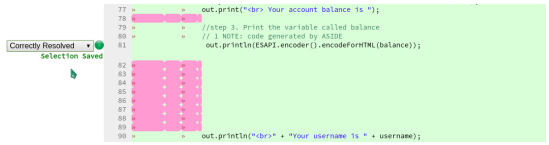

Figure 10: Security expert interface after a developer-resolved interactive static analysis warning is marked as correctly resolved

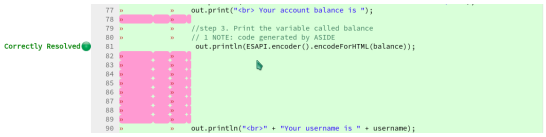

Figure 11: Developer view of an interactive static analysis warning marked as correctly resolved by the security expert during a security code review

5.4.1.2 Warning Details

The tool design supports all existing ASIDE warnings. These warnings represent all of the major categories of web security threats and include SQL Injection, Cross Site Scripting, Cross Site Request Forgery, and Access Control vulnerability warnings. These are presented in the IDE to the left of the code in the primary developer's interface. These warnings are also present in the security expert's web interface. Warnings are presented in the same manner and style as my ASIDE tool, building on lessons learned from my numerous studies on interactive static analysis. The specific warnings present in the web interface are as follows:

- Red Flag. Presented to the security expert when the developer has ignored a static analysis warning
- Blue Flag. Presented to the security expert when the developer has marked a static analysis warning as a false positive
- Green Flag. Presented to the security expert when the developer has attempted

to resolve a static analysis warning

- Red Devil. Presented to developers when a security expert has marked a warning as "Requires Modification"
- Yellow Orb. Presented to developers when a security expert has marked a warning as "Leave Unresolved"
- Green Orb. Presented to developers when a security expert has marked a warning as "Correctly Resolved"

Security experts and developers may contextually collaborate on the warnings at any time, regardless of whether or not the warnings are resolved. Security experts may also change their judgments of a warning at any time before the entire code review change has been completed.

Figure 12: Tooltip showing what type of sanitization was applied to a developer-resolved warning which was resolved incorrectly

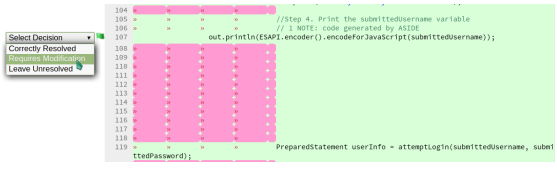

Figure 13: A security expert marking a developer-resolved warning which was resolved incorrectly

5.4.2 Collaboration

Collaboration is a key part of the tool's design. I have learned in several studies that developers wanted assurance as to whether or not their resolutions to vulnera-

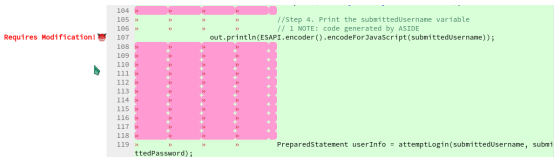

Figure 14: Developer interface showing warning marked as requiring modification

bility warnings and annotations were correct [61, 72, 71, 76]. Studies have also shown that security experts desire easily available records of vulnerabilities or issues [65]. The design of this tool should enable developers and security experts to communicate easily. It also groups records of this communication around the vulnerability warnings themselves. This enables users of the tool to avoid being overburdened with information which may come about as a result of single channel instant messaging. The records will also be retrievable later for the creation of reports. Additionally, developers and experts have additional buttons called "quote" "ack" and "done." The quote button is the same as the reply button, except it also copies the prior response and quotes it. The "ack" button stands for acknowledge, and is a one-click reply. The "done" button is also a one click reply, but says "done" instead of "ack." Ack can be used to indicate that a message was received and understood, while "done" can be used to indicate that a resolution has been carried out.

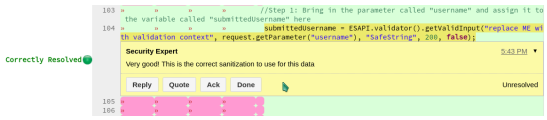

Figure 15: Screenshot showing the interface for contextualized warning collaboration

5.4.3 Implementation

Implementation of these features was achieved by the use of several key pieces of infrastructure. The Eclipse plugin of the primary developer submits the code to a

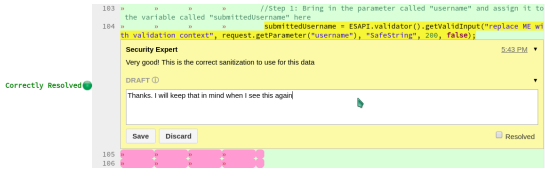

Figure 16: Developer replying to contextualized comment from security expert

git repository running Gerrit when the developer is ready to request a code review. When this occurs, the Eclipse plugin also submits HTTP requests containing the state of the user's annotations, interactions, and warnings. These calls will be made to a separate web and application server, which will accept and store the updates. The developer then logs into the modified Gerritt through a web based interface and selects a security expert to conduct a security code review. The source code of Gerritt was modified to include additional Javascript files and calls to those files. From this point on, tool-related logic is executed by custom Javascript included within Gerritt. When security experts perform security code review, they login to Gerrit's web based interface and begin the reviewing process. When they do so, the custom Javascript modifies the HTML produced by Gerrit and shows additional functionality not provided by Gerritt. This includes warning-contextualized comments, warnings themselves, and the solutions the primary developer has chosen. When the security expert interacts with these additional elements, the integrated web and application server receives the input and updates its database. The custom Javascript files pull updates from the web and application servers, using AJAX, and the HTML is modified to display these updates. When the review is finished, the final changes are submitted through Gerrit and the application and web servers maintain the additional content in their database for later retrieval and display.

5.4.4 Summary

Although building a working prototype tool was a significant endeavor, it provides many research contributions. The most important of these contributions is the ability to study the security code review process and find answers to my key research questions. Without this tool, it would have been difficult to study the interactions of participants in various roles during code review. It would also have been difficult to determine which features in a code review tool are useful for collaboration and how feedback from the security code review process should be displayed to users in all three roles. Lastly, the tool itself is a small research contribution, since other researchers could expand on the tool and use it for further security code review studies. Companies may also be able to build a commercial version of the tool and use it in their own security code review practices. SecView is available at http://aside.uncc.edu/secview.

CHAPTER 6: SECURITY EXPERT SECURITY CODE REVIEW TOOL EVALUATION

6.1 Introduction

In this chapter, I conduct a study to determine whether or not the addition of a human expert through a tool assisted security code review is beneficial for both developers and security experts. I provide motivation for the study, discuss the methodology for the study, provide my results, and discuss the implications.

In order to evaluate the security code review process, I must first study the two most important roles. These roles consist of the alternate developers and the security expert. The goals of this study are simple. I intend to determine the activities and contributions of participants in these roles. I also wish to learn their perceptions of the security code review process and what features security code review tools should possess in order to best support their efforts and activities. Moreover, I should gain an understanding of how participants in these two roles interpret, respond to, and resolve each major type of security warning.

For this study, I will use pre-written code and study the security experts. I will use one security expert who will review the code individually. This determination and specific features of the tool were decided based on the security auditor workflow from previous study in Chapter three.

6.2 Methodology

I evaluate this security code review tool with two studies. In the first study, I examine security experts. In my security code review tool design, developers first write a program or code change. They then conduct static analysis, either interactively or statically, and submit the code change to a security code review tool. The security code review receives the code change, resolved warnings, and unresolved warnings. The developer may add contextualized comments as well. At this point the security expert logs into the tool, views the resolved or ignored warnings from the developer, and provides judgements on the warnings. The security expert may also reply to the developer's contextualized comments or provide contextualized comments of his/her own. The developer then replies to the warnings, changes the code, and submits another change.

Experts were instructed to provide judgements, comments, and replies to developer's comments. The code under inspection was a simulated small banking application called Platinum Bank, made specifically for this study. Six predesigned warnings and two comments were placed into the tool before each session. Each participant observed the same code, warnings, and comments.

Screen recording software was used to record each participant's interactions, and database logs were recorded for redundancy. Semi structured interviews were conducted after the warning interactions, and demographic surveys were conducted at the end of the study. Interviews were audio recorded. The semi structured interviews were transcribed, open coded, and analyzed. The screen recordings were manually

inspected to determine behavioral patterns, warning judgement times, judgements themselves, accuracy, and comments. IRB approval was obtained for this study.

Security experts interacted with six warnings. See Table 4 for details. The first warning was a simple warning in which a developer had correctly resolved an input validation warning for a username. In the second warning, the developer applied incorrect input sanitization (filename) to an incoming password. The developer also added a comment "We should use different sanitization for passwords right?" In the third warning, a dynamic sql statement is being used to insert a new user account into a database. The developer has marked it as a false positive. In the fourth warning, data is output into a sensitive sink (the website). It was marked as a false positive and the developer has added a comment indicating that the data is likely not sanitized. In the fifth warning, data is being output into a sensitive sink as in the fourth warning. However, the developer has ignored the warning and left no comment. In the sixth warning, a dynamic SQL statement is being used to add data to a database. The developer has ignored the interactive static analysis warning. Additionally, the developer has left a comment stating "Not an issue since it was sanitized." This is meant to be intentionally misleading in two ways. First, no sanitization is actually applied. Second, prepared statements are the correct means of solving this problem.

6.3 Participants

I recruited participants from an advanced application security class offered at the University of North Carolina at Charlotte to serve as proxies for security experts. I

chose to use students from this class since it is often difficult to obtain a large sample of professional security experts for academic research, and most previous participants did not have time for additional research. Additionally, since most of these students are either almost finished with an undergraduate degree or pursuing a graduate degree, I feel that data obtained from them may be similar to that obtained from entry level professional security experts.

I recruited a total of twenty-nine participants. Twenty-four were male, and five were female. Eleven were White, seven were Asian, five were Indian, four were Black, one was Hispanic, and one was a Native Pacific Islander. Seven participants reported having professional experience as a security expert. Fifteen reported professional experience as a developer.

Participants were asked to rate their general security knowledge on a scale of 1 to ten, with 1 being very weak and 10 being very strong. Participants reported an average rating of 5.74 (SD 1.37). Participants were also asked to rate their secure programming knowledge and general programming skills on the same scale. They responded with ratings of 5.62 (SD 1.93) and 6.6 (SD 1.59) respectively. These numbers are much lower than those of the security expert study in chapter 3. This means that these participants are not true security experts, and are likely to be more like developers with some security training. Their perceptions may reflect those of security experts, but it is likely that security experts would perform with much greater accuracy.

Table 3: Demographics

Participant	Gender	Ethnicity	Sec. Experience (Years)	Prev. Dev. Experience (Years)
P1S	Female	White	0	3
P2S	Female	Indian	0	1
P3S	Male	Indian	0	4
P4S	Male	Indian	1	2
P5S	Male	Indian	1.5	2
P6S	Female	Indian	0	2.5
P7S	Male	Asian	0	0.83
P8S	Male	White	0	2.5
P9S	Male	Pacific Islander	1	0
P10S	Male	Asian	0	0
P11S	Female	White	0	1
P12S	Male	Asian	2	0
P13S	Male	White	1	0
P14S	Male	Black	0	0
P15S	Male	Black	0	0
P16S	Male	Black	0	0
P17S	Male	Hispanic	0	0
P18S	Male	White	0	0
P19S	Male	White	1	1
P20S	Male	Asian	10	3
P21S	Male	Asian	0	0
P22s	Male	Asian	0	0
P23S	Male	White	1.5	1.5
P24S	Female	Black	0	0
P25S	Male	White	0	0.4
P26S	Male	White	0	2
P27S	Male	Asian	0	0
P28S	Male	White	0	1
P29S	Male	White	1	0
Total/Avg.	24M/5F	11W/7A/5I/4B/1P	0.69	0.96/2.62

Table 4: Security Expert Warnings Encountered

Warning Attribute	Warning 1	Warning 2	Warning 3	Warning 4	Warning 5	Warning 6
Input Sanitization	X	X				
XSS	X	X		X	X	
SQL Injection			X	X		X
Output Encoding				X	X	
Dev. Resolved	X	X				
Dev. False Positive			X	X		
Dev. Ignored					X	X
Dev. Comment Question		X		X		
Dev. Misleading Comment						X
True False Positive						
Vulnerable		X	X	X	X	X
Immediately Exploitable		X	X		X	X

Table 5: Warning Accuracy and Judgment Behavior

Warning	Accuracy	Common Mistakes
1	62%	Marking correct autoremediation as incorrect
2	69%	Marking incorrect autoremediation as correct
3	79%	Believing warning was true false positive
4	69%	Incorrectly trusting data source and believing false positive
5	66%	Incorrectly trusting data source despite skipped warning
6	79%	Believing developer had sanitized input as claimed
Overall	71%	Incorrectly trusting data sources on output encoding warnings

6.4 Results

6.4.1 Warning Interaction and Behavioral Analysis

I first examine correctness of participants' judgements. Average accuracy across all warnings was 71%. This is somewhat expected due to the nature of the warnings. Accuracy was highest for the fourth and sixth warnings, at 79%. This is interesting, given that deliberate confusion was provided in the sixth warning through a developer comment, claiming that the input had already been sanitized. This insight suggests the following two things:

1. Experts are very aware of, and better at, detecting SQL injection vulnerabilities in security code review than input validation or output encoding issues.

2. Confused or defensive developers are unlikely to make a meaningful impact on this capability

Experts were effective at detecting input validation, output encoding, SQL injection, and incorrect false positives during security code review, regardless of developer questions. Accuracy was best for SQL injection and worst for output encoding. Encouragingly, misleading developer comments were unable to reduce the accuracy of

Table 6: Warning Resolution Time (in Seconds)

Participant	Warning 1	Warning 2	Warning 3	Warning 4	Warning 5	Warning 6	Total
P1S	71	60	97	65	6	179	478
P2S	33	128	87	103	97	224	672
P3S	451	159	252	197	19	348	1426
P4S	119	64	107	129	11	43	473
P5S	71	37	138	50	15	42	353
P6S	131	153	143	80	13	107	627
P7S	103	63	100	131	22	74	493
P8S	48	62	63	91	34	109	407
P9S	115	8	83	53	13	42	314
P10S	43	38	57	10	84	34	266
P11S	124	8	152	45	15	59	403
P12S	50	38	51	41	17	46	243
P13S	70	71	225	115	29	84	594
P14S	487	31	301	40	44	51	954
P15S	96	59	119	58	19	38	389
P16S	166	158	122	71	10	129	656
P17S	61	42	346	56	50	99	654
P18S	92	43	62	37	6	49	289
P19S	337	58	102	89	243	30	859
P20S	71	11	70	50	26	83	311
P21S	42	24	154	48	6	45	319
P22S	49	151	67	59	20	72	418
P23S	36	20	165	8	40	31	300
P24S	136	40	139	66	32	45	458
P25S	61	33	134	56	10	54	348
P26S	46	25	138	44	12	41	306
P27S	155	66	290	73	62	45	691
P28S	60	36	133	51	61	36	377
P29S	98	25	88	48	37	74	370
Average	118.00	59.00	137.41	67.72	36.31	79.76	498.21

SQL injection issues and helpful developer comments seemed to slightly increase the accuracy of output encoding issues. Experts were more hesitant to mark issues as correctly resolved than requiring modification, due to unfamiliarity with ESAPI and the specific functional use case. Experts seldom marked issues as "leave unresolved" and instead provided definitive judgements.

Participants took an average of about 1 minute and 20 seconds per warning (83.03 seconds). See Table 6 for details. The first thing that stands out about this is that the

first and second warning were actually very similar in nature, as both concerned simple input validation. However, the second warning took much less time on average. This suggests that warning resolution time may improve as the expert becomes familiar with the tool.

Surprisingly, the third warning took the most time to resolve. This was a simple sql injection warning. It took much longer to resolve than the other SQL injection warning. In the other warning, the developer is actually claiming that sanitization took place. This result suggests that security experts may take more time to evaluate cases of developer-marked false positives, at least the first false positive encountered.

An average time of 1 minute and 20 seconds is not ideal if many warnings are generated. Even if this average is lower, perhaps 1 minute, it still becomes cost prohibitive if many false positives are present. However, this suggests that if the false positive rate is low, it may result in great time savings, especially if it detects a security problem that would have been undetected and exploited, reduces the time required to train developers, or bypasses the need for a formal security audit.

Participants tended to be very decisive about their responses, and were unlikely to change them once made. Only 10 instances of changing responses were noted out of 174 observed warnings. This suggests that security code review tool designers do not need to invest as much effort in response changing functionality as other key functionality. The second warning was the most frequently changed response. This may be due to the fact that it is very similar in proximity and content to the first warning. Much variance is noted among experts. This means that a very limited number of experts were "changers" and practiced this behavior frequently.

6.4.2 Tooltip Observations and Scrolling Behaviors

Tooltips appear when the user hovers over a warning icon. They contain contextualized information about the warning, such as what triggered the warning and what action the developer has taken. Out of 174 total warnings, 36 contextualized tooltip observations occurred (21%). This is interesting as it suggests that the majority of tooltips went unobserved. Those who did observe the tooltips spoke highly of them during the semi-structured interview questions. Variance was also very high, with some participants discovering the tooltips and observing them frequently, and many not noticing any tooltips. This may be due to the fact that the target for generating a tooltip was simply the warning itself, not the actual dropdown box or line. At the time of design, this made sense since it would be less annoying when examining code. However, the amount of tooltip observations are smaller than expected. Therefore, security code review tool designers should be aware of this. Expanding the tooltip target area to cover the entire line of code may be more effective, with the risk that the tooltips may become annoying. This may be partially offset by allowing configuration options that disable tooltips for experienced experts.

Scrolling was noted in a total of 53 out of 174 displayed warnings (30.4%). This is interesting because it suggests, in part, what typical security expert behavior may look like when using a security code review tool. Specifically, it suggests that security experts generally like to review the warning itself and the immediate lines around it and make a decision. This is favorable behavior, as experts frequently touted the time saving potential in the semi structured interviews. However, the fact that scrolling

did occur some of the time suggests that experts did need to observe other sections of code when confused. Less variance is observed in this behavior than for tooltip observation, meaning that this was a behavior observed to a relatively small degree in most participants. Most scrolling occurred on the first, second, and third warning. This roughly correlates with the time spent on these warnings. This suggests that participants tended to scroll more at the beginning, and on the false positive, perhaps because they were less certain. They scrolled very little for output encoding issues and scrolled a moderate amount for the 6th warning, in which a developer incorrectly claimed that sanitization had been applied.

6.4.3 Conxtualized Commenting Behavioral Analysis

My security code review implementation allows for contextual inline commenting, tagged to individual security vulnerabilities. It also allows for replies to be provided to these comments. Participants were taught how to add comments to vulnerabilities and how to reply to vulnerabilities. They were instructed to add comments or reply as they would if they encountered the issue in the "real world." The point of this was to assess what kinds of responses and comments experts and developers would make and when they would make those comments. Buttons called "ack" and "done" were available when replying to a comment, which would provide one word responses.

Participants were provided with two precreated comments. The first comment was on the second warning and the second comment was on the sixth warning. The first comment was the neutral question "We need different sanitization for passwords right?" The second was the dismissive, and incorrect "Not an issue since it was

Table 7: Security Comment Behavior

Reply Type	Warning 2	Warning 6	Other	Total
Done	3	3	0	6
Ack	4	3	0	7
Very Short	6	0	0	6
Ignore	4	3	0	7
Counter Question	2	2	0	4
Neutral Detailed Response	10	9	2	21
Kind Detailed Response	0	3	2	5
Accusatory Detailed Response	0	6	0	6
Total	29	29	4	62

sanitized" comment attached to an SQL injection warning. In this case, the input was not sanitized and prepared statements were not used.

Participants provided 62 total comments. Of these 62 comments the vast majority, 58, were replies to an existing developer comment. Half of the new of comments were added to the fifth warning, which was an output encoding issue that participants performed relatively poorly at judging (66%).

One thing that stands out is that participants provided very short replies or clicks of the done and ack buttons roughly 50% of the time. They tended to ask questions about 10% of the time and provided detailed responses about 40% of the time. Most detailed responses were neutral. Some responses, but not the majority, were mildy accusatory in nature, and only occurred for the sixth warning. This warning contained a comment in which a developer was making an incorrect claim about input sanitization being applied. A few comments contained words such as "please" or were deliberately worded to avoid irritating the developer.

In short:

- Participants preferred to reply to most developer questions and other comments during security code reviews
- Participants added a few additional comments
- Participants acted in the role of a motivator when responding to developer questions

Organizations wishing to implement security code review tools should consider the dual-role of security experts in regards to answering questions and acting as an auditor. Also, organizations should be aware that, although comments were highly used and considered extremely useful, quick acknowledgements were quite common. Therefore, tools should support this capability, and use comments as a readily available tag when the acknowledgement, warnings, and tooltips are insufficient.

6.4.4 Warning and Contextualized Comment Perceptions

I gathered perceptions through semi-structured interviews. Most all of the participants were asked if the warnings made sense. They were then asked which were confusing and which were clear, as well as why they were confusing and clear. Around 85% stated that the warnings were clear. When asked about which parts were the clearest, most explained that the warning was able to direct their attention quickly to the appropriate line of code. That is, the location of the warning and it's ability to get attention were considered effective and more important than the specific type of warning being conveyed. Several participants indicated that comments provided

by developers made the meaning of the warnings much clearer.

"With the warnings we can get a hint like something is wrong here so yeah that makes sense" -p5S

"I mean that's exactly where the particular line needs to be checked so it makes more sense to be there we just so we don't need the entire code. We just need to get that particular line." -p3S

Several discussed the flags and made comments indicating that they understood red to be severe, blue to be not as severe, and green to be ok. However, they believed that the warnings were clear in this meaning. This is actually incorrect, as colors do not concern priority directly. Red flags mean that a static analysis warning was ignored by the developer. Blue flags indicate that the developer marked the static analysis warning as a false positive, and green flags indicate that the developer attempted to resolve the warning. However, in spite of this, if the warnings were to be prioritized, this prioritization may make sense. Therefore, a common, possibly prevalent trend of an incorrect but effective mental model is noted. Lastly, several participants did specifically mention false positives (blue flags) when discussing the warnings and indicated they were clear. Therefore, while this incorrect mental model was commonly observed, it should not be assumed to be held by the entire sample.

When participants did not perceive the warnings to be clear, the most common problem was that they did not feel the warning provided enough information about why it was raised. It is worth noting that most, but not all participants were not aware of, and did not discover, the contextualized tooltips. This response pattern is still observed but is much less common in participants who did discover the tooltips.

Interestingly, multiple participants indicated confusion regarding the industry standard diff colors and tab deletions. This highlighting is part of the core code review functionality and not related to the security. These participants, had, at some point in the study, perceived the diff red highlighting or diff green highlighting to have security implications related to the warning. Pink diff "tab" marks were also mistaken to have security implications. However, all of these participants seemed to be aware of this problem at the time of the interview, suggesting that they were able to correct their confusion during the study.

Participants did not indicate confusion regarding the warnings which were displayed after judgements were made (red devil, green orb, and yellow orb). They also did not indicate any confusion regarding "Requires Modification" or "Correctly Resolved." A few of them did indicate confusion about what would or should happen to a warning after "Leave Unresolved" was selected.

Several participants were asked if the comments and feedback from other users made sense. The majority (75%) said yes. Many of them explained that it lets them see the developer's thought process. Those that didn't think it made sense, and even one that did, all cited the warning in which the simulated developer claimed a vulnerability was not an issue since it was sanitized, when in fact no sanitization had been applied. Several explained that it would make sense if the developer was "experienced." but did not explain if that meant functional experience, security experience, or security collaboration experience. One expert suggested a feature that would enable a developer to highlight somewhere in the code that input sanitization or a prepared statement had been applied (interactive annotation) to prevent the

security expert from having to check for the missing code. I had not considered users' interactive annotation as a tool to feed data to a human for the purposes of time saving.

Many participants were asked what the best features for collaboration were in security code review. The vast majority said that the contextualized per warning commenting was ideal. However, many expressed concern about this method of communication if the conversation was prolonged. Several mentioned that a phone call or face to face conversation was best in these cases. A few mentioned that a global chat, such as Slack, would be best. A few also suggested a collapsible arrow button so that conversations could expand without taking up too much space. One mentioned an in-app phone calling capability. One mentioned that it would be best if comments left between warnings could be visually tied to a warning somehow so it would be clear which one was being referred to.

As a separate question, participants in my sample were asked what they thought the most key features are for any security code review tool. The vast majority of security experts responded that the contextual collaboration was the most important key feature that all security code review tools should support. Second to this and commonly mentioned were the contextualized warnings. Security experts did not always use the term "contextualized" but proceeded to describe the advantages of the comments and warnings within the code, emphasizing the importance of this context. Several developers considered the contextualized features to be major timesavers for experts. A few wanted a little more information about why warnings were triggered.

Participants in my sample were also asked how they felt about collaboration in

security code reviews. Of all those that were asked, every expert responded positively about collaboration and felt it was a good thing during security code reviews. Other than this, the most common theme was that it forced a collaboration process which allowed the experts to motivate the developers to solve the issues. One participant explained that this occurred because he could make the developer understand how a warning was vulnerable, which would motivate the developer to solve the problem.

"Working in industry... Developers must understand why you need to do that and collaborating with him and making them understand this is important. I think that is one of the best things about the code review process." -p4s

"Currently from my experience there's not enough collaboration. There's a division between the two going forward that needs to change so I like the fact that this is kind of enforcing the two to communicate." -p13S

This is interesting since it aligns with my results in chapter 3 that found that security experts often motivate developers to solve issues as a primary role. It also suggests that security code review tools should be a required part of the process, rather than an optional check in order to obtain the benefits of collaboration.

A few also mentioned that they liked collaboration in the security code review process because developers could provide their point of view. It is interesting that security experts would consider developers providing their perspective to be an advantage. However, one benefit of this would be that security experts would be able to determine what a developer was thinking when they resolved or did not resolve a warning. This would allow the security expert to tailor their response to best correct the developer.

It is worth pointing out that many mentioned the advantage of a second set of eyes reviewing the code for security issues when discussing their thoughts on collaboration in security code reviews. Interestingly, a few also mentioned and discussed a perceived advantage of shared problem solving. They explained that one individual may be approaching the problem from one perspective while another sees it differently and approaches it from a different perspective. Through collaboration during a security code review, they could share information and work together to better solve the problem. This is interesting because I have been considering security code review within the context of a developer receiving guidance from an expert. However, a junior security champion could potentially fulfill this role if a security expert is not available, and developers may have varying degrees of security knowledge. Likewise, multiple security experts may collaborate on warnings, producing this effect.

A few also mentioned that they loved the ability to quickly respond with the "ack" button and not write out a full response. One also explained that he liked the ability to simply select the judgement for the resolution and only provide the comments if needed. This is interesting because it suggests that security code review tools should loosely couple the warnings to the comments, not requiring responses or comments.

6.4.5 Perceived Contributions and Roles

One major aspect of communication in security code reviews is what role experts should play in the process. Most participants were asked what they thought the role of security experts should be in security code reviews. The vast majority felt that the primary role of the security expert was to understand security extremely well

and leverage that knowledge to catch vulnerabilities during the security code review process. Many of them also mentioned that the security expert should configure all scanners and security tools, reduce false positives in tools, and manage security policies. A few mentioned that they should help or educate the developers.

In any system where multiple roles are involved, the roles should make unique contributions to the system. In a collaborative system such as a tool assisted security code review process, it is important to understand how security experts perceive their communication. Most of the participants perceived their primary contribution as making the code more secure through contributing their knowledge during the process. Some of them (p20s) mentioned that it helps reduce the amount of legwork that is required to catch a vulnerability outside of the process, locate the developer, provide remediation knowledge, and verify the remediation. Many of the participants mentioned that it allows communication between the experts and the developers, which may impart security knowledge onto the developers. However, this was not as frequently mentioned as the direct security contributions. A few participants considered this process to be like static analysis with expert communication and assistance. One participant was aware of the operational distance between developers and experts, and the psychological problems associated with informing developers of problems. This participant suggested that this process may help bypass these issues:

"I think it would be a good resource, like a good communication point between the developer and the security expert and kind of close that gap. From what I've seen there's a division between the two a lot of times. The developers just look at the security guys and go 'Man you're killing me.' The security guy is like 'No you're

killing the company' so I think that really being able to communicate is a pretty big deal to me, and also you can give them positive feedback. This is how it should be done." -p13S

6.4.6 General Perceptions

Most of the security experts were asked what kind of burden, if any, they thought a tool assisted security code review process may place on the development process. The vast majority of them stated that it would not add too much of a burden. As expected, many of them explained that it would actually relieve a burden because it would save time otherwise spent catching the issues later or dealing with the aftermath. Several of them stated that it should not be considered a burden at all, but rather *"a part of the process" -p20S.*

Many participants expressed great concern over the possibility of large quantities of false positives, and stated that it would be a burden if that occurred. A few participants said that whether or not it is a burden depends on where the security code review falls in the development process. They were concerned about a stagnated code review during which an application security expert "unresolves," or marks "Required Modification" on vulnerabilities that were resolved some time ago by the developers. They were also concerned about development being halted if developers were waiting for security experts to review code.

A few participants also mentioned that the ability to reply to the developer may enhance the developers security training, although it was not nearly as common as expected. This may be due to the perceived goal of security code review being catching

and fixing vulnerabilities over developer security training. It should be noted that most of the "favorite" parts of the tool revolved around time savings in what was perceived to otherwise be a long scanning process of the entire codebase, out of band communication, and developer motivation.

Participants also noted that it would result in security vulnerabilities being found earlier on than without it, and therefore less likely to be left in the code and exploited.

"I think that if it was used in the regular code review process that a lot of security vulnerabilities would be found early on before they became very highly exploitable" -p1s

They also believed that developers knew more about coding and experts knew more about security. Many mentioned that security code review allows the expert to easily provide their knowledge to developers.

"I see a secure developer position (for this tool) because it's like you're developing and also trying to have the developers doing secure programming practices. They might not be particularly good at security so that's why" -p3s

Many participants seemed to assume that the tool made the most sense in large organizations or big projects, or at least imagined it in this context. Many of them provided answers that referenced big projects or large organizations.

"It's always necessary. It's something you need for all the big projects and industry." -p8S

Some thought it would be useful for the security champion to play the role of the security expert. Likewise, several noted that it may help to train developers. However, most felt that some security training was required for developers to use the tool.

"I think it would definitely help with people already in the field and hadn't had the appropriate security training. It would help the security engineers, analysts, whatever, help educate and identify possible threats and code from the people who aren't familiar with secure programming." -p18s

Many thought that security code review was going to become common practice in the future or was not common practice in the past and currently is common practice

"Security code review is huge. Maybe 5 years ago security code review wasn't a part of the process." -p20S

Several mentioned that it leveraged human intuition to find vulnerabilities that automated scanners could not detect.

"You need the human element to see proper mistakes. Machines don't really know the code, the purpose of code." -p23S

Others mentioned that simply having a second set of eyes glancing at the code can provide many security benefits:

"Initially when you make the code you let a lot of things slip by. It's always good to have a second opinion based on what other people might see in your code" -p17s

Several felt that it should be done during development (p28s) or before other, more rigorous security processes.

Participants mentioned that the process allowed them to observe how the developers were handling errors, and allowed them to easily detect whether or not the developer was handling errors in a secure fashion. This is interesting, as proper error handling is not trivial for static analysis tools to detect, but was considered an advantage in their general perceptions of this tool assisted security code review. This

suggests that security code review tools should consider functionality to support error handling code identification and error handling comments, or at least tool designers should be aware that the tool may be used in this way.

Participants mentioned that the process could useful in the prioritization of vulnerabilities for developers because the security expert can raise or lower the priority of vulnerabilities on the fly. They also mentioned that the security code review process produces immediate feedback. Developers have a timetable and work on different pieces of functionality for a set amount of time. If the security expert comments on certain warnings, priorities can be established and changed as the project functionality priorities change. Participants believed that this greatly reduced temporal gap will result in faster and easier fixing of the vulnerability by the developer.

"I do code reviews in my work. It's just basically literally going line through line with other developers and saying just 'This needs changing. this needs changing.' I think something like this could be useful to represent the code instead of everyone just hollering around the machine. And if you're saying okay fix that writing sticky notes and whatever, you could just have a visible representation " -p26s

However, it is worth pointing out that in order for this priority shifting technique to be effective, security experts would need to be acutely aware of evolving functional priorities. My results and findings from the security auditor study in chapter 3 suggests that, although they may be somewhat aware of the functionality of applications, they typically only learn enough about any given application to understand where the security sensitive components are located and determine whether architectural flaws exist. They also learn this information from the developers themselves, rather than

the project managers. Due to this environment, it may not be efficient for security experts to keep track of evolving functional priorities in order to raise or lower vulnerability priorities. That said, security experts may be able to infer priorities based on what parts of the code are being changed. Obviously, the highest functionally important code is written first, and therefore would be in the code change. Therefore, running static analysis on code changes and performing security review on code changes may incur the innate advantage of working on the highest priority pieces of a project.

In considering the effectiveness of a security code review tool, it is critical to ask specific questions and perform a detailed comparison. Most participants were asked if they felt that their vulnerability remediation advice was more likely to be correct with a security code review tool than without one. Overwhelmingly, the vast majority indicated that they believed that their vulnerability remediation advice was more likely to be correct with a security code review tool than without one. Many of them then cited the ability to provide specific, concrete, contextualized remediation instructions in the code in an organized fashion as the primary reason for this determination.

"I'd say yes because it's tied directly to the code that I'm giving advice off of. I'm not just talking conceptually, but I'm pointing to the code and what specifically is wrong with it and the supporting evidence that there was a warning there." -P1S

Most participants were also asked "Do you think that your (vulnerability resolutions/remediation advice) is more likely to be correct with a security code review tool and static analysis than with static analysis alone?"

Almost all of the participants answered yes to this question. This suggests that

participants believed that the addition of a security code review process enhances the security of the application being developed. Participants often explained why they provided a "yes" answer. The most commonly cited reason was the previously mentioned belief that scanners may not catch all issues and human intuition can help bridge this gap.

"Having this just helps more because sometimes a scanner like most things that's automated has errors in common sense it gives you like some memory points some address It doesn't like give you any solution to solve it so having humans helps it doesn't tell you how to solve it so having some human that just says here this is a false positive right here false positive" -p9s

This result is particularly encouraging since it demonstrates that the participants believed that:

- Security code review makes remediations more accurate than not doing security code reviews.
- Security code review and static analysis make remediations more accurate than only static analysis.

6.4.7 Design Suggestions

In order to design effective security code review tools, it is important to understand what features experts liked the most. Most of my participants were asked about their favorite part of the tool. The vast majority of experts (90%) responded that they liked two particular broad aspects of this tool the most. These aspects are the contextualization of the warning, and the ability to communicate with the developer in the

context of the warning. Some explained that this is due to the timesaving potential. Participants explained that the time saving advantages were due to following reasons:

- Time saving from observing only code changes or small pieces of files
- Time saving from avoiding out of band communication
- Reduced quantity of information needed for communication due to the contextualization of warnings and comments
- Time savings from quick visual warning severity indications
- Time saved from avoiding out of band audits or remediation verification

Others also cited the following three distinct "Advantages" of the tool when asked:

- Avoids the need to write down comments and line numbers out of band on paper or email.
- Developers are able to provide feedback easily.
- Easy to track and verify that vulnerabilities are correctly resolved.

Note that a few were confused about the pink tab space deletion in the Gerrit code review tool and thought that they were indicating some sort of error. Interestingly, one participant suggested that the red and green diff deletion and insertion highlighting be abandoned and repurposed to indicate whether or not a warning was fixed in an automated manner, a manual manner, or ignored. This is interesting as it would be a novel and potentially intuitive way of communicating this information to security

experts. It would also remove the code diff confusion problem. However, it would potentially introduce new problems as other users may assume the standard code review utilization of this highlighting remains. Also, it would not be possible to communicate which lines were added, and which were deleted through this mechanism.

A few participants expressed a desire for some direct means of triaging the warnings. One participant suggested a single sub menu for the "Requires Modification" option that would categorize it based on several common, broad, remediation terms such as "Sanitize" or "Prepared Statement." This design suggestion may have merit, as it could prevent the expert from having to specify this information in a comment. However, in cases where many warnings are generated, the time required to navigate this additional menu may prove counterproductive.

Many experts commented on the timing of the security code review process in the software development lifecycle. That is, security experts considered correcting or assisting developers as a key feature in security code review tools. They may not have liked the process as much if the developer did not attempt resolution of the static analysis warnings prior to the security code review.

In spite of this, a few experts did want a feature that would allow them to remediate the code in the security code review process. This is interesting because it may encourage dependence on the security expert for remediation. However, it may make sense for rare and complex issues when the effort required to explain the issue is greater than the time required for the expert to remediate the issue.

A few experts also desired a feature that would allow them to contact additional or third-party security experts for assistance on complex issues on a per warning basis.

This is interesting since off site security experts are becoming more commonplace. Companies, such as HPE, currently offer offsite static analysis. However, a per warning third-party security expert consultation feature does not currently exist in any known commercial or experimental security code review tool.

Lastly, a few security experts considered the ability for a security code review tool to support multiple languages and code repositories to be key features. This is not surprising, considering it aligns with results from my security auditor interview study in chapter 3.

Beyond this, several other features were mentioned as key to security code review tools to a lesser extent. A few developers mentioned the ability to arbitrarily leave notes and communicate with the developer on regions of the code was a key feature. This is particularly interesting because my design supports arbitrarily located comments, but I did not expect it to be used during the study. The design does not support arbitrary contextualized notes. A few experts did leave arbitrary comments, but this was not nearly as common as replying to existing warnings. However, the study script encourages warning by warning resolution, which suggests this feature may be much more critical to security code review than first expected. Some experts expressed concern that developers may be able to delete comments to avoid resolving warnings. My design and implementation support this feature, since comments cannot be deleted from the code change once they are submitted. However, since security experts expressed this concern, it is reflected in my design guidelines for security code review tools.

The ability to leave notes is particularly interesting since it was not expected. Notes

could be left in the code, tied to the warnings and include details that the security expert does not want to share with the developer or is not yet ready to share with the developer.

One expert mentioned that cross platform collaboration tools would be nice for collaboration. The design and implementation both support cross platform development tools, but not collaboration tools.

In summary, contextualized warnings and contextualized commenting were considered to be the most important parts of the tool by the participants in this study. Participants cited time savings as the main advantage. They believed that the contextualization provided a variety of time saving benefits, such as reduced communication overhead, and avoidance of the need to write down line numbers and comments among other reasons. The "ack" and "done" buttons were also heavily utilized and well liked by participants, as they helped to facilitate quick contextualized communication. Warnings successfully attracted the attention of the participants, and they were able to react appropriately. Tooltips were very well liked by those who observed them, but were not observed frequently.

6.5 Summary

This evaluation does contain some important limitations. First, student proxies were recruited and used in place of security experts. Although they acted in the role of security experts and did have some security training from a security course, they were not true security experts. True security experts would likely perform much more effectively than student proxies and may have different opinions. Second, the contex-

tualized conversations were simulated. Even though participants were instructed to respond as they would in the "real world," they were informed and aware that the comments were provided by a researcher. It is possible that participants would have reacted differently in a real world setting.

However, in spite of these limitations, I learned much from this evaluation. First, I know that participants performed best at SQL injection vulnerabilities and worst at output encoding vulnerabilities. Second, I learned that confused or defensive developers are unlikely to affect performance of users in the security expert role, as it draws attention to the issue. Third, I learned warnings are typically resolved in 1 minute and 20 seconds, with great reductions as participants gained practice. Additionally, I learned that participants do not like to change judgements, scroll roughly 30% of the time, and did not observe the tooltips frequently.

I also learned much about the communication of participants in the role and their perceptions of the warnings as well. I learned that 85% of participants felt that the warnings were clear, as they clearly directed them to the line of code. However, they misunderstood the color to represent severity. Most participants replied to comments frequently, made great use of "ack" and "done" buttons, and valued this communication. Most felt that inline contextual collaboration was the most important feature for any security code review tool. They also believed that the warnings should be contextualized to the line of code. Participants considered the primary value of the tool to be time savings, mostly brought about by the contextualized conversations.

From these results, I can suggest some design guidelines. First, security code review tools should support contextualized commenting and contextualized warnings

as a primary feature. Inline commenting should support at least one type of button which sends an acknowledgement of some kind. Limited and broad judgement options should be provided. Warning colors should also correspond to severity if at all possible. Security code review tools should also be designed to minimize unnecessary configuration or other time costs, as time savings is perceived to be the primary benefit.

CHAPTER 7: DEVELOPER SECURITY CODE REVIEW TOOL EVALUATION

7.1 Introduction

In this chapter, I conduct a second evaluation of my security code review tool. I study the role of the primary developer, and gain an understanding of their behaviors during tool assisted security code review. Additionally, I gain an understanding of their perspectives of the process and how they differ from those of experts.

Studying the security code review process from the vantage point of the security experts yields most of the answers to my key research questions. However, it is missing one key component- the primary developer. The person who wrote the code, or the primary developer, may have a very different perspective on the security code review process. As a result, to obtain true answers to questions such as how the participants in different roles collaborate with each other, the role of the primary developer cannot be ignored. Likewise, the primary developer may react very differently to the security code review process on code that he himself wrote. Therefore, another study is needed.

With this study, I examine how primary developer perceives the process of tool assisted security code review. I also investigate how the primary developer collaborates with other roles of the security code review process. Moreover, I identify the challenges and needs of the primary developer in security code review. Lastly, I examine how the primary developer perceives the process of tool assisted security code review.

I identify key communication challenges which arise when the primary developer communicates with security experts and other developers. I also identify the ways in which feedback and warnings should be displayed to the primary developer. Since the primary developer wrote the code being reviewed, feedback from the process of security code review may need to be displayed very differently than for security experts and alternative developers.

7.2 Methodology

To conduct this study, I again recruited students to serve as the primary developers. However, they were recruited from a Network Based Application Development class at the University of North Carolina at Charlotte to serve in the role of the primary developer and were given a task to code.

Developers were given very small coding tasks in a simulated small banking application, called Platinum Bank. This is the same application used to evaluate the security expert role in chapter 6. Due to a wide range of student programming ability and the time limitations of the study, tasks were very small and consisted of the minimum amount of coding necessary to produce warnings from ASIDE, my interactive static analysis tool. The first warnings concerned sanitization of an incoming username and password respectively. The third and fourth warnings concerned output encoding of a balance and a username. Table 8 summarizes the warnings.

As developers coded, pre-planned warnings from interactive static analysis appeared. They then were instructed to interact with the ASIDE warnings. Once this was done, they pushed the code to a security code review. A remote researcher then

Table 8: Developer ASIDE Warnings Encountered

Warning Attribute	Warning 1	Warning 2	Warning 3	Warning 4
Input Sanitization	X	X		
XSS	X	X	X	X
Output Encoding			X	X
Username	X			X
Password		X		
Balance			X	
HTML Sink Type			X	X

judged the warnings and left feedback, simulating the role of the security expert. The researcher left at least one comment on one warning. The comment was worded and the warning was selected based on where the developer appeared to be most confused or hesitate. Sometimes, if the developer seemed particularly confused, the researcher would leave an additional comment on another warning. Developer participants were then given a chance to view the judgements, leave comments, and reply. Once this was done, they were interviewed individually in a semi structured manner. Demographic surveys were administered at the end of each session as with the security experts.

I recruited a total of twenty-one participants. Seventeen were male, and four were female. Twelve were White, five were Asian, three were Black, and one was Hispanic. Seven participants reported professional experience as a developer. Participants had an average of .68 years of professional experience as a developer. The seven participants who reported professional experience as a developer reported an average of 2.05 years of professional development experience.

Table 9: Developer Demographics

Participant	Gender	Ethnicity	Sec. Experience (Years)	Prev. Dev. Experience (Years)
P1D	Male	White	0.25	0.25
P2D	Male	White	0	1
P3D	Female	Black	0	0
P4D	Male	White	0	0
P5D	Male	Asian	0	0
P6D	Female	Asian	0	0
P7D	Male	Asian	0	0
p8D	Male	Asian	0	0
P9D	Female	Black	0	0
P10D	Male	Hispanic	0	0
P11D	Female	White	0	0
P12D	Male	White	0	0
P13D	Male	White	0	0
P14D	Male	White	2	2
P15D	Male	White	0	5
P16D	Male	Asian	0	0
P17D	Male	Black	0	0
P18D	Male	White	0	0.6
P19D	Male	White	0	1.5
P20D	Male	White	0	4
P21D	Male	White	0	0
Total/Avg.	17M/4F	12W/5A/3B/1H	0.11	0.6833

Participants were asked to rate their general security knowledge on a scale of 1 to ten, with 1 being very weak and 10 being very strong. Participants reported an average rating of 4.37 (SD 1.98). Participants were also asked to rate their secure programming knowledge and general programming skills on the same scale. They responded with ratings of 3.84 (SD 2.22) and 5.89 (SD 2.18) respectively.

7.3 Results

7.3.1 Warning Interaction and Behavioral Analysis

Developers were provided with four warnings through use of ASIDE, the interactive static analysis tool. They could then use use the tool and create snippets of code to replace existing vulnerable code. Once this was done, the results were uploaded into the security code review tool. From another room, I was able to provide simulated feedback as a security expert. Once this was done, the participant was shown the code with the judgments and comments. They were then given a chance to respond.

Developers performed the best (86% correct) on the second warning, which concerned input validation of an incoming password. Most of those that chose incorrect solutions, chose solutions which may have been correct if the application was designed differently. It is likely that they were assuming that the application uses a social security number or no special characters for the password. Only one participant actually had trouble with this selection.

Accuracy was also very high for the first warning (76%), which involved input sanitization of a username. However, some participants selected options like “Only allow Filenames,” and applied sanitization for social security numbers, which is unheard

Table 10: Developer Accuracy in ASIDE remediations to warnings

Participant	Warning 1	Warning 2	Warning 3	Warning 4
Correct Answers	ABC order	Special	HTML	HTML
P1D	Ignore	Ignore	HTML	HTML
P2D	ABC order	Special	HTML	NA
P3D	Filename	Special	Attribute	Attribute
P4D	ABC order	Special	Attribute	HTML
P5D	ABC order	Special	Attribute	HTML
P6D	ABC order	Special	Javascript	Javascript
P7D	ABC order	Special	Attribute	HTML
P8D	ABC order	Special	Attribute	Attribute
P9D	Special	Special	Javascript	Attribute
P10D	Email	ABC order	Javascript	Javascript
P11D	ABC order	Special	HTML	HTML
P12D	ABC order	Special	HTML	Javascript
P13D	SSN	SSN	HTML	Javascript
P14D	ABC order	Special	HTML	HTML
P15D	ABC order	Special	HTML	Javascript
P16D	ABC order	Special	Javascript	Attribute
P17D	ABC order	Special	HTML	HTML
P18D	ABC order	Special	HTML	HTML
P19D	ABC order	Special	HTML	HTML
P20D	ABC order	Special	HTML	HTML
P21D	ABC order	Special	HTML	HTML
Average	76%	86%	57%	52%

of in username fields. It is possible that participants believed that these were safer sanitization algorithms and therefore preferable.

Surprisingly, developers performed much poorer on the third and fourth warnings. For these warnings, accuracy was 57% and 52% respectively. These warnings involved output encoding of variables being output into a webpage as HTML. What is most surprising about this result is that participants had roughly half as many options to choose from, which should increase the probability of choosing a correct answer. Additionally, the warnings were very simple, and involved strictly outputting a number and a username. Although the most commonly chosen incorrect answer was "Encode for HTML Attribute," which is very similar to "Encode for HTML," almost as many developers incorrectly selected "Encode for Javascript." This suggests that output encoding was not only avoided by my security experts in my current study, but poorly understood by developers.

If I compare the total accuracy rates among both the developers and security experts, I find that security experts had a rating of 71% for all of the warnings they encountered. Developers had a similar rate of 68% for all warnings they encountered. Developers were more accurate at resolving input validation issues (76% and 86%) than security experts were at judging them (62% and 69%). Both performed poorly on output encoding issues. This suggests the following:

Developers are reasonably effective at detecting and remediating input validation issues before a security code review if interactive static analysis is used.

Developers can be expected to perform poorly at detecting and remediating output encoding issues during interactive static analysis.

Table 11: Developer Comment Behavior

Reply Type	Warning 1	Warning 2	Warning 3	Warning 4	Total
Done	1	4	0	0	5
Ack	3	4	3	1	11
Very Short	0	0	0	0	0
Ignore	0	4	0	0	4
Question	0	1	1	0	2
Thank you/Verbal Ack	0	1	1	1	3
Defensive Detailed Response	0	7	1	0	8
Accusatory Detailed Response	0	2	0	0	2
Total	4	23	6	2	35

This implies that developers can be expected to perform reasonably well with their tasks, leading into security code review.

7.3.2 Contextualized Commenting Behavioral Analysis

Developers were provided with at least one comment, sometimes two, during the security code review process. One comment was attached to the second warning and an additional comment was attached to another warning that the developer answered incorrectly. Comments were worded politely and attempted to suggest alternatives resolutions rather than give commands.

Developers provided a total of 35 responses. The developers did not ask any questions on their own. 16 of the responses were clicks of the "acknowledgement" or "done" buttons. Two were comments. 10 were detailed responses. Of this, 8 were defensive in nature. They are categorized as such because the developers provided some sort of excuse for their action, though they may have admitted error. 2 suggested non specific alternatives such as a general use of ESAPI for the whole project and "a better method or logic for handling this (p20D)"

Developer comments were very interesting. One would expect developers to tend to ask questions on their own, with security experts providing replies. Instead they only replied and did not initiate conversations through the tool. However, roughly half of the time, they replied with a quick button press of the "done" or "acknowledgment" buttons. They also seldom had questions to the comments. The more detailed comments consisted of 1 to three sentences. Detailed replies were observed roughly one third of the time. It appears that 80% of these consisted of some kind of excuse. This suggests that the developer felt that they had made a mistake and needed to justify it to avoid blame. This occurred despite that fact that language was carefully selected to avoid sounding accusative. Encouragingly, only 2 of the total amount of replies actually suggested that the developer was unwilling to implement the change, or at least required additional convincing before implementing the change. In short:

Participants preferred to reply to most of the questions initiated by the experts during security code review, of which many are simple, short, acknowledgements of some kind.

Many developer replies contain some justification for why a particular piece of code is vulnerable.

Additionally, from these insights, it appears that organizations implementing tool assisted security code reviews could further reduce the human burden of code review by training developers to avoid defending their code if they intend to fix it. Likewise, security experts should be trained to expect developers to demonstrate these behaviors.

7.3.3 Warning and Contextualized Comment Perceptions

Most developers were asked whether or not the warnings made sense, as well as why they were clear and why they were confusing. Most simply stated that all the warnings were clear and did not elaborate. A few mentioned the tooltips favorably, but most developers did not encounter the tooltips. Several mentioned that the warnings provide additional feedback in the form of the comments, which made the warnings very clear and actionable. Several participants indicated that the yellow orb, or "Leave Unresolved" was less clear than the other two judgement types. The participant who did not consider the warnings to be clear said that it was because of "his inexperience." He also desired a different remediation options in the static analysis portion of the tool and felt that the warnings were subsequently confusing because that option was not available.

Participants actually had very negative perceptions of issues that the security expert marked "Leave Unresolved" when answering about their perceptions of the code review process, and not directly answering questions about the warnings. Many were confused as to what the designation meant, and were unsure how they would proceed if they encountered this in the real world. These participants appeared to be expecting a binary decision of "correctly resolved" indicating they chose the correct answer or another answer indicating they did not choose the correct answer. The yellow orb and "Leave Unresolved" appeared to indicate an issue that needed resolving, but was not as serious as those that were marked "requires modification." However, despite this confusion, the intended effect remains the same. Issues that are marked as "leave

unresolved" do not require action at the current point in time and are lower priority than issues marked as "requires modification" At a future point in time, the issue may need to be resolved. Using a neutral color may convey that the issue is either resolved or partly resolved, which is the not case.

When developers answered about the interactive static analyses portion of the tool, they were asked about the code review portion specifically. Upon shifting the focus of the discussion, most of the developers mentioned that they really liked the human "feedback" and the option to reply when they had questions. Several of them also mentioned that they liked the fact that the code was "inline." Many of the developers also liked the fact that they could leave issues unresolved if they were confused, delegating the responsibility for security of the code to the security expert(perhaps discouragingly for security)

"I liked the fact that I can respond and it stays tagged and I can click done with my response without having to actually click resolve issue. I don't know the policy of the website, so I don't know if that's something that I need to change or not, so if I can't make that decision as a developer, I can just make that comment unresolved and wait for their feedback." -p14D

A few of the developers did use the word "conversations" when referring to the feedback. Many developers were very appreciable of the human feedback, verification, and responsibility deflection, most of them did not think that the inline comments were ideal for long conversations (see burdens section). However, the vast majority of developers did feel the inline comments were ideal for short conversations. This implies that security code review tools should support some sort of functionality

that allows a conversation to be escalated through a different medium. At the very least, tool designers should be aware of the value of inline commenting for short conversations and aware that it is less valuable for longer conversations or cases where developers are extremely confused. One feature that was suggested by developers was collapsible and expandable conversations. This would allow contextual conversations around vulnerabilities to expand as needed but would prevent them from cluttering too much space.

A few of the developers also mentioned that they liked seeing "better alternatives" provided by the security expert. This, when combined with the fact that developers also liked the verification provided by the security experts, suggests that security code review tools may help to offset some of the developer perceptions of security experts being an obstacle. However in this security code review tool design, ultimately security experts decide whether an issue is resolved or not. Therefore, they are very much still performing the same role as they would in the absence of a security code review tool. Since many developers mentioned that they liked the ability to reply to the security expert, they may at least feel as though they have an easy appeals process.

"You could acknowledge it or reply to it. That part I feel like was useful for just being able to see what was said and letting people know that you saw it. I like really being able to communicate inline with the vulnerability." -p9D

Most developers were specifically asked how they felt about collaboration with security experts in the security code review process. When responding to this question, the vast majority stated that they liked it and that the inline contextualized comment-

ing was the best means of conducting the collaboration. Many of them mentioned that they liked being able to get feedback from the expert and verification from the expert as opposed to tracking down external security training resources, solving the warnings, and verifying they are resolved correctly on their own. Several developers mentioned that they liked collaboration because a human could explain what was wrong and why it was wrong, which is much more difficult for tools. Interestingly, only one person mentioned that lengthy conversations would be better on an out of band channel, and one person even mentioned that he liked the contextual collaboration because it prevented the need for a phone call to an expert. This is unlike the experts, who frequently mentioned that out of band channels would be better for lengthy conversations. This result suggests that experts may need to initiate out of band conversations when a developer is confused, or find ways of handling larger than ideal amounts of inline comments, as developers appear more likely to focus exclusively on the inline comments.

7.3.4 Perceived Contributions and Roles

Most developers were asked about their perceived contributions to the security code review process. The vast majority (95%) referred to the secure code generation aspect of ASIDE when asked. Like the security experts, developers often discussed the ways in which the tool would save time. Many commented that the auto code generation would save time googling. Some also commented that the auto code generation saved them time from actually writing the code as well *"I was able to double click and it was over -p14D."* Many of them also mentioned the red devil icon in the IDE as

well. They generally found it mildly humorous and accurately believed it indicated that an attacker may be able to manipulate the flagged line of code. Several also expressed favor that the results (from the IDE) were contextualized. These results suggest that the developers also perceive the primary value of the tool to be time savings. However, from their point of view, the time savings stems from the auto-code generation. When answering this question, they often perceived the code review as saving time since it could either 1) avoid audits, or 2) reduce the amount of issues they have to deal with during audits, or 3) avoid the extra effort required to fix issues when those issues appear during audits.

Most developers were asked what they thought the role of security experts should be in security code reviews. The majority, like the security experts, believed that security experts should make sure that the code is secure. Several also mentioned that the security experts should be aware of organizational security policies and implement those policies. Additionally, several mentioned that security experts should configure and maintain the security tools. A few mentioned that the security expert should be a final check and be responsible for the security of the code.

Surprisingly, several developers mentioned that either the main role or an equally important role of the security expert should be to train the developer. A few of them explained that this is important because it would greatly reduce the time burden on both security experts and developers that would occur if the developer did not have proper training. However, several other developers stated that security experts should avoid training developers. One reason given was that since security experts may not be good instructors and are in their role because of their technical expertise.

Another mentioned that their time would be more efficiently spent on other tasks. Lastly, a few felt that the developer would be able to learn on their own over time through this process.

7.3.5 General Perceptions

Most developers were asked what kind of burden, if any, they thought a tool assisted security code review process might place on the development process. Most felt that it would produce a small burden, but that it was well worth it for the security of the code. Surprisingly, several felt that the tool would save time due to not having to resolve the issue in a later audit.

In spite of this widespread support, many went on to describe situations in which it could become a major burden. Several expressed fears that if the security expert was not very knowledgeable, it could become tedious and burdensome. This is interesting because one security expert advocated for junior security experts to conduct this task. One was concerned that there is a risk of possible micromanagement, depending on the temperament of the expert. A few developers, like the security experts, were worried that a code review which becomes stagnated could greatly delay developers as they wait for the code to be reviewed. Like the security experts, several developers felt that if excessive false positives occurred, it would be a burden.

"Of course it's going to take a bit longer but I think this does the minimum impact possible because the developer is able to see the results of the static analysis tools before they submit it on to a security expert. They've got this application security right in the IDE. And then I got the feedback from an expert to see each of these potential

problem steps." -p1D

Most of the developers in the sample were asked if training was needed for developers to perform their role of security code reviews. Roughly half of those that were asked thought either no training at all was needed, or a short tutorial on the tool was all that was needed. Of the half that thought training in excess of a tutorial was required, the vast majority felt that basic security training, such as common vulnerability types and their remediations was all that was needed. A few developers felt that basic security training was only needed to teach developers the importance of security, so they do not reject the tool.

These results are particularly surprising and interesting. One would expect developers to feel that developers should have more training, as they should be less comfortable with the tool and process than security experts. However, unlike security experts, almost half of developers did not feel that security training was necessary. This finding has implications for tool designers and suggests the following:

1. Very basic developer security training, although not required, may be beneficial
2. Developers are less likely to feel they need training than security experts

Many developers were asked about the role of security code review in industry. Overwhelmingly, developers felt similar to security experts. Most all of them believed that security code review, at least as they observed it, was needed and desirable for industry. However, most of them considered the security code review process and "application security" in the same terms. When asked, many of them discussed the importance and need of security in general. This makes sense, when I consider they did not have the same level of security expertise. They did not often consider the

value of security code review when compared to other methods of application security, though they did sometimes consider the effects of code review specifically. Some developers (p18D, p19D) mentioned that it would be most useful for applications deemed "critical." Although they did not use the term, "higher risk," results in chapter 3 suggest that security experts would interpret it as such.

Perhaps most interesting is the lack of resistance and general appreciation for the tool. Developers could be expected to be resistant to the tool since it would add more work. However, this was not observed in my sample.

Developers seemed to put great value on being connected to the security expert and treated the security expert as a resource rather than an obstacle. However, developers seemed to have the perception that this would reduce the workload on security experts, rather than increase it.

"A lot it seems very useful because the developer can then go on developing their code, and have the security expert go through it and analyze what needs to be changed, and leave little footnotes, so all the actual security professional has to do is analyze the code, and he doesn't actually have to sit down and recode someone else's work." -p15D

Several of the developers noted that security code review would remove issues in code quickly, and placed value on that

"I definitely see a use for this because the tool will allow the developers to patch holes right off the bat." -p2D

Developers perceived the code remediation of ASIDE and remediation advice of security experts as valuable and educational if the developer has functional program-

ming knowledge, but did not think it would be effective for very junior developers with limited functional programming knowledge (p8d, p12d).

Some developers mentioned that the security code review would allow developers to focus more on functionality and less on security. This was perceived as positive (p15D).

Most developers were asked if they felt that their vulnerability remediations were more likely to be correct with a security code review tool and static analysis than with static analysis alone. Encouragingly, the vast majority felt that the their results were more accurate with both processes in place. Many of them made comments suggesting that they perceived the security code review as a sort of "check" to catch anything that slips through the static analysis detection and remediation process. A few of them expressed concern that small organizations may not be able to afford both processes and favored the static analysis if cost was a limiting factor.

7.4 Summary

This evaluation of the developer role contains some limitations. First, student proxies were recruited and used in place of developers. Although they were junior, senior, and graduate students acting in the role of the developer, they were not true developers. True developers may have different opinions, particularly in regards to preconceived notions and experiences with security processes. Second, participants did not have to implement any suggested changes. If they were required to implement changes, they may have been more hesitant to agree with or acknowledge the security expert. Likewise, they may have become more defensive about their code.

However, in spite of these limitations, I learned much from this evaluation. First, I learned that participants in the developer role perceived the process very similarly to those in the security expert role. Like the experts, the most important feature to participants in the developer role was contextualizing commenting. However, unlike the previous study, many of these participants discussed it within the context of human verification. Most of these participants also believed that the warnings were clear. However, they were confused by meaning of "Leave Unresolved" since they were not sure what to do with a warning in that state. Participants in this study also liked to see suggested alternative remediations. Participants in this study were very appreciative of human feedback, verification, and the delegation of security responsibility to the security expert despite actually resolving the vulnerabilities themselves.

Like the security experts, developers were very afraid of excessive false positives from static analysis carrying over into the security code review process. They were also concerned about stagnated reviews due to a limited quantity of security experts and unknowledgable security experts unresolving warnings which were resolved correctly, thus adding the need to justify and argue for each remediation. However, in spite of these concerns, most developers felt that the process, as they observed it, would not add much of a burden. They also overwhelmingly believed the security benefits outweighed the burden. Additionally, many of them perceived that it would save time as it would prevent or reduce audits, address issues immediately after development, and remove the effort to manually code remediations when combined with interactive static analysis.

CHAPTER 8: RESEARCH QUESTIONS, DESIGN GUIDELINES, AND APPLICATION SECURITY IMPLICATIONS

8.1 Introduction

In this final chapter, I lay out my research questions, and discuss how this body of work answers each research question. Moreover, I provide a collection of design guidelines for security code review tool designers. Lastly, I discuss notable implications for application security.

In this dissertation I begin by introducing the problem of efficient vulnerability detection and remediation in application source code. Following this, I identify a large coherent body of work that has investigated the use of different tools and techniques to move towards this goal. Following this, I identify an aggregate workflow model through a qualitative study of application security experts. More importantly, from this model, I infer a vulnerability lifecycle detailing how vulnerabilities are created, detected, and ultimately remediated in current business processes. I also gain an understanding of how and when interaction occurs between application security experts and developers. Lastly, I identify many communication, organizational, and technical challenges that my participants describe, as well as implications for application security.

Following this, I discuss my evaluation of an interface for a technique dubbed interactive annotation. I found that developers could successfully annotate checks

for security sensitive operations 77% of the time. This suggests that this approach may be effective for the detection and remediation of application-specific and access control vulnerabilities. However, it also provides a broader contribution. Specifically, I found that developers are capable of providing security input into a security tool during code development. I also showed that developers, although mostly accurate, had low confidence in their accuracy because the tool has no way to verify their checks are correct. It can only determine if a check is missing or incorrect, but has no way to verify that two identical checks are evaluating the correct criteria for a given resource. This prompted them to suggest that this approach be used in a security code review.

Following this study, I designed a security code review tool to introduce the idea of a security code review. This design is detailed in chapter 5 and is based on my findings from the evaluation of interactive annotation, the identified vulnerability lifecycle in chapter 3, the identified application security expert environment in chapter 3, current challenges to application security experts in chapter 3, and related work in chapter 2. I build an implementation of this design, and discuss this in chapter 5 as well.

In chapters 6 and 7, I learn the typical accuracy of the roles in security code review, behavioral patterns for both roles, frequency and type of communication, critical features and design suggestions, and perceptions of the tool from both roles. These findings are the result of two separate evaluations of security experts and developers.

In this chapter, I assess these contributions and use them to answer my research questions. Additionally, I use them to suggest design guidelines for security code review tools. Lastly, I provide application security implications based on these contributions.

8.2 Research Questions

In order to gain an understanding of the security code review process, it is critical to ask and answer specific research questions. I have elected to ask and answer research questions in three broad categories. I am assuming that two or three natural roles are apparent in security code review. The first category of research questions seeks to identify and support these roles. The second category of research questions seeks to identify ways in which security code review participants communicate and collaborate with one another and how to support this collaboration. Lastly, the third category of research questions seeks to identify what type of warnings should be displayed in security code review and how they should be displayed.

8.2.1 Roles

8.2.1.1 What are the activities and contributions of people in each of the three roles in security code review?

Understanding the activities and contributions of each of the roles of security code review is important since tools will be utilized to support those roles. From my evaluation of security code review in chapter 6, I learned much about the activities and contributions of each of the three roles. First, I learned that the alternative developers and experts are almost identical in function, as participants did not indicate a need for an "alternative developer." However, from my application security expert study in chapter 3, I know that security champions are frequently used by organizations implementing application security. I also learned that these security champions are actually developers, with an interest and possibly knowledge of, application security.

Therefore, they could, in effect, fill the role of alternative developer. These evaluations show that the design should be changed to reflect two roles, that of the primary developer and that of the security expert reviewer. Security champions would likely fulfil the role of the security expert in this model. However, participants suggested that forum avatars may be used to distinguish comments between senior security experts and junior security experts or security champions.

Additionally, I learned a great deal about the behavior of people in both of these roles. Security experts tend to make two primary contributions. First, they contribute their security knowledge to assist in vulnerability remediation. Secondly, they often, but not always, provide guidance to developers to improve their ability to fix security vulnerabilities. Developers, on the other hand, contribute the actual code and fixes into the security code review process. They contribute an attempt at solving a vulnerability, or mark it as a false positive. Although they do not frequently ask questions, they usually explain or defend their attempted remediations or false positive markings when the security expert questions it. Developers also delegate final security decisions to experts and are happy to do so, despite actually performing the remediations themselves.

8.2.1.2 How can a tool support these roles and activities?

Once natural roles have been identified, tools can be designed to support those natural roles. I can learn a great deal about how tools can support the identified roles and activities from examining the application security expert study in chapter three and the security code review evaluations. First, tools should be developed

with the context of the security expert and the developer in mind. By examining the aggregate workflow model in chapter 3, it appears as though a tool could best support security experts by either taking the place of an audit in small organizations, fitting into an audit after scans have been performed in a large organization, or fitting into a continues integration process in organizations that employ them.

Additionally, roles should consider the actual contributions that developers and security experts make. Namely they should be designed to allow streamlined judgements from application security experts, along with contextualized comments to help train the developers. On the other end of the spectrum, they should support developer replies and avoid bloated features that would detract from the developer's time.

8.2.1.3 What are the perceptions of people in different roles of performing security code review?

If the perceptions of people from different roles in security code review are known, tools can be designed to support positive facets of these perspectives and discourage negative ones. From the application security expert study in chapter 3, I know that security experts are favorable towards security code review, but do not conduct much of it. However, they often perform manual code inspections as part of audits. From the evaluations in chapters 6 and 7, I know that security experts perceive the tool to be very advantageous for two main reasons. First, it allows them to communicate within the context of the code, greatly easing the complexity in explaining fixes. Second, it allows for quick human visual inspection of the code. This, in turn, may

save them large quantities of time, which was the primary benefit. Developers also perceived the system as necessary. Surprisingly, most felt that it would add a small time cost, but would be greatly outweighed by the security benefits. A few of them even considered the tool to save time for them. They explained that this was due to being able to address the issue immediately, within the context of the code, rather than later during an audit. Both roles expressed concern over excessive false positives, and some developers feared stagnated reviews in cases of insufficient security expert human resources.

8.2.1.4 What type of training is required for developers to effectively perform security code review?

Knowing whether or not training is needed, and the type of training needed is critical in the implementation of any security code review tool. If certain types of training are required and not properly administered, the tool will not be adopted. Both security experts and developers were asked about what kind of training would be necessary for developers to perform their role of the process in chapters 6 and 7. I found that security experts tended to think more training was needed than developers. The majority of both roles felt that some training was required. Experts felt that basic knowledge of the most common vulnerabilities and how to solve them was needed. Developers also stated this, but were more likely than security experts to say less training or no training at all was needed. However, in spite of these opinions, both roles were very effective at their tasks with very little training. Therefore, training is likely not required for developers or experts to perform security code reviews.

8.2.2 Communication and Collaboration

8.2.2.1 How do participants in each role communicate and collaborate with each other regarding code review information?

How people from the various roles naturally communicate and collaborate with each other is extremely important. If this is known, tools can be developed to support these channels. I learned from chapters 6 and 7 that participants in both roles greatly prefer inline contextualized commenting to communicate and collaborate with one another to exchange code review information. However, I also learned that much communication is not textual. Many warnings were not accompanied by comments, and the judgement on the warning was sufficient to convey the message. Other times, comments were frequently responded to by a click of the "ack" or "done" buttons. Participants in both roles spoke highly of this system, often considering it to be their favorite part of the tool. Participants from both roles liked to reply, even in a lab study without real responses. They also tended to treat this commenting as a conversation. Therefore, this system is highly recommended for security code review tools.

8.2.2.2 Which features in a tool are most effective for collaboration between people in three roles and security experts?

Based on who participants naturally communicate and collaborate, certain features will be more effective for collaboration than other features. From the evaluations in chapters 6 and 7, it is clear that contextualized warnings coupled with contextualized inline commenting and the option of single button acknowledgements was an

extremely effective tool for collaboration between both roles. Participants often indicated as such when asked about their favorite features and favorite part of the tool. However, they also indicated that if the issue was sufficiently complex, different out of band channels would need to be employed. They provided many such options. The most common was a phone call or face to face meeting, followed by a real time global chat. Thus, options to automatically move discussion may be useful and link outcomes back to code.

8.2.2.3 How should feedback from other participants be displayed to users in each role?

It is clear that feedback from other users should be displayed as close to the code as possible. This works well with static analysis, but will become more challenging if additional scanners are used. I now know that security experts have serious organizational time shortages from chapter 3. I know that developers prefer to indicate security decisions in an IDE environment and require human verification from chapter 4. Lastly, I know that contextualized feedback with the code was highly favored in chapters 6 and 7 from the security code review tool evaluations.

8.2.3 Warnings

8.2.3.1 How to effectively communicate static analysis warnings, remediations, etc to code reviewers?

One major concern is how to effectively communicate static analysis warnings and suggested remediations to people in both roles conducting code reviews. From chapter 3, I know that security auditors need to see static analysis warnings at the proper time

in the development process. I also know that they are extremely short on time, do not like to use different tools for the same task and prefer options to be as cross platform as possible. Likewise, from chapter 4, I know that developers prefer autogenerated remediations whenever possible, followed by concrete human verification. In chapters 6 and 7, I learn that static analysis warnings with autogenerating code solutions in the IDE, followed by contextualized human verification in a code review was well liked by both roles. Therefore, this body of work suggests that the most effective means of communicating these warnings is to time them at the end of an audit, or ideally after integrated static analysis scans. Additionally, a cross platform, contextualized code-like environment should be maintained as the backdrop for presenting resolved static analysis warnings to both security experts and developers.

This process worked because remediations were displayed to security experts and developers. It is unclear if it would work without these remediations being displayed. It may be possible to use annotations or comments to generate these warnings in security code review if remediation-applying tools, such as ASIDE, are not used.

8.2.3.2 How do code reviewers interpret and respond to different kinds of security warning and vulnerability information?

Understanding how code reviewers interpret and respond to different kinds of security warnings and vulnerability information is critical in understanding how to design code review tools to communicate that information. I know from the interactive annotation evaluation in chapter 4 that developers are effective at providing useful and correct input for security tools in the IDE. However, they have low confidence doing so

and often did not understand what they were doing. From the evaluations in chapters 6 and 7, I know that developers greatly appreciated the human verification provided by security experts and appeared to have much higher confidence in their answers after this verification. Generally, developers accurately assessed expert judgements indicating an issue requires modification or is correct. However, they were confused as to what they should do when "leave unresolved" was provided as an answer. Many considered it to be a low priority issue that needed to be fixed rather than an issue the expert did not have time to properly evaluate. Experts tended to see warnings produced by the developer as "high," "medium" and "low" rather than "ignored by developer" "marked false positive" and "resolution attempted." However, they were consistently effective at catching issues where the developer had incorrectly marked issues as false positives or had incorrectly attempted resolutions. This was the case, even when the developer provided misleading comments.

Thus, I can conclude that developers can effectively make security decisions, but have low confidence. Security code review can provide this confidence as well as a degree of human verification that increases the accuracy of the remediations. Warnings should include some explanation for "leave unresolved" security expert judgements. Warnings colors should match warning severity if possible, or instead provide ways of indicating severity and prioritization of warnings. Warnings in which developers explain incorrect decisions are less likely to slip past security experts than warnings in which the developer makes no comment.

8.2.3.3 How do code reviewers resolve security vulnerabilities through code review?

People participating in security code review must resolve security vulnerabilities somehow. From the application security expert study in chapter 3, I know that application security experts tend to look for code smells and conduct risk assessment during security code reviews. They tend to make remediation decisions later, and support the developer during remediation if the developer has questions. From the interactive annotation evaluation in chapter 4 and studies outside of this work, I know that developers prefer to solve vulnerabilities immediately, with the aid of a tool in the IDE. From chapters 6 and 7, I know that developers prefer to resolve security vulnerabilities as quickly as possible after the human verification is complete, as evidenced by their concerns for stagnation and hopes of avoiding an audit. Ideally, the code review platform should be tightly coupled to the IDE so that developers can perform fixes right away, and submit the code into a repository for deployment. I did not examine that second interaction, but that might be important for future studies in security code review to consider.

8.2.3.4 How effective are they?

Understanding the effectiveness of the security code review process for the detection and remediation of security vulnerabilities is a complex question. However, it's answer is critical to the design and adoption of security code review tools. I know, from chapter 4, that developers could provide input to security tools with 77% accuracy, despite having a poor understanding of what they were doing. From the evaluations

in chapter 7, I know that developers correctly generated solutions for static analysis warnings with around 70% accuracy. It was best for input validation and slightly worse for output encoding. In chapter 6 security experts judged resolved vulnerabilities, ignored vulnerabilities, and vulnerabilities marked as false positives. I know from that evaluation that vulnerabilities that security experts made successful judgements in 71% of cases with an average time of around 1 minute and twenty seconds. For many of the ones that developers did not initially judge correctly, comments were provided that may have increased the accuracy if responded to by a developer. Likewise, some warnings that developers marked incorrectly may be caught and resolved by experts in security code reviews. Additionally, most security experts and developers were asked if they felt that their vulnerability remediations or vulnerability remediation advice was more likely to be correct with a security code review tool than without one. They were also asked if they thought that it was more likely to be correct with both static analysis and a security code review tool than static analysis alone. The mast majority of participants answered yes to both questions. From these results I can conclude that security code reviews do not catch all issues and should not be expected to do so. However, they can be expected to catch a majority of vulnerabilities, are unlikely to greatly slow down development, do not require much training, may actually train the developer as a side effect, and may actually speed up the development process. This would occur if it took the place of audits or resulted in a significant amount of vulnerabilities being fixed immediately rather than during an audit.They also greatly enhance the human component which is often leveraged during manual code inspections. Additionally, they are not perceived with much negativity unless

the underlying static analysis produces many false positives or the security experts do not review code quickly. Ultimately, the effectiveness of the process can only be judged in a real development environment. The current studies are encouraging and should influence tool developers to support communication around security.

8.2.3.5 What is the role of security experts in security code review?

Ultimately, security experts can be expected to assume some sort of role during the security code review process. If that role is understood, tools can be designed to support that role. After examining the results from chapter 3, it becomes apparent that security experts typically configure security tools and answer questions when the developer needs assistance. From the evaluations in chapter 6 and 7, I observed similar behavior among security experts. Security experts tended to answer developer questions when they were asked, avoided initiating questions, and mostly verified that fixed vulnerabilities were resolved correctly. From the interviews in chapters 6 and 7, I know that both security experts and developers considered the role of the security expert to be the person who is responsible for the security of the application. Security experts perceived the role to be that of finding vulnerabilities. Developers also perceived this to be their primary role but they perceived it to be more of "signing off" on each vulnerability while acting as a security resource. This is interesting, as it reinforces the notion that, despite the ability to supply decently accurate security inputs, developers have low confidence with security related decisions and therefore wish to delegate responsibility to the security expert. Security code review tools are in a unique position to support this role since they can allow a security expert

to quickly evaluate each developer remediation while placing the actual remediation burden on developers. Interestingly, developers also considered developer training to be a secondary goal for security code review in some cases, suggesting that they embrace the task of vulnerability remediation.

8.3 Design Guidelines

Now that the natural workflows and challenges of application security experts has been examined, along with two security code review tool evaluations, design guidelines can be suggested with reasonable confidence.

It is important to point out that I am making some assumptions about security code review. Fist, I am assuming that it should be done separately from a functional code review. Second, I am assuming that participants of a security code review will participate in some sort of role. Third, I am assuming that time from the part of security experts and developers is an extremely valuable resource, and the code review should be lightweight in nature.

Warnings:

- Security code review tools should contextualize static analysis warnings to the line of code where the warning applies.
- Emphasis should be placed on the visibility and location of warnings over the appearance of the warnings.
- Static analysis warnings in security code review should be prioritized based on the expected severity of the warning. Priority should be based on vulnerability

type and ease of exploitation.

Features:

- Security code review tools should feature contextualized comments with a one-button response option as a primary method of communication.
- Security code review tools should be designed based on a per code change utilization pattern.
- Security code review tools should be platform independent.

Non-Functional Design Considerations:

- Security code review tools should be designed to minimize the amount of user effort and total time of use for both experts and developers.
- Security code review tools should only be used if the amount of false positives from static analysis is low, or they will become tedious.

8.4 Future Work

Analysis of both of the studies in chapters 6 and 7 have led to direct answers for my research questions detailed here in chapter 8. Furthermore, it has resulted in a set of design guidelines also detailed here in chapter 8. I believe this has brought forth newfound knowledge which can be leveraged to detect and remediate security vulnerabilities with a minimum amount of effort, and therefore cost. However much additional future work remains to be done.

First, my evaluation of application security experts in chapter 3 revealed that security experts are an extremely limited resource. Although security code review tools can potentially save time by avoiding the need for audits or reducing the amount of effort spent manually inspecting code, they are not a magic bullet for freeing up security expert's time. Many more advances in other application security tool technologies are needed. First, additional work should be conducted to examine precisely the kinds of tasks that are consuming the time of security experts and are most tedious. With this knowledge, tools can be designed to automate these processes as well.

Further research in security code review should target this key challenge. Studies focused on the use of the security code review process to reduce the workload on security experts and increase the security of the application should focus on this process in a real world setting. Interactive static analysis tools are gaining support, but are not as frequently used as standard static analysis tools. Studies should focus on techniques for supplying warning resolution information to the security code review process through standard static analysis techniques. Annotations may be a good candidate for this, but other options should be considered.

Additionally, now that security code review fed by static analysis techniques has been shown to be an effective means of detecting and remediating vulnerabilities, further studies in this area should focus on applying this to real world development and security processes identified in chapter 3. Work in this chapter demonstrates that different organizations employ different security practices, although some trends stand out. Future work should examine security code review when it is added into these

practices. It is assumed that security code review would be best after a functional code review, but this may not be the case. Likewise, results from developer perceptions in chapter 7 suggest that the value of a security code review drops off greatly as time passes after the code is developed. Organizations may benefit from understanding exactly when a review becomes "stagnated" and when it is considered timely to maximize the benefits of the security code review. Therefore, future work should examine this key aspect.

Security breaches continue to occur, cost hundreds of billions of dollars in global damage, shatter lives, and close businesses. Security vulnerabilities are largely responsible for these breaches. I have examined the application security experts who are tasked with catching and fixing these vulnerabilities to stop these breaches. I have identified what sort of application security practices and workflows they use, as well as what challenges they are faced with and the interactions they have. I have designed a security code review tool fed by interactive static analysis warnings to partially offset some of the identified challenges, namely that of limited human resources. I have found that the tool is potentially effective at saving time and catching and remediation security vulnerabilities. Perhaps by developing innovative tools and processes, other challenges that application security experts face will be relieved. If only a 1% improvement is gained, tens of billions of dollars will be saved in the global economy. It is my hope that this research brings these gains, and others follow and expand them one-hundred fold.

REFERENCES

[1] Atlassian crucible website, 2016. https://confluence.atlassian.com/crucible/the-crucible-workflow-298977485.html.

[2] Checkstyle, 2016. https://maven.apache.org/plugins/maven-checkstyle-plugin/.

[3] Collaborator website, 2016. https://smartbear.com/product/collaborator/features/.

[4] Findbugs website and repository, 2016. http://findbugs.sourceforge.net/.

[5] Gerrit introduction, 2016. https://wiki.qt.io/Gerrit_Introduction.

[6] Hp fortify website, 2016. http://www8.hp.com/us/en/software-solutions/static-code-analysis-sast/.

[7] Mondrian website, 2016. http://www.niallkennedy.com/blog/2006/11/google-mondrian.html.

[8] Phrabricator website, 2016. http://phabricator.org/.

[9] Pmd, 2016. https://pmd.github.io/.

[10] Reitveld and mondrian google background article, 2016. https://code.google.com/p/gerrit/wiki/Background.

[11] Review board website, 2016. https://www.reviewboard.org/.

[12] Rietvald website, 2016. https://cloud.google.com/appengine/articles/rietveld.

[13] Secure programming definition, 2016.

[14] Department of Homeland Security National Institute of Standards and Technology National Vulnerability Database. https://web.nvd.nist.gov/view/vuln/statistics-results, 2017.

[15] What is Continuous Integration? https://aws.amazon.com/devops/continuous-integration, 2017.

[16] How Do Vulnerabilities Get Into Software? http://www.veracode.com/sites/default/files/resources/whitepapers/how-vulnerabilities-get-into-software-veracode.pdf, 2018.

[17] OWASP Organization Web Page. https://www.owasp.org/index.php/main_page, 2018.

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

[19] Y. Acar, M. Backes, S. Fahl, D. Kim, M. L. Mazurek, and C. Stransky. You get where you're looking for: The impact of information sources on code security. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 289–305, May 2016.

[20] Y. Acar, C. Stransky, D. Wermke, C. Weir, M. L. Mazurek, and S. Fahl. Developers need support, too: A survey of security advice for software developers. In *2017 IEEE Cybersecurity Development (SecDev)*, pages 22–26, Sept 2017.

[21] O. Albayrak and D. Davenport. Impact of maintainability defects on code inspections. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '10, pages 50:1–50:4, New York, NY, USA, 2010. ACM.

[22] E. L. G. Alves, M. Song, and M. Kim. Refdistiller: A refactoring aware code review tool for inspecting manual refactoring edits. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 751–754, New York, NY, USA, 2014. ACM.

[23] P. Anderson, T. Reps, T. Teitelbaum, and M. Zarins. Tool support for fine-grained software inspection. *Software, IEEE*, 20(4):42–50, July 2003.

[24] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh. Using static analysis to find bugs. *IEEE Softw.*, 25(5):22–29, Sept. 2008.

[25] A. Bacchelli and C. Bird. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 712–721, Piscataway, NJ, USA, 2013. IEEE Press.

[26] V. Balachandran. Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 931–940, Piscataway, NJ, USA, 2013. IEEE Press.

[27] M. Beller, A. Bacchelli, A. Zaidman, and E. Juergens. Modern code reviews in open-source projects: Which problems do they fix? In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 202–211, New York, NY, USA, 2014. ACM.

[28] A. Bosu and J. C. Carver. Peer code review in open source communitiesusing reviewboard. In *Proceedings of the ACM 4th Annual Workshop on Evaluation and Usability of Programming Languages and Tools*, PLATEAU '12, pages 17–24, New York, NY, USA, 2012. ACM.

[29] A. Bosu, J. C. Carver, M. Hafiz, P. Hilley, and D. Janni. Identifying the characteristics of vulnerable code changes: An empirical study. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 257–268, New York, NY, USA, 2014. ACM.

[30] M. Brandel. Cs online article, 2009. http://www.csoonline.com/article/2123602/application-security/source-code-analysis-tools–how-to-choose-and-use-them.html.

[31] C. Bronson. Iba magazine, 2016.

[32] J. Cohen. 11 proven practices for more effective, efficient peer code review, 2011.

[33] J. Cohen. Peer code review: An agile process, 2011.

[34] L. N. Q. Do, K. Ali, B. Livshits, E. Bodden, J. Smith, and E. Murphy-Hill. Just-in-time static analysis. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, pages 307–317, New York, NY, USA, 2017. ACM.

[35] M. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 1976.

[36] M. Fagan. Advances in software inspections. *Software Engineering, IEEE Transactions on*, SE-12(7):744–751, July 1986.

[37] M. E. Fagan. Design and code inspections to reduce errors in program development. *IBM Syst. J.*, 38(2-3):258–287, June 1999.

[38] S. Fahl, M. Harbach, H. Perl, M. Koetter, and M. Smith. Rethinking ssl development in an appified world. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 49–60, New York, NY, USA, 2013. ACM.

[39] G. J. Holzmann. Scrub: A tool for code reviews. *Innov. Syst. Softw. Eng.*, 6(4):311–318, Dec. 2010.

[40] C. D. Hundhausen, P. Agarwal, and M. Trevisan. Online vs. face-to-face pedagogical code reviews: An empirical comparison. In *Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education*, SIGCSE '11, pages 117–122, New York, NY, USA, 2011. ACM.

[41] P. Jaferian, D. Botta, F. Raja, K. Hawkey, and K. Beznosov. Guidelines for designing it security management tools. In *Proceedings of the 2Nd ACM Symposium on Computer Human Interaction for Management of Information Technology*, CHiMiT '08, pages 7:1–7:10, New York, NY, USA, 2008. ACM.

[42] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 672–681, Piscataway, NJ, USA, 2013. IEEE Press.

[43] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: a static analysis tool for detecting web application vulnerabilities. In *2006 IEEE Symposium on Security and Privacy (S P'06)*, pages 6 pp.–263, May 2006.

[44] M. Lavallée and P. N. Robillard. Why good developers write bad code: An observational case study of the impacts of organizational factors on software quality. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 677–687, May 2015.

[45] V. B. Livshits and M. S. Lam. Finding security vulnerabilities in java applications with static analysis. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, SSYM'05, pages 18–18, Berkeley, CA, USA, 2005. USENIX Association.

[46] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan. The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 192–201, New York, NY, USA, 2014. ACM.

[47] A. Meneely, A. C. R. Tejeda, B. Spates, S. Trudeau, D. Neuberger, K. Whitlock, C. Ketant, and K. Davis. An empirical investigation of socio-technical code review metrics and security vulnerabilities. In *Proceedings of the 6th International Workshop on Social Software Engineering*, SSE 2014, pages 37–44, New York, NY, USA, 2014. ACM.

[48] S. Morgan. Is poor software development the biggest cyber threat? https://www.csoonline.com/article/2978858/application-security/is-poor-software-development-the-biggest-cyber-threat.html, 2015.

[49] S. Müller, M. Würsch, T. Fritz, and H. C. Gall. An approach for collaborative code reviews using multi-touch technology. In *Proceedings of the 5th International Workshop on Co-operative and Human Aspects of Software Engineering*, CHASE '12, pages 93–99, Piscataway, NJ, USA, 2012. IEEE Press.

[50] E. Murphy-Hill, T. Zimmermann, C. Bird, and N. Nagappan. The design space of bug fixes and how developers navigate it. *IEEE Transactions on Software Engineering*, 41(1):65–81, Jan 2015.

[51] N. Nagappan, B. Murphy, and V. Basili. The influence of organizational structure on software quality: An empirical case study. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 521–530, New York, NY, USA, 2008. ACM.

[52] D. C. Nguyen, D. Wermke, Y. Acar, M. Backes, C. Weir, and S. Fahl. A stitch in time: Supporting android developers in writingsecure code. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 1065–1077, New York, NY, USA, 2017. ACM.

[53] S. Panichella, V. Arnaoudova, M. Di Penta, and G. Antoniol. Would static analysis tools help developers with code reviews? In *Software Analysis, Evolution and*

Reengineering (SANER), 2015 IEEE 22nd International Conference on, pages 161–170, March 2015.

[54] A. Poller, L. Kocksch, S. Türpe, F. A. Epp, and K. Kinder-Kurlanda. Can security become a routine?: A study of organizational change in an agile software development group. In *Proceedings of the 2017 ACM Conference on Computer Supported Cooperative Work and Social Computing*, CSCW '17, pages 2489–2503, New York, NY, USA, 2017. ACM.

[55] M. Rouse. Tech target application security definition, 2016.

[56] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspan. Lessons from building static analysis tools at google. *Commun. ACM*, 61(4):58–66, Mar. 2018.

[57] C. Sadowski, J. van Gogh, C. Jaspan, E. Söderberg, and C. Winter. Tricorder: Building a program analysis ecosystem. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 598–608, Piscataway, NJ, USA, 2015. IEEE Press.

[58] Y. Smeets. Improving the adoption of dynamic web security vulnerability scanners. In *Master's Thesis, Radboud University*, 2015.

[59] J. Smith. Identifying successful strategies for resolving static analysis notifications. In *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16, pages 662–664, New York, NY, USA, 2016. ACM.

[60] 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.

[61] T. Thomas, B. Chu, H. Lipford, J. Smith, and E. Murphy-Hill. A study of interactive code annotation for access control vulnerabilities. In *Proceedings of the 2015 IEEE Symposium on Visual Languages and Human-Centric Computing*, VLHCC '15, Washington, DC, USA, 2015. IEEE Computer Society.

[62] T. W. Thomas, M. Tabassum, B. Chu, and H. Lipford. Security during application development: An application security expert perspective. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI '18, pages 262:1–262:12, New York, NY, USA, 2018. ACM.

[63] P. Thongtanunam, X. Yang, N. Yoshida, R. G. Kula, A. E. C. Cruz, K. Fujiwara, and H. Iida. Reda: A web-based visualization tool for analyzing modern code review dataset. In *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*, ICSME '14, pages 605–608, Washington, DC, USA, 2014. IEEE Computer Society.

[64] N. F. Velasquez and S. P. Weisband. Work practices of system administrators: Implications for tool design. In *Proceedings of the 2Nd ACM Symposium on Computer Human Interaction for Management of Information Technology*, CHiMiT '08, pages 1:1–1:10, New York, NY, USA, 2008. ACM.

[65] R. Werlinger, K. Hawkey, and K. Beznosov. Security practitioners in context: Their activities and interactions. In *CHI '08 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '08, pages 3789–3794, New York, NY, USA, 2008. ACM.

[66] R. Werlinger, K. Hawkey, and K. Beznosov. An integrated view of human, organizational, and technological challenges of it security management. *Information Management & Computer Security*, 17(1):4–19, 2009.

[67] R. Werlinger, K. Hawkey, K. Muldner, P. Jaferian, and K. Beznosov. The challenges of using an intrusion detection system: Is it worth the effort? In *Proceedings of the 4th Symposium on Usable Privacy and Security*, SOUPS '08, pages 107–118, New York, NY, USA, 2008. ACM.

[68] R. Werlinger, K. Muldner, K. Hawkey, and K. Beznosov. Preparation, detection, and analysis: the diagnostic work of IT security incident response. *Information Management & Computer Security*, 18(1):26–42, 2010.

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

[70] G. Wurster and P. C. van Oorschot. The developer is the enemy. In *Proceedings of the 2008 New Security Paradigms Workshop*, NSPW '08, pages 89–97, New York, NY, USA, 2008. ACM.

[71] J. Xie, B. Chu, H. R. Lipford, and J. T. Melton. Aside: Ide support for web application security. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, pages 267–276, New York, NY, USA, 2011. ACM.

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

[73] J. Xie, H. R. Lipford, and B. Chu. Why do programmers make security errors? In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 161–164, Sept 2011.

[74] T. Zhang, M. Song, and M. Kim. Critics: An interactive code review tool for searching and inspecting systematic changes. In *Proceedings of the 22Nd ACM*

SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014, pages 755–758, New York, NY, USA, 2014. ACM.

[75] J. Zhu, B. Chu, H. Lipford, and T. Thomas. Mitigating access control vulnerabilities through interactive static analysis. In *ACM Symposium on Access Control Models and Technologies*. ACM, 2015.

[76] J. Zhu, J. Xie, H. R. Lipford, and B. Chu. Supporting secure programming in web applications through interactive static analysis. *Journal of Advanced Research*, 5(4):449–462, 2014.

APPENDIX A: INTERVIEW/PROMPTED QUESTIONS-CHAPTER 3 STUDY

- What do you do on a typical workday?
- What are your responsibilities towards software security?
- Please explain your organization's process for catching and mitigating security issues.
- Do you directly examine code for security vulnerabilities?
- How does your work get documented?
- Do you currently participate in developer code reviews?
- How do you interact with developers?
- What do you usually discuss during those interactions?
- How often do those interactions occur?
- Who fixes the vulnerabilities when they are discovered?
- How do you participate in that process?
- Does your organization train developers in secure coding?
- If so, in what ways?
- Has this been successful in improving security?
- Do you have any ideas on how to improve upon those things?

- What are the biggest security issues and challenges your company faces with software you develop?
- What are some of the biggest challenges you face in doing your job?
- Please explain your process for catching and mitigating security vulnerabilities in code.
- What sort of tools do you typically use as a part of your work?
- What do those tools do and why do you use them?
- Is there something that you would like to be able to do, but current tools do not support?

APPENDIX B: INTERVIEW/PROMPTED QUESTIONS-CHAPTERS 6 AND 7 STUDIES

- What role potentially do you see for security code review in industry?
- What do you feel you were able to contribute through a security code review process?
- What was your favorite part of secview?
- What do you think are key features that any commercial security code review tool should support?
- How do you perceive the security code review process in general?
- What kind of training is needed for developers to perform security code review?
- How do you feel about collaboration with (developers/security experts) in security code review?
- Does it provide any advantages or disadvantages?
- If so, what advantages and disadvantages?
- What is the best channel for that communication?
- Which security code review tool features do you think are best for communicating with (developers/security experts)?
- Do you think any additional features would be helpful?

- If so, why is that?
- Did the comments and feedback from other users make sense?
- How do you think comments and feedback from others in security code review should be displayed?
- Did the warnings make sense?
- Which ones were the clearest?
- Why were they clear?
- Which were the most confusing?
- Why were they confusing?
- How do you feel these warnings should have been displayed?
- Do you feel that most of your (vulnerability resolutions/ remediation advice) is correct?
- If not, why not?
- Do you think that your (vulnerability resolutions/remediation advice) is more likely to be correct with a security code review tool than without a one?
- Do you think that your (vulnerability resolutions/remediation advice) is more likely to be correct with a security code review tool and static analysis than with static analysis alone?

- What kind of burden, if any, do you think a tool-assisted security code review process may place on the development process?
- If so, why and in what cases?
- Could the use of Secview have increased your security knowledge?
- What do you think the role of security experts should be in security code reviews?
- How can this role best be supported in tools?
- What did you think about Secview?
- On a scale of 1 to 10 with 1 being very unlikely and 10 being very likely, how likely do you think it is that you would use a commercial, more robust version of a tool like this in the real world?