Why Can't Johnny Fix Vulnerabilities: A Usability Evaluation of Static Analysis Tools

ABSTRACT

Static analysis tools can detect security vulnerabilities early in the development process. To prevent security incidents, static analysis tools must also enable developers to resolve the defects they detect. Unfortunately, developers struggle to interact with the interfaces of these tools, leading to tool abandonment, and consequently the proliferation of preventable vulnerabilities. Simply put, the usability of static analysis tools matters. The usable security community has successfully identified and remedied usability issues in end user security applications, like PGP and Tor browsers, by conducting usability evaluations. Inspired by the success of these studies, we conducted a heuristic walkthrough evaluation of three security-oriented static analysis tools. Through the lens of heuristic walkthroughs, we identify several issues that detract from the usability of static analysis tools. The issues we identified range from workflows that do not support developers to interface features that do not scale. We make these preliminary findings actionable by outlining how our results can be used to improve the state-of-the-art in static analysis tool interfaces.

1. INTRODUCTION

Static analysis tools, like Spotbugs [19], Checkmarx [2], and Coverity [4] enable developers to detect code issues early in the development process. Among several types of code quality issues, developers rank security issues as the highest priority for these tools to detect [24].

To actually make software more secure, static analysis tools must go beyond simply detecting vulnerabilities. These tools must be usable and enable developers to resolve the vulnerabilities they detect. As Chess and McGraw argue, "Good static analysis tools must be easy to use, even for non-security people. This means that their results must be understandable to normal developers who might not know much about security and that they educate their users about good programming practice." [23].

Unfortunately, developers make mistakes and need help resolving security vulnerabilities due to the poor usability of security tools [25]. Recently, Acar and colleagues set forth a research agenda for remedying usability issues in developer security tools, explaining that "Usable security for developers has been a critically under-investigated area." [21]. As part of that research agenda, they call for usability evaluations of developer security tools, a research approach that has been successfully applied in the adjacent field of end-user security tools.

For instance, this paper references "Why Johnny Can't Encrypt: A usability Evaluation of PGP 5.0," in which Whitten and Tygar conduct a cognitive walkthrough evaluation to identify usability issues in an end user security tool [27]. In our study, we borrowed a similar evaluation technique, namely heuristic walkthroughs [26], to study usability issues in developers' security-oriented static analysis tools.

To that end, we conducted a heuristic walkthrough evaluation of three security-oriented static analysis tools, Find Security Bugs [9], RIPS [18], and Flawfinder [11]. To our knowledge, this study is the first to identify usability issues across multiple developers security tools using heuristic walkthroughs. As a result of this study, we identified several usability issues, ranging from missing affordances to interfaces that scale poorly. Each of these usability issues represents an opportunity to improve developers' security tools. Alongside these usability issues, we contribute our visions for how toolsmiths and security researchers can improve the usability of static analysis tools. Ultimately, by improving the usability of these tools, we can enable developers to create secure software by resolving vulnerabilities more accurately and efficiently. To support replication, we make our study setting available in a virtual machine [8], along with the study protocol [7] and the detailed list of usability issues we identified for the three tools [13].

The contributions of this paper are:

- A categorized list of usability issues that serves both as a list of known pitfalls and as a list of opportunities to improve existing tools. (Section 4)
- Design suggestions and discussion that demonstrates the actionability of this list. (Under *Discussion* throughout Section 4)
- Specifications for a heuristic walkthrough approach that researchers and practitioners can use to improve additional tools. (Section 3.4)

USENIX Symposium on Usable Privacy and Security (SOUPS) 2018.
August 12–14, 2018, Baltimore, MD, USA.

2. RELATED WORK

We have organized the related work into two categories. First, we will discuss relevant work concerning usability evaluations of end user security tools. Next, we will discuss prior evaluations of developer security tools.

2.1 Usability Testing End User Security Tools

Several studies have evaluated the usability of end user security tools. Through these evaluations, researchers identified usability issues in various end user security tools, ranging from encryption tools to Tor browsers. Collectively, these studies have improved the usability of end user security tools by contributing a better understanding of how users interact with these tools.

In their foundational work, Whitten and Tygar studied the usability of PGP, an end user encryption tool [27]. They evaluated PGP using a combination of a cognitive walkthrough and a laboratory user test to identify aspects of the tool's interface that failed to meet a usability standard. The issues revealed by their study covered topics such as irreversible actions and inconsistent terminology.

Since Whitten and Tygar's study, others have successfully applied similar approaches to study the usability of additional end user tools. For instance, Good and Krekelberg studied the usability of the Kazaa P2P file sharing system [**?**]. Their findings suggest that usability issues led to confusion and privacy concerns for users. Similarly, Gerd tom Markotten studied the usability of an identity management tool using heuristic evaluations and cognitive walkthrough [**?**].

Clark and colleagues conducted cognitive walkthroughs to examine the usability of four methods for deploying Tor clients [**?**]. Based on their evaluation, they make recommendations for facilitating the configuration of these systems. Also studying the usability of Tor systems, Gallagher and colleagues conducted a drawing study to elicit users' understanding, and misunderstandings, of the underlying system [**?**].

Like these previous studies, we are concerned with the usability of security tools and strive to better understand usability issues by conducting an empirical evaluation. We are encouraged by these studies' successful applications of evaluation techniques like cognitive walkthroughs and heuristic evaluations to end user tools. In our study, we evaluate static analysis tools to similarly identify usability issues in the domain of *developer* security tools.

2.2 Evaluating Developer Security Tools

Developers' security tools are constantly being improved, enabling them to scan more code and detect new types of defects. Like end user security tools, researchers have also evaluated the effectiveness of developers' security tools. However, the evaluations of developer security tools have primary focused on the technical aspects of the tools, such as the defects they detect and their false positive rates, rather than focusing on the usability of those tools. In this section we will summarize related work that has evaluated developer security tools. We will focus on studies, like our own, that either simultaneously evaluate multiple tools or consider usability as part of their evaluation criteria.

Several studies have conducted comparative evaluations of developer security tools. For instance, Zitser and colleagues evaluated five static analysis tools that detect buffer overflow vulnerabilities, comparing them based on their vulnerability detection and false alarm rates [**?**]. Comparing a wide range of Android security tools, Reaves and colleagues categorize tools according to the vulnerabilities they detect and techniques they use [**?**]. Their results do also include some usability experiences, such as how long it took evaluators to configure the tool and whether output was human-readable. Austin and colleagues compare four vulnerability selection techniques: systematic manual penetration testing, exploratory manual penetration testing, static analysis, and automated penetration testing [**?**]. Considering the number of vulnerabilities found, false positive rates, and tool efficiency, they conclude that multiple techniques should be combined to achieve the best performance. Finally, Emanuelsson and Nilsson compare three static analysis tools, Coverity Prevent, Klocwork K7, and PolySpace Verifier, in an industrial setting [**?**].

There have been a limited number of studies that account for usability in their evaluations of developer security tools. Assal and colleagues conducted a cognitive walkthrough evaluation of Findbugs to determine how well it helped developers determine the number of vulnerabilities in a codebase [**?**]. Based on the usability issues identified in this study, the authors created a tool, Cesar, designed to be more usable. Nguyen and colleagues describe some usability issues that affect Android lint tools to motivate the design of their tool, FixDroid, which uses data flow analysis to help secure Android apps [**?**]. However, the descriptions of Lint's usability issues are not based on a formal evaluation. Smith and colleagues conducted a user study which identified 17 categories of developers' information needs while using a security-oriented static analysis tool [**?**]. Thomas and colleagues leveraged Smith and colleagues' framework to evaluate the usability of ASIDE, an interactive static analysis tool [**?**].

Our work differs from these prior studies because we study the usability (rather than the technical capability) of static analysis tools and our usability evaluation includes multiple tools. To our knowledge, prior to our work, no study has conducted an in depth usability evaluation across multiple developer security tools.

3. METHODOLOGY

In this section we first justify our choice of tools to study and then describe the interfaces of those tools. Next, we describe the study environment, including the projects that each of the tools scanned. We also outline our heuristic walkthrough approach. Finally, we provide links to all the materials available in our replication package.

3.1 Tools

We evaluated three security-oriented static analysis tools, Find Security Bugs (FSB), Flawfinder, and RIPS. In this section we justify our choice of those three tools and describe those tools, focusing particularly on how they present information to developers.

Organizations and researchers have compiled lists containing dozens of different security-oriented static analysis tools [28, 29, 30, 31]. Combining these lists, we considered 61 candi-

Table 1: Tool interface dimensions

	Flawfinder	FSB	RIPS
Remediation Information[1]	✓	✓	✓
Trace Navigation[2]		✓	✓
Quick Fixes[3]		✓	
Graphical Rep. of Traces[4]			✓
Command-Line Interface	✓		
IDE Integration		✓	
Standalone Interface			✓

[1] Information that helps developers fix a vulnerability;
[2] Affordances that allow developers to trace dataflow;
[3] Features for applying patches automatically;
[4] Graphical representations of dataflow traces;

date tools. Because there are so many unique tools, it would be intractable for us to evaluate all types of tool interfaces. To narrow the selection of tools to use for our evaluation, we followed two criteria.

The first, and most straightforward, selection criteria was availability. We only considered tools we could access and run. There were several candidate commercial tools that fit our second selection criteria, but none had licenses that allowed for evaluation. Similarly, there were some research prototypes we could no longer configure to run.

Second, we faced the potential threat that the tools we selected would not represent the real-world diversity of tool interfaces. For instance, Lint, Jlint [12], and Pylint [17] are three distinct tools, but they share very similar command-line interfaces. To increase the generalizability of our results, we chose three tools that cover different aspects of the tool interface design space. This primarily translates to selecting tools with three different modes of interaction: command-line interface, IDE integration, and a standalone tool. Table 1 summarizes how Flawfinder, FSB, and RIPS vary along some of the interface dimensions we considered. (Note: Table 1 reflects the interfaces of the versions of the tools we chose to evaluate. For instance, FSB can also be run as a command-line tool.) The full table of tools and interface dimensions is available online [**?**].

3.1.1 Find Security Bugs

Find Security Bugs (FSB) [9] is an extension of the SpotBugs [19] static analysis tool that detects bugs in Java software. Find Security Bugs focuses on 125 different security vulnerability types described in the CWE [3] and OWASP top 10 [16] by detecting dangerous API signatures in Java programs. Like SpotBugs, Find Security Bugs is an open-source IDE plugin. We used the Eclipse [6] plugin, version 1.7.1.

Figure 1 presents the GUI of the Find Security Bugs plugin, running on the source code of Apache POI version 3.9 [1]. The list of vulnerabilities found by the tool is presented on the left view, under categories that indicate the severity of the errors: "Scary", "Troubling" and "Of Concern". In each category, the errors are grouped in sub-categories of confidence, indicating how certain the tool is that the error is a true positive. Finally, in each sub-category, the errors are grouped by vulnerability type (e.g., "Cipher with no integrity"). When double-clicking on an error, the tool highlights the line of code that causes the error in the editor. The left gutter of the editor shows bug icons to help users locate the errors. Tooltips for those icons provide information on all errors occurring at that particular line. The right gutter of the editor contains markers to other places in the file that have a bug. The bottom view provides more information about the vulnerability that is being examined. It contains a bug description, examples of similarly vulnerable code and how to fix it, links to useful information on this vulnerability, and tool-specific information. It also provides a "Navigation" panel that contains a step-by-step trace of the vulnerability. For example for an SQL injection, it reports a witness from the source to the sink.

The tool can be manually re-run by right-clicking on the project. It re-scans the entire project every time it is re-run. Aside from the list of errors shown in the IDE, it is possible to export a report of all bugs found in XML format.

With the configuration settings shown in Figure 2, it is possible to customize how the list of results is shown in the left view. Bug patterns and categories can be toggled in and out, in which case, errors matching the category will no longer show in the list. The different types of vulnerabilities can also be reclassified in different severity categories. It is also possible to include and exclude particular source files from the scan, and to choose which detectors are run by Find Security Bugs (e.g., the AndroidSqlInjectionDetector, or the AtomicityProblem detector).

3.1.2 RIPS

RIPS [18] is a static analysis tool for PHP code that can find more than 80 vulnerabilities described in the CWE/SANS top 25 and the OWASP top 10. It provides a standalone web interface from which the user can configure and launch scans, and consult the results. We used version 0.55 of RIPS.

Figure 4 presents the main screen of RIPS when run on version 2.0 of the Wordpress [20] source code. Upon finishing a scan, a pop-up presents a summary of the results with statistical data on the files scanned and the numbers of errors found. The errors are grouped by files, and in each file, ordered by vulnerability type. Each individual error is documented with a short description of the vulnerability and a code snippet in which it occurs. An icon on the left opens a code viewer view showing a non-editable version of the file containing the error. Sometimes, a "help" icon and a "generate exploit" icon are also shown on the left. The help icon opens a view in which the vulnerability is explained in more detail, with a code example and its fix. The generate exploit icon opens the view in Figure 3 in which the user can generate an example exploit for this vulnerability. The description box also contains information on other files in which the same error occurs.

On the top menu of the page, as shown in Figure 4, the user has access to additional views: The "stats" button opens the scan summary. The "user input" button shows the list of variables that may contain user-influenced inputs (e.g., $_GET[error]). The "functions" button opens a view in which the user can see the list of functions contained in the scanned files, and a call graph illustrating which functions call each other. The user can edit certain parts of the graph: rearrange the node layout, modify the function names, and edit the edges between the different functions. The "files" button

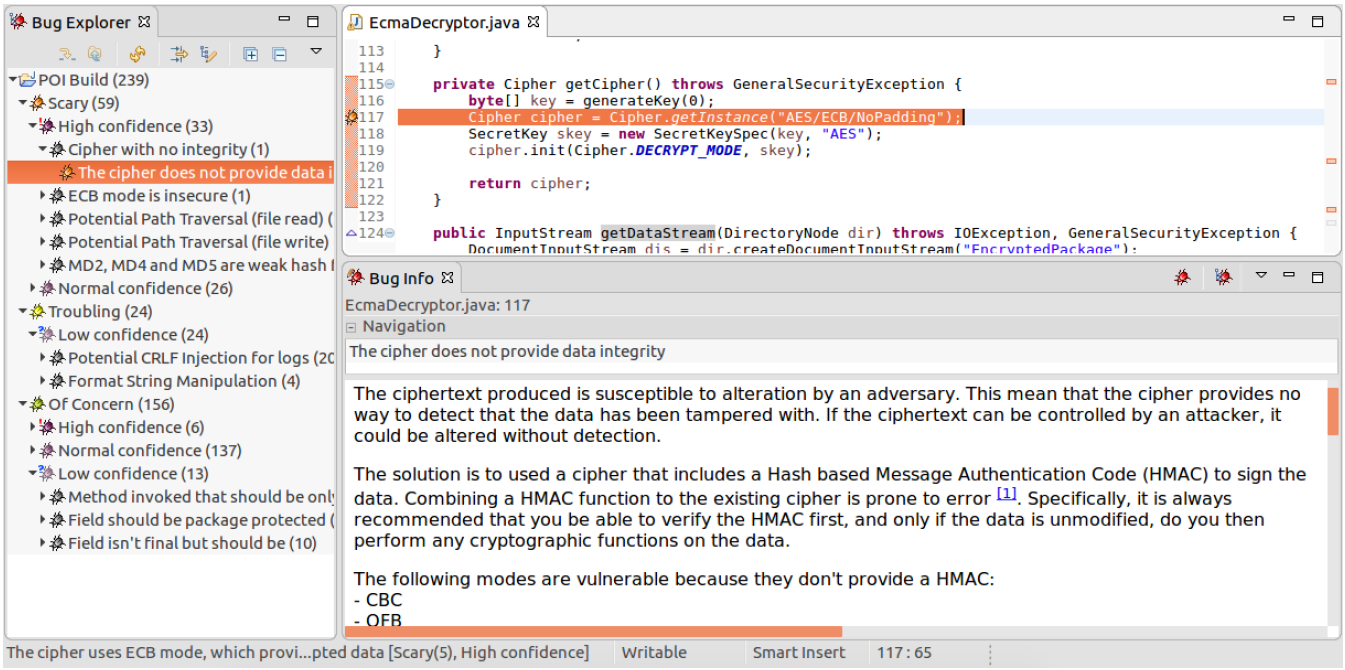

Figure 1: The Graphical User Interface of Find Security Bugs.

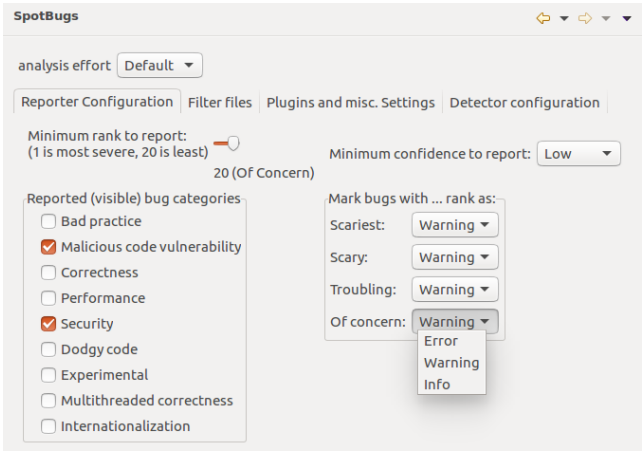

Figure 2: The Find Security Bugs configuration settings.

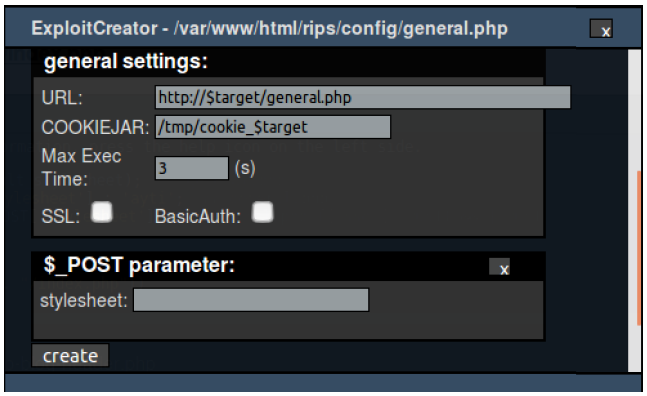

Figure 3: RIPS' exploit creator view.

provides the same functionalities as the "functions" button, on the level of the files.

The tool can be re-run by clicking on the "scan" button. RIPS scans the files for which the path in the file system is provided. The user can thus restrict which files have to be scanned. The user can set the verbosity level of the tool, which will make the scan return more or fewer results, and they can select what kind of vulnerabilities RIPS should search for. It is also possible to filter for which results are shown by using the "hide all" buttons which hide all errors found in a particular file, or searching for a particular regex pattern. The style of the errors (font, colour, background colour, etc.) can also be customized.

3.1.3 Flawfinder

Flawfinder [11] is a static analysis tool that detects uses of dangerous functions in C/C++ code as described in the the CWE/SANS top 25 [5]. This open-source tool is run using the command-line. We used version 2.0.4.

Figure 5 illustrates an HTML report produced by Flawfinder on the OpenSSL source code, version 1.0.1e [15]. On the top of the report, Flawfinder provides information on the tool and the ruleset it uses to find the vulnerabilities. It then lists all files that were scanned, and all errors it found, ordered by severity. For each error, it provides the location of the error (file name and line number), the severity score, the vulnerability type, and the dangerous function that is used in the code. A short description of the vulnerability is also given, to explain why the function is dangerous, along with a link to the CWE page of the vulnerability. A fix is then proposed, which is often the name of a safe function

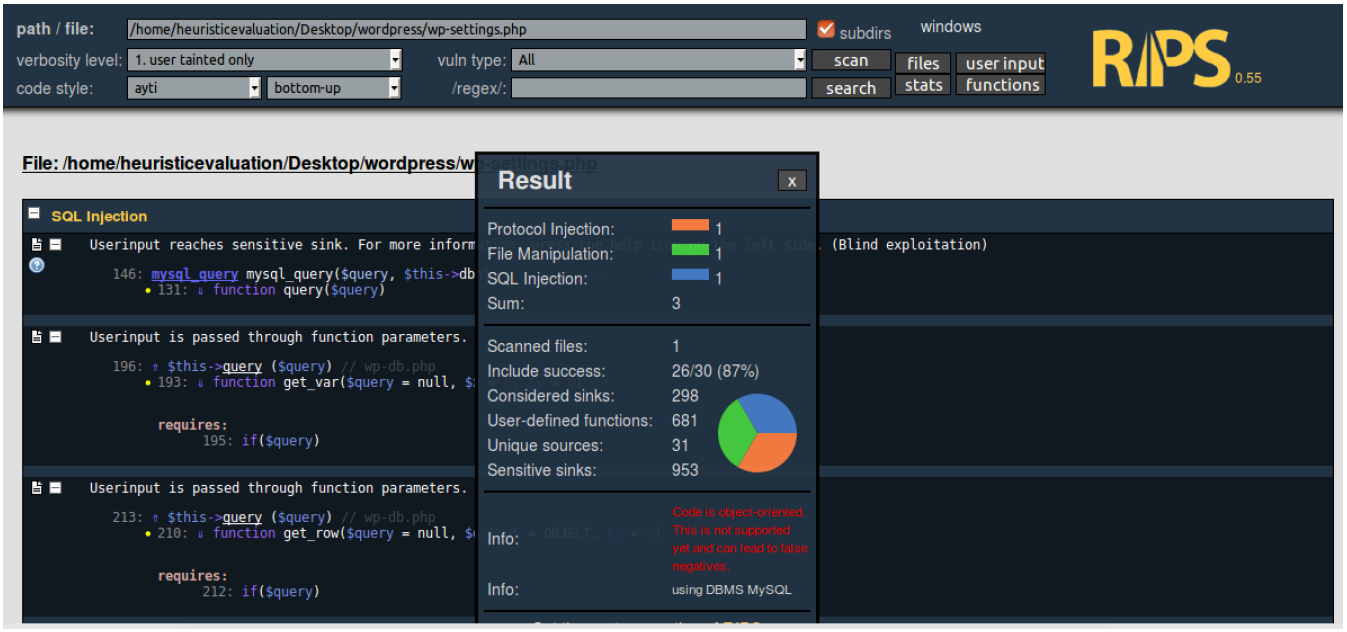

Figure 4: The Graphical User Interface of RIPS.

that can be used instead of the vulnerable one. The bottom of the report shows the analysis summary, which contains statistical data about the scan, such as with the number of files scanned, the number of errors reported, etc. The tool either prints its report in the command-line or produces reports in HTML or CSV format.

It is possible to restrict which files are scanned by Flawfinder using command-line options to include or exclude symbolic links. It can also only consider patchfiles, i.e. the diffs between two git commits. Errors can also be filtered out of the output based on regex patterns, risk level, likelihood of being a false positive, and whether or not the dangerous function gets input data from outside of the program.

3.2 Study Environment

To ensure the evaluators could exercise the tools in a variety of situations, we chose subject applications that contained multiple types of vulnerability patterns. Since each of the three tools can detect many different types of defects, we selected test projects that covered a variety of these vulnerability patterns. Synthetic suites, like the Juliet test suite [22], could have helped us ensure this coverage. However, they do not necessarily represent how vulnerabilities appear to developers in the wild. For instance, vulnerabilities in the Juliet suite are pre-labeled as false positives or true positives and generally strip away any code not relevant to triggering the vulnerability.

Therefore, while selecting test suites, we were also concerned with the ecological validity of our findings. We selected test suites from the "Applications" section of the Software Assurance Reference Dataset [14], because those test suites are derived from production code bases containing known vulnerabilities.

Since each of the three tools scans code written in different languages, we selected three test suites that satisfied the constraints described above: RIPS scanned WordPress version 2.0 (PHP); FSB scanned Apache POI version 3.9 (Java); and Flawfinder scanned OpenSSL version 1.0.1e (C). All three tools and there associated applications were configured in a single virtual machine image for evaluation. The virtual machine is available online [8].

3.3 Heuristic Walkthroughs

We identified usability issues in the security tools using heuristic walkthroughs [26], a two-phase method that combines the strengths of two usability evaluation techniques: cognitive walkthroughs [**?**] and heuristic evaluations [**?**]. Heuristic walkthroughs have been shown to be more thorough than cognitive walkthroughs and more valid than heuristic evaluations on their own [26].

We chose this evaluation technique, because heuristic walkthrough evaluations have successfully identified usability issues in other domains, such as electronic health records systems [**?**], music transcription tools [**?**], and lifelong learning systems [**?**]. Compared with traditional laboratory and field studies, this technique does not require the onerous recruitment of participants with security expertise. Researchers have noted that recruiting sufficiently many participants for security-focused studies can be prohibitively difficult [**?**]. One reason, for instance, is that individuals and organizations may be reluctant to openly discuss their security practices, for fear of revealing weaknesses in their security practices. Addressing this problem, heuristic walkthroughs enable relatively few evaluators to identify usability issues by providing evaluators with a structured process and requiring the evaluators to be double experts (experts in the application domain and usability principles). In compliance with these requirements, the two authors who conducted the heuristic walkthroughs for all three tools were familiar with security tools and usability principles. Each evaluator spent approximately four hours with each tool, totaling approxi-

Figure 5: The Graphical User Interface of Flawfinder.

mately 24 hours of evaluation time. The study protocol [7] and the detailed list of usability issues [13] are available online.

3.3.1 *Phase 1: Task-Oriented Evaluation*

Phase 1 of a heuristic walkthrough resembles a cognitive walkthrough, where evaluators approach a system with a list of predefined tasks. The goals of this phase are for the evaluator to familiarize themselves with the system and complete tasks similar to those an actual user would try to complete. In Phase 1 of our study, evaluators used the tools with a particular task in mind: fixing as many errors as possible in a limited time, as a security auditor at a software company. To do so, they were given the following guidelines, though they were also permitted to freely explore the system:

- Choose a vulnerability that you'd be most likely to inspect first.
- Determine whether the reported vulnerability is actually a vulnerability.
- Propose a fix to the vulnerability.
- Assess the quality of your fix.

To help evaluators think critically about each tool, we provided them with the list of guiding questions determined by Sears and colleagues for heuristic walkthroughs [26]. These questions ask evaluators to consider whether users will: know what to do next; notice the correct controls; know how to use the controls; see progress being made. During Phase 1, evaluators recorded the vulnerabilities they inspected, along with their impressions of the tool, and any usability issues encountered.

3.3.2 *Phase 2: Free-Form Evaluation*

Phase 2 of a heuristic walkthrough resembles a heuristic evaluation, where evaluators freely explore an entire system using a set of usability heuristics to identify issues. After completing Phase 1, evaluators have familiarized themselves with the system and, thus, are prepared to conduct a heuristic evaluation.

Among the many sets of available heuristics, we chose to use Smith and colleagues' 17 information needs [**?**] as the basis for our heuristics. These 17 information needs pertain specifically to security-oriented static analysis tools and cover a range of relevant topics such as: vulnerabilities, attacks, and fixes; code and the application; individuals; and problem solving support. As opposed to using generic usability heuristics, like Nielsen's ten heuristics [**?**], we developed our heuristics from these information needs, as prior studies have shown that domain-specific heuristics can be more effective [**?**, **?**, **?**].

The two evaluators considered each of the 17 information needs and recorded any usability issues that related to those information needs. Of course, during this phase, the evaluators also recorded additional usability issues that did not

precisely fit any of the provided heuristics.

3.4 Replication Package

To support the replication and continuation of this work, we have made all of our materials available, including the virtual machine containing the static analysis tools and the code bases they were used on [8], the study protocol [7] and the list of usability issues we detail in the following section [13].

Our replication package includes the virtual machine image used during our heuristic evaluation. This image contains FSB, RIPS, and Flawfinder configured to run on their respective codebases, and instructions describing how to get initial scan results.

Because heuristic walkthrough evaluations of security tools are beneficial beyond the scope of what was feasible during our study, we also provide the heuristic evaluation guide we developed. With this guide, a qualified evaluator with expertise in static analysis tools and usability principles could extend our work to any additional tools he/she could access.

4. RESULTS

Our heuristic walkthrough evaluation identified 155 usability issues. We do not intend for the presence or quantity of these issues to be a statement about the overall quality of the tools we evaluated. Instead, each of these issues represents a potential opportunity for tools to be improved. For completeness, we provide the full list of usability issues in the supplemental materials [13].

Because similar issues were identified across tools, heuristics, and evaluators, we present our findings thematically — each of the following six sections represents one theme. In each section, we will give a general description of the usability issues in that theme, explain how instances of those issues impact developers, and sketch how our insights could be used by tool designers and researchers to improve static analysis tools. Next to the title of each theme, we report the number of usability issues in parenthesis (X) that we identified in that theme. This number simply characterizes our findings and should not be interpreted as the ranking or severity of the issues in that theme. Also note that these counts sum to slightly more than 155, because some usability issues span multiple themes.

To further organize the results, we have bolded short titles that describe types of usability issues within each theme; To enable the reader to browse the results by tool, in {braces} next to each short title are the tools that each issue applies to. For instance, "**Immutable Code {RIPS, Flawfinder}**" denotes that the Immutable Code usability issue applies to RIPS and Flawfinder, but not FSB.

4.1 Missing Affordances (31)

Beyond presenting static information about code defects, analysis tools include affordances for performing actions, such as navigating code, organizing results, and applying fixes. Issues in this category arose when tools failed to provide affordances.

Managing Vulnerabilities {FSB, RIPS, Flawfinder}: The first missing affordance we observed, was that while all three tools showed a list of all errors they found, the options to manage the list of all potential vulnerabilities were limited. This is an important affordance, because it would allow a developer to quickly find the vulnerability they would like to inspect. After scanning the source code, tools can identify any number of potential vulnerabilities, all of which must be made accessible to the developer inspecting the results. Flawfinder, for example can generate a single text file, csv, or HTML page containing all the scan results. As Figure 5 depicts, these results are presented as a list that cannot be reorganized, searched, or sorted. Consequently, it is difficult for developers to find and fix related vulnerabilities.

Applying Fixes {FSB, RIPS, Flawfinder}: We also identified two missing affordances pertaining to applying fixes to the code. First, both RIPS and Flawfinder do not allow the developer to modify the code they scan. Instead, developers must view scan results in one window and edit the code using a separate text editor. This workflow is problematic, because developers are burdened with maintaining a mapping between the tool's findings and the code editor. Secondly, the tools did not fully support quickfixes that could automatically patch vulnerable code and did not otherwise offer assistance applying changes. Only FSB included some quickfixes, but this feature was available for just a few defect patterns. Without these affordances for applying fixes, developers must exert extra effort to resolve the defects presented by their tools.

Discussion: Many of the affordances that we noted as missing from these tools do not represent revolutionary breakthroughs in user interface design. In fact, features like sorting and filtering lists are commonplace in many applications. Integrating these well-known affordances into static analysis tools could be one low-cost way to improve the usability of these tools. On the other hand, some affordances will require more effort to incorporate into analysis tools. For example, affording developers the ability to accurately apply automated patches remains an open research area. We are encouraged by systems like FixBugs [**?**], which assists developers in applying quickfixes with FindBugs. Our results suggest that security-oriented static analysis tools would benefit from advances in this area.

4.2 Missing or Buried Information (75)

Static analysis tools can provide developers with a wide range of information about the defects they detect. For example, all three tools we studied give information about the location and defect-type of the vulnerabilities detected. The issues in this theme correspond to instances where tools failed to provide information that was needed to resolve defects. In this theme we discuss both missing information and buried information. These two issues are intertwined, because buried information that a developer never unearths is effectively missing.

Vulnerability Prioritization {FSB, RIPS, Flawfinder}: Since tools can generate many alerts from a single scan, before fixing a vulnerability, developers must decide which alert to inspect first. To varying extents, all three tools failed to provide information that, had the information been present, would have helped developers decide which vulnerabilities to inspect. We noted several different types of missing information, such as information about: which files contained clusters of vulnerabilities; a vulnerability's severity; and how to interpret severity scales. For example, unlike

RIPS, FSB provides information about the severity of each vulnerability, typically in the following form: "Rank: Of Concern (18), confidence: Normal." However, even FSB does not provide information about how to interpret this report. A developer might be left wondering whether 18 is high or low, or what other confidence values are possible. This issue disproportionately affects users who are using a tool for the first time and still learning to interpret the scales. Nonetheless, lacking information about how to prioritize vulnerabilities, developers might misallocate their limited time by fixing low-severity vulnerabilities.

Fix Information {FSB, RIPS, Flawfinder}: The tools we evaluated also failed to provide some information that developers would need to accurately fix vulnerabilities. The types of missing information spanned many different categories. To name a few, the tools were missing code examples, fix suggestions, definitions of unfamiliar terms, and explanations of how vulnerabilities could be exploited. Furthermore, some types of information that were present, were not detailed enough, such as when the tools provided terse issue descriptions or when the tools listed possible fixes, but did not articulate the tradeoffs between those solutions.

Discussion: One solution to these types of issues would be to simply add more information to tool notifications. This simple solution would ensure all the information needed to select and fix vulnerabilities is present for the developer. However, overstuffing notifications with too much information might bury the most pertinent information at a given time. Instead, the challenge for static analysis tools is to discern when developers need a particular piece of information and deliver that information.

4.3 Scalability of Interface (10)

As static analysis tools scale to find more defects in larger codebases, so too must their interfaces for presenting those defects. The issues in this section arose when tools struggled to present large amounts of information about vulnerabilities. Here we distinguish between scalable interfaces and scalable tools because we are interested in the usability of these tools' interfaces, not their technical capabilities. Each of the three tools we examined exhibited an interface scalability issue.

Vulnerability Sorting {Flawfinder}: As we previously discussed in Section 4.1, Flawfinder does not provide affordances for managing the list of vulnerabilities it detects. This issue is magnified as Flawfinder scales to identify more vulnerabilities in a project. Lacking the ability to manage this list, developers must resort to sub-optimal task selection strategies, such as searching linearly through the list for a vulnerability they would like to inspect.

Overlapping Vulnerabilities {FSB}: Like the other tools we evaluated, FSB can detect multiple different patterns of vulnerabilities. When multiple vulnerability patterns are detected on the same line, the bug icons for those vulnerabilities overlap. As a result, developers cannot identify lines containing more than one vulnerability at a glance and instead have to hover over each line with a bug icon to check whether it contains multiple defects. Scaling up the number of defect detectors increases the likelihood that these instances of overlap will occur. The decision to display vulnerability markers in this way deprives developers of information and forces FSB users to manually inspect each line for duplicates. As a consequence, developers might overlook severe vulnerabilities if their indicators are hidden beneath minor vulnerabilities.

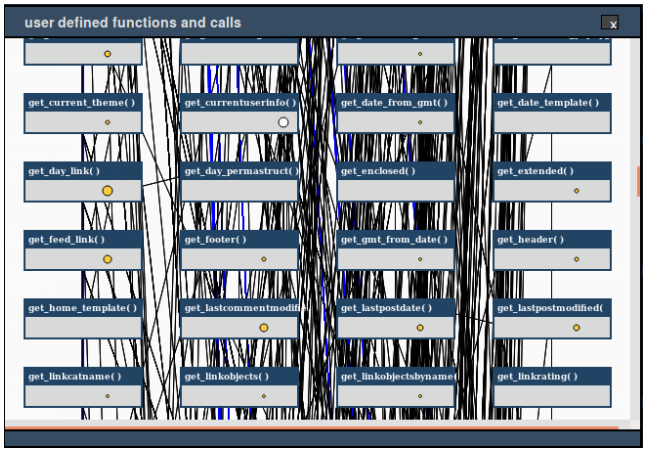

Figure 6: Scalability of RIPS' function view.

Scalable Visualizations {RIPS}: Finally, the clearest scalability issue we encountered was the call graph visualization in RIPS. An example of this visualization is depicted in Figure 6. The graph is intended to show the flow of tainted data from sources to sensitive sinks. However, when functions along the flow are called from many sites, all the call sites are added to the graph, resulting in a crowded visualization. Furthermore, when these call sites span more than 50 files, RIPS will not generate any visualization (Figure 7).

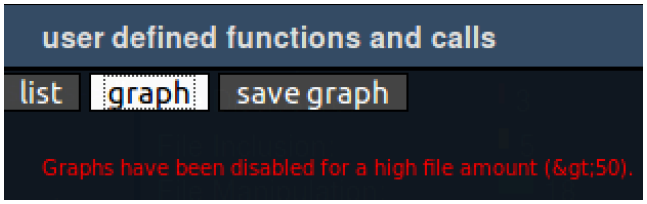

Figure 7: RIPS call graph visualization for more than 50 files.

Discussion: We propose two potential design changes that would improve FSB's overlapping vulnerabilities issues. One possibility is that FSB could use the stack metaphor to present the co-located vulnerabilities. Rather than draw the icons directly on top of each other, the tool could offset each subsequent bug by a few pixels. Alternatively, the tool could annotate a single bug icon with the number of vulnerabilities present on that line (e.g., 3).

RIPS provides a call graph visualization, where the other two tools we evaluated provided no similar feature. Such a feature could help visually oriented developers trace vulnerabilities and reason about the flow of tainted data through their system. However, if tools are to implement successful graph visualizations to support developers, the scalability of the visualization must be considered. Tool designers could consider implementing features such as those in Reacher [**?**], which enable developers to expand and highlight only the

relevant paths through the graph.

4.4 Code Disconnect (12)

Static analysis tools generate reports based on the code they scan. However, we identified usability issues when the content of those reports were disconnected from the source code.

Mismatched Examples {FSB, RIPS, Flawfinder}: The first issue in this category relates to the code examples used by all three tools. Many FSB notifications, for instance, contain hard-coded examples of vulnerable code and also examples of suggested code patches. Providing any code example is certainly more helpful than giving no information. Nonetheless, because the examples are hard-coded for each pattern, the burden of figuring out how to apply that example to the current context falls on the developer. Even if the example is written in the same programming language, this translation can be non-trivial, especially if the example is drawn from a source using different libraries or frameworks.

Immutable Code {RIPS, Flawfinder}: We encountered usability issues while trying to apply changes. Only FSB presented its results alongside code that could be edited directly. Flawfinder and RIPS include snippets from the scanned code in their reports, but do not enable the developer to edit the source code directly. As we previously described in Section 4.1, this disconnect forces developers to transfer their findings from where the tool presents its results to where they need to make changes. Furthermore, without being able to access the complete original source code while using a static analysis tool, developers cannot use navigation tools to browse code related to a vulnerability.

Discussion: Presenting results within an IDE, like FSB does, helps developers maintain connections between a vulnerability and the code surrounding that vulnerability. Developers also have the added benefit of being able to use code navigation tools while inspecting a vulnerability within this environment. However, our findings reveal opportunities to establish deeper connections between vulnerable code and how tools present those vulnerabilities to developers. Considering the mismatched examples usability issue, we imagine that analysis tools' code examples could be parameterized to use similar variable names and methods to those in the scanned code. Achieving a closer alignment between source code and examples will hopefully reduce developers' *cognitive load* while translating between the two, freeing them up to concentrate on resolving the vulnerability.

4.5 Workflow Continuity (34)

Developers do not use static analysis tools in isolation. Tools must synergize with other tasks in a developer's workflow, such as editing code, testing changes, and reviewing patches. Issues in this category arose when tools dictated workflows that were not compatible with a developer's natural way of working.

Tracking Progress {FSB, RIPS, Flawfinder}: Developers work with static analysis to reduce the number of vulnerabilities in their code, making code changes to progressively resolve warnings. However, the way tools present their results does not support developers in tracking their progress. Instead of reporting which vulnerabilities were added and removed between scans, the tools we evaluated only provide snapshots of all the current vulnerabilities in a project at a given time. This is only somewhat problematic when a developer wants to consider the effects of their changes on a single vulnerability notification. For example, if the tool at first reports 450 vulnerabilities, and then reports 449 after the developer applies a patch, then they can assume their patch fixed the one vulnerability. However, when a developer or his/her team makes sweeping changes to address many several vulnerabilities simultaneously, it becomes much more difficult to determine which issues were added and removed between scans based only on two snapshots.

Batch Processing {FSB, RIPS, Flawfinder}: Secondly, all three tools we evaluated dictate that developers process notifications on a case-by-cases basis. This is problematic because projects can contain many occurrences of the same vulnerabilities; fixing these vulnerabilities serially can be tedious and error-prone. Since tools are technically capable of scanning many more files than developers could manually scan, they must also enable developers to fix similar vulnerabilities in batches. Otherwise, the tool far outpaces the developer, adding vulnerabilities to his/her work queue much faster than they can dismiss them.

Discussion: Static analysis tools can be improved to better support developers' workflows. By keeping track of the vulnerabilities added and removed in each scan, tools could provide the option of presenting developers with scan diffs. These diffs could help developers identify changes that add many potential vulnerabilities to the code as well as draw developers' attention to changes that remove vulnerabilities. Tools could also support developers' workflows by enabling them to process similar vulnerabilities in batches.

4.6 Inaccuracy of Analysis (12)

Whereas the previous five sections describe usability issues with the *design* of FSB, RIPS, and Flawfinder, the issues in this category could comparatively be categorized as *implementation bugs*. While evaluating these tools we encountered some usability issues that seemed to arise from the tools' unexpected behavior, rather than bad design. Nonetheless, implementation bugs in a tool's interface may affect a developer's confidence in a tool and ability to complete tasks.

Most of the issues in this category affected FSB, specifically its code navigation features. For instance, applying a quick fix caused the tool to unexpectedly jump to a different file. In other instances, the tool's navigation pane in the bug info window contained duplicated entries and entries that seemed to have no effect.

5. LIMITATIONS

In this section we discuss the limitations of our study:

We only evaluated three static analysis tools and certainly expect that evaluating additional tools with unique interface features could yield new usability issues. Relatedly, we were unable to evaluate commercial tools in this study due to licensing restrictions. To mitigate these threats to generalizability, we selected analysis tools that each covered distinct interface features. We also make our evaluation guide available so that our results can easily be extended to include additional tools. Still, we do not claim that any of the

usability issues we identified necessarily generalize to other static analysis tools.

We also acknowledge that the evaluators' individual tool usage styles might have influenced the issues they identified. For instance, the evaluators' ad hoc approach to reading documentation might not align with some users who read all available documentation before using a tool. However, in this example, many issues we identified, like scalability issues, would not be influenced by documentation reading. Nonetheless, to bolster the ecological validity of our study, we selected real-world applications and instructed evaluators to perform realistic tasks in Phase 1 of their evaluation.

Our choice of usability evaluation technique also influences our results. We chose to use heuristic walkthroughs, because they enable relatively few evaluators to identify usability issues. Compared with laboratory studies, where qualified participants must be recruited and might typically spend only an hour using a tool, evaluators conducting heuristic walkthroughs have more time to find deeper issues (approximately four hours per tool in our study).

6. CONCLUSION

This paper responds to a call for usability evaluations of *developer* security tools. In this work two authors conducted heuristic walkthrough evaluations of three tools, Find Security Bugs, RIPS, and Flawfinder. Our results reveal usability issues that detract from all three of these tools. For instance, we noted that the interfaces of these tools do not scale as more vulnerabilities are detected in a project. We hope that by identifying these issues this work enables practitioners to improve the usability of their tools and inspires researchers to continue evaluating the usability of developers' security tools.

7. REFERENCES

[1] Apache poi source code. `https://poi.apache.org/subversion.html`, 2018.

[2] Checkmarx home page. `https://www.checkmarx.com/`, 2018.

[3] Common weakness enumeration home page. `https://cwe.mitre.org/`, 2018.

[4] Coverity home page. `https://scan.coverity.com/`, 2018.

[5] Cwe/sans top 25 most dangerous software errors. `http://cwe.mitre.org/top25/`, 2018.

[6] Eclipse home page. `http://www.eclipse.org/`, 2018.

[7] Evaluation guide. `https://figshare.com/s/087f103905189f1a7ca0`, 2018.

[8] Evaluation vm. `https://figshare.com/s/bf91e24a3df3c4fff77c`, 2018.

[9] Find security bugs home page. `http://find-sec-bugs.github.io/`, 2018.

[10] Findbugs home page. `http://findbugs.sourceforge.net/`, 2018.

[11] Flawfinder home page. `https://www.dwheeler.com/flawfinder/`, 2018.

[12] Jlint home page. `http://jlint.sourceforge.net/`, 02 2018.

[13] List of issues. `https://figshare.com/s/71d97832ae3b04e0ff1a`, 2018.

[14] Nist test suites. `https://samate.nist.gov/SRD/testsuite.php`, 2018.

[15] Openssl source code. `https://github.com/openssl/openssl`, 2018.

[16] Owasp top 10. `https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project`, 2018.

[17] Pylint home page. `https://www.pylint.org/`, 2018.

[18] Rips home page. `http://rips-scanner.sourceforge.net/`, 2018.

[19] Spotbugs home page. `https://spotbugs.github.io/`, 2018.

[20] Wordpress source code. `https://github.com/WordPress/WordPress/`, 2018.

[21] Y. Acar, S. Fahl, and M. L. Mazurek. You are not your developer, either: A research agenda for usable security and privacy research beyond end users. In *Cybersecurity Development (SecDev), IEEE*, pages 3–8. IEEE, 2016.

[22] T. Boland and P. E. Black. Juliet 1.1 c/c++ and java test suite. *Computer*, 45(10):88–90, Oct 2012.

[23] B. Chess and G. McGraw. Static analysis for security. *IEEE Security and Privacy*, 2(6):76–79, Nov. 2004.

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

[25] M. Green and M. Smith. Developers are not the enemy!: The need for usable security apis. *IEEE Security & Privacy*, 14(5):40–46, 2016.

[26] A. Sears. Heuristic walkthroughs: Finding the problems without the noise. *International Journal of Human-Computer Interaction*, 9(3):213–234, 1997.

[27] A. Whitten and J. D. Tygar. Why johnny can't encrypt: A usability evaluation of pgp 5.0. In *USENIX Security Symposium*, volume 348, 1999.

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

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

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

[31] Static analysis tools for security. `https://www.dwheeler.com/essays/static-analysis-tools.html`.