Cesar: Visual Representation of Source Code Vulnerabilities

Hala Assal*
School of Computer Science
Carleton University

Sonia Chiasson†
School of Computer Science
Carleton University

Robert Biddle‡
School of Computer Science
Carleton University

ABSTRACT

Code analysis tools are not widely accepted by developers, and software vulnerabilities are detected by the thousands every year. We take a user-centered approach to that problem, starting with analyzing one of the popular open source static code analyzers, and uncover serious usability issues facing developers. We then design Cesar, a system offering developers a visual analysis environment to support their quest to rid their code of vulnerabilities. We present a prototype implementation of Cesar, and perform a usability analysis of the prototype and the visualizations it employs. Our analysis shows that the prototype is promising in promoting collaboration, exploration, and enabling developers to focus on the overall quality of their code as well as inspect individual vulnerabilities. We finally provide general recommendations to guide future designs of code review tools to enhance their usability.

1 INTRODUCTION

Detecting software vulnerabilities is a classic problem in computer security. Microsoft Security Response Centre defines a software vulnerability as *a security exposure that results from a product weakness that the product developer did not intend to introduce and should fix once it is discovered* [15].

Major software companies are taking the initiative to integrate security in the Software Development Lifecycle (SDLC), starting from the early stages. For example, Google has an independent *Security Team* responsible for aiding security reviews during the design and implementation phases, as well as providing ongoing consultation on relevant security risks and their remedies [1]. Microsoft has been following a security-oriented software development process since 2004. The Microsoft Security Development Lifecycle (SDL) introduces security early in the development and throughout the different stages of the traditional SDLC [16]. However, software vulnerabilities are discovered daily; close to 6500 vulnerabilities were reported in 2015.[1] Heartbleed[2] and Shellshock[3] are two prominent examples of software vulnerabilities demonstrating the importance of integrating security in the different stages of the SDLC, and conducting security code reviews to reduce chances of implementation mistakes.

Static analysis [22] is a method of software testing that can be performed throughout the different stages of the development to reduce risks of software vulnerabilities. Since it does not require the code to be executed, this method can be employed during early stages of the software when errors are less expensive to fix [9, 3]. Static-code Analysis Tools (SATs) are tools that automatically analyze static-code to uncover vulnerabilities. However, despite their benefits [9], SATs are not widespread in the Software Engineering community [13], *e.g.*, due to lack of support for collaboration [12].

The cybersecurity visualization research community has acknowledged the need for using user-centered design methodologies and evaluation through the entirety of the design process of visualization tools [14]. We approach the topic of usable security for software development from the users' perspective, and take a user-centered approach in designing a visual analysis environment to effectively support developers analyze the security of their code. Such an environment also aims to support collaboration amongst the development team analyzing source code vulnerabilities by encouraging discussion and exploration of potential issues.

This paper presents the following contributions.

- Study usability issues facing software developers while using code analyzers by evaluating one of the popular open-source SATs (FindBugs).
- Introduce Cesar, an initial visual analysis prototype to support what we call Collaborative Security Code Review (CSCR), where developers/testers collaborate to reduce risks of security vulnerabilities in the code under review.
- Evaluate the usability and effectiveness of the prototype, and propose additional features for future design iterations.
- Provide general recommendations for designing collaborative code review tools.

In section 2, we provide a background on SATs, software visualizations, and the two usability evaluation methods used herein. In section 3, we present the usability evaluation of FindBugs. We then present Cesar in section 4 and evaluate its usability in section 5. In section 6, we present additional features for the prototype. Finally, we provide general design recommendations for collaborative code review tools in section 7.

2 BACKGROUND

2.1 Static Analysis Tools (SATs)

Current SATs are not perfect; they build a non-exact model of the code to be analyzed and this approximation inevitably leads to false negatives. A false negative occurs when a vulnerability exists in the program being analyzed, yet the tool fails to detect it. False positives are also a problem with SATs, where the tool mistakenly reports a vulnerability. Thus, SATs require human inspection of discovered vulnerabilities; SAT users need to inspect vulnerability reports produced by the tool in order to determine whether the discovered vulnerabilities are true or false positives, in addition to looking for potential false negatives. The process of classifying detected vulnerabilities is referred to as *bug triaging* [3]. For the success of the triaging process, it is important to consider how to make the best use of a SAT despite the shortcomings of both the tool (e.g., false positives and negatives) and the human user (e.g., cognitive biases and short attention span) [3]. A tool that overwhelms the user with a long list of detected vulnerabilities containing many false positives is likely to be ignored by the user or increase the chances of user errors in triaging.

Johnson *et. al.* [12] explored the different factors influencing developers' decision to use SATs to uncover vulnerabilities in their code. One of the top reasons for using SATs was that these automated tools enable developers to discover potential vulnerabilities

*e-mail: HalaAssal@scs.carleton.ca
†e-mail: chiasson@scs.carleton.ca
‡e-mail: robert.biddle@carleton.ca

[1] https://web.nvd.nist.gov/view/vuln/statistics
[2] http://heartbleed.com
[3] http://seclists.org/oss-sec/2014/q3/650

978-1-5090-1605-1/16/$31.00 ©2016 IEEE

in their code in a faster and less-effortful manner compared to manual line by line inspection. Johnson *et. al.* and Lewis *et. al.* [13] explored reasons for the underuse of SATs by developers, and they found that developers are reluctant to use SATs because they do not adequately support collaboration between team members. In addition, because of false positives, it is sometimes inefficient for developers to use these tools, especially in large projects. Many tools also require complicated steps to customize, and yet they do not fully enable developers to make the customizations they want.

Visual representations of static analysis output are features missing from most current SATs. Some of the available proprietary tools provide simple visualizations (*e.g.*, bar graphs showing the number of issues in a project), yet to the best of our knowledge no tool provides comprehensive visual analysis support. Johnson *et. al.*'s participants expressed the desire for integrating visual representations of source code vulnerabilities in SATs to support analysis.

2.2 Software Visualizations

The majority of the software visualization literature focuses on supporting software development tasks, and little work addresses software security from the developer's standpoint [11]. Research on collaborative User Interfaces (UIs) and multitouch surfaces has recently gained traction in Human-Computer Interaction (HCI) literature. In this section, we present two relevant proposals for supporting software development using multitouch surfaces, as well as two proposals for software security visualizations.

Müller *et al.* [18] explored how multitouch interfaces could support code reviews to help improve software quality. Aiming to encourage and support collaborative code reviews, their prototype combines different approaches, such as source code visualizations, code smell detection based on software metrics, and support for annotations of code. Anslow *et al.* [4] developed a source code visualization system using a large multitouch table as the interface. The aim of this system was to help developers collaborate in exploring the structure and evolution of the different versions of the software systems. The visualization system supports multiple visualizations that provide an overview of the system being analyzed with the ability to dig deeper for more details, as well as discovering problematic entities such as particularly small or large classes. Anslow *et al.* found their system encouraged collaboration and team discussion.

As for visualizations specifically for software security, Fang *et al.* [10] proposed a tool that automatically produces diagrams necessary for software security analysis tasks, such as threat modelling. The detailed diagrams are automatically generated by the tool after it collects trace data from the system during run time. The tool allows security analysts to explore the diagrams through time and with different levels of detail. Automatically generating the diagrams using this tool is significantly faster than the manual method; the proposed tool reduced the time taken to the initial diagrams from months to hours. Goodall *et al.* [11] developed a visual analysis system to allow developers to explore vulnerabilities detected in their source code. They aimed to help developers gain a better understanding of the security state of their code by providing them with a visual representation of the aggregate results from different code analysis tools. The visualization presented each source code file as a block, where the block's width depends on the number of potential vulnerabilities detected in the file it represents. Although the authors explain different use cases of their proposal, there was no user testing or usability evaluation done to evaluate its efficacy.

2.3 Cognitive Dimensions (CDs) Framework

We use the Cognitive Dimensions (CDs) framework [8] as one of the methods for evaluating the existing SAT and Cesar. The framework seeks to determine whether users' intended activities are appropriately supported by the system in question. In cases where there are a deficiencies, the designer explores how the system can be fixed and the trade-offs of different design alternatives guided by the framework. The CDs framework was designed to aid, even the non-Human Computer Interaction specialists, in evaluating the usability of their systems. Blackwell and Green [7] developed a CDs questionnaire for use by prospective users for evaluating system usability. One of the aims of this framework is to improve the quality of discussion between designers and those evaluating the design by providing a common vocabulary—the cognitive dimensions. Evaluating the usability of a system using the CDs framework consists of three main steps: (1) classifying users' intended activities, (2) analyzing the CDs, and (3) determining whether the system appropriately supports the users.

The CDs framework classifies six generic activities: incrementation, transcription, modification, exploratory design, searching, and exploratory understanding. Activities in Cesar fall under searching, and exploratory understanding. There are 13 main CDs, and more have been proposed in literature [8]. Each dimension gives a reasonably general description of an information structure. The purpose of the system defines whether it would be better for its usability to be high or low on a CD. We provide a brief description the five CDs [8] selected for the usability evaluations presented herein.

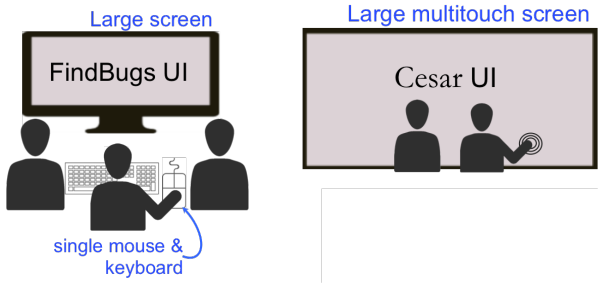
- *Viscosity* is the system's resistance to change. A viscous system requires users to perform many actions to fulfill a single task. Although viscosity could be tolerable for transcription and incrementation activities, it is harmful for modification activities and exploratory design. For ease of readability, we will rename this CD to its desirable counterpart *Fluidity (FLUI)*. Being at the opposite end of the viscosity spectrum, fluidity is useful for modification activities and exploratory design. Due to the nature of our application (code review tools), we include this dimension to relate to changes to representations of vulnerability information.
- *Hard Mental Operations* is defined as the system's high demand on users' cognitive resources. A system exhausting users' working memory makes tasks more complicated to perform. Likewise for ease of readability, we will rename this CD to its desirable counterpart *Low Cognitive Load (LCOG)*, where a system does not place a high cognitive load on the user.
- *Abstraction (ABST)* is whether the system uses abstraction mechanisms and the types of abstractions used. Abstractions help make information structures more succinct and could reduce viscosity. Employing the proper level of abstraction is useful for exploration activities.
- *Closeness of Mapping (CLOS)* is providing a match between representations of information and its domain in a way that allows users to build on their domain knowledge to solve problems.
- *Visibility and Juxtaposability (VIJU)* is the ability to view components easily and to view any two components side-by-side. This CD is useful for transcription and incrementation activities, and is especially useful for exploratory design.

FLUI, *LCOG*, *ABST*, *CLOS*, *VIJU* are all particularly desirable cognitive dimensions for exploratory design.

2.4 Cognitive Walkthrough (CW) Evaluation

The Cognitive Walkthrough (CW) [20] is a method used for evaluating the usability of systems from the perspective of users. It focuses on evaluating the system learnability by focusing on users' cognitive activities to ensure the ease of system learning through exploring the interface. Users' tasks are identified, and one or more evaluators work through the steps to perform these tasks providing suggestions to improve the system learnability. This method has been used in several usability evaluation studies; *e.g.,* Allendoerfer *et al.* [2] evaluated the usability of their visualization system using the CW methodology.

In our usability evaluation studies, we combine both the CDs framework and CW methodology. We evaluate each interface using the CW methodology, where evaluators go through the different



(a) FindBugs (b) Cesar

Figure 1: Cognitive Walkthrough session setup.

tasks provided by the interface, to take advantage of the in-context discussion among the evaluators. Next, we reflect on the results of the CW, and evaluate the interface using the CDs framework.

3 FindBugs' Study

After surveying different open source and proprietary SATs [19], we chose to evaluate the usability of FindBugs.[4] It is a popular open source tool used in Microsoft SDL and is widely used in similar research projects [12, 5, 23]. This study does not focus on FindBugs' underlying vulnerability detection mechanisms, rather on the tool's UI as it is the element with which developers interact. FindBugs analyzes Java code to detect potential vulnerabilities, and divides them into nine categories, *e.g.*, Security, Malicious Code, and Performance. Each vulnerability has two metrics: "Rank" indicating its severity and "Confidence" indicating the tool's confidence that it is an actual issue.

3.1 Study Design

We conducted a CW of FindBugs v.2.0 with a group of four evaluators who are experts in the fields of security and usable security. We refer to FindBugs' evaluator i as Ei_{FB}. The session lasted 90 minutes and was voice recorded. Members of the research team were present to observe and take notes during the session. FindBugs UI was displayed on a 47-inch screen using a single mouse and keyboard for interaction (see Fig. 1a). The session started with running FindBugs analysis on the source code of Apache Tomcat v.6.0.41.[5] Next, the evaluators performed some tasks to explore FindBugs' UI and its warnings of potential vulnerabilities. These tasks include:

T_1 Choose a package and view its vulnerabilities.
T_2 How many Security vulnerabilities are there in the codebase?
T_3 Choose a class and view its number of vulnerabilities.

The evaluators then focused on some warnings and worked towards classifying them as false positives or true vulnerabilities. Finally, the evaluators discussed different UI features they wished to have been available in FindBugs.

3.2 Results

We found, also aligned with previous research [12], that FindBugs' UI does not adequately support collaboration. The evaluator managing the input devices, E2$_{FB}$, was more engaged in exploring the interface and vulnerabilities than the rest of the evaluators. Communicating ideas was problematic; evaluators tried to draw each other's attention by pointing to the screen, and they sometimes resorted to asking E2$_{FB}$ to point the mouse to what they wanted to discuss by describing its position (*e.g.*, top right of the screen).

The UI has poor fluidity ($-FLUI$),[6] hard to navigate, and often the evaluators would be silent trying to determine how to perform a certain task or how to make sense of the information presented on the screen. In addition, to complete some tasks, the evaluators needed to perform many steps. For example, FindBugs presents potential vulnerabilities as a tree structure, where the details of the individual vulnerabilities are accessible via the leaf nodes. For every vulnerability the evaluators wanted to inspect in details, they had to click through all the nodes down the tree branch containing that vulnerability. This deterred the evaluators from extensively exploring vulnerability details, and likewise had a negative effect on collaboration between them, as they were very consumed in the steps that they forget to discuss the information they were shown.

The default setting of FindBugs does not maintain the codebase hierarchy ($-CLOS$); the tree structure focuses on individual vulnerabilities rather than on their distribution in the codebase. Granted the UI allows users to structure the tree by the codebase packages, however this option is hidden in the UI in a way that the evaluators did not discover throughout the entire CW session. Although the default hierarchy of the tree structure is adjustable, it was not clear for the evaluators how to perform this task and the role of some elements of the UI was not clear. For example, while trying to adjust the structure of the tree, E1$_{FB}$ said, "*I don't know what the arrow part is. [Does] the arrow means ignore everything to the right?*"

The structure of the tree, focusing on the vulnerabilities rather than the codebase, swayed the evaluators to become too consumed in the first vulnerability they viewed. This was exacerbated by the fact that in order to view other vulnerabilities, they would have to go through many steps and clicks. This led the evaluators to become absorbed by their attempt to assess a vulnerability without assessing the overall quality of the code, or thinking first about their strategy to evaluate the codebase.

There was confusion related to other aspects of the UI as well, such as the *Rank* and *Confidence* metrics. The Rank of a vulnerability is presented as an integer ranging from 1 to 20, while the Confidence is presented by colours (*e.g.*, if a vulnerability was detected with high confidence, it is marked red). After inspecting the interface, the evaluators deduced that the lower the rank, the more severe the vulnerability. They were inspecting a low rank vulnerability that was marked red. E2$_{FB}$ said, "*So lower [rank] is higher [severity]. Because it's red? and red means bad.*" However, towards the end of the CW, the evaluators noticed another vulnerability that did not match their reasoning. At this point E2$_{FB}$ exclaimed, "*Wait, hold on! I thought lower was higher [severity]! So, maybe the colour and the thing [rank] [aren't] related.*" This confusion resulted from the inconspicuousness of the Confidence metric ($-VIJU$). FindBugs' UI does not explain what the integer values or colours meant, thus the evaluators erroneously linked the colours (*i.e.*, Confidence) to the integer values (*i.e.*, Rank), and attributed them both to the vulnerability's severity. The discussion trying to decipher the meaning of, and relation between, the colours and the integer values ended by E1$_{FB}$ saying, "*I think the rank needs some kind of explanation. What's a 7 mean? [...] Is there anything in the documentation that would help? [all laughing].*"

On the other hand, FindBugs' UI allows users to adjust the size of its components, *e.g.*, they could increase the size of the code pane when they want to focus on the source code. In addition, when a user clicks on the vulnerability leaf node in the tree structure, the UI displays the details of this vulnerability in a separate pane, as well as the source code where the potential vulnerability, with the vulnerable lines highlighted. FindBugs also provides the ability to add and save textual annotations to the detected vulnerabilities.

Although FindBugs allows users to focus on specific vulnerabilities, it fails to encourage users to develop a strategy for evaluating the overall quality of the code. In addition, it does not adequately support collaboration nor exploration activities.

[4] http://findbugs.sourceforge.net/findbugs2.html

[5] http://tomcat.apache.org/download-60.cgi

[6] A $+$ or $-$ sign before a CD indicates that the interface is high or low on this dimension, respectively. A $+$ sign throughout this paper indicates an advantage of the interface.

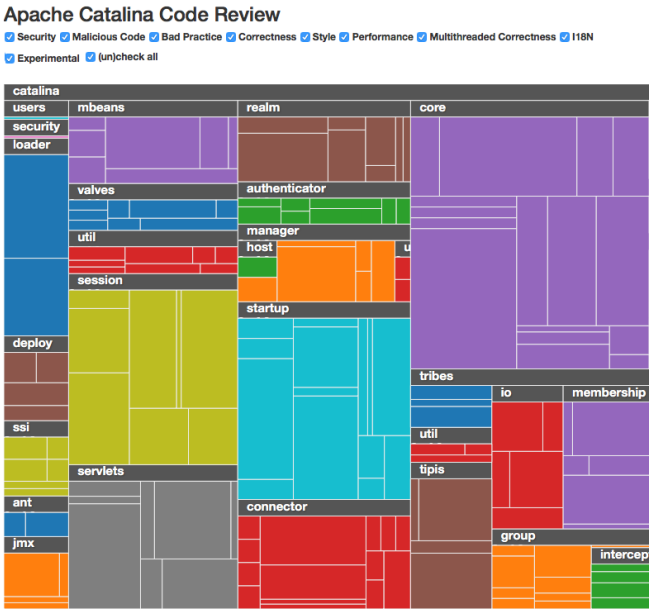


Figure 2: Cesar's treemap showing the distribution of selected vulnerability categories in package `Catalina`.

4 CESAR

We designed and implemented Cesar, a prototype aiming to leverage the benefits of SATs (*e.g.*, FindBugs), while addressing their shortcomings. We chose FindBugs for demonstration purposes, however, our general approach could be applied to the results of any SAT. The prototype was developed as a web application using JavaScript and D3.[7] This allows it to be used on many platforms needing only a browser, thus eliminating the need to install more software applications. Cesar offers a visual representation of the output of FindBugs in the form of a treemap [21], where the codebase (sub)packages are interior nodes and the classes are the leaf nodes. The size of a leaf node depends on the number of potential vulnerabilities in the class represented by this leaf node relative to the total number of vulnerabilities in the codebase. In contrast to Goodall *et al.*'s proposal [11], Cesar's treemap maintains the codebase hierarchy to which developers are accustomed to maintain closeness to the programming environment. Cesar uses a large multitouch vertical surface as an interface. Large Multitouch displays have shown promising results in supporting and promoting collaboration between team members [4].

Cesar is designed to satisfy the following objectives:

O_1 Support and encourage collaboration
O_2 Encourage exploration
O_3 Support focusing on the quality of the code as a whole
O_4 Support focusing on details of specific vulnerabilities

The first step to building the prototype was to run FindBugs analysis on a codebase. We analyzed the `Catalina` package of Apache Tomcat written in Java. However, our implementation is extensible to source code in any programming language that is organized in a hierarchical structure, either implicitly through the language (*e.g.*, Object-oriented Programming (OOP)) or through the programmers' file organization. The result of FindBugs' analysis is an XML file of potential vulnerabilities in the software analyzed; the file contains each vulnerability's name, its severity and priority, the category under which it falls, and information about its location in the codebase. However, the file is arranged by vulnerabilities, ignoring the codebase hierarchy. Thus, we extracted the XML file and built a JSON file in the proper format for use by Cesar, maintaining the codebase hierarchy. We also added the description[8] of every detected vulnerability to the JSON file.

Through checkboxes above the treemap visualization pane, Cesar enables developers to choose the categories of issues they want to include in the visualization. Figure 2 shows Cesar with all the categories included in the treemap visualization. The visualization is interactive, *i.e.*, the treemap view is adjusted in real-time to add/remove each category the user selects/unselects. However, although the relative size of the treemap rectangles change with changing the categories, the rectangles remain in the general vicinity to help developers maintain perspective.

Users can change the view of the treemap to focus on a subpackage by tapping it. The visualization thus zooms-in on that particular package filling the entire treemap pane with it, as in Figure 3 showing sub-package `catalina.realm`. When the user zooms to the class level, they can view the number of vulnerabilities in each class. Each rectangle is labelled with x of y, where y is the number of issues in a class and x shows how many of those belong to the selected categories. For example, in Figure 3, class `catalina.realm.RealmBase` is labelled (1 of 9), indicating that it has a total of 9 vulnerabilities, and only one of them is of the selected category (Malicious Code).

When the user performs a long tap on a class, two additional panes appear: "details" and "source code". The former lists all vulnerabilities (in the tapped class) that belong to the selected categories. Vulnerabilities are grouped by categories, and a brief description of each vulnerability is available. The description is collapsable to save screen space. The "source code" pane displays the class's code, highlighting vulnerable lines. As shown in Figure 3, the "details" pane lists the only Malicious Code vulnerability in class `catalina.realm.RealmBase`, whereas the "source code" pane displays its code with the vulnerable line highlighted.

The current version of Cesar classifies the treemap using FindBugs' set of categories (e.g., Security, Performance). However, this set could be customized to best fit the code review goals. For a security-focused code review tool, the set (Cesar's checkboxes) would include exclusively security vulnerability categories (e.g., Buffer overflow, Cross-site scripting).

5 CESAR'S STUDY

To analyze how well Cesar fulfills its objectives, we conducted a CW of its UI and used the CDs as the evaluation framework.

5.1 Study Design

We held two independent CW sessions, each with two evaluators. The four evaluators were recruited participants with industry programming experience and high experience in Java programming. The evaluators performed all the tasks on the prototype, while members of the research team observed and took notes. The sessions were audio recorded and structured as follows. First, a researcher introduced the study to participants and explained how a CW is conducted, then the participants started working together interacting with the prototype on a 75-inch vertical multitouch screen (see Fig. 1b) to get an idea of how it works. The researcher then gave participants some tasks to perform. After all the tasks were done, each participant independently filled out a short survey soliciting their opinion of the prototype. The researcher then interviewed the pairs to discuss their opinion of the prototype and their recommendations. Both the survey and interview questions were discussing the CDs, and were adapted from the CDs questionnaire [7].

We followed the CDs framework when designing the study. We categorized activities done using the prototype according to the

[7] `https://d3js.org`

[8] `http://findbugs.sourceforge.net/bugDescriptions.html`

Figure 3: Cesar's visualization, details, and source code panes.

CDs framework into *exploratory understanding* aiming to advance developers' understanding of the overall security-level of the program (*e.g.*, identifying packages and classes that contain the most security vulnerabilities) and *searching activities* (*e.g.*, looking for the explanation of the vulnerability type). We divided the tasks into "get to know your system" tasks where participants spend some time exploring how Cesar works, followed by some "specific" tasks that are representative of all that could be conducted using the prototype, and finally some "general" tasks which allows them to reflect more on Cesar's overall purpose.

The "Get to know your system" tasks included:

- KT_1 Select (one or more) categories of bugs to see their distribution in the code base
- KT_2 Zoom in one package to see the number of bugs
- KT_3 Change the bug categories selected to see the difference in the number of bugs
- KT_4 Display the source code for a class
- KT_5 Find the different bug types present in a class

The "Specific" tasks included:

- ST_1 Find out how many security vulnerabilities in `catalina.core` package
- ST_2 Find out how many packages have security vulnerabilities?
- ST_3 Find out which (sub)package is the least/most vulnerable?
- ST_4 Find out which line of code has Malicious code vulnerabilities in class `catalina.realm.RealmBase`?
- ST_5 Find out what does "MS: Field isn't final but should be (MS_SHOULD_BE_FINAL)" mean?

For the "General" tasks, we asked the evaluators to imagine they are in charge of approving this software for deployment. We asked them to describe how they would do an appraisal of this software and how would they prioritize which vulnerabilities to fix.

5.2 Cesar's Strengths

In this section, we analyze the sessions' outcomes based on the CDs, and discuss how the prototype fulfills each of its objectives (Section 4). In Section 6, we discuss improvements to address some of Cesar's weaknesses and enhance the user experience. Cesar's CW evaluator *i* is referred to as E*i*.

Participants rated how Cesar fulfills objectives 2-to-4 on a scale of 1 (*very well*) to 4 (*not at all*).[9] Survey responses were positive, with mean scores of 1 for O_2, 1.75 for O_3, and 1.25 for O_4.

[9]Participants did not numerically rate Cesar with respect to O_1, rather we rely on the verbal discussion of their opinion and our observations.

O_1 Support and encourage collaboration. In contrast to a personal computer with a mouse and keyboard, the large multitouch interface offered all evaluators the same view and level of control over the interface. In FindBugs, the evaluator managing the input devices was more engaged than the others (Section 3.2), whereas Cesar's evaluators were almost equally engaged in interacting with the interface. They were actively discussing their understanding of the different parts of the interface, as well as discussing the steps they thought were necessary to perform the different tasks and the implications of the different types of vulnerabilities. At no point during Cesar's CW sessions was one evaluator monopolizing the interface while the other stood silent. E3 mentioned, "*since it's touch, the control is accessible. You don't have to hand over the mouse or anything, so that's good.*" When an evaluator successfully reached a desired view, they were keen to return to the main view to show their teammate the steps followed to reach that view. As an evaluator was waiting for their teammate to complete interacting with the interface, they were focusing on the steps taken by their teammate and they would sometimes say words like "aha" expressing that they have discovered something. The evaluators attributed this to the natural feel of the interface and how easy it is to move from one view to another. This implies that Cesar has high fluidity (+*FLUI*). In addition, because the interface did not exhaust the evaluators' working memories (+*LCOG*), *e.g.*, it does not require them to carry information from one step to the next, the evaluators did not feel that discussing the steps with their teammates would break their train of thought, and so were able to discuss and share information before moving to the next step.

O_2 Encourage Exploration. Due to the abstract nature of the visualization (+*ABST*) and the way the treemap structure maintained the codebase hierarchy familiar to developers (+*CLOS*), the evaluators were stimulated to explore the different aspects of the interface as soon as they started interacting with the prototype. In addition, E2 mentioned, " *[the interface] supports exploration pretty well, just by the nature of touching the things that are on the screen, like that encourages you to go back and look at other categories and compare.*" The flexibility of the system and the fluidity of switching from one view to another (+*FLUI*) helped the evaluators not feel reluctant to change a view and explore more. For example, when the task was to find the line number that contains a "Malicious Code" vulnerability in a specific class (ST_4 above), the evaluators returned to the main view after inspecting that class, and looked for the package that had the most "Malicious Code" vulnerabilities even though this was not an assigned task.

The interface reduces cognitive load (+*LCOG*) when performing exploration tasks. For example, the evaluators would explore which packages were most/least vulnerable by looking for the visually biggest/smallest areas in the treemap when only the "Security" category was selected. The prototype thus provides a quick overview as opposed to Findbugs where users would have to go through each package in the tree structure and look for the number of security vulnerabilities in each one. Cesar's evaluators also mentioned that the treemap allowed them to easily discover areas where there are many problems and to look for different trends in the code. The structure of the interface guided the evaluators to continuously consider the overview along with the detailed view, and the interface induced them to continuously explore available information and ask themselves questions such as "*What happens if we uncheck this vulnerability category?*", "*Why is this package so big?*", and "*Why does this class have all these security vulnerabilities?*".

O_3 Support focusing on the quality of the code as a whole The different packages and classes in the codebase are represented with rectangles in the treemap and the relative area of the rectangles represents the number of potential vulnerabilities relative to the total number of vulnerabilities in the codebase. This form of abstraction (+*ABST*) helped promote the focus on the overall quality of the codebase. For example, the evaluators were not drawn into individual vulnerabilities before acquiring an overall view of the quality of the codebase. Instead, they initially developed a strategy for appraising the software, and then compared the areas of the rectangles to each other and to the overall area of the visualization in order to begin exploring the most problematic packages. E1 said, "*I think we did take a step back to think, 'okay what [is] our approach, do we stay here in a details view or do we go back to the overview, or you know should we filter things more, filter things less'. So, we kind of took a step back I think before every task to kind of figure out what our strategy is.*"

We mentioned earlier that the interface guided the evaluators to continuously consider the overview of the code base along with the detailed view. This was accomplished by the ease of starting from the main view containing the package in question and the ability to zoom-in on it to get more details, where the level of zoom depends on the detail depth. For example, in order to perform task ST_4, evaluators started from the main view, showing the `catalina` package, zoom in on the `realm` package, and then displaying the vulnerabilities and source code of the `RealBase` class. Doing so, the evaluators were able to evaluate the state of the `realm` package with respect to the codebase before digging deep for the more detailed information. The ease of switching between views (+*FLUI*) prevented distraction from the evaluators' objective and focused their attention on the overall code quality. Nevertheless, the interface does not force users to dig through the interface for every task, as it allows for general exploration tasks, such as finding the most vulnerable package, without delving into details. The evaluators mentioned that the interface was helpful in allowing them to gain a general understanding of how vulnerable the codebase is, and that it does this in a more interesting way than going through a list of potential vulnerabilities. E4 said, "*I think it's good that it's pictorial and that people can discuss on it [...] It's not a pile of text that I'm going through. It's not a list of errors.*"

The interface allows for some shortcuts, *e.g.*, it allows users to display vulnerability details and source code of any class by a long tap on its leaf node's rectangle from any view. This could be useful, for example, in case a user wants to explore the class that has the most vulnerabilities, which would be represented by the biggest rectangle in the treemap, or for someone who memorized the location of a class on the treemap. We discuss more shortcuts that we anticipate to be useful for advanced users in Section 6.

O_4 Support focusing on details of specific vulnerabilities When discussing their strategy for appraising the codebase, the evaluators mentioned how they would use filters provided by the interface and zoom-in on important packages to assess how well the codebase fulfills their coding standards. E3 mentioned, "*Depends on what our standards are, or what we consider a bug that must be fixed, or one that's maybe not that important,*" whereas E4 said, "*Maybe I would start with the security issues, maybe after I get all the security [issues] fixed, I would look at another [category], say performance,*" while tapping on the respective checkboxes. By providing filters that allowed them to inspect vulnerability categories that are more critical to them, and by adjusting the visualization in realtime (+*FLUI*) as more categories are (un)checked, the interface allowed the evaluators to narrow down their focus to those critical vulnerabilities. For example, when the evaluators did not want to inspect "Style" issues, they unchecked it to disregard it from their analysis. Maintaining the structure of the codebase hierarchy in the treemap visualization (+*CLOS*) aligned with the evaluators' familiarity with the hierarchical nature of the codebase. This helped the evaluators focus their attention on specific vulnerabilities, rather than trying to resolve to which package a class belonged.

In addition to filtering out irrelevant vulnerabilities, evaluators consistently checked the description of the vulnerabilities available through Cesar. However, their behaviour varied ; some started looking at the source code, discussing why a specific line was problematic before looking at the provided explanation,while others started with the explanation before checking the code. In either case, they all inspected the vulnerability description area, mentioned that it was useful, and that its placement close to the source code window invited them to consistently refer to it (+*VIJU*).

6 Future Enhancements

In this section, we address how Cesar may be improved in future iterations.

Display the number of vulnerabilities in a package on the treemap. Minimizing vulnerability details displayed by the treemap visualization encouraged the evaluators to inspect the overall code quality, *e.g.*, by allowing them to compare the vulnerability of a package relative to other packages or to the codebase as a whole. However, our evaluators mentioned that displaying the absolute number of vulnerabilities in the package could help them perform some tasks faster (+*LCOG*, +*VIJU*). For example, evaluator E1 said, "*It would be nice if the number of vulnerabilities that are in a package could be written next to the package [name] [...] When we were looking for the smallest package, we were like "is this the smallest?", "is this the smallest?", whereas if it was just a number, we would have been able to spot it quicker*". Thus, displaying the absolute number of detected vulnerabilities in a package beside its name would further augment users' focus on the overall code quality and speed up some tasks. For consistency, the number beside the package name should follow the same format as that displayed on class rectangles. It should be in the form of (x of y), where x is the number of issues from the selected categories, and y is the total number of vulnerabilities in the package from all categories.

Add breadcrumbs trail. To further increase the fluidity of the interface, we would use a breadcrumbs trail [17] to trace and display the hierarchy of the user's current treemap view in relation to the codebase structure. This would allow users to switch easily between package levels with a single tap and to quickly identify the location of the package/class in the treemap's current view in relation to the codebase (+*FLUI*). For example, rather than having to tap twice on the screen to zoom-out of the fourth-level sub-package (*e.g.*, package `interceptors` with full path `catalina.tribes.group.interceptors`) to the second level (package `tribes`), the user would tap on the name of the second-level sub-package in the following breadcrumbs trail: *catalina* ▷ *tribes* ▷ *group* ▷ *interceptors*. Although the evaluators did not find it annoying to have to tap on the screen multiple times

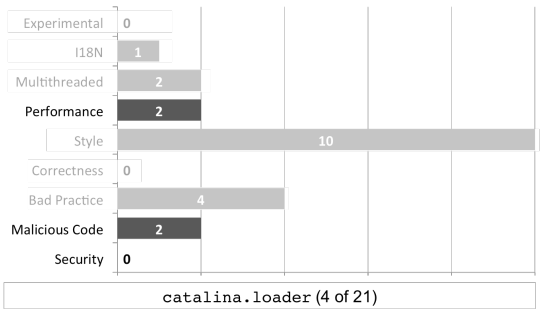


Figure 4: A secondary visualization showing the distribution of vulnerabilities in each category. Selected categories are highlighted.

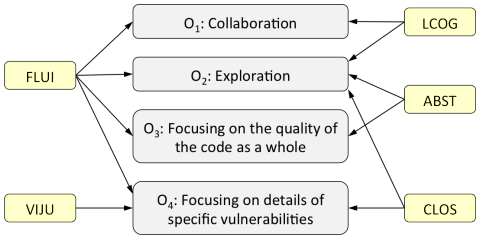


Figure 5: Relation between CDs and four select objectives of a CSCR tool. The figure can be read following the arrows, *e.g.*, the arrow from FLUI to Collaboration indicates that Fluidity supports Collaboration.

to reach a higher package, *e.g.*, evaluator E2 thought that, "*Getting in [zooming-in] is really quick and then getting back up [zooming-out] to the top is pretty quick as well*", we believe that this feature would be particularly useful as a shortcut for more advanced users and especially for large projects with deep package levels.

Use colours to represent data. In addition, as was suggested by the evaluators during both CW sessions, we could use colours to introduce another dimension of the data visualized by the treemap. For example, rather than using random colours for the rectangles, packages and classes that are more critical to the application, or those that need to be thoroughly investigated against security vulnerabilities (*e.g.*, classes handling databases), would be in warm colours, and the others in cool colours. Alternatively, packages/-classes could be coloured according to the severity of their vulnerabilities, the warmer colours indicting more severe vulnerabilities. This mechanism would provide yet another way for filtering information and minimizing the cognitive load on users (+*LCOG*).

Use a secondary visualization. Finally, to increase the visibility of the vulnerability state of the code, we propose a secondary visualization to show the distribution of different vulnerability categories in the package/class in the current treemap (+*VIJU*). Initially, both visualizations would show the distribution of all vulnerability categories in the codebase as a whole (package `catalina` in our example). In addition, we will employ the *linking and brushing* techniques [6]. The secondary visualization and the treemap will be linked, such that whenever the treemap changes, the secondary visualization will be updated in realtime to match the current treemap. For example, if the user zooms-in on a sub-package, the secondary visualization will be updated to show its distribution. Figure 4 depicts the secondary visualization as a bar graph showing the distribution of all the vulnerability categories in the `catalina.loader` sub-package. It shows the absolute number of vulnerabilities in each category on its respective bar, as well as the package name, the number of selected vulnerabilities, and the total number of vulnerabilities (+*LCOG*). The colour of the bars will match that used in the treemap—grey bars for packages and the same colour used in the treemap for classes. In Fig. 4, the user selected only a subset of vulnerability categories (Security, Malicious code, and Performance); unselected categories are faded out on the bar graph. Focusing on categories can be done by checking their checkboxes, or using a *shadow highlight brushing* operation on the secondary visualization (+*FLUI*). Brushing can be done by tapping on the category's bar, its name, or by finger-tracing around them (*e.g.*, drawing a circle). This will highlight the selected categories, and will update the linked treemap accordingly. Highlighting information about the selected vulnerability categories allows users to focus on them, while the background information helps users maintain cognizance of the overall code quality. We note that in some cases, a category with a particularly large number of vulnerabilities will occupy most of the bar, thus making it hard to see the rest of the categories. Thus, the secondary visualization will enable users to zoom-in/out to focus on the desired categories.

7 Discussion

In this section, we reflect on the results from our evaluations to discuss the effect of five CDs on achieving Cesar's objectives (see Fig. 5), and provide general recommendations for collaborative code review tools based on the Cognitive Dimensions framework.

Fluidity (*FLUI*). Fluidity is useful for *all* four objectives of a collaborative code review tool. A fluid interface does not compel the user to perform many actions to fulfill a certain task, thus reduces disruptions to the code review workflow. In particular, without fluidity, the user would be consumed in the many steps required to perform a task, negatively affecting collaboration between team members, and distracting them from the overall code quality and from analyzing specific vulnerabilities. A user absorbed by the task at hand is less likely to have discussions with their teammates that might break their line of thought, and they might be reluctant to redo all these steps to teach their teammate how to perform a certain task. In addition, being distracted by the steps they need to perform takes the user's focus off the original objective of assessing the quality of the code and inspecting vulnerabilities. A poorly fluid system would also deter users from exploring—there is only so many times a user may be willing to go through multiple steps to explore information. As evident by Cesar's CW, the UI of a collaborative code review tool should be fluid enough to invite users to explore the different aspects of the data presented to discover hidden trends, rather than settling for the most obvious explanations.

Recommendation. It is important for a code review tool to have fluidity. The tool should act as a transparent interaction medium; users should feel as if they are directly interacting with their codebase, rather than the burden of learning a new code review tool.

Low Cognitive Load (*LCOG*). A tool that reduces the cognitive load on users is valuable for collaboration and exploration. Useful information should be available to the user throughout the different steps taken to perform a task. Reducing the load on the user's working memory allows the user to be more open to discussion and exploration. A tool that provides users with opportunities for exploration without exerting too much cognitive effort is more likely to succeed in persuading them to perform deeper analyses.

Recommendation. A code review tool should conserve the user's cognitive resources to the actual objective of reviewing their codebase, determining how to secure it, and identifying critical issues.

Abstraction (*ABST*). Using the correct level of abstraction to present vulnerability information supports exploration and helps the user assess the overall quality of the codebase. Providing vulnerability information in an abstract form, while maintaining closeness of mapping as discussed below, encourages the user to explore

available information and invokes their analysis mindset. Additionally, the difference in the approach taken by the Findbugs and Cesar's evaluators demonstrates the usefulness of abstraction in preventing users from getting engrossed in the details of specific vulnerabilities without being mindful of the big picture.

Recommendation. A collaborative code review tool should use abstractions to invite the user to explore available information and to apprise the user of the overall quality of the codebase.

Closeness of Mapping (*CLOS*). Closeness of mapping of the interface elements to the domain motivates users to explore information provided by the code review tool, and concentrate on vulnerability details. Thus, a visualization that maintains the codebase hierarchy familiar to the developers allows them to build on their existing knowledge of the codebase to review their code, looking for hidden patterns and trends. In addition, it allows them to focus on the details of vulnerabilities, rather than *deciphering* how the vulnerability information presented to them relates to their code.

Recommendation. A code review tool should maintain closeness of representation to the domain of software development, *e.g.*, by maintaining the codebase hierarchy in its visualizations. Users who are familiar with the structure of the information do not have to go through the first steps of determining how to make sense of the presented data, and could delve right into analyzing this data utilizing their existing knowledge.

Visibility and Juxtaposability (*VIJU*). This dimension supports focusing on details of specific vulnerabilities. As noted by Cesar's evaluators, when vulnerability information (such as its description and location in the codebase) is easily accessible to the user without cognitive effort, the user can focus their attention on the specifics of the vulnerability, identifying its criticality and how to solve it. Components that are relevant to each other should be placed side-by-side to allow the user to easily access the required information, making comparisons and inferences.

Recommendation. Components' visibility and their placement juxtaposed allows users to direct their cognitive resources to investigating vulnerability details, making inferences and decisions, rather than on finding information in disparate parts of the interface.

8 Conclusion

We applied a user-centered approach to address the issue of usability of source code analyzers. Usability evaluation was two fold: the Cognitive Dimensions (CDs) framework and Cognitive Walkthrough (CW) method. We evaluated the usability of the UI of FindBugs, one of the most popular open source code analyzers. To address some of FindBugs' usability issues we designed Cesar, which provides developers and testers with a visual analysis environment to help them reduce risks of source code security vulnerabilities. Cesar uses a vertical multitouch display as an interface, and a treemap as its primary visualization element. The treemap presents vulnerability information while maintaining the codebase hierarchy familiar to developers. We evaluated the usability of an initial Cesar prototype, and discussed additional potentially useful features based on the evaluation. Finally, we presented general recommendations for designing collaborative code review tools.

Acknowledgements

Hala Assal acknowledges NSERC for her Postgraduate Scholarship (PGS D). Sonia Chiasson acknowledges NSERC for funding her Canada Research Chair and Discovery Grant. This work is partially funded by NSERC SurfNet.

References

[1] Google's Approach to IT Security. https://static.googleusercontent.com/media/1.9.22.221/en//enterprise/pdf/whygoogle/google-common-security-whitepaper.pdf. [Accessed July-2016].

[2] K. Allendoerfer, S. Aluker, G. Panjwani, J. Proctor, D. Sturtz, M. Vukovic, and C. Chen. Adapting the cognitive walkthrough method to assess the usability of a knowledge domain visualization. In *IEEE Symposium on Information Visualization*, INFOVIS '05, pages 195–202, Oct.

[3] P. Anderson. Measuring the Value of Static-Analysis Tool Deployments. *Security Privacy, IEEE*, 10(3):40–47, May 2012.

[4] C. Anslow, S. Marshall, J. Noble, and R. Biddle. SourceVis: Collaborative software visualization for co-located environments. In *IEEE Working Conference on Software Visualization*, VISSOFT '13, pages 1–10, Sept 2013.

[5] N. Ayewah and W. Pugh. The Google FindBugs Fixit. In *International Symposium on Software Testing and Analysis*, ISSTA '10, pages 241–252, New York, NY, USA, 2010.

[6] R. A. Becker and W. S. Cleveland. Brushing Scatterplots. *Technometrics*, 29(2):127–142, 1987.

[7] A. Blackwell and T. Green. A Cognitive Dimensions questionnaire optimised for users. In *Annual Meeting of the Psychology of Programming Interest Group*, pages 137–152, 2000.

[8] A. Blackwell and T. Green. Notational systems–the cognitive dimensions of notations framework. In *HCI Models, Theories, and Frameworks: Toward an Interdisciplinary Science*. Morgan Kaufmann, 2003.

[9] B. Chess and G. McGraw. Static Analysis for Security. *IEEE Security & Privacy*, 2(6):76–79, 2004.

[10] W. Fang, B. P. Miller, and J. A. Kupsch. Automated Tracing and Visualization of Software Security Structure and Properties. In *IEEE Symposium on Visualization for Cyber Security*, VizSec '12, pages 9–16, New York, NY, USA, 2012.

[11] J. Goodall, H. Radwan, and L. Halseth. Visual Analysis of Code Security. In *IEEE Symposium on Visualization for Cyber Security*, VizSec '10, pages 46–51, New York, NY, USA, 2010.

[12] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge. Why don't software developers use static analysis tools to find bugs? In *International Conference on Software Engineering*, ICSE '13, pages 672–681, May 2013.

[13] C. Lewis, Z. Lin, C. Sadowski, X. Zhu, R. Ou, and E. Whitehead. Does bug prediction support human developers? Findings from a Google case study. In *International Conference on Software Engineering*, ICSE '13, pages 372–381, May 2013.

[14] S. Mckenna, D. Staheli, and M. Meyer. Unlocking user-centered design methods for building cyber security visualizations. In *IEEE Symposium on Visualization for Cyber Security*, VizSec '15, pages 1–8, Oct 2015.

[15] Microsoft Corp. Definition of a Security Vulnerability. https://msdn.microsoft.com/en-us/library/cc751383.aspx. [Accessed June-2016].

[16] Microsoft Corp. Microsoft Security Development Lifecycle. https://www.microsoft.com/en-us/sdl. [Accessed June-2016].

[17] J. Mifsud. 12 Effective Guidelines For Breadcrumb Usability and SEO. http://usabilitygeek.com/12-effective-guidelines-for-breadcrumb-usability-and-seo/. [Accessed June-2016].

[18] S. Müller, M. Würsch, T. Fritz, and H. C. Gall. An Approach for Collaborative Code Reviews Using Multi-touch Technology. In *International Workshop on Co-operative and Human Aspects of Software Engineering*, CHASE '12, pages 93–99, Piscataway, NJ, USA, 2012.

[19] V. Okun, Delaitre, Aurelien, and P. E. Black. Report on the Static Analysis Tool Exposition (SATE) IV. In *NIST Special Publication 500-297*. 2013.

[20] P. G. Polson, C. Lewis, J. Rieman, and C. Wharton. Cognitive walkthroughs: a method for theory-based evaluation of user interfaces. *International Journal of Man-Machine Studies*, 36(5):741 – 773, 1992.

[21] B. Shneiderman. Tree Visualization with Tree-maps: 2-d Space-filling Approach. *ACM Trans. Graph.*, 11(1):92–99, Jan. 1992.

[22] I. Sommerville. *Software Engineering*. Pearson Education, 9 edition, November 2011.

[23] H. Zhong and Z. Su. An Empirical Study on Real Bug Fixes. In *International Conference on Software Engineering*, ICSE '15, pages 913–923, Piscataway, NJ, USA, 2015.