UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

Faculty for Computer Science, Electrical Engineering and Mathematics

User-Centered Tool Design

for Data-Flow Analysis

Lisa Nguyen Quang Do

Doctoral Thesis

Submitted in partial fulfillment of the requirements for the degree of

Doktor der Naturwissenschaften (Dr. rer. nat.)

Advisors
Prof. Dr. Eric Bodden
Prof. Dr. Karim Ali

Paderborn, October 21, 2019

Abstract

In the past decades, static analysis tools have been known to have specific user-experience issues such as a high number of false positives, a lack of responsiveness, or the poor warning descriptions that they provide. Their increasing use in industry makes those issues more relevant, especially as static analysis tools become more powerful, detect more complex bugs and vulnerabilities, and support an increasing number of languages, third-party libraries and coding concepts. The way in which an analysis interprets the code it analyzes may differ from how the developer views the analyzed code, causing major user-experience issues such as warning misunderstandings or wrong fixes.

To address user-experience issues in static analysis tools, we apply the user-centered design methodology, first aiming to understand the users' motivations for using the tools, and what they need to easily interact with them. With this knowledge, we then derive design recommendations for building static analysis tools, which differ from the ones identified in past studies that only focus on the user-experience issues themselves. Finally, we prototype and evaluate tools for static analysis following the recommendations, showing the usefulness of the user-centered process.

In this thesis, we focus on two groups of users. First, we study *analysis developers*—who write the code of a static analysis—to discover how to assist them in writing and debugging static analysis code. Through a survey of professional analysis developers, we show that current development tools do not sufficiently support static analysis development. To help analysis developers handle two codebases (the analysis code and the analyzed code), we identify desirable design requirements for a debugging tool for static analysis, such as displaying internal analysis results, providing graph visualizations, or introducing two sets of breakpoints for the two codebases. We apply some of those requirements in VISUFLOW, a coding environment we designed specifically for static analysis. Through a user study, we were able to show that VISUFLOW allows analysis developers to debug static analyses more efficiently than with currently used coding environments.

Second, we focus on *software developers*—who write the code that is analyzed by an analysis tool—and report on developer motivations and strategies through a study consisting of a survey of professional developers, a study of analysis logs from a large software company, and a small-scale cognitive walkthrough. Through our study, we discover that the usage of static analysis tools in industry is heavily influenced by time constraints: the decisions made by software developers depend on how much time they can allocate to understanding and fixing analysis warnings. We thus derive a set of design recommendations for analysis tools, such as improving the responsiveness of the tools to provide developers with quicker updates, including developer knowledge in the analysis to reduce the rate of false positives, or improving the explainability of analysis results to help developers understand warnings more efficiently. We address some of those recommendations through the Just-in-Time Static Analysis concept with which developers can guide the analysis towards results of interest, postponing other paths for later. This

allows our implementation, CHEETAH, to be responsive enough to be integrated in an Integrated Development Environment. Focusing at the overlooked use case of analysis configuration, we address other recommendations through the concept of rule markers. The concept is used to expose internal analysis information to the developer and help them better understand how the analysis works and why it reports certain results, so that they can adjust the analysis rules. Through user studies and empirical evaluations, we show that when addressing our design recommendations, the two concepts of Just-in-Time analysis and rule markers allow developers to perform their tasks better than with current tools.

Through this thesis, we motivate the need for more user-centered approaches for addressing decades-old user-experience issues in static analysis, putting the user at the center of the design process in order to create tools that suit their needs.

Zusammenfassung

In den letzten Jahrzehnten waren statische Programmanalyse-Tools berüchtigt für User-Experience-Probleme wie die hohen Anzahl von Fehlalarmen, das langsame Produzieren von Ergebnissen oder für unvollständige Warnungen. Die zunehmende Nutzung der Tools in der Industrie macht diese Probleme noch relevanter, insbesondere da statische Programmanalyse-Tools immer mächtiger werden, mehr Sprachen, externe Bibliotheken und Coding-Konzepte unterstützen und komplexere Bugs und Schwachstellen erkennen. Die Art und Weise wie eine Analyse den analysierten Code interpretiert, kann sich vom Verständnis des Entwicklers des Codes unterscheiden und so zu signifikanten User-Experience-Problemen führen, wie z.B. missverständliche Warnungen oder falsche Fehlerbehebungen.

Um die Probleme mit statischen Programmanalyse-Tools zu adressieren, wenden wir benutzerzentriertes Design an, um zu verstehen, wie und warum Programmanalyse-Tools benutzt werden. Dabei untersuchen wir die Motivation der Anwender zur Verwendung der Tools und ihre Bedürfnisse in der Interaktion mit diesen Tools. Mit diesem Wissen extrahieren wir Designempfehlungen für den Aufbau statischer Programmanalyse-Tools, die sich von denen früherer Studien unterscheiden, welche sich nur auf die User Experience-Probleme konzentrieren. Dann erstellen wir Prototypen, bewerten diese nach den Designempfehlungen und zeigen den Nutzen von benutzerzentriertem Design.

Zu diesem Zweck konzentrieren wir uns auf zwei Benutzergruppen. Zuerst studieren wir die *Analysenentwickler*—die den Code einer statischen Analyse schreiben—um herauszufinden, wie man sie beim entwickeln und debuggen des Codes der statischen Analyse unterstützen kann. Durch eine Umfrage unter professionellen Analysenentwicklern zeigen wir, dass aktuelle Entwicklungswerkzeuge die Entwicklung statischer Analysen nicht ausreichend unterstützen. Um den Analysenentwicklern zu helfen, mit zwei Codebasen (dem Analysecode und dem analysierten Code) umzugehen, bestimmen wir Designanforderungen eines Debugging-Werkzeugs für statische Analyse, z.B. die Anzeige interner Analyseergebnisse, der Bereitstellung von Diagrammvisualisierungen oder der Einführung zweier Breakpoint-Systeme für die beiden Codebasen. Einige dieser Anforderungen implementieren wir in VISUFLOW, einer Programmierumgebung, die wir speziell für statische Analyse entwickelt haben. Durch eine Nutzerstudie konnten wir zeigen, dass VISUFLOW es Analysenentwicklern ermöglicht, statische Analysen effizienter zu debuggen als dies mit aktuellen Entwicklungswerkzeugen möglich ist.

Zweitens konzentrieren wir uns auf *Softwareentwickler*—die den Code schreiben, welcher von einem Analysetool analysiert wird—und berichten über Motivationen und Strategien der Entwickler. Dies tun wir durch eine Studie, die eine Umfrage unter professionellen Entwicklern, eine Auswertung von Log-Dateien der Code-Analysen eines großen Softwareunternehmens, und einen kleinen Cognitive Walkthrough umfasst. Die Studie zeigt, dass die Nutzung von statischen Programmanalyse-Tools in der Industrie stark durch Zeitdruck beeinflusst wird: die Entscheidungen der Softwareentwickler hängen davon ab, wie viel Zeit sie haben, um Analysewarnungen

zu verstehen und zu beheben. Wir bestimmen Designempfehlungen für Programmanalyse-Tools, z.B. schnellere Ergebnisse durch die Tools, Integration von Entwicklerwissen in die Analyse, um die Anzahl der Fehlalarme zu reduzieren oder die Verbesserung der Erklärbarkeit der Analysewarnungen, um Entwicklern zu helfen, Warnungen effizienter zu verstehen. Wir setzen einige dieser Designempfehlungen durch das Just-in-Time Static Analysis Konzept um, mit welchem Softwareentwickler die Analyse zu ausgewählten Schwerpunkten leiten und andere Teile auf später verschieben können. Dies ermöglicht es unserer Implementierung, CHEETAH, schnell genug zu sein, um in einer integrierten Entwicklungsumgebung effizient zu laufen. Darüber hinaus stellen wir das Rule Markers-Konzept vor, welches den Softwareentwicklern interne Analyseinformationen zur Verfügung stellt, um ihnen zu helfen, die Funktion der Analyse besser zu verstehen, und warum bestimmte Ergebnisse gemeldet werden, sodass die Entwickler die Analyseregeln zielorientiert umkonfigurieren können. Durch Nutzerstudien und empirische Bewertungen zeigen wir, dass die beiden Konzepte der Just-in-Time-Analyse und der Rule Markers es den Entwicklern ermöglichen, ihre Aufgaben besser zu erfüllen als mit aktuellen Programmanalyse-Tools.

Wir begründen daher die Notwendigkeit von benutzerzentrierteren Ansätzen, um die Jahrzehnte alten User-Experience-Probleme der statischen Analyse zu lösen. Dabei stellen wir den Benutzer in das Zentrum der Designmethodik, um Programmanalyse-Werkzeuge zu entwickeln, die den Bedürfnissen des Benutzers entsprechen.

Acknowledgements

Contents

1

Introduction

This chapter motivates and presents the main research question of this thesis: *can we build more usable tools for data-flow analysis by putting the user at the center of the design process?* It presents the five main contributions of the thesis, and details their corresponding publications.

1.1 Motivation and Research Question

Static program analysis is a method of automatically analyzing the the source code or the bytecode of a program without running it [98]. From assistant utilities for compiler optimizations [51, 140] to simple style-checking analyses to the detection of complex software bugs and security vulnerabilities [136], the use of static analysis is widespread in industry, from the individual developer to large companies [22, 47, 65]. One particular type of static analysis, *data-flow analysis* [52, 98], is able to perform complex reasoning and to report warnings covering a large range of software issues such as privacy leaks [9, 71], data races [49, 84], and API misuses [2, 60]. Because static analysis reasons about the program without running it, it needs to approximate all potential runtime scenarios. As a result, the users of static analysis tools, in particular of analysis tools that perform complex reasoning, often encounter common user-experience issues such as poor responsiveness [22, 47] (as the analysis needs time to run through all scenarios and is thus slow to provide updates), or dissatisfactory warning explainability [14, 65] (because explaining the analysis' complex reasoning to the user can be challenging). Such mismatches between how an analysis tool works and how its end-user—the software developer—thinks it should work are often caused by unadapted tool design, in which the features offered by the analysis tool do not appropriately support the developer in their tasks. Those user-experience issues can cause warning mishandlings and tool abandonment, thus preventing developers from appropriately using the analysis tools to their full potential [10, 11, 14, 22, 47, 65].

When designing software, methods of *user-experience design* focus on the user's needs in order to bridge mismatches between the user's perception of the environment and tasks, and the tool's [25, 40, 108]. Unlike functional design, user-experience design approaches the design of an interface from the angle of the user's needs rather than from the functionalities a given program can support, thus yielding more usable interfaces and new functionalities that are closer to the user's expectations. For example, in this thesis, we discover the need to support analysis developers in debugging static-analysis code, a topic that has seldom been researched in the past, and design dedicated functionalities that are more adapted to static analysis than currently used debugging tools.

In this thesis, we explore the application of user-centered methods to the design of tools for static analysis to address current user-experience issues. We consider two distinct user groups: the *software developers* (who are the main target users of analysis tools), and the *analysis developers* (who write and debug the static analysis code used by the former group). While analysis developers do not use static analysis tools to write and debug analyses, the code editors and debuggers that they use to write the analysis code still need to be able to run analyses and explain the inner-workings of the analyses for debugging purposes. As a result, analysis developers also run into similar static analysis-specific user-experience issues as software developers. We refer to analysis tools used by software developers and code editors and debuggers used by analysis developers as *tools for static analysis*, or *tools for data-flow analysis*.

Throughout the thesis, we apply a *user-centered* approach to design tools to support both user groups. We first conduct surveys and studies to understand user-experience issues with the current state-of-the-art tools. We then focus on particular issues found in our studies, design tools that address those specific issues, and evaluate them through empirical evaluations for soundness and precision, and user studies for usability, with the goal of answering the main research question of this thesis:

> *Can we build more usable tools for data-flow analysis by*
> *putting the user at the center of the design process?*

1.2 Contributions and Structure of the Dissertation

For the two user group of interest to this thesis—software developers and analysis developers—we assess the current state-of-the-art tools for static analysis, survey how they use those tools, determine user-experience issues and design recommendations when building tools to support them, and propose, implement, and evaluate approaches to address the user-experience issues. After presenting background information on static analysis and user-centered design in Chapter 2, the thesis makes five contributions. The related work specific to each contribution is detailed at the beginning of the corresponding chapter.

In a first contribution, the thesis presents the current state-of-the-art tools used by analysis developers to write and debug static analysis, and details their known user-experience problems. This is achieved through a survey of 115 analysis developers in which we determine which problems analysis developers typically encounter when debugging static analysis code, which tools they use to help fix those problems, which user-experience issues they encounter with those tools, and which debugging features would best support analysis developers write and debug static analysis. This contribution is presented in Chapter 3.

The second contribution of the thesis is the design and evaluation of VISUFLOW, a debugging environment built to support the *development* of data-flow analysis. VISUFLOW is designed to address the main user-experience issues found in the survey from Chapter 3 and focuses in particular on features that provide more clarity in the interactions the of two codebases (i.e., the analysis code and the analyzed code) instead of just one, like in traditional debugging scenarios. A user study conducted over 20 analysis developers showed that when debugging data-flow analyses, VISUFLOW allows analysis developers to identify 25% and fix 50% more errors than with a traditional debugging environment. This contribution is detailed in Chapter 4.

Chapter 5 reports on the third contribution of the thesis. It explores the current state-of-the-art static analysis tools used by software developers to detect bugs and security vulnerabilities in their application code, and expands on their user-experience issues. Through a survey of 87 developers in industry, a study of analysis logs, and a small-scale cognitive walkthrough on eight software developers, we determine where and when developers typically use static analysis

tools, what their motivations are when using those tools, which user-experience issues they encounter, and how those aspects influence their strategies when fixing analysis warnings. From this information, we identify requirements for designing usable static analysis tools for software developers.

The fourth contribution of the thesis focuses on a prominent user-experience issue encountered by software developers: the lack of responsiveness of certain analysis tools, which we present in Chapter 6. We address this issue by introducing the concept of Just-in-Time data-flow analysis that prioritizes certain analysis paths based on developer preferences. We present a framework for adapting existing data-flow analysis solvers to support that concept, and present CHEETAH, an instantiation of the concept for Android taint analysis. In an empirical evaluation, CHEETAH was able to report its first analysis results in less than a second. Its user study conducted over 18 developers showed that participants were able to fix data leaks twice as fast with CHEETAH than with a traditional taint analysis.

The fifth contribution of this thesis is presented in Chapter 7, in which we target another prominent user-experience issue reported by software developers: the poor explainability shown by some analysis tools when detailing a warning, which is particularly needed when developers configure an analysis tool. We address the issue by introducing the concept of rule graphs, which gather analysis-internal information—which is traditionally not exposed to the end-user—and uses it to assist the developer in the following tasks: improve warning understanding, classify warnings in given categories, and learn analysis patterns for which the analysis is more likely to make a mistake. A user study over 22 participants on MUDARRI, an implementation of the usage of rule graphs, shows that using analysis-based information helps developers understand warnings significantly better than without. An empirical evaluation also reveals that rule graphs allow the automated classification of data leaks and can help identify causes for false positives in data-flow analysis.

Chapter 8 summarizes the improvements in usability gained from a user-centered approach for tool design for both analysis developers and software developers, and suggests potential future work in the area, advocating for the importance of integrating usability from the start of the tool design process.

1.3 Publications Details

The work presented in Chapter 3 and Chapter 4 on debugging environments for analysis developers has been published in the IEEE Transactions on Software Engineering journal (TSE) in 2018 [91]. The VISUFLOW debugging environment was presented at the tool track of the International Conference on Software Engineering (ICSE) in 2018 [92].

The study on the usage of static analysis tools in industry described in Chapter 5, with the exception of Section 5.5.2 on gamification, is currently under submission [93].

The work on gamification from Section 5.5.2 was presented at the New Ideas and Emerging Results track of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering [89] in 2018.

The concept of Just-in-Time analysis from Chapter 6 has been presented at the ACM SIGSOFT International Symposium on Software Testing and Analysis conference (ISSTA) in 2017 [88] for which it received an ACM SIGSOFT Distinguished Paper Award. Early work on this concept was presented at the ACM conference on Programming Language Design and Implementation (PLDI) in 2016, where it won the first place at the ACM Student Research Competition. CHEETAH was demonstrated at the tool track of the International Conference on Software Engineering (ICSE) in 2017 [87].

The work on rule graphs presented in Chapter 7 is currently under submission [90].

We make the evaluation artifacts for all five contributions available online [86]. The artifacts contain—when applicable—the survey questions and anonymized answers, the user study questionnaires and results, the benchmark suites used for evaluation, the source code of the implementations, and video demonstrations of the interfaces.

The author of this thesis is the main and lead contributor to all of the publications mentioned above.

2

Background

This chapter presents background information on static analysis and user-centered design. In particular, we focus on *taint analysis*, a static data-flow analysis, and present two data-flow analysis frameworks: the monotone framework [50, 55] and the IFDS framework [15, 110], both of which we use in this thesis. We also introduce the notion of user-centered design, and how we apply it to the design of tools for static analysis. Further background information on the specific topics addressed by each chapter (e.g., responsiveness of static analysis tools) is given in the corresponding chapters.

2.1   Taint Analysis

Taint analysis is a type of static data-flow analysis that tracks *tainted data* through a program. It is typically used to detect *injection flaws* or *data leaks.*

Injection flaws are reported by OWASP as the first category of their current top ten application security risks [102, 103]. They occur when a program executes untrusted data, as illustrated in Listing 2.1. The code example is an excerpt from a web servlet written in Java, which answers GET requests through the method *doGet*(). It contains a simple SQL injection (CWE-89) [79] where a potentially attacker-controlled string enters the program through one of the request's parameters at line 2, is appended to the variable *query*, and is executed on the database at line 5. Because the program uses the parameter as is, an attacker can get malicious payloads executed by issuing GET requests with well-crafted parameters, and can thus run arbitrary SQL queries on the database. To avoid injection flaws, it is recommended to sanitize external inputs. Manually crafted sanitizers should be avoided in favor of trusted existing libraries, as illustrated in the example with the *sanitize*() method at line 3 that does not sanitize and returns the parameter instead. In the particular case of SQL injections, developers should prefer prepared statements instead.

To detect an injection flaw, a taint analysis marks program variables that contain potentially malicious data. Here, *uId* is marked (tainted) at line 2, because it receives the external data from *getParameter*(). The variable *input* is then tainted in the *sanitize*() method (line 9), because the argument passed to the method at line 3 is tainted. *id* is therefore tainted at line 10, and since it is the return value of the function, *userId* is tainted in turn at line 3. At line 4, the potentially malicious string is appended to *query*, which is thus tainted. The call to *executeQuery*() at line 5 is executed with the tainted variable *query* as a parameter, which causes the taint analysis to report a potential injection.

```
1  protected void doGet(HttpServletRequest request, HttpServletResponse response) throws
       Exception {
2    String uId = request.getParameter('userId');
3    String userId = sanitize(uId);
4    String query = "SELECT * FROM User WHERE userId='" + userId + "';";
5    ResultSet res = statement.executeQuery(query);
6    ...
7  }
8
9  private String sanitize(String input) {
10    String id = input;
11    return id;
12  }
```

Listing 2.1: SQL injection from line 2 to line 5. The *sanitize*() method is incomplete and does not sanitize the input correctly.

Methods that create taints such as *getParameter*() are called *sources*, methods at which taints are reported (e.g., *executeQuery*()) are referred to as *sinks*, and methods that kill taints (thus preventing the malicious data from being propagated) are *sanitizers* (e.g., *sanitize*(), if it was correct).

As with injection flaws, taint analysis can also be used to detect data leaks—vulnerabilities which were part of the OWASP top 10 in 2007 [100, 101] but that have now been absorbed by the injection flaws category [102, 103]. Data leaks present a problem of information exposure [75], where private data such as banking credentials, passwords, or other application information can be disclosed via email, or by being written to a log for example. To detect this category of vulnerabilities, we configure the taint analysis with the sources being accessor methods to private information, and the sinks being methods that send information out of the system.

Depending on which sources, sinks, and sanitizers it uses, a taint analysis can detect a large array of bugs and vulnerabilities, for example, 17 of MITRE's top 25 most dangerous software errors [80] (e.g., SQL injections (CWE-89) [79], use of hard-coded credentials (CWE-798) [78], or use of a broken or risky cryptographic algorithm (CWE-327) [76]) can be found with taint analysis [106]. In this thesis, we use the example of taint analysis for the detection of data leaks in Android, and thus use dedicated sources and sinks definitions described by Rasthofer et al. [8] for the Android framework.

Other types of data-flow analysis can detect other types of bugs and vulnerabilities. For instance, *typestate analysis*, a generalization of taint analysis, can detect another five of MITRE's top 25 errors, *nullness analysis* can find null pointer dereferences (CWE-476) [77]. In this thesis, we focus on the case of taint analysis, but our approaches can be generalized to all types of data-flow analysis, as long as they can be expressed in the frameworks we present in the next sections.

2.2 The Monotone Framework

The *monotone framework* is often used to define and solve classical data-flow analysis problems. First introduced by Kildall [55] through the example of a constant propagation analysis used for compiler optimization, it has then been refined by Kam et al. [50] for the general case of data-flow analysis. In the monotone framework, an analysis problem is expressed with the following five properties:

- A *Control-Flow Graph* (CFG) representing all possible paths of a method that can be executed at runtime. The nodes of the CFG typically represent *statements* of the method.

6
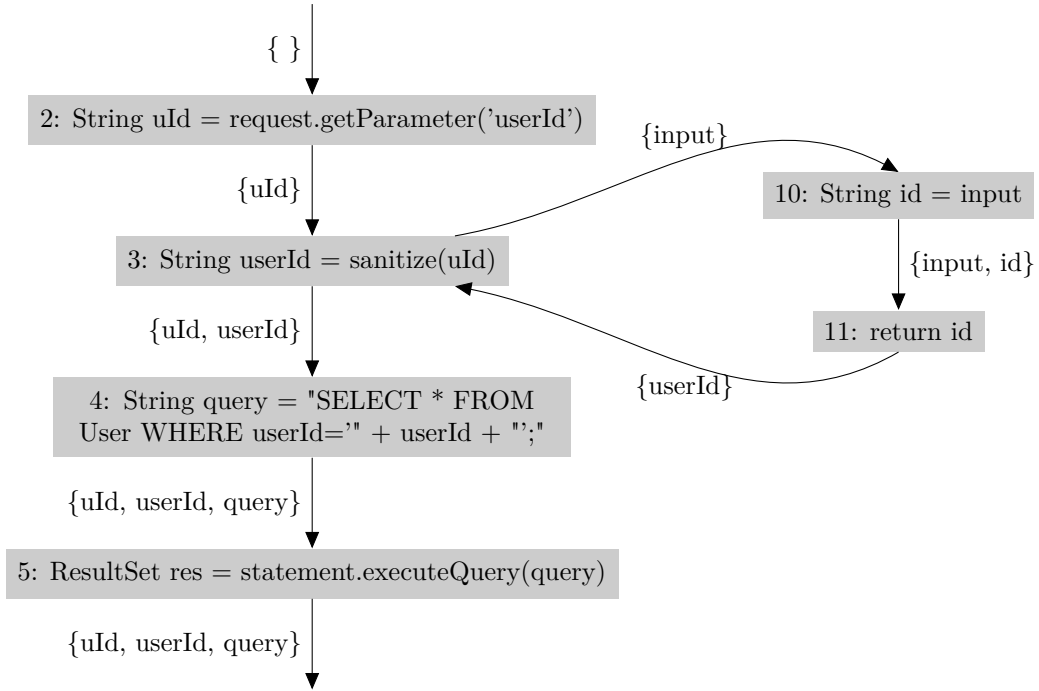
Figure 2.1: Interprocedural Control-Flow Graph (ICFG) of the example Listing 2.1, and the data-flow sets generated at each statement for a forward taint analysis.

Extended to a full program, the combination of multiple CFGs connected with call and return edges is called an *Interprocedural Control-Flow Graph* (ICFG). Figure 2.1 depicts the ICFG of the example from Listing 2.1.

- A *data-flow domain* $\mathcal{D}$ containing the *data-flow facts*, which are the values that the analysis results can take. For the example Listing 2.1 with a taint analysis, the lattice $\mathcal{D}$ contains the different combinations of the data-flow facts: the program's variables.

- A *flow function* $f_n : \mathcal{D} \mapsto \mathcal{D}$ defining how n—a node in the ICFG—is interpreted by the analysis. Given the program statement n and an *in-set* of data-flow facts $in \in \mathcal{D}$ representing the analysis results before n is executed, $f_n(in)$ determines the *out-set out* $\in \mathcal{D}$ that represents the analysis results after n's execution. The flow function thus encodes the rules of the analysis. For example, in the case of a taint analysis, it can encode the generation of a taint at a source statement, the transfer of a taint at an assignment statement, or the kill of a taint when a variable is sanitized, among other rules.

- An *initial set* $d_0 \in \mathcal{D}$ corresponding to the in-set at the program's entry point e. It is used to initialize the analysis. In Figure 2.1, e corresponds to line 2, and d_0 is the empty set.

- A *join operator* or *merge function* $\sqcap : \mathcal{D} \times \mathcal{D} \mapsto \mathcal{D}$ defines how to merge two in-sets at meet points of the ICFG, which are places where a node n has multiple predecessors (e.g., after $if/else$ branches, or after method calls such as line 4 in Listing 2.1). There, the out-sets of the parents are merged into a single in-set for n, before the flow function can be applied.

Once an analysis is defined with the five properties, the monotone framework can compute the analysis results. The Kildall approach computes MOP_n, the *"meet over all paths"* solution for each node n. For a given node n, the MOP is obtained by first calculating the analysis results

---

Algorithm 1 Fixed point iteration algorithm

```
 1: procedure ANALYZE
 2:     waiting_list := e
 3:     IN[e] := d_0
 4:     while waiting_list ≠ ∅ do
 5:         pop n off waiting_list
 6:         OLD := OUT[n]
 7:         IN[n] := ⊓ { OUT[m] | m ∈ predecessors(n) }
 8:         OUT[n] := f_n(IN[n])
 9:         if OLD ≠ OUT[n] then
10:             wl ∪ = successors(n)
```

---

along every single execution path of the ICFG from the entry point to n, and by merging them to obtain the general results over all paths of the program leading to n.

$$MOP_n = \underset{p \in paths(n)}{\sqcap} f_p(d_0)$$

While the MOP solution yields sound and precise results, it is generally undecidable, for example in the case of loops where an infinite number of paths exist. To remediate the issue, an over-approximation of the MOP, the MFP (*maximal fixed point*) is introduced by the Kam et al. approach [50]. Instead of merging the results at the end, the results are merged at each meet point of the ICFG. The meet information thus encompasses multiple paths at once, and at each meet point, it limits the number of paths that are explored by the analysis.

$$MFP_e = d_0$$
$$MFP_n = \underset{m \in predecessors(n)}{\sqcap} f_m(MFP_m)$$

We see that MFP over-approximates MOP: for each node n, MFP is a superset ($\sqsupseteq$) of MOP, making the analysis results obtained with MFP still sound, but less precise.

The monotone framework computes the analysis results by using a *fixed-point iteration* algorithm over the nodes of the ICFG, as shown in Algorithm 1. Once the fixed point is reached and the out-sets do not change anymore, the analysis results correspond to the MFP solution. In Figure 2.1, the MFP results of the taint analysis can be read in the edge labels and can be interpreted as such: if a variable is included in a set, it is tainted at the corresponding point of the program.

Intermediate Representation

In this thesis, we use the monotone framework provided by the Soot analysis framework [139,140] for the analysis used in our evaluation of VISUFLOW in Section 4.2. To simplify the analysis rules, many analysis frameworks first transform the source code or bytecode into an *intermediate representation* that is semantically easier to analyze. In the case of Soot, the intermediate representation is called *Jimple*. Jimple contains 15 types of statements, while Java bytecode contains more than 200, making Jimple programs easier to analyze. In addition, Jimple makes all classes and methods explicit by using their fully qualified names, so the analysis does not need to infer them. Jimple programs are also expressed in three-address code, making each call and each assignment explicit and simpler to analyze than more complex constructs.

Distributivity

The original approach by Kildall assumes the *distributivity* property on the flow functions, requiring that $\forall x, y \in \mathcal{D}, f(x \sqcap y) = f(x) \sqcap f(y)$, meaning that whether the join operator is applied before or after the flow function, the results of the analysis are the same. Because many analysis problems are not distributive, and can therefore not be expressed with distributive flow functions, the Kam et al. approach relaxes the requirement to satisfy only monotonicity: $\forall x, y \in \mathcal{D}, f(x \sqcap y) \sqsubseteq f(x) \sqcap f(y)$ (with $\sqsubseteq$ the partial order of the lattice $\mathcal{D}$), from which the name of the framework—the monotone framework—originates. If the problem is distributive, the MFP is equal to the MOP. This is the case for taint analysis, since it is a distributive problem. Distributivity allows us to define the Just-in-Time analysis concept in Chapter 6, and to express taint analysis problems in the IFDS framework (which we detail in Section 2.3), as done with the analyses presented in Section 6.4 and Section 7.6.

Soundness and Precision

The flow functions of an analysis describe how each statement of a program is interpreted by the analysis. Real-world programs can be very complex, for example, containing constructs that are difficult to express in an analysis, such as multithreading, or by using external libraries or native code which cannot be analyzed easily. It is impossible to predict all of the code constructs used by all real-world programs, making it impossible to write an analysis that is completely sound and precise [67]. However, analysis developers can improve the soundness and precision of their analyses as much as possible by precisely analyzing aliasing, or the analysis being field-sensitive, object-sensitive, flow-sensitive, or context-sensitive [104, 120, 126, 129].

The taint analysis we use in Section 4.2 is field-sensitive, up to a maximum access path length of 4, meaning that objects with more than 4 nested fields can lower the precision of the analysis. The analysis does not support aliasing, but is flow-sensitive, object-sensitive, and context-sensitive, the latter being achieved through inlining. Despite the low performance caused by inlining and the lack of aliasing support, this analysis is sufficient to evaluate VISUFLOW in the context of the user studies presented in Section 4.2, as they do not evaluate the analysis' soundness or precision, but the usability of the tool that presents the analysis results.

The two taint analyses used in Section 6.4 and Section 7.6 support fields, up to access path lengths of 5 and 3, respectively. Both analyses support aliasing by relying on a black-box, field-sensitive alias analysis derived from Soot's intraprocedural $LocalMayAliasAnalysis$[1]. The analyses are flow-sensitive, object-sensitive, and context-sensitive. Context-sensitivity is provided through the framework that is used to express the analyses: IFDS, which we introduce in the next section.

2.3 The IFDS Framework

Context-sensitivity is a major issue when defining a data-flow analysis: the analysis must keep track of the different call sites it visits, so that it can return back to the correct caller method when it finishes analyzing a callee method. If not, the analysis must return to all callers to ensure soundness, causing a loss in both precision and scalability. To address this, the *call-strings approach* [120] explicitly records the call sites along with the data-flow facts, while the *functional approach* computes general analysis summaries that can be reused in any context.

The Interprocedural Finite Distributive Subset (IFDS) framework [110] belongs to the latter category. The taint analyses used in Section 6.4 and Section 7.6 use the IFDS implementation

---

[1] https://github.com/secure-software-engineering/cheetah/blob/master/Cheetah/src/layeredtaintplugin/icfg/LocalMayAliasAnalysisWithFields.java
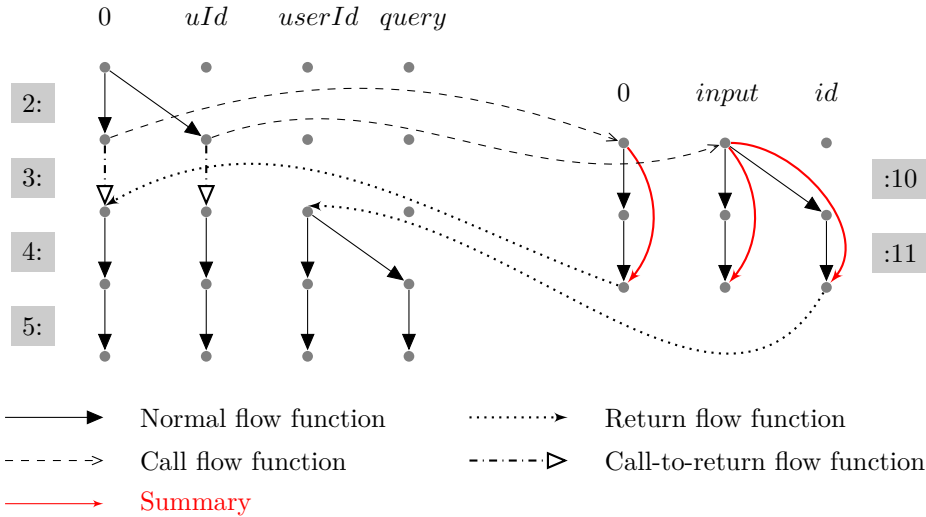
Figure 2.2: Exploded Super Graph (ESG) of the example Listing 2.1 for a forward taint analysis. The statements are represented by their line numbers (highlighted in gray).

Heros [15]. Built on top of the monotone framework, IFDS requires an analysis to be expressed as an interprocedural finite distributive subset problem, namely adding the constraints of distributivity, and the join operator being $\cup$, the set union. In addition, the framework separates the flow function into four distinct ones:

- *normal* flow functions, which encode the propagation of the data-flow facts in a method,

- *call* flow functions, which are used at call sites to map data-flow facts from the context of the caller to the context of the callee,

- *return* flow functions, which are also used at call sites and map data-flow facts back to the caller,

- and *call-to-return* flow functions, which transfer the data-flow facts that are not influenced by a call at a call site.

Figure 2.2 illustrates all four types of flow functions on an Exploded Super Graph (ESG). Unlike the monotone framework which only uses an ICFG, IFDS runs its fixed-point iteration by building an additional ESG. In an ESG, the statements are displayed vertically (represented by their line numbers in Figure 2.2), and the in-sets are separated in their individual data-flow facts and are represented horizontally. At a statement, the flow functions are called for each existing data-flow fact of the in-set instead of the entire in-set. Just like the in-sets are transferred to the out-sets in the monotone framework, each data-flow fact of the in-set is transferred to the data-flow facts composing the out-set. This separation of the data-flow facts is made possible due to the distributivity constraint of the flow functions: whether the join operator ($\cup$ in IFDS) or the flow function is run first, the analysis results are identical. Thus, the union of all data-flow facts at any program point will always be the same as the in-set or out-set the monotone framework would compute at that point.

IFDS transforms the analysis problem into a reachability problem: if a data-flow fact is reachable from the original data-flow fact 0 at the program's entry point, it means that the fact holds. In the example in Figure 2.2, if a data-flow fact can be reached from the 0 at line 2, it is tainted. For example, the data-flow fact *query* after line 5 can be reached from the original 0

through *id*, *input*, and *uId*, denoting that *query* is tainted after that statement. On the other hand, *userId* is not tainted after line 2, because it cannot be reached from the original 0. 0 is an artificial data-flow fact introduced in IFDS that is always reachable at any point of the program, ensuring that all parts of the program have the possibility to be reached from the original 0.

Since the analysis problem is a reachability problem in IFDS, the framework can analyze a callee once per data-flow fact and reuse the computations at the next call. This results in the creation of *summaries* that can be plugged into the caller methods regardless of the context. In the example in Figure 2.2, the *sanitize*() method has three summaries: $0 \rightarrow 0$, *input* $\rightarrow$ *input*, and *input* $\rightarrow$ *id*. If *sanitize*() is called a second time, the summaries can be reused without reanalyzing the method. The use of summaries allows analyses expressed in IFDS to be context-sensitive by default and to reduce the number of times a callee is analyzed, by only computing information along individual data-flow facts instead of entire in-sets.

We define below the flow-functions of the taint analyses we use in Section 6.4 and Section 7.6, marking with $\langle stmt \rangle(\alpha)$ the flow function applied to an access path $\alpha \in \mathcal{D}$ at a statement *stmt*.

Normal-flow function

$$\langle x \leftarrow y \rangle(\alpha) = \{\alpha\} \setminus \{x.*\} \cup \{x.\lambda \mid \alpha = y.\lambda\}$$
$$\langle x \leftarrow y \otimes z \rangle(\alpha) = \{\alpha\} \setminus \{x.*\} \cup \{x \mid \alpha = y \vee \alpha = z\}$$
$$\langle other \rangle(\alpha) = \{\alpha\}$$

The normal-flow function is the identity function except for assignment statements where one of the right-hand side operands is tainted. For a direct assignment, we remove all taints for the left operand (x and all of its fields), and apply to its base variable the taints that exist for the right operand and its fields ($x.\lambda$). In the case of a binary operator $\otimes$, Jimple requires all operands to be local variables, so we taint the left operand if any of the right operands are tainted.

Call-flow function

$$\langle x \leftarrow a.m(p) \rangle(\alpha) = \begin{cases} \{\alpha\} & \text{if } \alpha \text{ is static} \vee \alpha = 0 \\ \{this.\lambda\} & \text{if } \alpha = a.\lambda \\ \{arg.\lambda\} & \text{if } \alpha = p.\lambda \\ \emptyset & \text{otherwise} \end{cases}$$

For the call-flow function, taints for the base variable and the parameters are propagated into the callee. They are mapped to the *this* variable and the method's arguments in the scope of the callee. Static variables are propagated as well.

Return-flow function

$$\langle x \leftarrow a.m(p) \rangle(\alpha) = \begin{cases} \{\alpha\} & \text{if } \alpha \text{ is static } \vee \alpha = 0 \\ \{a.\lambda\} & \text{if } \alpha.\lambda = this.\lambda \\ \{p.\lambda\} & \text{if } \alpha.\lambda = arg.\lambda \wedge |\lambda| > 0 \\ \{x.\lambda\} & \text{if } \alpha.\lambda = retVal.\lambda \\ \emptyset & \text{otherwise} \end{cases}$$

The return-flow function maps the *this* variable and the arguments back to the base variable and parameters of the caller scope. Static variables are propagated back into the caller.

Call-to-return-flow function

$$\langle x \leftarrow a.m(p) \rangle(\alpha) = \begin{cases} \emptyset \text{ if } \alpha \text{ is static} \vee \alpha.\lambda = p.\lambda \vee \\ \qquad\qquad \alpha.\lambda = a.\lambda \\ \{\alpha\} \setminus \{x.*\} \cup \{newTaints(m) \mid \\ \qquad\qquad \alpha = 0 \wedge isSource(m)\} \\ \qquad\qquad\qquad \text{otherwise} \end{cases}$$

The call-to-return-flow function ensures that the variables that are affected by the call are not just propagated across the call on the side of the caller. Static variables, parameters, and the base variable are killed by returning $\emptyset$. Their taints, as well as that of the overwritten variable x, are carried over by the corresponding return flow function, depending on what happens in the callee(s). The function also propagates further such access paths not referenced by the call. If the method m is a source method, the newly tainted variables are added to the tainted set. Sinks are also handled in the call-to-return flow function: if the method m is a sink and if it leaks a tainted variable α, it is reported.

The Interprocedural Distributive Environment Framework

The Interprocedural Distributive Environment (IDE) framework is a generalization of IFDS to analyze problems that compute more data-flow information than a simple taint analysis, e.g., a *linear constant propagation*. While the presence or absence of a variable is enough to encode whether or not it is tainted, a linear constant propagation needs to convey the constant value of the variable. To do so, the IDE framework first runs an IFDS pass to determine all reachable data-flow facts, and then annotates the edges of the ESG with the so-called environment transformers that can, for instance, compute the constant values. Naturally, the requirements for expressing an analysis in IFDS also apply to IDE. In HEROS, the IFDS solver is a specialization of the IDE solver.

2.4 User-Centered Design

Many different design strategies can be adopted when designing a software product. A popular one is *functional design*, in which the product is divided in different functionality-specific modules to reduce the coupling between modules. Unlike functional design which focuses on optimizing the product's implementation, *user-centered design* aims at optimizing the product's usability, by understanding the user's needs and requirements and including them at every step of the design process [68, 143]. The goal of user-centered design is to create products with high usability, that answer the users' needs as closely as possible, as opposed to functional design where the product's functionality is the first priority, and the user is expected to adapt to the product. An example is the introduction of the desktop metaphor [1] in 1970 which proposed a new interface for interacting with computers: instead of a command-line interface proposing abstract commands to the users, it presented the computer like an abstraction of the user's work desk, with a file system, folders, and a trash can, in a graphical format. This metaphor matched the working environment of computer users at the time, and allowed them to understand and use computers more efficiently. The desktop metaphor quickly became the standard on all platforms, and is still in use today.

User-centered design is an iterative process along which prototypes are designed and tested, involving the user at every step of the process. Different versions of the user-centered design process exist, all revolving around three main steps: user research, design, and evaluation, as shown in Figure 2.3 [68, 112, 138, 143].
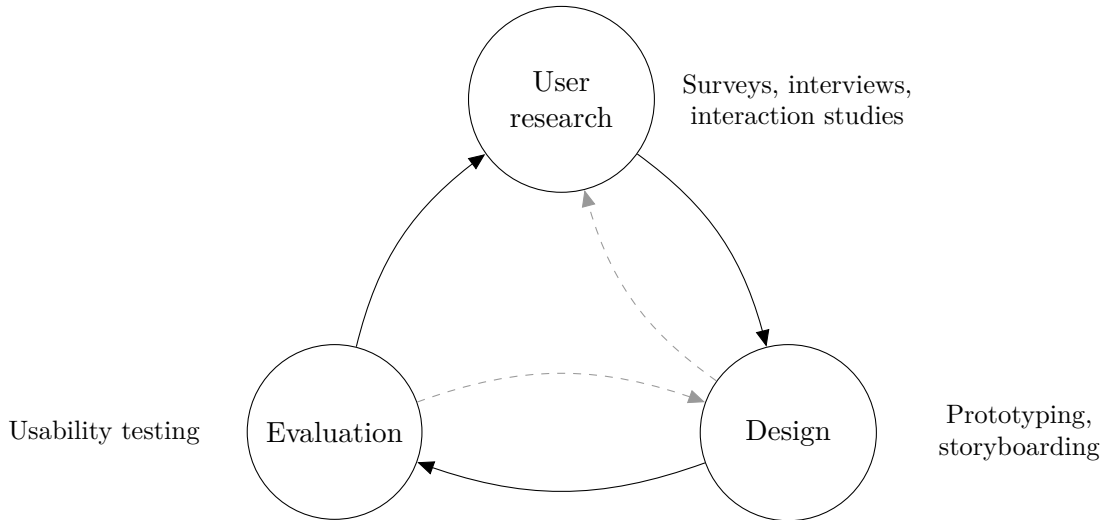
Figure 2.3: User-centered design process and tools for each step of the process.

User research is the first step to take when designing software for particular users. Its aim is to identify the users' needs and establish the requirements that the software must meet to satisfy those needs. At this stage, creating personas, scenarios, or user stories helps building an understanding of the users and their needs. More information can be gathered through *surveys*, *interviews*, or studies of the usage of existing tools, with the goal to gather knowledge about the context in which the users interact with the tool, and what they expect from it. With those needs, software designers then derive requirements and functionalities for their tool. The software architecture that results from this process can differ from the one that the functional design would yield: a small change in the interface can have large consequences on the implementation in the background.

The *design* step comes after identifying the user requirements. Designers produce alternative designs, focusing on how the user interface supports those requirements. Here, we refer to the user interface as not necessarily a graphical interface, but any means through which the user may interact with the system. Interfaces can be designed using various levels of *prototypes*, from low-fidelity sketches or paper prototypes, to high-fidelity wireframes or working software. The design step must be performed with the user in mind, and can lead to going back to the user research to determine use cases through *storyboarding*, or to conduct more studies that can help validate the design ideas.

In the *evaluation* step, software designers validate the design of their product. To involve users as much as possible in the design process, the evaluation typically includes *user studies* of different types, such as comparative studies where the differences between two prototypes are evaluated. In this thesis, we mainly conduct *within-subjects* studies [96], in which each participant tests all prototypes. As opposed to a *between-subjects* study—where a participant only interacts with a single prototype, this design allows us to compare a participant's behavior across different prototypes and to minimize the noise created by the personal knowledge of the participants. This is especially effective since our studies are held in a controlled lab environment. To reduce the learning effect between the different prototypes, we change the order in which the prototypes are shown to the participants. To ensure learning-fairness between the different prototypes, we follow a *latin-square design* [30]. In our studies, we also prime [97] our participants, so that they start the study with the same background knowledge. For example, we ask participants to perform with a simple example task, to reduce uncertainties due to task understanding or rule out usability issues unrelated to the prototypes. Another evaluation for-

mat is cognitive walkthroughs, in which the participant walks through a prototype step by step, detailing their reasons for taking specific actions, thus providing insight into how they interpret the interface. Once the evaluation is finished, the design process can stop if the prototype meets the requirements to satisfaction. Most often, new constraints and user-experience issues are revealed through the user studies, and the prototype is taken to the design step for adjustments, or back to the user research for a new iteration.

In this thesis, we apply the user-centered design process to create tools for analysis developers and software developers. For the first user group, we first research the usage context and user needs for debugging tools through a survey of analysis developers, and derive tool requirements from the survey results (Chapter 3). We then build VISUFLOW based on those requirements, evaluate it through a comparative user study with an existing debugging tool, and report the outcomes of the study (Chapter 4). For software developers, we run a survey, study usage logs of an analysis tool, and interview users using a paper prototype as part of the user research step (Chapter 5). Focusing on different user-experience issues, we design the tools CHEETAH and MUDARRI, and conduct comparative studies to verify the validity of our design ideas (Chapter 6 and Chapter 7).

Khoo Yit Phang's thesis [53] on *User-centered Program Analysis Tools* creates visualizations for analysis paths, and combines the results of multiple static and dynamic analyses together in a single, user-friendly tool. Unlike Khoo's approach which focuses on design and only evaluates one of its tools with the end-users, we apply the full cycle of the user-centered design process, starting with thorough studies to understand user motivations and needs. While Khoo's research address some of the requirements that we identify (e.g., visualizations to enhance warning explainability), we focus on others (e.g., responsiveness, or the integration of developer-specific knowledge in the analysis).

Identifying Tool Requirements for Analysis Developers

<div style="text-align:right">**3**</div>

In practice, static analysis tools are used by companies and individual developers to optimize programs, verify their compliance to security or quality guidelines, or to vet third party applications—for example in Google's Play Store, static analysis helped take down over 700,000 malicious applications in 2017, 99% of which were removed before anyone could install them [4].

As applications grow with new features, code concepts, frameworks, and libraries, static analysis tools must model increasingly complex systems and analyze larger codebases to properly support their users. However, static analysis code is also prone to bugs itself. An error in the rules of the analysis could have a large security impact on the applications made available to the public every day: over 86,000 applications were released on the Google Play Store in April 2018 [5]. While more complex analyses are written and used in production systems every day, the cost of debugging and fixing them also increases tremendously.

In this chapter, we focus on tool support given to analysis developers to write and debug their analysis code. We will refer to such tooling as *debugging environments*. We first introduce related work on existing tools and approaches for writing and debugging static analysis. Then, to understand the difficulties of debugging static analysis, we conducted a survey on 115 static analysis developers. In this survey, we aim at understanding the context of use of debugging tools for static analysis code (e.g, when and why analysis developers use those tools), which user-experience issues they encounter with those tools, and which features would best support them when developing analyses. From the survey results, we derive a list of requirements for building debugging environments for static analysis tools [91].

The work presented in this chapter has been published in the IEEE Transactions on Software Engineering journal (TSE) [91].

3.1 Related Work

Debugging static analysis has been overlooked in the software engineering community, and to our best knowledge, our work is the first one on the topic. Therefore, in this section, we discuss currently popular debugging tools and techniques for general software, and prior work on visualizing static analysis information.

3.1.1 Debugging Tools

The runtime environments of most programming languages are shipped with debuggers provided by the language maintainers (e.g., GDB [132]). Many IDEs, such as Eclipse [32] and IntelliJ [46],

integrate debugging functionalities for major programming languages natively in their tool sets. As a result, many analysis developers use IDE-integrated debuggers to debug static analysis code. As our survey shows, such tools are designed for general application code, and do not have specific support for static analysis. More complex debugging techniques such as delta debugging [150], omniscient debugging [64], and interrogative debugging [57] also suffer from the same issue. Moreover, they are not integrated into commonly used debugging tools.

3.1.2 Debugging Static Analysis

The problem of debugging static analysis code compared to more general application code revolves around the nature of the program's input: the analyzed code. Because the analyzed code is a program, it is typically more complex than other types of input (e.g., parameters for a web request, or text file to parse). An analysis must reason about how its input program behaves when it is run, adding another layer of indirection to the development and debugging tasks.

We are not aware of any tool that is tailored to address issues specific to debugging static analysis. In past work, Andreasen et al. [3] suggest to employ soundness testing, delta debugging, and blended analysis to debug static analysis. Through examples, they discuss how the combination of these techniques (both pairwise and all three of them) have helped them locate and fix bugs in their static analyzer TAJS. In our approach, we aim to determine feature requirements based on what analysis developers need to build an appropriate debugging tool.

Other tools provide a subset of the features that we identify in this chapter for a static analysis debugger, especially in terms of visualization of information and data flows. For example, Atlas [27] visualizes data-flow paths based on the abstract syntax tree (AST) of a given program. Phang et al. [54] present a tool that visualizes program paths to help the user track where an error originates from to improve user understanding and error report evaluation. However, none of these tools enable static analysis developers to debug their own analyses, but are rather tailored to the users of static analysis tools (e.g., software developers), and therefore focus more on visualization features than debugging features.

A similar domain to static analysis is metaprogramming, in which a program can operate on other programs—in most cases, itself. In this domain, the work closest to our own is Porkoláb et al.'s [107], on debugging C++ template metaprograms. Porkoláb et al. name feature requirements similar to the ones we discover for static analysis, such as the need to make the CFG and intermediate analysis values visible to the developer. Our study is focused on the domain of static analysis, and aims to identify all requirements needed to build a debugger for this use case. Unlike Porkoláb et al., we study the usage context and motivations of analysis developers, and identify additional requirements, such as omniscient debugging for example.

3.2 Survey: Debugging Tools for Static Analysis

We designed the following survey to understand the usage context of debugging tools for static analysis and their user-experience issues. In the survey, we asked the participants to distinguish between *analysis code* (the static analysis code) and *application code* (the code that is typically analyzed by the analysis code). The latter may range from small test cases to large, complex systems, and encompasses all programs that are not analysis code.

With the survey, we aim to answer the following research questions:

- **RQ1**: Which types of analysis are most commonly written?

- **RQ2**: Which errors are most frequently debugged in analysis code and application code?

16

- **RQ3**: Do analysis writers think that analysis code is harder/easier to debug than application code, and why?

- **RQ4**: Which tools do analysis writers use to support the debugging of analysis code and application code?

- **RQ5**: What are the limitations of those tools and which features are needed to debug analysis code?

The survey contains 32 questions (grouped into five thematic sections) that we refer to as **Q1**–**Q32**, in the order in which they were presented to participants. Unless specified otherwise, all questions are multiple-choice questions, with an open "Others" text field. The questions are detailed in Appendix A.

1. **Participant information:** We use **Q1**–**Q10** to gather information about the participants. **Q1** asks for their background—academia or industry, **Q2** for their research topic—in free text, and **Q3** for how long they have been writing analysis code. In **Q8**, we also ask the participants if they have written analysis code for commercial tools, and which commercial tools those are.

2. **Hardness of debugging:** We ask which type of code is easier to debug on a Likert scale from 1 (application code) to 10 (analysis code) (**Q11**), and why (**Q12**), in free text. **Q13** then queries participants on how long they spend on writing analysis code compared to debugging it on a Likert scale from 0 (100% writing, 0% debugging) to 10 (0% writing, 100% debugging). **Q20** is the same question as **Q13**, but for application code.

3. **Debugging context:** To understand when and how analysis developers use debugging tools, we ask questions about the analyses participants have written in the past. **Q4** queries participants about the programming languages they have written analyses for, **Q5** asks them which purposes those analyses serve (e.g, security, program optimization, etc.), **Q6** for the types of analyses (e.g, data-flow, abstract interpretation, etc.), **Q7** for the program level at which the analyses operate (e.g., line-based analyses, full-program analyses, etc.), **Q9** for the analysis frameworks they used, and **Q10** for a few examples of analyses that they have written, in free text. In free text, we then ask for the reasons why the participants start debugging the analysis code (**Q14**), and what the root causes of errors are (**Q15**). **Q21** is the same question as **Q15**, for application code.

4. **Debugging tools:** This section gathers information about the debugging environments used by analysis developers. We first ask them in **Q28** if their preferred editor is a text editor (e.g., vim, emacs) or an Integrated Development Environment (IDE) (e.g., Eclipse, IntelliJ). We then ask which particular editors they use (**Q29**), and why they prefer those editors (**Q30**), both in free text. Also in free text, we ask participants which debugging features they most use (**Q16**), like (**Q17**), dislike (**Q18**), and would like to have (**Q19**) when debugging analysis code. **Q22**–**Q25** are the same as **Q16**–**Q19**, applied to application code. Finally, with **Q26**, participants rated how important certain debugging features are to them on the following Likert scale: *not important*, *neutral*, *important*, *very important*, and *not applicable*.

5. **Contact information:** In **Q31** and **Q32**, we ask participants if they would be willing to participate in further studies, and if yes, which email address we could contact them at.
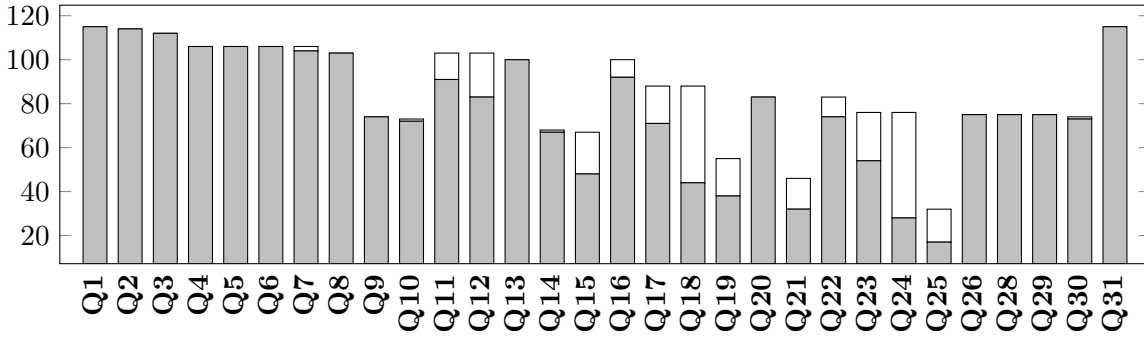
Figure 3.1: Number of valid (gray) and invalid answers (white) per question.

Pilot Survey

We sent a pilot survey to ten pilot participants (both students and researchers). Using their feedback about length, quality, and understandability, we modified the survey and obtained the final version described above. We namely shortened the survey from 48 to 32 questions, removing questions that were too specific to particular areas of static analysis. The pilot participants also noted that similar questions such as the ones that became **Q13** and **Q20** in the new survey were confusing when asked directly one after the other. We thus grouped the questions about static analysis together and those about application code together, clearly focusing the first part of the survey on the participants and their analyses, the second part on analysis development, and the last part on application development.

Participants

Analysis developers in industry were more difficult to find and approach, and could not divulge some of the information that we requested. Although most of our participants are from academia (85.2%), we also include analysis developers from industry (15.7%), with three participants affiliated to both (**Q1**). For our survey, we emailed authors of static-analysis papers published between 2014 and 2016 at the following top conferences in Software Engineering and Programming Languages, and their co-located workshops: ICSE, FSE, ASE, OOPSLA, ECOOP, PLDI, POPL, SAS. We manually extracted the static-analysis related papers from the conference proceedings, and reached out to the 450 authors we gathered. We received responses from 117 researchers, and discarded two for quality issues.

Most participants are experienced static analysis developers. Approximately 31.3% of the participants have 2–5 years of experience writing analysis code, 22.3% have 5–10 years of experience, 26.8% have more than ten years of experience, and only 9.8% have less than two years. The rest has never developed an analysis, or did not answer **Q3**.

Survey Analysis

In the following sections, we present the answers given by analysis developers to our survey. Because some of the survey questions were in free text format, and the participants were given the choice of an open "Others" answer to the multiple choice questions, we applied an open card sort [45] methodology to distribute such answers in different categories. In this type of card sorting, the categories are derived as the answers are classified. Two researchers created the categories as they were classifying the answers, and a third researcher then classified the answers into the categories derived by the first two. Answers that could match multiple categories were sorted in all matching categories, such as "I use breakpoints and stepping", which matches both of

the "Breakpoint" and "Stepping" categories. Responses that do not answer the question were classified in an "Invalid" category (e.g., "n/a" **Q24**). Such answers were more often found in the last part of the survey about application code, where some participants were confused by the familiarity of the questions to the analysis code part, and did not notice that they now applied to application code. Answers such as "I think I just answered this one." (**Q22**) were thus classified into the "Invalid" category.

To verify the validity of the classification, we compared the agreement between the two rounds of classifications between the first two researchers and the third one. Since one answer can be classified into multiple categories, we calculated the agreement for each category of each question. Because of the imbalance in the distribution of the answers, we ran into a paradox of inter-rater agreement [141], making the Cohen's Kappa [61] an unreliable statistic for this survey (average $\kappa = 0.66$, median $\kappa = 0.7$, min = -0.08, max = 1, $\sigma = 0.33$). We thus computed the percent agreement instead. The average percent agreement over all categories for all questions is 96.3% (median = 98%, min = 65.2%, max = 100%, standard deviation $\sigma = 0.05$).

We only report the most relevant answers to the survey questions: in general answers that are chosen by more than one participant. Our statistical tests include all valid answers. All questions and anonymized answers are available online [86]. Since participants were given the possibility to choose multiple answers to the same multiple-choice question and an answer to a free-text question could match multiple categories, the percentages we report for each question may add up to more than 100%. Due to optional questions and participants who did not finish the survey, some questions received fewer answers than others. Figure 3.1 reports the number of valid (gray) and invalid (white) answers per question. The percentages we report for each question are based on the number of valid answers, and not on all 115 answers.

3.3 Usage Context and Developer Motivation

To gain a better understanding of how analysis developers use their debugging tools, we first gather information about the types of analysis written by the developers, the reasons for which they start to debug an analysis, and the root causes of the analysis bugs they find at the end of a debugging session. We also report on which type of codebase (analysis code or application code) the participants think is harder to debug, and why.

3.3.1 Most Commonly Developed Analyses

Table 3.1 presents an overview of the different types of analyses written by the participants. The main analyzed language is Java, with 62.3% of the participants writing analyses for this language. C/C++ comes second with 59.4%, and JavaScript is third, with 23.6%. After that, 34 more languages were named, but the number of participants analyzing each of them was under 8%. Those results are consistent with the state of research in static analysis at the time of the survey (in 2017), which was mostly applied to Java and its related programming languages (e.g., Android) (**Q4**).

Most of the analyses revolve around four main uses: security and privacy analyses aiming at finding security vulnerabilities and data leaks (58.5%), analyses that support software developers in program understanding (55.7%), analyses that detect functional bugs (52.8%), and analyses that support performance optimization (29.4%). All other 20 types of analyses are reported by less than 2% of the participants (**Q5**). This is also consistent with current analysis tools used in industry: the vast majority of them aim to find security vulnerabilities (e.g., Checkmarx [20], Coverity [26], FindBugs [121], etc.)

Table 3.1: Analyses developed by the participants.

Analyzed Language (Q4)	
Java	62.26%
C / C++	59.43%
JavaScript	23.58%
Every other language	$\leq 7,55\%$

Purposes (Q5)	
Security and privacy	58.49%
Program understanding	55.66%
Functional correctness	52.83%
Performance optimization	27.36%
Every other purpose	$\leq 1.89\%$

Framework (Q9)	
Soot	55.41%
WALA	31.08%
LLVM	21.62%
Every other framework	$\leq 9.46\%$

Analysis Type (Q6)	
Data-flow	74.53%
Abstract interpretation	65.09%
Symbolic execution	36.79%
Model checking	21.70%
Every other type	$\leq 3.77\%$

Analysis Level (Q7)	
Program-based	71.15%
Function-based	50.96%
Line-based	40.38%
Module-based	26.92%
System-based	14.42%

The same trend appears with the branches of static analysis that participants write for (**Q6**). Data-flow analysis is the most popular (74.5%), because it can be used to find most of the top security vulnerabilities (e.g., SANS top 25 [106,117]). Abstract interpretation, symbolic execution, and model checking come next, with 65.1%, 36.8%, and 21.7% of the participants respectively. The remaining nine categories are each used by less than 4% of the participants.

Depending on the property that the analysis aims to verify and the complexity of the analysis, the analysis considers different subsets of the program (**Q7**). For example, simple style checkers typically verify properties based on one line of code, while the detection of injection flaws may traverse an entire program or system so an analysis must be able to consider system-wide data flows. The program level that is most used is program-based, with 71.2% of the participants writing analyses that take the entire program as a context, 51% write intra-procedural analyses—on the method level, and 40.4%, on the line level. Analyses considering modules (e.g., packages) or entire systems are less popular (26.9% and 14.4%, respectively).

We asked the participants which frameworks they use to write static analyses, and received three main answers: Soot [140] is the most popular framework, with 55.4% of the participants using it. WALA [135] and LLVM [133] are second and third (31.1% and 21.6%, respectively). The 32 other frameworks named by the participants are used by less than 10% of them (**Q9**).

3.3.2 Most Frequently Debugged Errors

Knowing what kind of analyses are written most often, we now focus on the types of errors that are debugged most often. Asking for why analysis developers start a debugging session (**Q14**), we find that the top three reasons revolve around analysis correctness. As shown in Table 3.2, 47.1% of the participants start debugging because the analysis did not produce the correct results, and 31.4% and 29.4% explain that they debug because the respective precision and soundness of their analysis is not satisfying. Unlike general application code, static analysis code has stricter requirements for correctness: wrongly handling one statement can result in a wrong data-flow fact being propagated throughout the entire codebase, in turn causing the

Table 3.2: Reasons why analysis developers start a debug session (**Q14**).

Reason	% of Devs
Unexpected results	47.06%
Precision	31.37%
Soundness	29.41%
Performance	15.69%
Bad termination	13.73%
Program understanding	5.88%
Memory issues	3.92%
Intermediate representation handling	3.92%
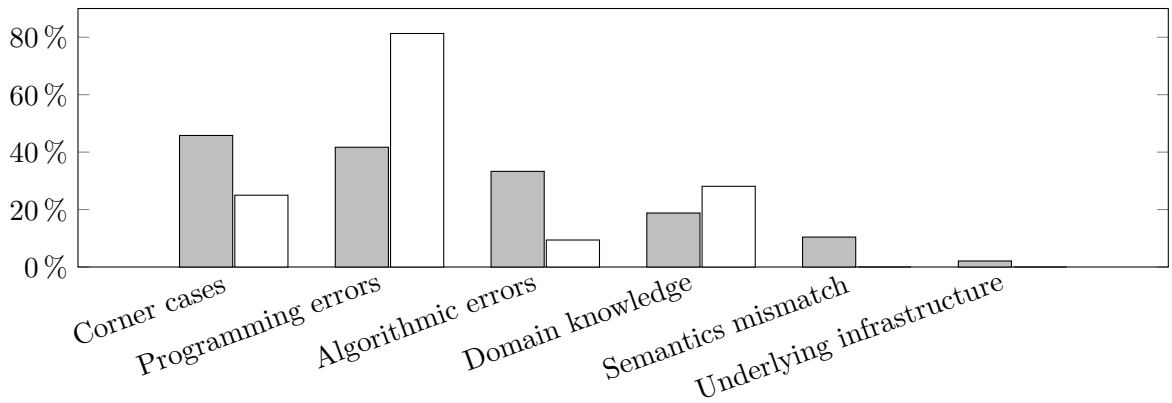
Figure 3.2: Root causes of the errors found when debugging analysis code (gray) and application code (white) (**Q15** and **Q21**).

propagation of other wrong data-flow facts, resulting in false positives or false negatives. More minor reasons for starting a debugging session are due to the bad performance of the analysis (15.7%), and the wrong termination of the analysis algorithm (13.7%).

At the end of a debugging session, analysis developers know the root cause of the error. Figure 3.2 presents the most typical root causes found when debugging analysis code (**Q15**) and application code (**Q21**), classified in six categories. We notice that when debugging analysis code, the main cause for errors is handling *corner cases*, which is twice as prevalent as when writing analysis code. This category includes overlooked cases that the developer normally knows of. For example, for **Q15**, a participant answered "Forgot to consider the effect of certain, rare instructions". On the other hand, *domain knowledge* refers to code behavior that the developer is unaware of in the application code or the analyzed code, such as "Unexpected values returned by an API". This category has the smallest difference between analysis code and application code.

Programming errors are also found in both analysis code and application code, but they occur twice as often in application code. This category includes implementation errors, such as "wrong conditions, wrong loops statements", as opposed to *algorithmic errors*, which contain errors due to a wrong design of the program's algorithm (e.g., "non-convergence" of the analysis) (**Q15**). Algorithmic errors are 3.5× more prevalent in analysis code than in application code.

Two categories are specific to analysis code: *semantics mismatch*, which is specific to how the analysis interprets the analyzed code (for example "The code does not take [into] account the abstract semantics correctly"), and *underlying infrastructure*, which is similar to *domain knowledge*,
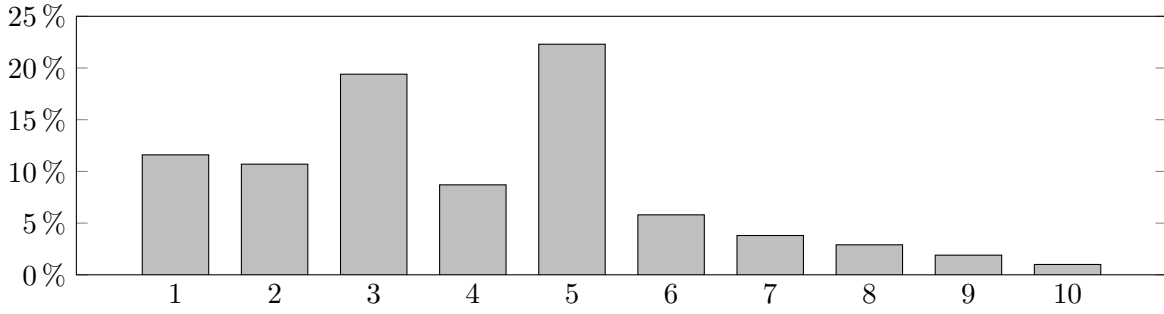
Figure 3.3: Difficulty of debugging analysis code compared to application code on a scale from 1 (analysis code is harder) to 10 (application code is harder) (**Q11**).

but instead of the knowledge of the analyzed code, it is about the knowledge of the analysis framework, e.g., "Can't load classes/methods successfully".

We observe different interests when debugging analysis code or application code. While bugs in application code are mainly due to programming errors, the root causes of static analysis bugs are distributed over multiple categories. We attribute this to the heightened interest of analysis developers to produce not only functional analyses, but sound and correct ones. Testing functional correctness typically requires validating input/output relationships. For analysis code, those relationships are always imperfect due to necessary approximations. Hence, it is difficult to define functional correctness for static analysis. Another specificity of static analysis, handling two code-bases, is also the cause of errors belonging to analysis-specific categories: semantics mismatch, and underlying infrastructure.

Because of the specific requirements of static analysis, the bugs that developers investigate in application code have different causes than the bugs found in analysis code. Therefore, there is a need for supporting debugging analysis code for the specific kind of errors that are of interest to analysis developers.

3.3.3 Comparison with Application Code

Based on the knowledge of which errors are most commonly found in analysis code compared to application code, we ask participants which errors are easier to debug, and why.

In **Q11**, participants rated how hard debugging analysis code is compared to debugging application code on a Likert scale from 1 (analysis code is harder to debug) to 10 (application code is harder to debug). The average rate is 4.0 (standard deviation $\sigma = 2.1$). Figure 3.3 shows that 50.5% of the participants find static analysis harder to debug than application code, 28.2% are neutral, and 9.5% think that application code is harder to debug.

This observation is confirmed with **Q13** and **Q20**, where participants reported that they spend more time debugging a piece of static analysis code than writing it, and the contrary for a piece of application code. On average, they spend 46.8% of their time writing analysis code, and 53.2% of their time debugging it. For application code, they spend 57.5% of their time writing, and 42.5% debugging. A $\mathcal{X}^2$ test of independence does not detect significant correlations ($p > 0.05$) between the rating of **Q11** and the participants' background (seniority, coding languages, editor type, or analysis frameworks), suggesting that no matter the type of analysis, debugging analysis code is perceived as more difficult than debugging application code.

When asked why they found one type of code harder to debug than the other (**Q12**), participants gave different answers. We classified them and present them in Table 3.3. The classification yields ten reasons for which static analysis is more difficult to debug than application code, seven

Table 3.3: Reasons why developers think that analysis code is harder to debug than application code, and vice versa (**Q12**).

Harder to debug	Reason	% of Devs
Analysis code	Abstracting two types of code	15.7%
	Greater variety of cases	15.7%
	More complex structure of static analysis tools	6.0%
	Evaluating correctness is harder	6.0%
	Soundness is harder to achieve	3.6%
	Intermediate results are not directly accessible	4.8%
	Static analysis is harder to debug	3.6%
Both	Both are application code	13.3%
	They cannot be compared	7.2%
	No opinion	3.6%
Application code	Used to developing static analysis	6.0%
	Application code is more complex	2.4%

reasons for why they are equally difficult to debug, and nine reasons for why application code is harder to debug.

The main reasons cited by participants who thought that static analysis was harder to debug focused on the complexity of handling two codebases at the same time: the analysis code and the analyzed code: "Static Analysis requires to switch between your analysis code and the Intermediate Representation which you actually analyze". This difficulty creates more corner cases that need to be handled by the analysis developer. Another argument was the requirements of static analysis tools. While both the application code and the analysis code need to be functional, static analyses also have to ensure soundness, making static analysis code more complicated to write. To quote a participant: "'correct' is better defined [in application code]". The last reason mentioned by the participants is that "static analysis is harder to debug", pointing to one particular cause: the intermediate results of the analysis are not directly verifiable, as opposed to the output of application code: "Static analysis code usually deals with massive amounts of data, i.e., large programs. Therefore, it is harder to see where a certain state is computed, or even worse, why it is not computed at some point."

Participants who noted that static analysis and application code were equally hard to debug were split between two main arguments. One was that both codebases were application code: "a static analyzer is an application, albeit a sophisticated one", and the other, that they were so different that they could not be compared: "These two difficulties are qualitatively different and hence incomparable."

Participants who find application code more difficult to debug than static analysis mentioned that application code was more complex than static analysis code and can contain a high number of corner cases the application developer has to comprehend: "Static analysis code usually includes very limited number of possible cases." Some participants also wrote that the reason why they found application code harder to debug was that they were used to developing static analysis.

Discussion (RQ1–RQ3)

We observed that the most analyzed programming language is Java, that analyses most often target security vulnerabilities, that data-flow analyses, and program-level analyses are written more often than other types of analysis, and that Soot is the most popular analysis framework.

The primary concern of analysis developers when starting a debugging session is the correctness (soundness and precision) of their analysis. When debugging such errors, the most recurrent root causes are corner cases, algorithmic errors, semantics mismatches, and unhandled cases in the underlying analysis infrastructure. In comparison, programming errors are the one main cause of bugs in application code. We see that because of the specific requirements of static analyses, the bugs that developers investigate in application code and analysis code have different causes, thus motivating the need for supporting debugging static analysis for the specific kind of errors that are of interest to analysis developers.

Static analysis developers spend 10.7% more time debugging static analysis than they spend debugging application code. In fact, more than half of the participants (50.5%) find static analysis harder to debug because of three main reasons: handling two codebases at the same time, the soundness requirements of analysis tools, and the lack of debugging support for static analysis. With this knowledge, we explore how to best design features to support analysis debugging in the next section.

3.4 Desirable Features for Debugging Static Analysis

Knowing why developers debug analyses, we now focus on the tools analysis developers use to debug analyses, in particular which features are most useful, inconvenient, and wanted. We compare tool features for debugging analysis code compared to application code, and derive a list of requirements for building a debugging environment for static analysis.

3.4.1 Debugging Tools Used by Analysis Developers

When asked which editors they use to write analysis code in **Q28** and **Q29**, 56.0% of the participants answered that they use an Integrated Development Environment (IDE) such as Eclipse or IntelliJ, 42.7% use text editors like vim or emacs, and 1.3% use another solution that they did not specify. The most popular editor is vim [16], used by 33.3% of the participants. Eclipse [32] is second, with 28.0%. Emacs [38] and IntelliJ [46] come third and fourth, with 21.3% and 17.3%, respectively. The other 21 tools are each used by less than 10% of the participants. We note that all of the tools named here are general debugging tools for application code, thus motivating the need for analysis-specific debugging environments.

We asked participants which features of their coding environments are most useful when debugging analysis code (**Q17**) or application code (**Q23**). Table 3.4 shows the features mentioned by more than one participant. We group the answers in four categories: IDE users writing analysis code, text editor users writing analysis code, IDE users writing application code, and text editor users writing application code.

We can see that many features are common to all four groups of users. *Breakpoints* are used by 35.2% of the participants when debugging application code, and 28.2% of the participants when debugging analysis code. *Coding support* (such as auto-completion), is appreciated by 29.6% of the participants when writing analysis code, and 20.4% when writing application code. *Variable inspection* functionalities are used by 27.8% of the participants when debugging application code, and 19.7% when debugging analysis code. *Debugging tools* such as "GDB/JDB" are also used: 20.4% of the participants use them when writing application code, and 16.9% when writing analysis code. In addition, IDE users highlight IDE-specific features such as *type checkers*, *stepping*, and *hot-code replacement*. It is interesting to note that *printing* the intermediate results is a popular solution for debugging both analysis and application code.

A $\mathcal{X}^2$ test of independence shows a strong correlation between the type of editor (IDE or text editor) and the most useful debugging features ($p = 0.01 \leq 0.05$) for application code. The

Table 3.4: Useful features for debugging analysis code and application code for IDE users and text editor (TE) users (**Q17** and **Q23**).

Feature	Analysis code		Application code	
	IDE	TE	IDE	TE
Printing	✓	✓	✓	✓
Breakpoints	✓	✓	✓	✓
Debugging tools	✓	✓	✓	✓
Coding support	✓	✓	✓	✓
Variable inspection	✓	✓	✓	✓
Automated testing	✓	✓	✓	✓
Expression mode	✓	✓	✓	✓
Memory tools		✓	✓	✓
Graph visualizations		✓		✓
Stepping	✓		✓	
Type checker	✓		✓	
Hot-code replacement	✓		✓	
Visualizations				✓
Stack traces		✓		
Drop frames			✓	
Documentation	✓			

test does not find such a correlation for analysis code, indicating that the debugging features used when writing analysis code are the same in both types of coding environments.

Overall, we observe that the most popular debugging features are traditional debugging functionalities. Regardless of the coding environment, analysis developers use the same debugging features to debug analysis code and application code, e.g., breakpoints, variable inspection, coding support, and printing intermediate results.

3.4.2   User-Experience Issues with Current Debugging Tools

Through **Q18** and **Q24**, we focus on the limitations of the debugging tools presented in Section 3.4.1. The two questions ask participants about which features of their coding environments are inconvenient to use when debugging analysis code and application code, respectively. We show the features mentioned by more than one participant in Table 3.5.

We see that the features that gather the most dissatisfaction from the participants are general debugging features. Of all participants, 29.5% find the *standard debugging tools* lacking when debugging analysis code, against 25% for application code. The *lack of immediate feedback* when a change is made in the code is pointed by 11.4% of the analysis developers, and 17.9% of the application code developers. *Coding support* is also disliked by 18.2% of the participants when writing analysis code, against 25% when writing application code.

To our surprise, two of the most disliked features—*debugging tools* (disliked by 29.5% when debugging analysis code and 25% when debugging application code) and *coding support* (18.2% for analysis code and 25% for application code)—are also among the most used and appreciated (Table 3.4). This contradiction suggests that although current tools are useful, analysis developers require more specific features to fully support their needs. For example, a participant wrote: "While the IDE can show a path through [my] code for a symbolic execution run, it doesn't show analysis states

Table 3.5: Unsatisfactory features for debugging analysis code and application code for IDE users and text editor (TE) users (**Q18** and **Q24**).

| *Feature* | *Analysis code* | | *Application code* | |
	IDE	*TE*	*IDE*	*TE*
Debugging tools	✗	✗	✗	✗
Immediate feedback	✗	✗	✗	✗
Coding support	✗	✗	✗	✗
Multiple environments		✗	✗	✗
Intermediate results	✗	✗		
Handling data structures	✗		✗	
Support for system setup			✗	✗
Scalability	✗			
Visualizations	✗			
Conditional breakpoints	✗			
Memory tools			✗	
Bad documentation				✗

along that path". Debugging tools could thus be improved by showing more of the *intermediate results* of the analysis, and by providing more support to handle different environments (e.g., participants complained about the "Manual work to setup complex build/test systems").

In particular, participants using an IDE to write analysis code find that debugging tools are not *scalable*, lack *visualizations* of analysis constructs (e.g., "It's mostly text-based"), and need *special breakpoints* that can be controlled from both the analysis code and the analyzed code, allowing them to step into both codebases independently (e.g., "Missing an easy way to add a breakpoint when the analysis reaches a certain line in the input program (hence having to rerun an analysis)").

Following from our survey results, we see that current debugging tools lack specific functionalities to support static analysis such as showing the analysis' intermediate results, or providing clear visualizations of the analysis and special breakpoints.

3.4.3   Tool Features for Debugging Analysis Code

To identify which debugging functionalities would best support analysis developers, we asked participants to suggest useful features for debugging analysis code (**Q19**), and application code (**Q25**). We present the features mentioned more than once in Table 3.6.

We see a clear distinction between the features requested for debugging application code and analysis code. For application code, participants ask for better *hot-code replacement*, and *coding support* (including help with handling complex data structures: "Better support to record complex data coming from external services"). For static analysis code, 18.4% of the participants ask for better *visualizations of the analysis constructs*, and 23.7% indicate that they would like to have *graph visualizations*: "Easier way to inspect "intermediate" result of an analysis, easier way to produce state graphs and inspect them with tools". *Omniscient debugging* is requested by 13.2% of participants to help show the intermediate results of the analysis: "Stepping backwards in the execution of a program". Participants also request better *test generation tools* and *special breakpoints*. We also see demands for better *test generation*, better *debugging tools* in general, and the *special breakpoints* mentioned in Section 3.4.2.

Table 3.6: Requested features for debugging analysis code and application code for IDE users and text editor (TE) users (**Q19** and **Q25**).

Feature	Analysis code		Application code	
	IDE	TE	IDE	TE
Graph visualizations	✓	✓		
Omniscient debugging	✓	✓		
Visualizations	✓	✓		
Hot-code replacement	✓		✓	
Coding support			✓	✓
Test generation		✓		
Debugging tools		✓		
Intermediate results	✓			
Conditional breakpoints	✓			
Handling data structures			✓	

A $\mathcal{X}^2$ test on the features of Table 3.6 shows a correlation between the type of code (analysis code or application code) and the features requested by participants ($p = 0.04 \leq 0.05$), motivating the need for specific tooling for debugging static analysis code in particular. The same test does not yield significant correlation coefficients between the type of code and the features marked as useful or inconvenient by participants, with p-values of 0.97 and 0.69, respectively. This correlation suggests that while analysis developers use currently available debugging tools—which are designed for general application code—debugging functionalities that are more specific to static analysis would be needed to properly support the development of analysis code.

In addition, the test shows strong correlations between the type of editor (IDE or text editor) and the requested features for analysis code ($p = 0.02$) and application code ($p = 0.04$). Such correlations cannot be found for features that participants deemed useful or inconvenient, suggesting that the tools used to debug application code and analysis code contain features that all types of users equally like and dislike. We see that regardless of the editor, the requested features for writing analysis code and application code are quite different.

We further evaluate the importance of the features identified with **Q19** in a debugging environment for static analysis. The participants were asked to grade the features shown in Figure 3.4 on a Likert scale from *Not important* to *Very Important* (**Q26**).

All features are generally well-received. Namely, four features are deemed *Important* or more by over 73% of the participants. Among visualizations, *graph representations* are considered most important. Access to the *intermediate representation* and to the *intermediate results* of the analysis is also very important, along with *breakpoints* that take both the analysis code and the analyzed code into account, and their associated *stepping* functionalities. Those features have the particularity of considering both the analysis code and the analyzed code. Although they are less popular, other features such as *Other types of visualizations*, better *test generation*, and *quick updates* are considered *Important* or more by at least 52% of the participants.

A $\mathcal{X}^2$ test of independence with the participants' backgrounds (seniority, coding languages, editor type, or analysis frameworks) only shows a strong correlation between the desirable features and the editor type (text editor or IDE) ($p = 0.02 \leq 0.05$). This observation suggests that the desirable features are generalizable to the different types of static analysis and analysis developers represented by our participants, and that the set of desirable features only differs according to whether the analysis developer uses a text editor or an IDE.
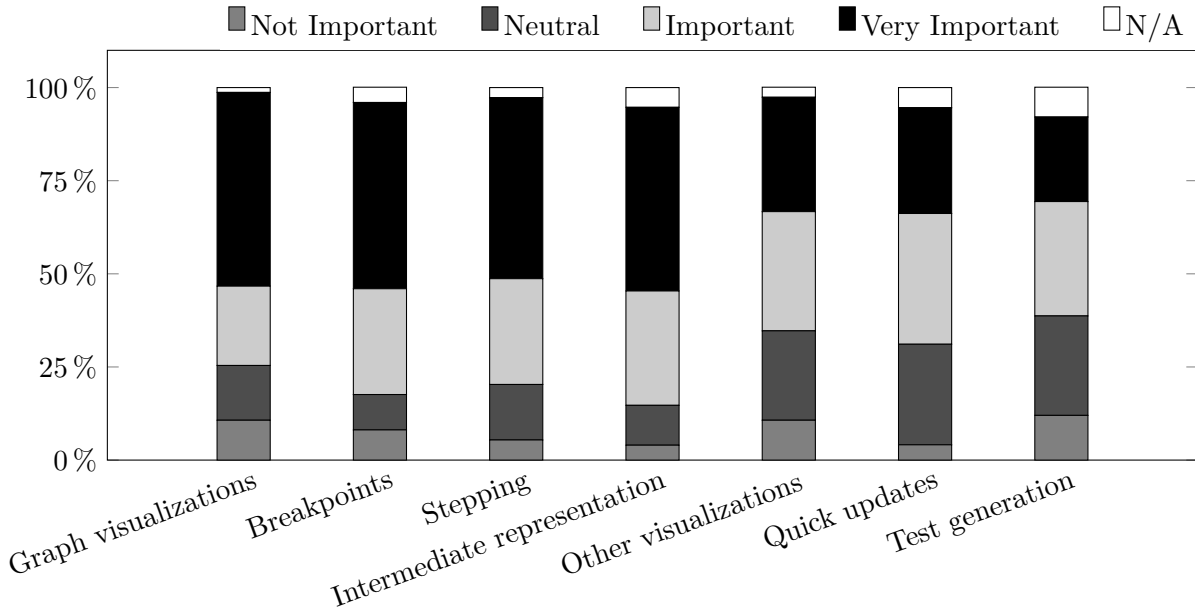
Figure 3.4: Ranking the importance of features for debugging static analysis (**Q26**).

Discussion (RQ4–RQ5)

We saw that to debug their code, analysis developers mainly use the traditional debugging tools and features included in their coding environments such as breakpoints and variable inspection. While those tools are helpful, they are not sufficient to fully support static analysis: debugging features such as simple breakpoints fall short to filling the requirements debugging static analysis entails, and force analysis developers to handle parts of the debugging process manually. Debugging analysis code therefore requires features that are not provided by current debugging tools. Those features revolve around improving the visibility of how the analysis code represents and analyses the analyzed code, a process that is typically hidden when using traditional debuggers that only handle one codebase.

3.5 Limitations and Threats to Validity

In our survey, we reached out to 450 authors of static analysis papers published between 2014 and 2016. The 117 participants who answered the survey are mainly researchers in static analysis, with only 15.7% from industry, so our sample may not fully represent analysis developers in general. We argue that the requirements and tools used by analysis developers in academia or in industry should not widely differ, as researchers in industry gave us information consistent with the other participants'. Another survey gathering information from the industry may be interesting to conduct, to verify the outcomes of our survey.

Because the survey participants were specialists in static analysis, their experience of developing and debugging application code may not be on par to their experience with debugging analysis code. An interesting observation is the reported 42.5% of the participants' time dedicated to debugging application code (**Q20**), which is different from the average time programmers spend on debugging their code shown in a study conducted by the University of Cambridge: 49.9% [17]. While a better option would have been to report on two surveys: one aimed at analysis developers and one for code developers, only few software developers know about static analysis development so the answers we would have gathered would be less com-

plete. In addition, we conducted a similar survey aimed at developers in industry, but only received ten answers over the span of two months, so we did not report on it.

We classified the free-text answers into categories by hand, which is, of course, subjective. To make the classification as objective as possible, two of the authors classified the answers together, and a third author verified the classification, following the open card sort methodology [45]. We report on the agreement measurements in Section 3.2.

Q11 of the survey was misinterpreted by a few participants: their answers do not match the explanation given in **Q12**. For example, a participant wrote in **Q12** that "debugging SA [static analysis] is still a bit harder [than application code]", and gave **Q11** a score of 7 (scale from 1 to 10), denoting the contrary. In such clear cases, we reversed the score (in this example, the new score is 4). We reversed only 12 scores out of 103 responses.

The features identified in this survey are applicable to the domain of static analysis. Some of them can also generalize to other branches of software engineering that deal with two codebases that interact with each other, e.g., testing or meta-programming. However, those branches have their own specificities and their own challenges when it comes to debugging. For example, specific debugging features would be needed for code generation in meta-programming. It would be interesting to study the extent to which the features we identified in our survey can be applied to other fields of software engineering.

3.6 Summary

With a survey of analysis developers, we have determined the different types of analyses currently developed by experts in the domain, and the reasons why they start debugging sessions. We saw that analysis correctness is their main priority, and that the root causes of analysis errors are due to the involvement of two codebases and the lack of *visibility* of the results that are generated when one interprets the other.

Our survey collects extensive data that we have not used to its full extent in this chapter, not only about debugging features for static analysis, but also about debugging features for general application code, motivations for writing static analysis, types of analysis written by participants, detailed analysis examples, reasons why participants debug static analysis, and why participants use particular debugging environments. Our data is made available online for others to use [86].

We now conclude by summarizing seven design requirements **RE-A1**–**RE-A9** (REquirements for Analysis developers), which we derived from our survey to build a coding and debugging environment for static analysis code.

- **RE-A1**: Analysis developers are particularly interested in the *correctness* of their analyses (i.e., precision and soundness). As a result, debugging tools should contain functionalities that allow them to track down code locations that cause loss in precision and soundness, e.g., points where a data-flow fact is created while it should not be, and vice versa.

- **RE-A2**: The root causes of errors in analysis code are due to algorithmic errors, semantics mismatches, and unhandled cases in the underlying analysis infrastructure. To help analysis developers debug such errors—especially the latter two, coding environments should provide more support and visibility in how the analysis code interprets the analyzed code, e.g., insights in the intermediate representation. Particular features that provide such support are described in **RE-A3**–**RE-A9**.

- **RE-A3**: *visualizations* of the analysis' constructs. *Graph visualizations* are particularly requested by both IDE users and text editor users.

- **RE-A4**: Access to the *intermediate results* generated by the analysis for all points of the analyzed code is asked by both groups, through *omniscient debugging*, where the intermediate results can be tracked back-in-time.

- **RE-A5**: Access to the *intermediate representation* of the analyzed code.

- **RE-A6**: *Breakpoints* and *stepping* controlled from both codebases are also deemed very important by IDE users, so that they can stop the program execution at any point of the analysis code for any point of the analyzed code.

- **RE-A7**: *Quick updates* are requested to recompute the analysis results on-the-fly when a modification in the analysis code or the analyzed code occurs. They prevent users from having to rerun the analysis and losing track of the state that they are monitoring.

- **RE-A8**: In addition, IDE users also ask for better *hot code replacement.*

- **RE-A9**: Users of text editors request *automated test generation* functionalities to verify the results of a change in the analysis.

Through a user-centered approach, we have determined the design requirements to build coding environments for static analysis development. In the next chapter, we use those requirements to build VisuFlow, a coding environment for Soot-based data-flow analysis in the Eclipse IDE [92].

4
Debugging Data-Flow Analysis

In Chapter 3, we motivated the need for debugging environments to support analysis developers write and debug static analysis code. Such features are not provided in traditional coding and debugging environments and have not been researched yet. In this chapter, we address the two main problems that current tools face when supporting static analysis: abstracting from both the analysis code and the code that it analyses at the same time, and tracking the analysis' internal state throughout both codebases. Based on **RE-A1**–**RE-A9**, we have designed and implemented VISUFLOW, a debugging environment for static data-flow analysis [91, 92].

In Section 3.3, we saw that the most analyzed programming language is Java, and that the most used analysis framework is Soot [140], which supports analysis of Java programs. As a result, we implemented VISUFLOW on top of Soot, and focus on the use case of Java programs, and analyses written in Java. To support advanced visualizations, we integrated VISUFLOW into the most popular Integrated Development Environment (IDE): Eclipse [32]. We also observed that the most developed types of analyses target security vulnerabilities, and are data-flow and program-level analyses. As a result, we apply VISUFLOW to the problem of data leaks, with a whole-program taint analysis.

In this chapter, we present VISUFLOW's Graphical User Interface (GUI), detail its implementation, and explain how it can be reused for other analysis frameworks. In our evaluation of VISUFLOW's usability, we have verified the validity of the requirements from Section 3.6, by conducting a comparative user study with the Eclipse debugging environment. We show that VISUFLOW's features, in particular its graph visualizations and custom breakpoints help analysis developers to identify 25% and fix 50% more errors in the analysis code.

VISUFLOW's design has been published in the IEEE Transactions on Software Engineering journal (TSE) [91]. Its implementation was presented at the tool track of the International Conference on Software Engineering (ICSE) [92].

4.1 VisuFlow, a Debugging Environment for Data-Flow Analysis

In this section, we present VISUFLOW, a debugging environment for Soot-based data-flow analysis implemented as an Eclipse plugin. We first introduce the GUI of VISUFLOW, and detail how its features match the requirements that we identified in Chapter 3. We then explore the implementation of the tool and how to adapt it to analysis frameworks other than Soot. The source code of VISUFLOW is available online, along with a video demonstration of its GUI [86].

Figure 4.1: The GUI of VISUFLOW. Features (A)–(F) are described in Section 4.1.1.

4.1.1 User Interface

We detail VISUFLOW's GUI (Figure 4.1) and how it addresses **RE-A1**–**RE-A9**.

Analysis Code and Analyzed Code

VISUFLOW shows the analysis code in a *Java Editor* (A) and the analyzed code in the *Jimple View* (B) side by side, to help work with both codebases at the same time. The analyzed program is displayed in both Java and the Jimple *intermediate representation* manipulated by the analysis (**RE-A5**). Echoing the survey findings from Chapter 3, this layout adds *visibility* in the debug process (**RE-A3**), and helps *avoid errors* due to Jimple-related semantic mismatches (**RE-A2**).

Graph Visualizations

To help the user better visualize the structure of the analyzed code, VISUFLOW presents a *Graph View* (C) that displays the call graph and the CFGs of the different methods of the analyzed code. By showing the intermediate results on the edges of the CFGs, it provides a more visual approach to debugging, allowing the user to determine with a quick glance where a particular piece of data-flow information is generated, killed, or transferred, instead of manually inspecting the intermediate results statement by statement.

After our first study, we have chosen to lay out the graphs using a Sugiyama-style graph-layering algorithm [130], to add zooming and panning features, tooltip information, and with a search bar that allows the user to locate a particular statement. The graphs can be customized through node drag-and-drop and coloring. To draw the graphs, we use the GraphStream framework [43], because it scales up to a large number of nodes and edges.

The *graph visualizations* display information about the structure of the application code (**RE-A3**), and the edge labels provide *access to the intermediate results* (**RE-A4**). This information helps analysis developers ensure the *correctness* of the analysis (**RE-A1**).

Breakpoints and Stepping

Traditional debugging environments allow analysis developers to set breakpoints only in the analysis code. When a user needs to debug an analysis at a specific statement of the analyzed
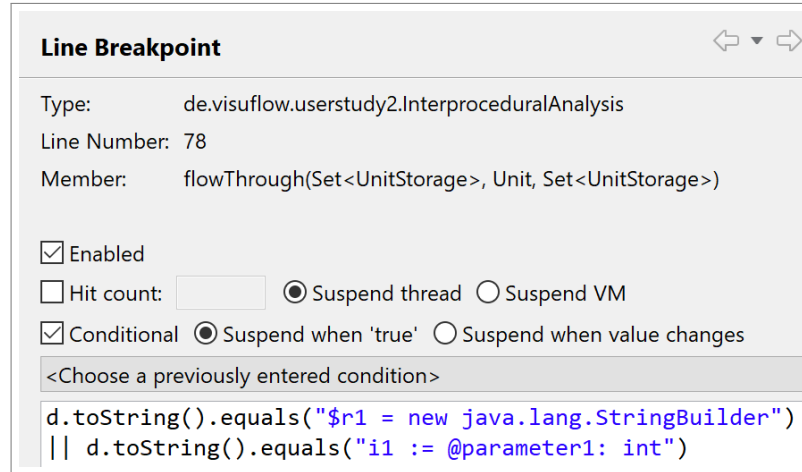
Figure 4.2: ECLIPSE conditional breakpoint dialog.
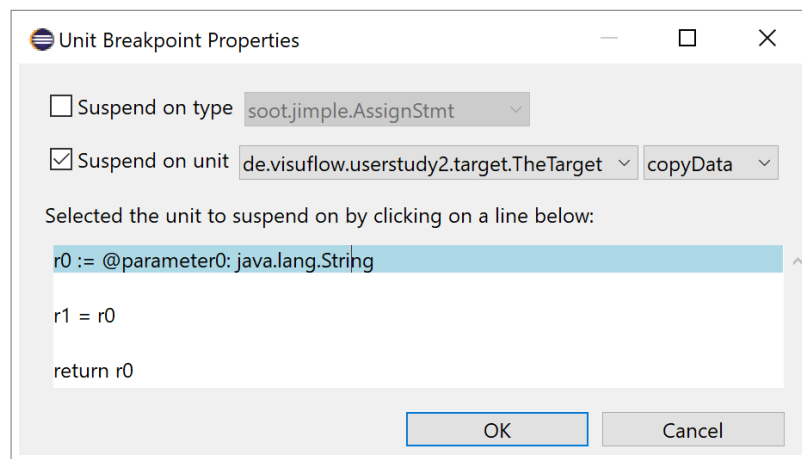
Figure 4.3: VISUFLOW breakpoint dialog.

code, they have to use conditional breakpoints to suspend the analysis for that specific statement, as illustrated in Figure 4.2. This workaround is quite limiting, since it requires the user to know in advance which statements need inspection, and to manually edit the breakpoint condition.

VISUFLOW allows analysis developers to set breakpoints in both the analysis code (in the *Java view*) and the analyzed code (in the *Jimple View* (F)). Analysis developers can thus stop the execution for specific statements of the analyzed code. VISUFLOW provides stepping commands for the analyzed code (i.e., step-over and resume), which are independent from the stepping commands of the analysis code. As a result, analysis developers can step through both codebases as they wish (**RE-A6**). A breakpoint dialog (Figure 4.3) allows developers to set multiple breakpoints at once, on specific statements, or on groups of statements.

Other Visualizations

A set of different views displays various information that can be used to help reason about the analysis code. The *Results View* (D) provides a compact summary of the *Graph View*, with search and filter options: it details the intermediate information for each statement of the application-code, and also allows users to mark specific statements with custom tags.

The *Unit Inspection View* Ⓔ shows a list of the statements of the analyzed program so that a user can inspect details of how a statement is constructed (i.e., type of the statement and types of its components). This feature is useful to novice Soot users who might have little to no knowledge of Jimple, but need to handle Jimple statements.

Both views have been enhanced with search and filtering options. As part of VISUFLOW's *visualizations* (**RE-A3**), they aim at providing a clearer picture of how the analysis works internally (**RE-A5**), to help developers guarantee analysis *correctness* in the case of the *Results View* (**RE-A1**), or avoid *semantic mismatches* for the *Unit Inspection View* (**RE-A2**).

Integration with Eclipse

As an Eclipse plugin, VISUFLOW aims at following Eclipse's usability paradigms, and provide a consistent user experience.

During the development of VISUFLOW, we have discovered that having many views reporting different information on the same statement could disrupt the user's understanding of the analysis. When the information shown on different views was not synchronized, the users had to manually scroll through all of the views to keep the focus on a given statement. VISUFLOW's views contain automated synchronization options, navigation menus, and highlighting features to allow users to switch between views more smoothly. Selecting a statement in one view also highlights it in the other views, and stepping through either of the codebases steps through all views of VISUFLOW to keep the focus on the current statement.

The VISUFLOW perspective introduces a new type of Eclipse project that links the analysis project and the analyzed project. This system provides VISUFLOW's builder and debugger components with the projects whose information should be displayed in the VISUFLOW views.

The top toolbar provides easy access to VISUFLOW's functionalities, such as rebuilding the project (which repopulates the data model if needed), running the analysis (which populates the in-sets and out-sets), and the breakpoint and stepping functionalities for both codebases.

To stay consistent with Eclipse's breakpoint system, VISUFLOW's double breakpoint and stepping systems extend Eclipse's (thus allowing synchronization with the variable inspection view and the stack frame view), the editors extend Eclipse's Java Editor, and the navigation functionalities are available on right-click as well as in the Eclipse native menu.

4.1.2 Implementation

Figure 4.4 illustrates the component diagram of our implementation. The three main components are the Soot framework, the Eclipse platform, and VISUFLOW. The VISUFLOW component is an Eclipse plugin that acts as a buffer between the IDE and the analysis framework to provide debugging support to the analysis developer when they debug an analysis for a particular piece of analyzed code.

Data Model

To provide visibility into the analysis, VISUFLOW maintains a data model of the analysis' representation of the analyzed code. In data-flow analysis, the analyzed code is abstracted into call graphs (over the methods of the analyzed code) and Control-Flow Graphs (CFG) (over the statements of the analyzed code). The data model in VISUFLOW models this information with $VFClass$, $VFMethod$, and $VFUnit$ objects (a Unit is a statement in the analyzed code). $VFEdge$ and $VFNode$ objects are used to represent the CFGs and call graphs. Each $VFUnit$ also contains its *in-set* and *out-set*. In data-flow analysis, those sets contain the data-flow information computed for each statement of the analyzed code (i.e., they contain the intermediate results of the analysis).
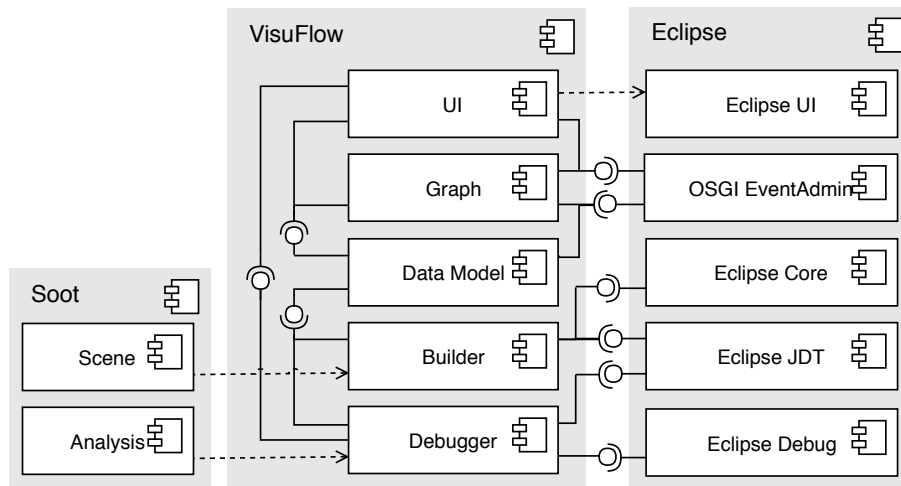
Figure 4.4: Component diagram of VisuFlow.

Every time the analyzed project is built in Eclipse, it updates the data model, except for the in-sets and out-sets: Eclipse Core's $IncrementalProjectBuilder$ calls VisuFlow's Builder, passing the analyzed project to Soot's pre-analysis phases. Soot creates different Units, Methods, Classes, CFGs and a call graph that are available from Soot's own data model: the Soot *Scene*. VisuFlow taps into the *Scene* and populates its data model by wrapping around the Soot constructs with its own.

The in-sets and out-sets are populated at runtime, when the analysis developer runs the analysis on the analyzed code. Eclipse's debug component is run, which calls VisuFlow's debug component through $ILaunchConfigurationDelegate2$. VisuFlow then runs the analysis using Soot, and retrieves the in-sets and out-sets by instrumenting the flow function of the analysis. As the analysis runs, VisuFlow employs a Java agent to establish inter-process communication between the running Soot analysis and the VisuFlow plugin. The latter runs a TCP server that populates the data model. The corresponding client instruments the flow function of the analysis so that, at the beginning of the flow function, it sends the values present in the in-set, and at the end of the flow function, it sends the out-set that has been generated.

The data model contains the internal state of the analysis, which is typically hidden from the analysis developer. This state is used by the UI components of VisuFlow and by the Eclipse's OSGi *EventAdmin* interface as an answer to a user event (e.g., to show more information in a tooltip), to display information in the different views of the VisuFlow plugin. The following debugging features can be obtained from the data model: *graph visualizations* (CFGs and call graphs), *access to intermediate results* (in-sets and out-sets), and *access to the intermediate representation* ($VFUnit$ in Jimple format).

Breakpoints and Stepping

On top of the Eclipse breakpoint system in the analysis code (in the *Java View*), VisuFlow provides an independent set of breakpoints for the analyzed code (in the *Jimple View*). The second set of breakpoints extends Eclipse's breakpoints and is attached to the analyzed code. VisuFlow transforms breakpoints in the analyzed code into conditional breakpoints in the flow function of the analysis code: if the user had previously set breakpoints in the flow function, VisuFlow adds a condition to stop the execution only for the specific statements of the analyzed code marked with the breakpoints in the analyzed code. If no breakpoints exist in the analysis code, VisuFlow creates one at the start of the flow function. This system is transparent to

the user: it creates the abstraction of a second set of breakpoints by translating them into conditional breakpoints. Likewise, VISUFLOW also simulates separate stepping functionalities to help users step through both codebases by adding breakpoints at the successor statements of the currently examined statement, and removing them later.

We use the Eclipse JDT to set *breakpoint* and *stepping* events. When triggered, those events transmit an $IJavaModel$ of the project under execution to the VISUFLOW debugger and builder. When the analysis generates new in-sets and out-sets at execution time, VISUFLOW populates the data model and—if breakpoints are set—updates its views to highlight the corresponding statements and update the in-sets and out-sets.

Unimplemented Features

VISUFLOW implements the debugging features that participants marked as most important in an IDE (Figure 3.4). Other features such as *omniscient debugging* (**RE-A4**), *automated test generation* (**RE-A7**), hot code replacement (**RE-A8**), and *quick updates* (**RE-A9**) constitute significant bodies of work which have been researched for general application code [64,113,116], and whose integration we keep for a future iteration of the user-centered design process.

Generalization of the Approach

The design of VISUFLOW's architecture separates VISUFLOW from the underlying analysis framework. It is thus possible to plug in another data-flow analysis framework such as WALA [135], the IFDS/IDE solver Heros [15], or IDEal [128], if it can expose its data model and flow functions to the VISUFLOW builder and debugger components.

Beyond reusing VISUFLOW, it is also possible to apply its tool features to non data-flow of analyses. We describe a high-level recipe for this application.

1. *Define a data model:* the data model represents the internal state of the analysis that is typically not exposed to the analysis developer. The data model must contain an intermediate representation of the analyzed code and the intermediate results of the analysis.

2. *Generate the data model:* the debugging tool must be able to query the analysis framework to populate the data model. Retrieving the intermediate results may need to be done at runtime. For VISUFLOW, this retrieval involves hooking into the Soot *Scene* and instrumenting the flow function.

3. *Define breakpoints in the analyzed code:* following the idea of constructing conditional breakpoints on the analysis code to simulate breakpoints in the analyzed code, the debugging tool must be able to stop the analysis at certain points of the analyzed code. Those points must be defined beforehand. In the case of VISUFLOW, they are the statements of the analyzed code.

4. *Design a usable GUI* to visualize the data model and to support the breakpoint system.

4.2 Evaluation

To evaluate how efficiently VISUFLOW supports analysis developers, we conducted a user study with 20 participants in the form of a comparative study between VISUFLOW and the default Eclipse debugging environment—which we refer to as ECLIPSE. The main goals of the study are to evaluate the usefulness of the different features of VISUFLOW, the environment's overall usability, and verify some of the outcomes of the survey presented in Chapter 3. To this end, we evaluate the following three research questions.

- **RQ6**: Which features of VisuFlow and Eclipse are most useful to the participants?

- **RQ7**: Can VisuFlow help developers identify and fix more analysis bugs than Eclipse?

- **RQ8**: Does VisuFlow's GUI help understand and debug analyses better than Eclipse's?

4.2.1 User Study: Usability of VisuFlow Compared to Eclipse

We describe the experimental setup of our comparative user study between Eclipse and Visu-Flow, the study participants, and the data extraction methodology.

Experimental Setup

To compare how analysis developers interact with VisuFlow compared to Eclipse, we performed a within-subjects study in which we asked each participant to perform two tasks: to debug a static analysis with VisuFlow and to debug another one with Eclipse. In the latter case, participants had access to the Eclipse debugging functionalities such as the breakpoints, the variable view, and the stack frame view. We also provided them with the Jimple intermediate representation of the analyzed code.

The two test analyses are hand-crafted taint analyses that contain three errors each. Running either analysis on hand-crafted analyzed programs does not compute the correct results. For each task, the participants had 20 minutes to identify and fix as many errors as possible in the analysis code. To counter the learning effects, we used a simple latin-square design in which half of the participants performed their first task with VisuFlow, and the other half with Eclipse. Both groups switched tools for the second task. Before each task, we primed participants through simple training tasks with a demonstration analysis to familiarize themselves with the tools. The full tasks were then performed with different analysis code and analyzed code.

During the two tasks, we counted the number of errors that participants identified and fixed. We consider an error as identified if a participant is able to locate it in the code, and to correctly explain why it yields the wrong data-flow facts. We consider an error as fixed if a participant manages to edit the analysis and rerun it, and if that new analysis generates the correct data-flow facts. We also logged how long the mouse focus was for each view of the coding environment to measure the time spent using each view.

After the tasks, participants filled a comparative questionnaire of the two debugging environments, followed by a short interview of their impressions of the tools.

We first ran a pilot study on six participants. Based on their feedback, we extended the time limit for the tasks from 10 to 20 minutes, and adjusted the difficulty of the tasks so that they were not trivial, but still doable in the time limit.

Questionnaire

The post-task questionnaire contains 41 questions, referred to as **Q1**–**Q41**, and is detailed in Appendix B.

1. **Participant information:** **Q1**–**Q5** gather information about the participants. **Q1** asks for how long they have been developing analysis code in a multiple-choice question, **Q2** asks for their main coding environment—in free text, and **Q3**–**Q5** ask participants to rate their familiarity with Eclipse, data-flow analysis, and Soot, on a Likert scale from 0 (novice) to 10 (expert).

2. **System Usability Scale: Q6**–**Q15** ask the ten questions of the System Usability Scale (SUS) [18] for the coding environment that the participants used for their first task

(Eclipse of VisuFlow). Each question is rated on a Likert scale from 1 to 5. **Q16**–**Q25** are the same SUS questions for the second coding environment.

3. **Perceived Performance: Q26**–**Q31** ask users with which coding environment they felt they did a better job at understanding errors (**Q26**), fixing errors (**Q27**), understanding the analysis code (**Q30**), understanding the analyzed code (**Q31**), with which environment they fixed more errors (**Q28**), and with which environment they fixed errors faster (**Q29**). Participants could choose either of the two coding environments, or a "Neutral" answer.

4. **Net Promoter Score:** Using the Net Promoter Score (NPS) [109] in **Q32**–**Q33**, participants rated on a Likert scale from 0 to 10 how much they would recommend one of the coding environments over the other. In **Q34**–**Q35**, they rated how much they would recommend VisuFlow or Eclipse over their own coding environment.

5. **Open questions: Q36**–**Q41** gather general comments about the coding environments, and capture any usability issues participants encountered. Those full-text questions serve as guidelines for the short interviews. In **Q36**, we ask participants to describe the features of their usual coding environment. **Q37**–**Q38** ask them about which features of Eclipse and VisuFlow they would like to use in their own coding environment. **Q39**–**Q40** ask which tasks the participants would use the two coding environments for, and **Q41** asks participants what they would change in VisuFlow or Eclipse.

The full-text answers were categorized by two researchers in an open card sort [45]. Because each answer could be categorized in multiple categories, we calculated a Cohen's Kappa for each category of each question. The average Kappa score over all questions and categories is $\kappa = 0.98$ (median = 1, min = 0.66, max = 1, standard deviation $\sigma = 0.07$), which is above the 0.81 threshold, indicating an almost perfect agreement [61]. The questionnaire, the anonymized answers, the test applications, and the results of the user study are available online [86].

Participants

We conducted our user study with 20 participants of diverse backgrounds: researchers in academia (65%), researchers in industry (5%), and students (30%). Eleven participants have less than a year's experience of developing static analysis, six have from one year to five years of experience in the domain, and three have more than five years of experience developing static analyses. In the following, we refer to the participants as **P1**–**P20**. Eighteen participants are IDE users, which was our target group for this tool. The other two participants mainly use vim to write analyses and were included in the study to evaluate how text editor users would adapt to VisuFlow.

On a scale from 0 (novice) to 10 (expert), the familiarity of participants with data-flow analysis is well distributed, with an average score 5.7 (min: 1, max: 9) (**Q4**). Eclipse has an average of 5.9 (min: 2, max: 8) (**Q3**), and Soot, an average of 3.3 (min: 0, max: 7) (**Q5**). We thus gathered a variety of both novice and expert users of data-flow analysis and Eclipse. Expert Soot users are rarer, as they are more difficult to find.

Eighteen of the 20 participants had not participated in the survey from which VisuFlow had been derived (Chapter 3). We have not observed any significant difference between their results and other participants' results. This gives us confidence in the impartial evaluation of those participants.

Table 4.1: Average time spent using features of VisuFlow and Eclipse.

	VisuFlow		Eclipse	
Feature	*#users*	*Time (s)*	*#users*	*Time (s)*
Java Editor	14	486	14	653
Graph View	14	201	n/a	n/a
Jimple View	11	58	12	60
Breakpoints / Stepping	11	174	11	313
Variable Inspection	3	78	8	67
Results View	8	50	n/a	n/a
Console	5	24	7	40
Drop Frame	5	12	3	5
Breakpoints View	3	13	2	110
Unit View	3	7	n/a	n/a

4.2.2 Study Results

In this section, we present the results of the user study. We discuss the most useful debugging features of Eclipse and VisuFlow, show how many analysis bugs the two debugging environments allow participants to identify and fix, and compare how they help users understand and debug analyses.

Most Useful Debugging Features (RQ6)

Table 4.1 shows the number of participants who used the features of VisuFlow and Eclipse, and the median focus time that they spent on each feature. Due to technical difficulties, we could only process the logs of 14 participants. As expected, the *Java Editor* is the most commonly used feature. The *Jimple View* is also often used, showing that access to the intermediate representation is helpful when debugging analysis code. Other frequently used features include *breakpoints*, *stepping*, and *variable inspection*. The VisuFlow-exclusive features that were used the most are the *Graph View* and the *Results View* (100% and 57.1% of participants, respectively), demonstrating the use of visualizations, and the need to expose the intermediate results of the analysis.

Using VisuFlow, participants spent 25.6% less time using the *Java Editor*, and 44.4% less time stepping through the code. Instead, they spent this time using the *Graph View*, the *Results View*, and the *Variable Inspection View*. This observation shows that graph visualizations and access to the intermediate results of the analysis are desirable features for debugging. Participants used the *Breakpoints View* 88.2% less often in VisuFlow compared to Eclipse, which we attribute to VisuFlow's dual breakpoints system. Since VisuFlow allows users to step through both codebases simultaneously, it spares them the effort of writing conditional breakpoints in the *Breakpoints View*.

The *Unit View* was only used by three participants, all of whom were unfamiliar with Jimple. We believe that the *Unit View* may be more popular for tasks requiring more knowledge about Jimple statements (e.g., writing an analysis rather than debugging it). However, we cannot verify this hypothesis with this study.

Using a $\mathcal{X}^2$ test of independence, we did not find a significant correlation ($p > 0.05$) between the participants' background and the tool features that they used most, suggesting that those results are generalizable to all user groups represented by our participants.

Table 4.2: Number of errors identified and fixed with ECLIPSE (E) and VISUFLOW (V) by each participant.

| | Task 1 (E) | | Task 2 (V) | | | Task 1 (V) | | Task 2 (E) | |
	Iden.	Fixed	Iden.	Fixed		Iden.	Fixed	Iden.	Fixed
P1	0	0	1	1	**P11**	2	2	1	1
P2	0	0	1	1	**P12**	1	0	2	1
P3	1	1	1	1	**P13**	2	2	1	1
P4	1	0	1	1	**P14**	2	1	0	0
P5	0	0	0	0	**P15**	1	1	0	0
P6	3	3	3	3	**P16**	1	1	2	1
P7	2	1	2	2	**P17**	2	1	1	1
P8	2	1	0	0	**P18**	2	1	1	1
P9	2	1	0	0	**P19**	3	2	2	1
P10	1	1	2	2	**P20**	1	0	0	0
Sum	12	8	11	11	**Sum**	17	11	10	7

Identifying and Fixing Analysis Errors (RQ7)

Table 4.2 reports the number of errors identified and fixed by each participant. For Task 1, participants identified and fixed 1.4× more errors with VISUFLOW than with ECLIPSE. In particular, they identified 17 errors and fixed 11 with VISUFLOW compared to 12 and 8 with ECLIPSE for that task. For Task 2, participants identified 1.1× and fixed 1.6× more errors when using VISUFLOW. Overall, 11 and 10 participants identified and fixed, respectively, more errors with VISUFLOW than with ECLIPSE. Using ECLIPSE, only 4 and 3 participants identified and fixed more errors, respectively. The remaining participants found and fixed the same number of errors with both tools. VISUFLOW thus allowed participants to identify 25% more errors, and fix 50% more errors than with ECLIPSE.

We do not compare the number of errors found by the same participant with different tools, because the two tasks were run on different, and thus incomparable, analyses.

Twelve of the 20 participants are Eclipse users, making the learning curve for VISUFLOW steeper than for ECLIPSE. Despite this factor, 7 of those 12 participants found and fixed more errors with VISUFLOW than with their original debugging environment.

We found no significant correlations between the number of errors identified and fixed, and participant background (coding environment, seniority in analysis development, and knowledge of Soot, Eclipse, and data-flow analysis).

Analysis Understanding and Debugging (RQ8)

Overall, the participants positively received VISUFLOW. In the NPS questions of the post-task questionnaire, the 20 participants rated their likelihood of recommending a debugging environment over another one to a friend for a task similar to the ones that they performed in the study. VISUFLOW has an NPS score of 63 compared to ECLIPSE (denoting an excellent score), and 21 (good) compared to the participants' own debugging environments. Participants gave ECLIPSE a mean score of -100 compared to VISUFLOW, and -75 compared to their own debugging environments.

When asked which debugging environment made it easier for them to find/fix the errors and understand the static-analysis code, all participants answered that identifying errors was easier with VISUFLOW ("It is pretty obvious that that's what static analysis needs."). Sixteen found it

easier to fix errors with VisuFlow; the other four participants answered that both debugging environments made it equally easy. Seventeen participants wrote that VisuFlow helped them understand the analysis code better ("What I was looking for in the first coding environment [Eclipse] was given to me by the second one [Visuflow]"), while one participant preferred Eclipse, and two remained neutral. To our surprise, the 12 participants who were already familiar with Eclipse still preferred VisuFlow, showing that VisuFlow is better suited than traditional debugging tools for debugging analysis code.

Participants were asked what they would use both debugging environments for. Sixteen wrote they would use VisuFlow to write and debug analysis code ("[I would use VisuFlow for] visualizing an analysis and finding unexpected values included or excluded from expected results"). Eleven participants found Eclipse more useful for "standard software development" or "general Java programming".

When asked which features of VisuFlow and Eclipse they would like to have in their own debugging environments, three participants requested Eclipse's *integrated debugger*, which echoes our survey findings (Table 3.4). We received more requests for features that VisuFlow provides. In particular, ten participants asked for the *Graph View* ("visualizing for provenance was useful"). Seven required visualizing intermediate results ("[Visuflow] is useful, because I get the abstract view of the situation, what's happening inside. Before [with the other coding environment], you have to [go through all] the variables."). Five participants mentioned dual breakpoints ("[Visuflow] is more comfortable; you can set Jimple breakpoints. It is clearly better."). Three asked for the synchronization between multiple views ("I think [Visuflow] is helpful because of the linkage between the Java code, the Jimple code and the graphic visualization: all that I had to keep in my mind [earlier]"). The *Jimple View* and the *Unit Inspection View* were only mentioned once. Two novice Soot users wrote that they wanted "All of them". The features that participants found most useful confirm the prioritization made by the survey participants (Figure 3.4), and match the most used features during the user study tasks.

Novice analysis developers noted a gentler learning curve when using VisuFlow: "I think this approach of debugging in the CFG is easier to learn for starting with taint analysis", "For someone who doesn't do this style of debugging analysis code at all, it kind of surprised me how quickly I was able to track down bugs for a bunch of code that I don't understand."

Using the System Usability Scale, VisuFlow obtains a mean score of 65.9, which is below the average SUS score of 68. This is explained by the fact that VisuFlow is obtained from the first iteration in the user-centered process, and still runs into usability issues such as improvements for breakpoints ("I would like to set breakpoints from the (uncompiled) source code under analysis rather than Jimple"), or quick updates "The refresh of the results is buggy"). We marked them for future improvements. In comparison, Eclipse scores a 43.4—which is below the 10th percentile, denoting its unsuitability for the task of debugging analysis code.

4.3 Limitations and Threats to Validity

We conducted the user study in a controlled environment (20 participants, 20 minutes per task, two tasks) rather than in a development setting. In practice, users would have more time to investigate more complex analyses. Given the time limits, we simplified the analyses while keeping them as realistic as possible: we based them on ~300 LOC-long taint analyses written by experienced students in our graduate course, and introduced typical errors made by those students. We verified with our pilot participants that the tasks could be achieved within the time limits. To avoid further external threats to validity, we recruited participants from different backgrounds: academia, industry, students, and professionals. VisuFlow is built on top of Eclipse and Soot, which are well-established both in industry and academia. It would, however, be interesting to conduct a future study in real-life conditions.

The user study was conducted as a within-subjects study, which created a learning effect between the prototypes, in which participants performed better on the second task, since they were used to the task and the tool. We countered this effect by applying a latin-square design, in which half of the participants used VisuFlow first, and the other half, Eclipse first, and reported on the aggregated results. In addition, we primed the participants before each task, to reduce the learning curve for both tools.

The times reported in Table 4.1 represent the use times of the different views of VisuFlow and Eclipse. We measured them based on the mouse focus events caught by the underlying Eclipse framework. The times might not be exact, because participant attention may be divided between multiple views while the mouse can focus on only one of them. We argue that in our user study, participants mainly used the mouse to navigate between views. In the absence of an eye-tracking device, our measurements approximate real user data sufficiently well. Averaged over all users, we can use the relative difference between the times spent in each view as a reliable metric to draw the same conclusions.

In its current implementation, VisuFlow does not gracefully scale up for two reasons. First, the size of the analyzed codebase can create large graphs that clutter the interface. GraphStream is able to render millions of nodes, however, as we discovered in our user study, the human eye cannot process more than a few dozens on the same screen. Solutions such as collapse-expand functionalities can be developed as part of future work.

Secondly, long update times are an issue when running complex analyses. Real-life analyses can take a long time to terminate: from minutes to hours, to days, depending on the complexity of the analysis and the size of the analyzed code. As a result, rerunning the analysis for each change in the analysis code or the application code would take a long time. Approaches such as incremental analysis [116] or Just-in-Time analysis [88] can be used to address this issue on the analyzed code. In external research, we have explored different ways to apply them to the analysis code. An early implementation achieves a 60% time improvement compared to a full recomputation of the analysis, on a dedicated micro-benchmark.

VisuFlow is not a complete tool. While we address the main features identified in Section 3.6, other features have been overlooked, such as *test generation* or *quick updates*, which can be explored and integrated in VisuFlow. *Omniscient debugging* has been integrated in VisuFlow as part of external research, allowing users to step forward and backward to any point of both programs, thus providing users with insight about the intermediate results of the analysis at any point of the execution without needing to rerun it.

4.4 Summary

In Chapter 3 and Chapter 4, we have illustrated the user-centered design methodology for a user group that has not been researched before: static analysis developers. Through a survey of analysis developers, we have derived tool requirements for a coding environment for static analysis. We have applied those requirements, keeping the using contexts and developer motivation in mind, and designed VisuFlow, a debugging environment for Soot-based data-flow analysis integrated in the Eclipse IDE. VisuFlow can be adapted for other analysis frameworks, and its general design methodology can be reused for other applications of static analysis.

In a comparative user study between VisuFlow and Eclipse, we empirically show that VisuFlow enables analysis writers to debug static analysis more efficiently. VisuFlow was well received by analysis writers, its analysis-specific features (e.g., graphs, breakpoints, and access to the intermediate results and to the intermediate representation) allowed them to identify 25% and fix 50% more analysis errors than with Eclipse. Participants find VisuFlow more useful than both Eclipse and their own coding environment to debug analysis code. In the

questionnaire and interviews, they confirm that the features identified as most important in our survey allow novice and expert analysis writers to more easily understand and fix bugs in analysis code. In particular, novice users find VisuFlow useful to learn about data-flow analysis. The study results show that VisuFlow succeeds in providing a good coding environment for developing analysis code while Eclipse is more suited to write general application code. The study thus confirms the outcomes of the survey in Chapter 3, and validates the usefulness of the debugging features implemented in VisuFlow.

We developed VisuFlow as part of the *design* and *evaluation* steps of the user-centered design process for analysis developers. While it successfully proves that this approach can yield good designs, we discovered user-experience issues during the user study, such as scalability, or long update times. Such issues, along with the debugging features that we did not integrate in VisuFlow should be addressed in the next iteration of the design process. VisuFlow is open-sourced [86], and we encourage contributions by other researchers and practitioners. This work can be used to design better tool support for debugging static analysis and help analysis writers secure application code.

With this work, we demonstrate the use of the user-centered approach for designing tools for static analysis, and advocate for involving end-users in the design process of tools for static analysis. In the past two chapters, we have focused on analysis developers. In the next chapters, we turn towards a more studied user group: software developers.

5

Identifying Tool Requirements for Software Developers

Past studies in the domain of usability of static analysis tools in practice focus on analysis correctness and usability issues reported by the tool users. In this thesis, we apply principles of user-centered design to offer a new perspective on the usage of static analysis tools in practice. In this chapter, we focus on the usage context of the tools, and the developer motivations and strategies when interacting with them. Some of our findings confirm past research (e.g., soundness issues in analysis tools), others contrast with it (e.g., developers are not as interested in style warnings as they are in performance warnings), and others highlight novel research areas for static analysis (e.g., tool support for collaborative interfaces).

To understand the usage context of analysis tools in practice, we conducted a survey in industry [93], in collaboration with Software AG [127]. Software AG is an international software vendor based in Germany and present in more than 70 countries. It is active in areas such as database management systems, big data analytics, business analytics, networking, software development, data transfer, and cloud solutions. As a large software vendor, Software AG has a strong interest in the functionality and security of their software products. Among other security measures, they use a large array of static analysis tools, which are the focus of our survey. Our survey focuses on the usage of 17 analysis tools at Software AG, with 87 of its software developers. We report on the developers' goals, motivations, and strategies when they use analysis tools, how those three aspects influence the way they interact with the tools, and which tool features could thus support them best. In addition, we confirm some of the findings of our survey through a study of the analysis results of Checkmarx [20], a major analysis tool used by Software AG, on two large projects of the company. We also use an independent smaller-scale cognitive walkthrough to further research the last section of our survey on desirable features for analysis tools [89]. Those studies allow us to derive recommendations for building and using analysis tools in industry.

The survey and the study of the Checkmarx analysis results are currently under submission [93]. The cognitive walkthrough was presented at the New Ideas and Emerging Results track of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering [89].

5.1 Related Work

We present related work on the use of static analysis tools in industry, and past studies reporting on their user-experience issues.

5.1.1 Usability of Static Analysis Tools in Practice

Static analysis has been used in industry since the 1970's. At first aimed at compiler optimization, its uses are now extended to bug and security vulnerability spotters, or coding automation such as code completion or refactoring [81]. The first large static analysis tools dedicated to bug and vulnerability detection appeared in the early 2000', such as Coverity [26] or Fortify [37]. Since then, the adoption of static analysis tools in industry has steadily increased, to detect bugs as early as possible in the software development process, thus cutting fixing costs by multiple factors [111]. In more recent years, applications of static analysis tools have expanded, supporting more languages and use cases, and detecting a growing set of bugs and vulnerabilities. From lightweight checkers run in the Integrated Development Environment (IDE) such as FindBugs [121] to more complex analyses such as Checkmarx [20] which are typically run during nightly builds, open-source and commercial tools are used and recommended by security authorities like OWASP [99] or CERN [19].

Despite its success, static analysis has been known for specific user-experience issues since its first applications in industry. Vorobyov et al. [142] compare model checking and static program analysis by focusing on the precision of both approaches, and discuss the causes of their limitations in the approach's functionalities (e.g., handling internal libraries). Bessey et al. [14] report on the experience of software developers with Coverity since its release in industry in 2002, showing that bad warning explainability, unclear tool configurations, and a high number of false positives have been issues with static analysis tools early on. Christakis et al. [22] survey developers and study live site incidents at Microsoft to extract pain points (e.g., false positives and bad warning messages) and potential improvements to the analyses (e.g., sources of unsoundness and reporting locations).

Those studies primarily focus on the tools and their usability issues, deriving features for better developer support with respect to the analysis (e.g., improving precision). In contrast, our focus is not the tools themselves but we approach the problem from the developers' perspective, and report on their motivation for using the tools, their strategies for using the tools, and, as a result, derive a different set of supporting features (e.g., collaboration features).

5.1.2 Developer Motivation and Behavior

The two studies closest to our own also report on developer motivations and how they influence the requirements for static analysis tools. Layman et al. [62] study the strategies of 18 student developers using the analysis tool AWARE. They extract requirements that differ from other studies such as the need for integrating the user's perception of severity in the severity rating of a warning. Johnson et al. [47] interview 20 developers on their experience of various analysis tools, focusing on the usability issues encountered in those tools and why they occur, from the point of view of the developer. This approach enabled the authors to present different requirements such as the need for better explanations in warning descriptions.

We build on both studies and conduct ours on a company-wide scale. In addition, instead of reporting on usability issues, we focus on the usage context, including company policies and developer schedules, and how that context affects the way that developers work with analysis tools, deriving requirements for building and using analysis tools in practice.

Ayewah et al. [11] report on developer usage of FindBugs, in particular the importance of warning severity in selecting which warnings to fix first, and how developers handle false positives. In a follow-up study, Ayewah et al. [10] observe student developers while using FindBugs to extract the factors that impact warning understandability. The findings of both studies are consistent with our survey, but we further investigate the developer motivations and the reasons for which they make those choices. Additionally, our study covers more aspects of how develop-

ers work with analysis tools such as how they decide which warnings are true / false positives, and what they do with warnings they do not understand.

Lewis et al. [65] compare different analyses and observe which one enables Google developers to be more efficient. Factors such as a bias towards new warnings or actionable messages were shown to be factors of interest to the developers. Similar to Ayewah et al. [10], the authors conduct their study in a controlled environment, thus not accounting for realistic usage contexts such as developer time constraints and motivations in real-life, which we highlight in our study.

5.2 Study: Developer Behavior and Motivation

To understand how developers interact with static analysis tools, we conduct a three-part study. First, we sent a survey across the main development teams at Software AG, asking developers about their experience with static analysis tools. In a second part, we were given access to the reports of analysis runs with Checkmarx [20], one of the major analysis tools used by Software AG. Checkmarx is a static analysis tool that supports 20 programming languages, and that can be used as a standalone tool with a web interface or in different Integrated Development Environments as a plugin. It is part of the continuous integration system at Software AG, and is used by a large variety of projects to detect software bugs and security vulnerabilities. Its interfaces provide management overviews of the projects' health, and detailed information about individual warnings, which developers use to communicate about the warnings and to fix them. The analysis reports we use in this thesis include information on how developers handled the warnings over several months, for two of Software AG's major projects, which we anonymize as Road Runner and Coyote at the company's request. We use this data to complement the survey answers. The third part of our study consists in a cognitive walkthrough, focusing on the user interface (UI) of an analysis tool, in which we evaluate the usefulness of tool features identified in the survey.

Our study targets the following research questions:

- **RQ9**: How are analysis tools integrated in Software AG's development environment?

- **RQ10**: In which usage contexts do developers use analysis tools, and with which goals?

- **RQ11**: Which strategies do developers apply when working with analysis tools?

- **RQ12**: Which features should be present in analysis tools?

In this section, we present the composition of the survey, the analysis reports, the cognitive walkthrough, and our methodologies for designing the entire study and extracting the data.

5.2.1 Survey of Industry Developers

To answer **RQ9**–**RQ12**, we designed a survey composed of 40 questions (referred to as **Q1**–**Q40**) grouped into the following six categories and detailed in Appendix C. Unless specified otherwise, all questions are multiple choice questions with an "Others" free-text field.

1. *Participant information:* We asked participants about their background: how long they have worked as a developer (**Q1**) and which programming languages they work with (**Q2**).

2. *General use of static analysis tools:* We asked developers general questions on how analysis tools are used at Software AG. This category includes questions on which analysis tools they use at the moment (**Q4**), when those tools are run in the project (**Q5**), who configures them (**Q6**), what kind of issues are detected (**Q7**), what kind of issues they
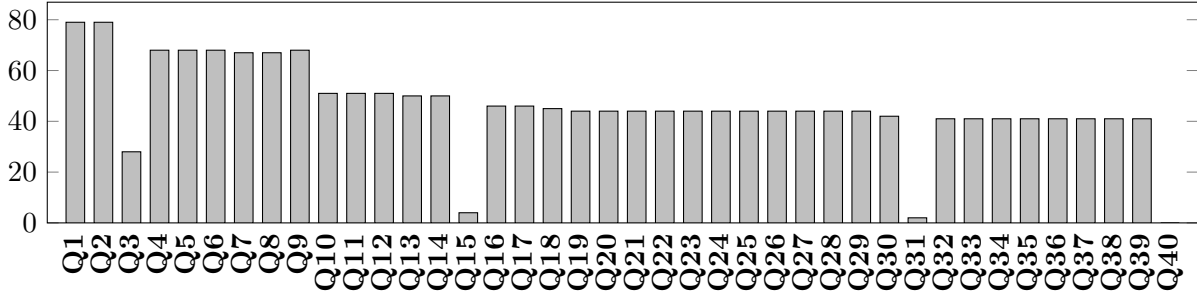
Figure 5.1: Number of responses per question. **Q40** received 0 responses.

would like the tools to report (**Q8**), if they fix the analysis warnings themselves (**Q9**), and who reviews the fixes (**Q29**).

3. *Reporting warnings:* This category contains questions about the format in which the tools report warnings (**Q10**), which format the developers would prefer (**Q11**–**Q12**), how long developers take to fix a warning (**Q13**), and how long they typically wait before their fix is verified (**Q14**). In a free-text question, we asked developers to comment on the reporting systems of analysis tools (**Q15**).

4. *Working context:* We asked developers which analysis tool they use the most (**Q16**), and focus on that tool for the rest of the category. We asked them how often (**Q18**), when (**Q19**), and where (**Q20**) they use it, how long they use it for (**Q21**–**Q22**), and why they use (and stop using) it (**Q23**–**Q24**). We also queried developers about which parts of the tool's interface they use the most, when they open (**Q26**), work with (**Q28**), and close (**Q27**) the tool, and if they are using the default layout of the interface (**Q25**).

5. *Features of analysis tools:* **Q30** and **Q31** (the latter, as a free-text question) asked developers to evaluate how important different tool features are to them.

6. *Fixing analysis warnings:* This category reports on the ratio of warnings developers investigate (**Q32**), understand (**Q35**), and for which they seek help from colleagues (**Q38**). It details the strategies used by developers to choose which warnings to investigate (**Q33**–**Q34**), the reasons why certain warnings are difficult to understand (**Q36**–**Q37**), and why developers ask for help (**Q39**). Finally, (**Q40**) asks about final comments on analysis tools as a free-text question.

We ran a pilot survey with five developers, after which we compacted the survey so that it could be completed in approximately 20 minutes. We namely removed questions similar to **Q30** and **Q31** that asked developers to which extent their current tools support the features.

We reached out to 120 developers at Software AG (two thirds of the development force) and received 87 responses, yielding an exceptionally high response rate of 72.5%. From those participants, 53 developers completed the survey in full, yielding a drop rate of 39.1%. Figure 5.1 details the response rates. As expected, the lowest response rates are found for the free-text questions (**Q15**, **Q31**, and **Q40**). Throughout this chapter, when we report on percentages of participants, we take the number of responses to the corresponding question as the baseline, instead of the overall number of 87 participants. The percentages may also add up to more than 100%, because some questions allow the selection of multiple responses. We only report on responses reported by more than one participant. We have made the complete list of questions and anonymized responses available online [86].

48

In the survey, 46.8% of the participants have 2–5 years of experience as software developers, 25.3% have 5–10 years, 13.9% have 1–2 years, 10.1% have more than 10 years, and 3.8% have less than a year of experience (**Q1**). While the large majority work with Java and Android (91.1%), Javascript (38%), C/C++ (10%), PHP (7.6%), Python (7.6%), and 12 other languages, each used by fewer than 2.5%, are also used (**Q2**). Due to Software AG's policies on the usage of static analysis tools, all participants have experience with them. Our survey thus gathers information from a diverse group of developers.

All survey questions but three are multiple-choice questions, for which we straightforwardly report the results. We either recategorized responses provided in the "Others" fields in existing categories when possible (e.g., "15" was recategorized in "> 10 years" for **Q1**), ignored them when they clearly did not answer the question (e.g., "Not applicable" for **Q24**), or kept in the "Others" category. The three free-text questions received the least answers (three for **Q15**, two for **Q31** and none for **Q40**). We detail and discuss the outcomes of the survey in the remainder of this chapter.

5.2.2 Analysis Reports

To complement the survey with respect to developer strategies for triaging and prioritizing analysis warnings (**RQ11**), we analyzed the reports of Checkmarx, one of the main static analysis tools used at Software AG. Checkmarx is deployed for a large number of projects at Software AG, as part of their global effort to use analysis tools for improving the quality of their code, and is used by 51.2% of the survey participants.

Checkmarx is a *dedicated tool*, meaning that it is independent of the tools used by developers (e.g., code editors or project management systems). Checkmarx has an elaborate web-based Graphical User Interface (GUI) that provides developers with detailed information such as warning categories (e.g., SQL injection), an estimate of their severity, general warning statistics, etc. Checkmarx also allows developers to comment on the warnings, for example, marking them as false positives or fixed.

Here, we study the analysis reports of two projects, which we name Coyote and Road Runner. Coyote contains 8 sub-projects, varying from 56,000 to 1,550,000 LOC. Road Runner has 4 sub-projects, ranging from 270,000 to 6,650,000 LOC. We report on analysis scans from spring 2017 to December 2018, except for 6 sub-projects of Coyote, three of which have been using Checkmarx since winter 2018, and three others, since winter 2017. We present the outcomes of our analysis of the reports in Section 5.4.

5.2.3 Cognitive Walkthrough

In an evaluation of how to present different tool features in a way that motivates and engages users (**RQ12**), we conducted a small-scale cognitive walkthrough with eight participants on a paper prototype. The prototype is built using a simple wireframe prototype with Balsamiq [13]. To raise user engagement, we gamified the prototype, as was done in previous studies in the more general field of software engineering [6, 39, 42, 134]. We gamified the different functionalities of the prototype to different degrees, ranging from ones completely dedicated to user engagement (e.g., badges) to functional ones with no gamification (e.g., quick fixes). We used a paper version of the prototype to conduct the experiment, and determined which levels of engagement are most attractive to users [89].

We ran a 30-minutes cognitive walkthrough of the prototype with eight researchers who have knowledge of how static analysis tools function. Five of them had worked with static analysis tools as developers in the past. Participants performed 23 tasks grouped into five themes: navigate the selection screen, (un-)assign bugs, navigate the debug screen, fix a bug, mark a
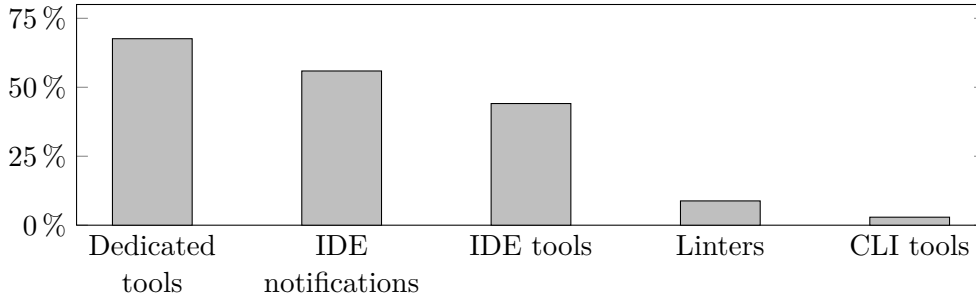
Figure 5.2: Types of analysis tools used at Software AG (**Q4**).

bug as a false positive. For each task, we noted the answers to the four cognitive walkthrough questions [144]:

- Will the user try to achieve the right action?

- Will the user notice that the correct action is available?

- Will the user associate the correct action with the effect they are trying to achieve?

- If the correct action is performed, will the user see that progress is being made toward the solution of the task?

The answers to those questions served two purposes. The first one was to ensure that the prototype was usable, and the second, that the participants gained a good understanding of the prototype before getting to the follow-up interview. All participants managed to perform the correct actions, with rare exceptions.

In a 15-minutes follow-up interview, we asked participants whether they found each of the tool functionalities (1) useful to complete their tasks and (2) engaging (i.e., they enjoyed using it). Finally, we asked them to list the top functionalities of the tool. The study protocol, the questionnaire, the prototype, and the results are made available online [86].

Two of the eight participants were meant to be pilot participants. Since they did not run into issues during the first phase of the cognitive walkthrough, we did not modify the prototype or protocol, and included their walkthroughs in our main results. We discuss the results of the cognitive walkthrough in Section 5.5.2.

5.3   Usage Context of Static Analysis Tools

In this section, we answer **RQ9** and **RQ10**, aiming to understand how analysis tools are used at Software AG, how developers interact with the tools, and their primary goals when using them.

5.3.1   Industrial Deployment of Analysis Tools

In the past few years, Software AG has strived to ensure the quality and security of its software through the use of analysis tools. Individual developers and projects can use their own analysis tools independently, but global efforts across the company have recently resulted in the deployment of common analysis tools and platforms over most major projects. We now discuss which tools are used at Software AG, how the company integrates them in its development process, and which types of issues they find.

Software AG developers report using a total of 17 different analysis tools that we group by interface types in Figure 5.2 (**Q4**). *IDE notifications* refer to analyses run by the developers'
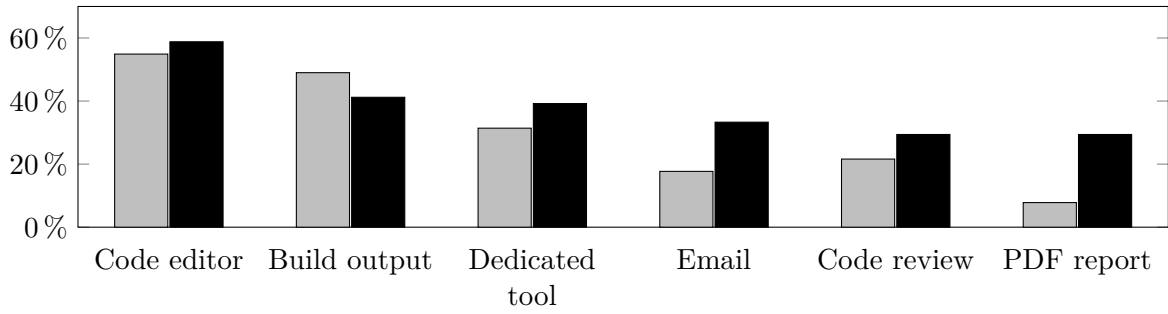
Figure 5.3: Reporting locations of current analysis tools (gray) and ideal reporting locations (black) (**Q10**–**Q11**).

Integrated Development Environment (IDE) (e.g., uninitialized variables). *IDE tools* designate analysis tools integrated in the IDE (e.g., FindBugs). *Dedicated tools* provide interfaces that are separate from the developer's coding environment (e.g., Fortify, Checkmarx, or CodeSonar for example). *CLI tools* provide a Command-Line Interface (CLI). We can see that Software AG uses a wide variety of analysis tools. Among the survey participants, 67.6% use dedicated tools, which conforms with Software AG's policy of using such tools in their projects. Participants also receive analysis information from IDE notifications (55.9%) and IDE tools (44.1%). Overall, 36.8% of the participants use only one analysis tool, and 48.5% of the participants use only one type of analysis tools.

With **Q5**, we observe that the use of analysis tools is spread over the software development lifecycle: 55.9% of the participants run their analysis tools at coding time, 52.9% during nightly builds, 29.4% at commit time, and 17.6% at major project milestones. We attribute this behavior to the use of different types of analysis tools—IDE notifications run at coding time, while longer running tools are not usually able to do so. We see that Software AG puts efforts in raising awareness about the use of analysis tools and exposes its developers to large system of tools comparable with other large companies studied in past research, thus making Software AG a good case study for evaluating developer behavior and motivation towards static analysis tools.

Once the analysis tools have run, they display warnings in various places, as shown in grey in Figure 5.3 (**Q10**). Reports in the code editor, build output, and dedicated tools are expected from the tool types that are most used at Software AG. When asked which reporting media they would prefer to use (**Q11**, black bars in Figure 5.3), participants confirmed wanting to use their current reporting platforms. However, alternative means were requested to a much higher extent: PDF reports were requested 3.75× more than currently used, email reports were requested 1.89× more, and the code review platform was requested 1.36× more. We attribute this higher demand to the ability of those media to aggregate results from multiple analysis tools in one place, which makes it easier to have an overview of the analysis results. Although we cannot confirm this claim with our current dataset, it is partially supported by the responses to **Q12** where 5.5× more developers indicated that they would prefer having the results of multiple analysis tools into one reporting place rather than in different ones (74.5% against 15.7%).

The survey responses show that 82.4% of the participants typically fix analysis warnings themselves (**Q9**). Once the warnings are fixed, they are reviewed by colleagues (79.5%), managers (15.9%), or dedicated teams (9.1%). Out of all fixes, 9.1% go unverified (**Q29**)

According to 47.1% of the participants, analysis tools used at Software AG are configured by a dedicated team. However, 36.8% wrote that they configured some of their analysis tools themselves, and 16.2% that some of their tools run on default settings (**Q6**). We see that a high number of developers set up their own tools themselves, which we attribute to the use of tools
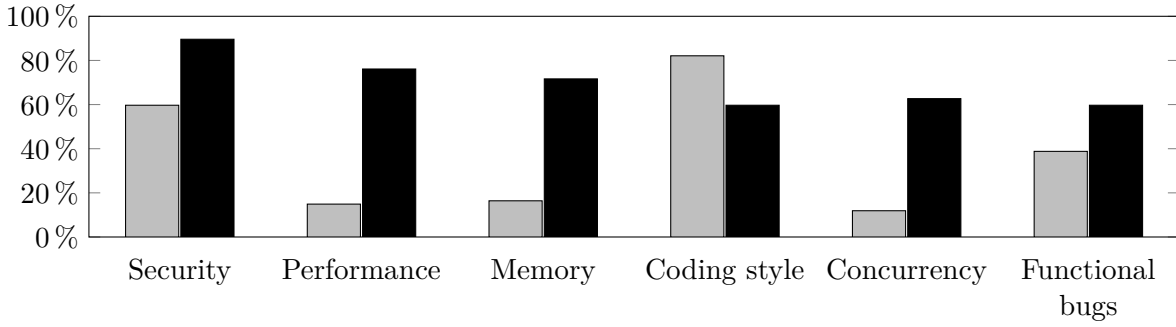
Figure 5.4: Warning types that are reported by the analysis tools (gray) and that developers want (black) (**Q7**–**Q8**).

Table 5.1: Conditional probabilities of a warning being of a certain type (**Q7**).

	Security	*Perf.*	*Memory*	*Coding style*	*Concurrency*	*Functional bugs*
Dedicated tools	0.31	0.06	0.08	0.34	0.04	0.16
IDE notifications	0.26	0.06	0.04	0.4	0.06	0.19
IDE tools	0.27	0.05	0.06	0.31	0.09	0.21
Linters	0.27	0.09	0	0.36	0	0.27
CLI tools	0.50	0	0	0.25	0	0.25

not proposed by the global company effort. This behavior, along with the responses to **Q17** where 78.3% of the developers said that they use analysis tools because it "helps me code better" against only 30.4% because of "company policy", suggests that Software AG generally encourages the use of analysis tools and spreads awareness among its developers about the importance of fixing bugs and security vulnerabilities.

In their responses to **Q7**, participants indicated the vulnerability types that are reported by analysis tools in their projects (grey bars in Figure 5.4). The warnings that are most reported are coding style-related (according to 82.1% of the participants), followed by security vulnerabilities (59.7%), and functional bugs (38.8%). Table 5.1 displays the conditional probabilities of warnings being of a certain type given the tool type. We see that for each tool type, the warning types most likely to be reported are security vulnerabilities, coding style issues, and functional bugs. This matches the distribution of warnings that code developers listed as most often reported. Dedicated tools and CLI tools are the most likely tools to report security vulnerabilities, and linters, IDE notifications and tools are more likely to report coding style issues.

When asking developers which types of warnings they would like analysis tools to report (**Q8**, black bars in Figure 5.4), we observe that the warning types are more distributed. While security vulnerabilities and functional bugs are still high (89.6% and 59.7%, respectively), the number of participants asking for other types of warnings (performance, memory, and concurrency) is higher than the number of participants getting access to such warnings by factors of 4.4× to 5.3×. On the other hand, participants wish to see less of the most frequently reported class of warnings: coding style.

Table 5.2: Conditional probabilities of the length of a working session given the tool type (**Q22**).

	< 10 min	10–30 min	> 30 min	hours
Dedicated tools	0.16	0.52	0.13	0.1
IDE notifications	0.31	0.35	0.12	0.12
IDE tools	0.40	0.44	0.04	0.04
Linters	0.75	0	0	0.25
CLI tools	0	1	0	0

Table 5.3: Conditional probabilities of fixing a warning in a certain time given the tool type (**Q13**).

	minutes	< 1 hour	< 1 day	< 1 week	< 1 month
Dedicated tools	0.15	0.15	0.38	0.21	0.06
IDE notifications	0.19	0.26	0.35	0.10	0.30
IDE tools	0.23	0.19	0.42	0.12	0
Linters	0.50	0.25	0	0	0
CLI tools	1	0	0	0	0

5.3.2 Analysis Tools in a Developer's Daily Work

Here, we detail when and how Software AG developers use analysis tools, and expand on the reasons that make them use those tools. We focus on the tool types that developers use the most: IDE tools and notifications, and dedicated tools.

Although the usage of analysis tools is distributed evenly during the day (morning 11.4%, afternoon 13.6%, and evening 11.4%), the largest group of developers (22.7%) use analysis tools in their spare time, i.e., when they have a few minutes between meetings, or spare hours during the work day (**Q19**). Analysis tools are used frequently in the work week: 75.6% of the participants say they use them multiple times in a week, 24.5% of whom use them more than once a day (**Q18**). This usage pattern indicates that working with analysis tools is not a large task that requires a developer to block a part of their schedule, but instead a set of short tasks that can be interrupted and resumed later. This observation is further supported by responses to **Q13** and **Q22** where we see that the median time for one working session with an analysis tool lasts for 10–30 minutes while the median time for fixing one warning is between an hour and a day. We can thus infer that in many cases, developers spread their treatment of an analysis warning over multiple working sessions.

The length of a working session with IDE notifications, IDE tools, and dedicated tools mainly vary between a few minutes to 30 minutes (Table 5.2). While the session length for IDE notifications and tools is evenly distributed between *< 10 min* and *10–30 min*, dedicated tools clearly lean towards the longer end of the spectrum. The typical fix times for one warning are shown in Table 5.3. We find the same trend over the time span of minutes to under a week: with IDE notifications and tools, a warning is fixed in around a shorter time (between an hour and a day) than with dedicated tools (around a day). This trend is explained by the fact that analyses running in the IDE must be able to yield results in a matter of seconds, which restricts them to fast and simple-to-compute checks. Their warnings are thus relatively easier to fix. We see that for the individual tool types, working sessions are typically shorter than the time to fix a warning, the only exception being CLI tools.

Table 5.4: Developer goals when opening analysis tools (**Q23**).

	Goal	*% of Devs*
O1	Fix all warnings	36.4%
O2	Fix warnings in a given time	31.8%
O3	Consult warning list	31.8%
O4	Fix a set number of warnings	9.1%
O5	Fix warnings up to a certain standard	4.6%

Table 5.5: Reasons for closing analysis tools (**Q24**).

	Reason	*% of Devs*
C1	Finished fixing everything	45.5%
C2	Professional obligation	25%
C3	Wait for the analysis tool to update	18.2%
C4	Office distraction	13.6%
C5	Never close the tool	13.6%
C6	Cannot fix an issue	9.1%

Having determined the usage context of analysis tools, we now explore the reasons why developers start and stop using analysis tools in their daily work.

When asked for the reasons why they use analysis tools in general (**Q17**), 30.43% of the participants reveal that it is because of company policies, and 21.74%, that it is because they help them code faster. In addition, 78.26% of the developers report that the tools help them code better, showing that developers value the use of analysis tools outside of company obligations.

In Table 5.4, we see that, independent from the tool type, most of the reasons for which developers open an analysis tool revolve around fixing warnings, with the variation of how many warnings they aim to fix (**Q23**). Conditional probabilities show that a relationship exists between tool types and fixing goals. When using dedicated tools, participants mostly aim at fixing as many warnings as possible in a given time ($Pr = 0.31$), which is a sensible strategy when dealing with complex warnings. With all other tools, participants mainly aim to fix all warnings when they open the tool. Consulting the list of warnings is a frequent reason why developers open the tools for all tool types ($0.24 \leq Pr \leq 0.28$). Table 5.5 details why developers close an analysis tool (**Q24**), independent from the tool type. The main reason is that they finished fixing all warnings, which we attribute to the use of lighter analysis tools that yield easier-to-fix warnings. The second cause is time limit. The third one is that they wait for a tool update (complex analyses can take minutes to hours to process an update). A minor reason for which developers close the tool is that they cannot fix a warning, an issue that is likely encountered when dealing with complex warnings that are not properly explained by the tool.

Table 5.6 shows that, regardless of the reason why the tool was opened, a popular reason for closing it is that all warnings were fixed (**C1**). However, when developers open the tool with a certain limit in mind (time or number of warnings), fixing all warnings is not the main closing reason. Developers are also likely to close the tool due to professional obligations (e.g., a meeting) or waiting for a tool update. We also see that when developers open the tool with the intention to fix all warnings, they only manage to reach their goal 45% of the time. Otherwise, they are either stuck on a warning or waiting for a tool update. This suggests that when developers do not have time constraints, they have a fair chance to eventually run into warnings that they cannot fix in one working session.

Table 5.6: Conditional probabilities of reasons for closing an analysis tool given the reason for opening it. The legends for Ox and Cx are found in Table 5.4 and Table 5.5 (**Q23**–**Q24**).

	C1	*C2*	*C3*	*C4*	*C6*
O1	0.45	0.05	0.15	0.05	0.15
O2	0.22	0.33	0.17	0.17	0
O3	0.35	0.25	0.15	0.10	0.05
O4	0.20	0.20	0.20	0.20	0

Discussion (RQ9–RQ10)

Software AG developers are involved with a variety of analysis tools at all stages of the software development process, the most frequently used being dedicated tools, IDE tools, and IDE notifications. Warnings are reported in various places across the working tools, which is consistent with the case study of Microsoft [22], but developers indicate that they would prefer a common interface for all analysis warnings. Centralizing warnings in the same user interface would reduce developer effort in switching between different interfaces.

Different types of tools are more likely to find different types of warnings. The most reported warnings are about security vulnerabilities, coding style, and functional bugs. In addition, developers would like to obtain information about performance, memory, and concurrency bugs in their code. A major difference from the Microsoft study is the requirements for coding style. While second most asked in their list, it is significantly less required by Software AG developers, which we attribute to two factors: their tools report too many such warnings, and with time, developers need more help with complex properties of the code than shallower ones.

Software AG developers use analysis tools at short points in time, mostly in their spare time. For them, fixing warnings is a continuous task spread over short working sessions, which length depends on the tool type. Warnings reported by IDE notifications and tools allow developers to spend less time fixing warnings, and to have shorter working sessions with the tools. More complex warnings produced by dedicated tools have longer fix times, and cause developers to work with the tools for a longer period of time.

In turn, this makes time limitations the main reason for stopping to use an analysis tool. Time limitations generate different working goals: while developers most often open a tool with the intention to fix warnings, they choose to fix different sets of warnings depending on their available time. This constraint introduces different interaction experiences with the tools, namely how to support developers in choosing which warnings to fix in the given time, which we explore in the following section. As a result, when designing an analysis tool—and modeling the workflow of a user within the tool, it is recommended to take into account how long the user intends to use the tool for in a single session.

More minor causes for developers ending a working session are explainability issues for complex warnings, and the long time taken to update analysis results, which have been reported in past studies [14, 22, 47].

5.4   Developer Motivations and Strategies

We now answer **RQ11** by exploring developer behavior when fixing warnings, in particular how they prioritize which ones to investigate first, what they do with warnings that they do not understand, and how they collaborate with their colleagues to fix them.

Table 5.7: Developer strategies to prioritize which warnings to address first (**Q34**).

Strategy	% of Devs
Prioritize warnings affecting the developer's code	46.3%
Prioritize warnings with the most impact	43.9%
Prioritize warnings the developer can fix	31.7%
Follow the order of the warning list	31.7%

5.4.1 Prioritizing Warnings

Before they start to fix warnings, developers must first select which warnings to fix. To help them choose, analysis tools typically provide them with additional information such as warning type (e.g., SQL injection), code location, or severity. Table 5.7 shows the four main strategies adopted by the survey participants when choosing which warnings to investigate (**Q34**). One of the most popular is to prioritize the warnings by impact, which aligns with Software AG's policy of addressing all of the most severe warnings before a major release. A developer will also preferentially work on warnings that impact their own code or that they know how to fix, because they have the necessary knowledge to do so. The last strategy is to go from the top down in the warning list, which is a sensible methodology for simple lists of warnings that are all fixable within one working session. Such strategy is also useful for longer, more complex lists that the tool already sorts by importance, which is often the case in dedicated tools.

To gain a deeper understanding of which warnings are fixed first, we studied the analysis reports of Checkmarx on two projects: Coyote and Road Runner[1]. Figure 5.6 shows the number of warnings found for nine of their sub-projects, grouped by confidence as labeled by the developers when they handled the warnings, over the time span of a few months for Coyote to nearly two years for Road Runner. Figure 5.5 shows the same data, grouped by severity, as provided by Checkmarx.

Except for Coyote G, only a fraction of the warnings are labeled by developers as true or false positives. We see that the variations of the number of labeled warnings follows the variations of the general number of warnings, suggesting that developers actively handle new warnings but usually only look at a fraction of the total number of warnings, or do not often label warnings. We also observe that developers tend to keep the number of warnings with a high severity at a minimum: the plots consistently remain close to 0, confirming our survey results (**Q34**: developers tend to fix warnings with the most impact). This observation is supported by Table 5.8: the probability of a high severity warning to be in the *to verify* list is very low compared to other types of warnings, and the high severity warnings that remain are most often false positives. Confirmed true positives are handled similarly: they are kept to a minimum and eventually removed, (e.g., Road Runner C).

Most projects also have a small number of low severity warnings, which we attribute to the relative ease of fixing such warnings, matching the developer strategy of fixing what they know they can fix. For example, *unchecked return value*, and *null pointer dereference*, likely to be classified with a low severity with a probability of 1, are simple to fix.

In the longer-running Road Runner projects, we also observe that the number of warnings regularly increases and plummets, which (knowing Software AG's release schedule) we suppose with fair confidence corresponds to compliance tests before major product releases or milestones. Outside of those times, the number of warnings decreases slowly due to continuous work done by developers on their spare time, as we have discussed in a previous section.

---

[1]The project names are anonymized at Software AG's request.
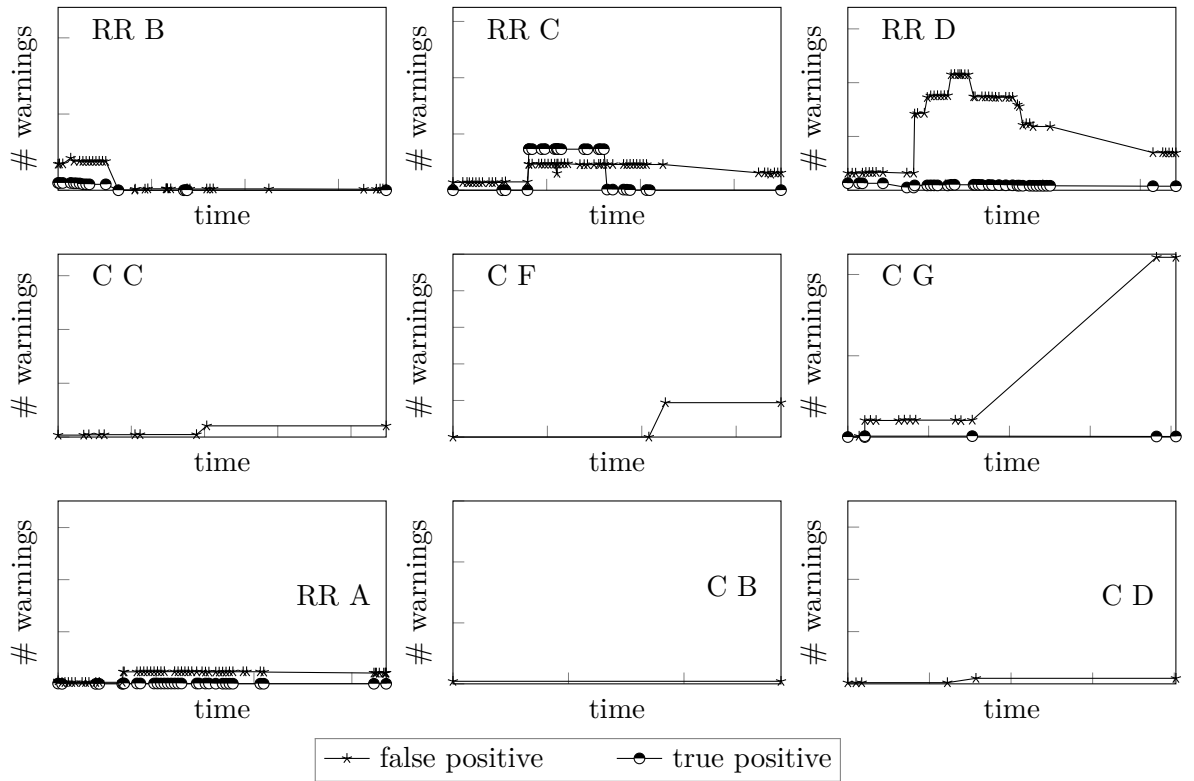
Figure 5.5: Number of warnings for four sub-projects of Road Runner (RR), and five sub-projects of Coyote (C) classified by *confidence* as labeled by Software AG developers. At the request of Software AG, we do not disclose the axis labels. All axes are in linear scale, the y-axes all start at 0. For the same sub-project, the maximum value on the y-axis is the same as for the corresponding sub-project in Figure 5.6.
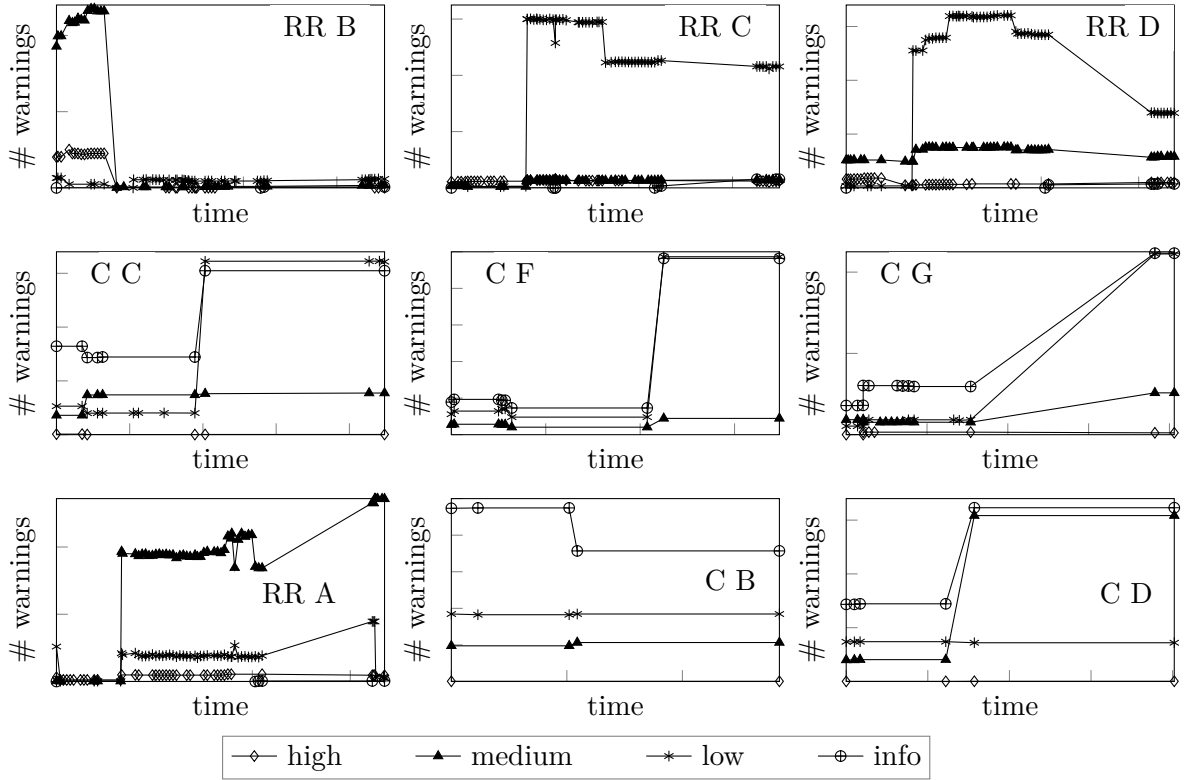
Figure 5.6: Number of warnings for four sub-projects of Road Runner (RR), and five sub-projects of Coyote (C) classified by *severity*. At the request of Software AG, we do not disclose the axis labels. All axes are in linear scale, the y-axes all start at 0. For the same sub-project, the maximum value on the y-axis is the same as for the corresponding sub-project in Figure 5.5.

Table 5.8: Conditional probabilities that a warning is marked with a certain confidence, given its severity.

	False Positive	*True Positive*	*To Verify*
High	0.92	0.03	0.04
Medium	0.15	0.03	0.83
Low	0.05	0.01	0.93
Info	0	0	1

Table 5.9: Developer strategies to detect false positives (**Q33**).

Strategy	*% of Devs*
Categories of issues are known false positives	43.9%
Code constructs are not handled by the analysis	39.0%
Warning witness is not executable	31.7%
Code locations are never executed	22.0%
Conditions along the warning are never true	12.2%

5.4.2 Detecting False Positives

To help developers decide whether a warning is a false positive or a real issue, analysis tools often provide them with additional metrics. For example, the main screen of Checkmarx allows developers to filter analysis warnings by warning type (e.g., cross-site scripting), code location, severity, and other information the developer can edit (e.g., confidence).

Table 5.9 shows five main strategies that participants use to evaluate if a warning is a false positive (**Q33**). The main strategy is looking at the warning type. This strategy can be a quick way of triaging through warnings, because certain warning types are more often labeled as false positives than others. For example, *memory leak* or *use of uninitialized variable* are very likely to be marked as false positives ($Pr = 0.8$ and $Pr = 0.71$ respectively). However, this strategy can be misleading: without investigating each warning in detail, true positives may be overlooked.

The second-most popular strategy is related to misinterpretations of some code constructs by the analysis (for example, the behavior of specific libraries or objects). Such constructs seem to be known by developers, and are used to differentiate true from false positives. Another 43.9% of the participants said that they recognize false positives because the warning witness is incorrect, meaning that the analysis' interpretation of the code's runtime behavior is faulty. This situation requires from the developer to investigate the warning in detail, which is a very accurate, but time-consuming strategy. On average, participants investigate 65.1% of the warnings in detail (min = 20%, max = 100%, $\sigma = 26.1$) (**Q32**). The last strategy is to mark as false positives warnings that go through areas of the code that are never executed. While this strategy helps remove false positives, it is insecure to keep vulnerable non-executed code in the codebase, as it could be exploited in the future.

5.4.3 Understanding Warnings

To understand if a warning is a true positive, if they should prioritize it, and how they could fix it, developers have to gain an understanding of the warning. We have previously discussed that developers often use heuristics over easily accessible data to make a decision, because they cannot spend time investigating all warnings. Therefore, the ability of the analysis tool to explain the warning and showcase relevant data is key to supporting its users.

Table 5.10: Reasons why warnings are difficult to understand (**Q36**).

Reason	% of Devs
Unfamiliar with the issue	48.8%
Explanation given by the analysis tool is unclear	48.8%
Span over too much of the codebase	31.7%
The codebase is unclear	4.9%

Table 5.11: Developer actions for warnings that they do not understand (**Q37**).

Action	% of Devs	Behavior type
Leave for later	56.1%	Neutral
Ask for help	51.2%	Good
Ignore	14.6%	Bad
Research and fix	7.3%	Good
Suppress	7.3%	Bad
Escalate	2.4%	Good

On average, participants understand 36.1% of the warnings they investigate (min = 0%, max = 80%, $\sigma = 32.6$) (**Q35**). To explain this low number, we asked participants for the reasons why warnings can be difficult to understand, which we detail in Table 5.10 (**Q36**). Three major reasons stand out. First on the list is that the warning is new to the developer, so they need to learn a lot: what the warning means, how it applies to their code, and how to safely fix it in the context of their code. Another reason is that the tool's explanation is unclear. While some tools simply give a generic description of the warning type, others, such as Checkmarx, provide more detailed information, yet it is still difficult to completely explain to developers how the analysis reasons about the warning, especially when the warning is complex and spans over a wide part of the codebase, which leads us to the third reason: the size of the warning. While some analysis warnings can affect small parts of the code (e.g., use of potentially dangerous function), other complex warnings can involve larger parts (e.g., an SQL injection going from an input form all the way to the database).

Table 5.11 details the treatment of warnings that developers do not understand (**Q37**). Overall, we see three main types of behavior appear: neutral, positive, and negative. A majority of the developers (56.1%) adopt the neutral behavior of leaving the warnings for later. More negative solutions are to ignore or suppress the problematic warning. Respectively 14.6% and 7.3% of the developers admit to using them, which should be discouraged. Other participants opt for positive actions and spend more time asking for help, escalating, or researching the warning. On average, participants ask for their colleagues' help for 27.8% of the warnings in an analysis report (min = 0%, max = 70%, $\sigma = 42.2$) (**Q38**).

When developers ask for help (**Q39**), they are interested in the three particular aspects that we discussed in **Q36**: the issue, the codebase, and the analysis tool (in particular, what the tool means when explaining the warning), as seen in Table 5.12. The first three aspects confirm our observations from **Q36**. In particular, with the second one, we see that of the warnings developers ask about, 46.3% are due to codebase issues, while only 4.9% of the participants find warnings confusing due to lack of understanding of the relevant codebase. We infer that developers rarely ask about confusing warnings, and that a large fraction of the ones they ask about are due to codebase clarity issues. This finding confirms the need for better warning explanations, especially with respect to information about the issue and the analysis tool, which

Table 5.12: Reasons why developers ask for help (**Q39**).

Reason	*% of Devs*
Others have experience with the codebase	46.3%
Others have experience with the type of issue	39%
Others have experience with the analysis tool	31.7%
The developer does not ask for help	14.6%

is less at hand to the developers than codebase information they can ask colleagues about. Lastly, 14.6% of the participants do not ask for help. We suppose that this behavior could be caused by time constraints, discouragement of working on a warning for too long, or the social consequences of admitting that they do not understand the warning.

Discussion (RQ11)

When choosing which warnings to fix first, developers aim for those they know they can fix, or for the ones with the most impact. The order suggested by the tools is secondary to that, which we attribute to the time constraints discussed in the previous section. As a result, developers base their judgement on their knowledge of the codebase, their experience of the warning types, and warning severity. While current recommender systems—which highlight to the developer warnings they should fix in priority—mainly center around severity, they should also take into account the length of a working session and developer experience.

To distinguish false positives from true positives, developers often use heuristics derived from their common experience, which can be sound or flawed. Allowing development teams to integrate sound heuristics in the analysis would help bridge the gap between how the analysis understands the codebase and how the developer does. Flawed heuristics stem from warnings that are ill-explained by the analysis tool, as confirmed by previous studies [14, 47, 65], which can result in negative warning treatments such as inappropriate silencings or dismissals.

Facilitating explainability is not only restricted to finding better explanations than the ones already provided by the tool. When asking for help from other colleagues, developers seek knowledge about the analysis tool, the warning type, or the codebase. Again, we see that taking developer experience into account could be a benefit to analysis tools, by suggesting to a stuck developer the name of a colleague who might have the knowledge they seek, or by building a knowledge database to look up warning into. Instead of using side-channels for asking for help or looking up warnings, analysis tools could take into account the usage context of a company with a community of developers to encourage collaborative positive behavior.

5.5   Desirable Features of Static Analysis Tools

In light of the developers' motivations identified in the previous sections, we discuss which features are of most interest in the user interface of an analysis tool, and how to present them to the developer, answering **RQ12**.

5.5.1   Tool Layout and Features

From CLI interfaces to standalone applications to IDE tools, the different static analysis tool types used at Software AG offer a wide choice in interfaces. To understand the developers' preferences with respect to the UIs of analysis tools, we asked them which kinds of layouts they use most often (**Q25**). Of all developers, 70.5% use the default layout of the tool, 18.2%, their

own custom layout, 4.6%, the company layout, another 4.6% change the tool layout according to their needs of the moment, and 2.3% do not use a particular Graphical User Interface (GUI) and stick to CLI tools only. We see that even though Software AG provides developers with a company-specific interface, they prefer the tools' default layouts.

In **Q26**–**Q28**, we asked participants which UI features they most often look at when opening, using, and closing an analysis tool. Developer attention is most attracted to the dashboard (a high-level summary of the project's health) when opening the tool (47.7%) and to the warning list when closing it (59.1%). The warning list is central at all times, in particular, when using the tool (68.2%), showing the importance of the information conveyed in this list: the issues, where they are located, and how much still needs to be achieved to meet the company's standards.

We identified a set of 19 UI features from commercial (e.g., Checkmarx [20], CodeSonar [41], etc.), academic (e.g., FlowDroid [9], Cheetah [88], etc.), and open-source (e.g., FindBugs [121], IntelliJ's Code Inspection [46], etc.) analysis tools, and from past work on their usability [10, 22, 47, 89], and asked participants to rank their importance between six categories: *should not exist*, *neutral*, *low importance*, *important*, *very important*, and *indispensable* (**Q30**). In the remainder of this thesis, we will refer to those features as **F1**–**F19**, all listed in Figure 5.7. **F1**–**F2** focus on the responsiveness of the tool, **F3**–**F6** address different aspects of explainability, **F7**–**F9** deal with fixing warnings, **F10**–**F12** aim at visualizing the project's health, **F13**–**F15** help keep track of individual warnings, **F16**–**F18** concern analysis configuration and feedback, and **F19** focuses on collaboration.

Figure 5.7 shows that the most popular features are **F3** (explain a bug), **F4** (bug severity), **F7** (explain how the bug can be fixed), and **F9** (quick fixes). The first three have a total of 28, 26, and 24 developers respectively marking them as *very important* or higher, with an overall average of approximately 19 developers. **F9** has 10 developers marking it as *indispensable*, against an average of approximately 6. The popularity of **F3** and **F4** echoes our findings from Section 5.4: since developers are more interested in severity and understanding the warning, those two features are most important to them. **F7**, which explains how the warning can be fixed on a high level, is highly appreciated. However, **F8**—which does the same but gives more specific recommendations with regards to the codebase—receives less support. Although we cannot be certain, it is possible that manually verifying a fix generated by the analysis would add to what developers currently have to understand, and the risk of introducing more potential bugs in the codebase is too high. Those reasons would also explain the low ratings of **F9**, which has the highest score for *should not exist* with 4 developers compared to an average of approximately 1. **F9** is thus among both the most popular features (supposedly for its gain of time) and the least popular ones.

Features such as collaboration (**F19**), customization of the analysis rules (**F18**), and visualization features (**F10**–**F11**)—which we have identified in Section 5.4 as ones that can potentially enhance the user-experience of the developers—have received the lowest ratings by a margin of four developers or fewer when compared to the average, despite **F10** and **F11** being often used by the developers (**Q26**–**Q28**). With our current data, it is difficult to say whether those lower ratings are caused by the developers disliking such features in their current analysis tools, them disliking the general idea behind those features, or, if having not experienced those features yet, they are wary of them. On the other hand, all features in the survey were deemed *important* or more by at least 75% of the participants, showing that when designing an analysis tool, even the least popular features would be worth including.

5.5.2 Motivation Through Gamification

To further explore how to introduce desirable features into a more concrete interface while keeping up developer engagement, we have run a cognitive walkthrough using the gamified paper

☐ Should not exist
◼ Low importance
☐ Very important

F1 Responsiveness of the analysis
F2 Responsiveness of the UI
F3 Explain how a bug works
F4 Explain how bug affects the code
F5 Explain how the bug is executed
F6 Explain how warnings are related
F7 Explain how to fix the bug on a high level
F8 Explain how to fix the bug in the code
F9 Quick fixes
F10 Visualizing warnings in the code

◼ Neutral
◻ Important
◼ Indispensable

F11 Dashboard of the warnings
F12 Search and sort through warnings
F13 Prioritization of bugs to fix
F14 Keep a list of the developers' bugs
F15 Tracking progress of what was achieved
F16 Configurability of the analysis
F17 Provide feedback to the tool
F18 Customization of the analysis rules
F19 Collaboration options (with colleagues)

Figure 5.7: Ranking the importance of tool features (**Q30**).
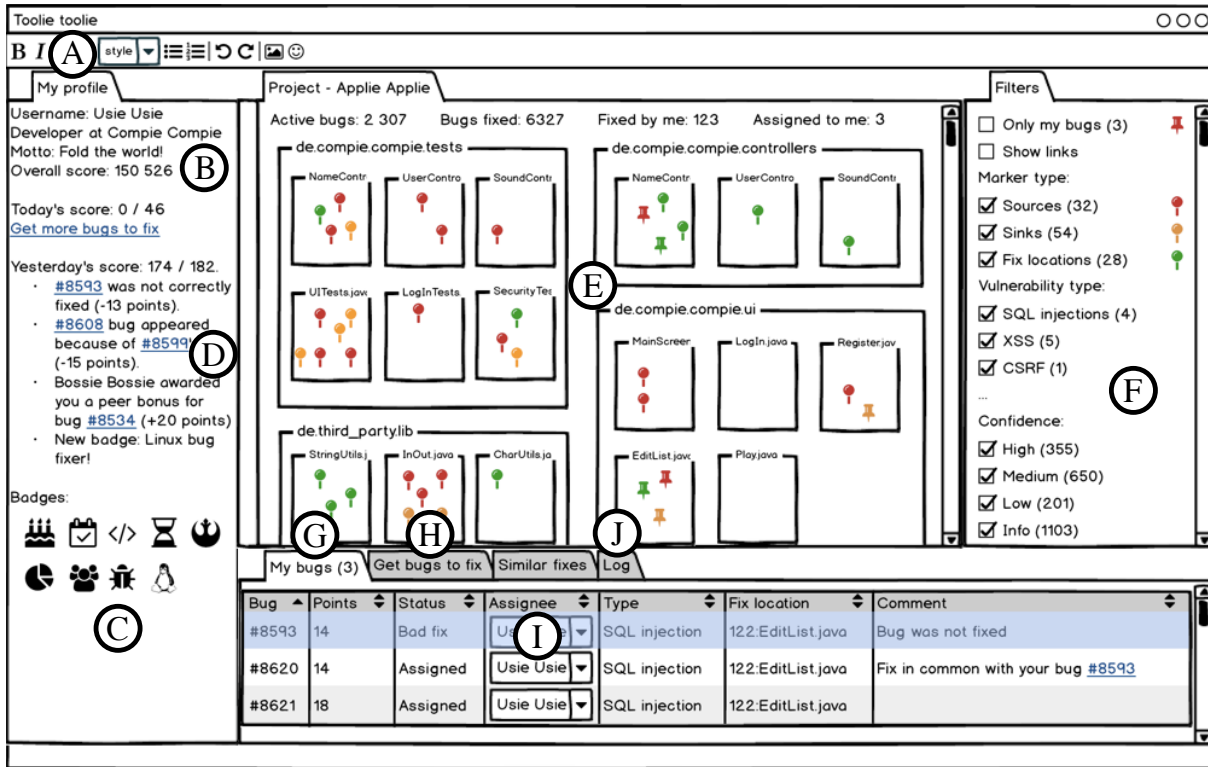
(a) Selection screen. (A)–(J) and (P) are detailed in Table 5.13.
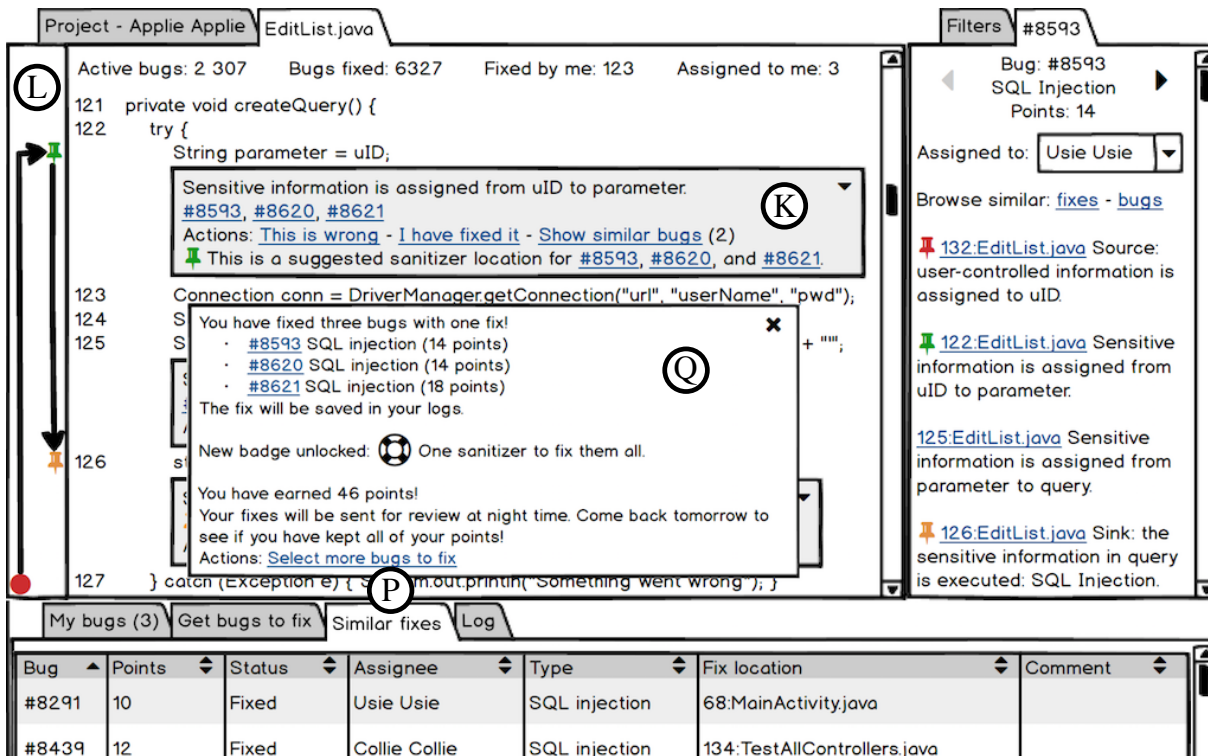
(b) Close-up of the debug screen. (K)–(Q) are detailed in Table 5.13. (M)–(O) are included in the grey box ((K)).

Figure 5.8: Paper prototype used for the cognitive walkthrough.

Table 5.13: Percentage of participants who found the functionalities in Figure 5.8 useful/engaging to achieve their tasks, and who have named them among their preferred ones.

	Functionality	*% Useful*	*% Engaging*	*% Preferred*
Ⓐ	Developer profile.	50	75	25
Ⓑ	Point system.	12.5	37.5	25
Ⓒ	Badges.	37.5	62.5	12.5
Ⓓ	Overview of yesterday's achievements.	62.5	75	37.5
Ⓔ	Map/overview of the warnings.	75	37.5	0
Ⓕ	Filtering functionalities for the map.	100	50	50
Ⓖ	Warnings assigned to the user.	100	33.3	25
Ⓗ	Suggestions of bugs to fix.	100	16.7	0
Ⓘ	Assignment system.	100	25	0
Ⓙ	Warning history.	62.5	37.5	0
Ⓚ	Warning information in the code.	100	62.5	87.5
Ⓛ	Gutter icons.	85.7	0	12.5
Ⓜ	Mark false positives ("This is wrong").	100	62.5	12.5
Ⓝ	Mark as fixed ("I fixed it").	100	80	12.5
Ⓞ	Cancel Ⓜ or Ⓝ with one click.	100	0	0
Ⓟ	Fix suggestions.	100	50	50
Ⓠ	Notification popup.	87.5	87.5	37.5

prototype shown in Figure 5.8 [89]. The figure shows the two main screens of the prototype, corresponding to the two main phases of using a static analysis tool: selecting which warnings to work on, and fixing those warnings. In the first phase, developers need an overview of all warnings, of the warnings they want to fix, and clear information that helps them select warnings to fix (e.g., incentives or impact of the fix). In the second phase, developers focus on one warning in particular, so they need detailed information about that one warning.

The gamified tool functionalities Ⓐ–Ⓠ are described in Table 5.13. While designing the functionalities, we have focused on two aspects: customizing the information based on the developer's current work (e.g., their currently assigned bugs) and experience (Ⓗ and Ⓟ), and providing them with customized feedback and actionable controls. For example, Ⓚ presents information embedded in the code where it is relevant: explanations on why the tool reports a bug, its relationship to other bugs, and access to possible actions: "This is wrong", "I fixed it", etc. Ⓠ appears when the developer fixes a bug, shows them their status (new points, achievements...), and gives them access to possible actions, e.g., "Get more bugs to fix". Functionalities Ⓐ–Ⓠ are designed to respect features **F1**–**F19**, with the exception of **F16** and **F18**, because they are part of different types of screens that are less used by software developers.

Table 5.13 presents the results of the post-cognitive walkthrough interviews, in which we asked the participants for their impressions on each of the tool functionalities. In particular, if they were (1) useful to complete their tasks, (2) engaging (i.e., they enjoyed using it), and (3) one of the top functionalities of the tool in their opinion. We see that functionalities Ⓓ–Ⓟ were perceived as useful by an overwhelming majority of the participants (between 62.5% and 100%). The information embedded in the code Ⓚ was mentioned by 87.5% of the participants as part of their top useful functionalities, confirming the need for **F7**–**F8**, and **F14**–**F15**. The

filtering functionalities (F) come next (50%, **F1**–**F2**, and **F11**–**F13**), followed by fix suggestions (P) (50%, **F7**–**F8**).

We see that functionalities (A)–(C)—which correspond to typical gaming functionalities (badges, profile, points)—have a lower usefulness score, and are generally perceived as more engaging than useful. Many participants overlooked them, as they "are unrelated to what I am doing". In contrast, functionality (Q) (notifications) is also a typical gaming functionality, but received a much higher usefulness and engagement score (87.5%), and was also chosen as one of the top functionalities by 37.2%. This functionality matches **F1**–**F2**, **F7**–**F8**, and **F11**–**F15**.

The cognitive walkthrough results suggest that for static analysis, gamified functionalities generate a higher engagement. However, developers favor functional usefulness before pure gamification. Functionalities that achieve both are best received (e.g., (Q)), which confirms observations made by past research in the more general field of gamifying software engineering [94].

Discussion (RQ12)

When first opening an analysis tool, Software AG developers often look at a dashboard to get a general overview of the warnings in the codebase. When closing the tool and while using it, they most often look at the warnings list, which seems to be their primary working tool. We have identified 19 concrete features for static analysis tools, all of which received positive ratings, and should be considered when designing an analysis tool. One should keep in mind that the integration of certain novel, or less popular features should be designed with care, finding the right balance between usefulness and motivation, to best assist developers.

5.6 Limitations and Threats to Validity

Our study is limited to Software AG and therefore does not necessarily generalize to every software company. However, the participants showed a large diversity in experience and programming languages, and have years of experience working with many static analysis tools at Software AG (**RQ1**–**RQ2**). Thus, the results of our survey reasonably generalize to similar companies who use analysis tools internally.

The formulation of the survey questions could also be subject to misunderstandings. To minimize such errors, we ran a pilot survey with five developers from Software AG. During the data extraction process, we did not find any responses that we could interpret as a response to a misunderstood question.

Another threat to validity is the subjective interpretation of the free-text (**Q15**, **Q31**, and **Q40**) and "Others" survey responses when we reclassified them in different categories. However, for this survey, we did not use the free-text questions, and we verified the classification of the "Others" responses with two raters, with 100% agreement. We have made the survey questions and anonymized responses available online [86].

While Road Runner and Coyote do not fully represent all of Software AG's projects, they are major projects of the company, and are contributed to regularly. We included all of their sub-projects in our study, which offers diversity in terms of project type, target platforms, and exposure time to Checkmarx.

The cognitive walkthrough has been run on a small set of eight participants. While they do not represent developers in industry, five of them have worked with static analysis tools as software developers in the past, and all of them are knowledgeable enough in the field of static analysis to know of its main issues, as would be expected from a software developer taking the walkthrough. Running a larger-scale study would yield more accurate results, which we leave for future work.

5.7 Summary

Through a developer survey and an analysis of two projects at Software AG, and a small-scale cognitive walkthrough, we have drawn a picture of how static analysis tools are used in industry. We conclude by summarizing a few tool requirements and recommendations for the design and usage of static analysis tools in practice, which we refer to as **RE-S** (REquirements for Software developers).

- **RE-S1**: *Time constraints* are the primary concern of developers when using an analysis tool. The length of a working session with the tool should be taken into account when designing the workflow of an analysis tool.

- **RE-S2**: Linked to time constraints, the lack of *responsiveness* of some analysis tools is a user-experience issue encountered by developers at Software AG. The analysis responsiveness and the tool interface should be crafted to minimize waiting times.

- **RE-S3**: As they work with analysis tools, developers build project-specific or company-specific heuristics to deal with warnings faster. Some are useful, and developers should be allowed to *contribute them to the analysis.*

- **RE-S4**: Other heuristics can be harmful, and can be avoided by improving the *explainability* of analysis warnings. Explaining a warning revolves around three knowledge bases for developers: past exposure to certain warning types, knowing the codebase, and knowing the analysis tool.

- **RE-S5**: The *developer knowledge* mentioned above should be integrated in *recommender systems* to provide users with personalized warning suggestions, given their abilities and their working time.

- **RE-S6**: Developer knowledge should also be made available to all users, through *collaboration options* in the analysis tool, to provide more official alternatives of communication than the currently used side-channels.

- **RE-S7**: More generally, analysis tools should be designed to *encourage good behavior* in situations when users are blocked. Tool and interface features such as the ones suggested in Section 5.5 should be designed with this in mind.

- **RE-S8**: Tool features can be ill-received if badly designed, especially if they are new or give risky advice. Therefore, they should be carefully designed to *prefer usefulness over motivation.*

- **RE-S9**: In practice, while it is generally recommended to use multiple static analysis tools in conjunction and recoup their warnings together, we also recommend adding *different types of tools* (e.g., dynamic analyzers, profilers) to cover aspects that are less reported on by static analysis tools (e.g., performance, memory).

- **RE-S10**: When using different tools, we recommend the use of *a single reporting platform* to handle all warnings. On top of a general overview of the project's health, such a platform can also provide a unified reporting format, which would reduce developer work in consulting multiple interfaces and learning how the different tools work.

- **RE-S11**: The success of analysis tools directly depends on the company's backing. *Policies* enforcing the use of analysis tools and spreading awareness, should be maintained and

developed, with for example, initiatives and trainings to sensitize developers towards good behavior when using analysis tools.

With this study, we advocate for a more user-centered approach of designing static analysis tools, in which usage context and user motivation can offer a different design perspective and yield different requirements. In the next chapters, we focus on a few of the tool requirements identified here.

6

Just-in-Time Analysis for Responsiveness

More companies integrate static analysis checks in their development process. However, most static analysis tools—such as Microsoft's PREfix/PREfast [72], Fortify [37], and Coverity [26]—are designed to be used in *batch mode* (i.e., offline, during nightly builds or at major software releases), because analyzing real-life projects can easily take hours. This use of analysis tools requires developers to pour over long lists of warnings (often in the order of thousands of warnings for real-life projects) and decide which messages correspond to real errors that require a fix, hours or days after they have committed their code [14, 65]. Running analysis tools in batch mode thus limits the potential utility of static analysis: the coding and fixing tasks are separated, creating a context switch for the software developer. Several human-subject studies have highlighted challenges related to workflow integration with static analysis tools, finding that developers dislike when static analysis disrupt their workflow [22, 47, 65], and that developers whose security tools help them do their work quickly are more likely to adopt those tools [146, 148].

We present the concept of Just-in-Time (JIT) static analysis, in which developers can encode priority rules to accelerate the analysis of particular paths. Warnings of interest are thus returned first, and other warnings are delayed while the developer handles the first ones (**RE-S2**). The ability to encode developer knowledge about the analyzed code addresses **RE-S3**. Our particular instantiation of a JIT analysis, CHEETAH, takes advantage of the responsiveness provided by the JIT system to incorporate a taint analysis into the active development environment, allowing developers to see the impact of their changes as they code, without blocking them from performing other coding tasks (**RE-S1**). In addition, CHEETAH makes use of the priority system to report more manageable, "digestible" sets of warnings first, instead of providing the user with a long list of warnings at once (**RE-S4**).

In this chapter, we describe general guidelines for designing JIT analyses. We also present a general recipe for transforming static data-flow analyses into JIT analyses through a concept of layered analysis execution. We illustrate this transformation with CHEETAH, a JIT taint analysis for Android applications implemented as an Eclipse plugin. Our empirical evaluation of CHEETAH on 14 real-world applications shows that our approach returns warnings in under a second—quickly enough to avoid disrupting the normal workflow of developers. This result is confirmed by our user study over 18 developers, in which they fixed data leaks twice as fast when using CHEETAH compared to an equivalent batch-style analysis.

The concept of Just-in-Time analysis has been presented at the ACM SIGSOFT International Symposium on Software Testing and Analysis conference (ISSTA) [88]. CHEETAH was presented at the tool track of the International Conference on Software Engineering (ICSE) [87].

6.1 Related Work

In this section, we present past work in improving the interactions between static analysis tools and developers, focusing on the responsiveness of analysis tools, the prioritization of analysis warnings, and the integration of developer-specific knowledge in the analysis rules.

6.1.1 Responsiveness of Static Analysis

Significant work has been done to make static analyses more responsive. For example, Solstice [83] runs an offline analysis on a replica of the developer's workspace, and reports results in a non-disruptive manner. In contrast, CHEETAH is an interactive analysis that operates on the original codebase, reporting its results in a timely fashion.

Incremental analyses such as Reviser [7], ECHO [151] or Lu et al.'s approach [69] reanalyze only the recent code changes, updating the analysis results quickly enough to be used in an IDE. The key component of incremental analyses is the computation of which data-flows need to be invalidated and recomputed [116]. In Reviser's case, this is done by traversing the ESG. ECHO uses static happens-before graphs to derive this information. Lu et al. [69] achieve the same result through Context Free Language (CFL) reachability. In contrast, CHEETAH relies on a prioritization system to compute a certain set of results quickly, reanalyzing the whole program at every run. In addition, CHEETAH gives developers control on the prioritization.

Parfait [23] runs different analyses in layers of increasing order of complexity and decreasing order of efficiency. Unlike Parfait, CHEETAH layers a single analysis, making it more responsive in general. Moreover, later analyses in Parfait may invalidate the results that the initial analyses have already reported. On the other hand, later layers in CHEETAH do not refute the results that have been reported by earlier analysis layers.

6.1.2 Warning Prioritization

Choosing which warnings to fix first is a major task for software developers. Analysis tools propose several ways to prioritize which warnings developers should address first. Most often, tools provide software developers with filtering and ordering options, for example by warning type or code location. They also rely on heuristics, like FindBugs [121] which classifies warnings as low, medium, or high priority.

Surveying the research, Muske and Serebrenik [82] organize prioritization approaches into three main categories: statistical, historical, and user-feedback. As an example of a user-feedback based approach, Heckman and Williams [44] use machine learning to prioritize actionable warnings over unactionable ones. Kim and Ernst [56] use code history to prioritize defects. Other approaches do not easily fit into these categories. For example, Shen et al. [122] deprioritize predicted false positives, then use developer feedback for future prioritization. As another example, Liang et al. [66] use resource leak defect patterns to prioritize potential resource leaks. In contrast, the JIT approach allows end-users to choose which warnings are returned first, allowing them to customize the prioritization order. CHEETAH in particular prioritizes using a developer's working context, and uses that context to guide the analysis itself. As this prioritization system is part of the analysis, it is not run as a post-processing module like other approaches, and can thus be used in combination with them.

6.1.3 Integration of Developer-Specific Knowledge in the Analysis Tool

Commercial tools sometimes allow developers to customize the analysis rules. For example, Fortify [37] and Checkmarx [20] provide their own languages and allow end-users to write their

own analyses. Other methods interface the user and analysis, for example with Microsoft SAL annotations [73] that denote the intent behind certain pieces of code. Those annotations are interpreted by the analysis, providing it with user-specific information. Another example is ASIDE [149], in which IDE annotations are used to interact with the user, to ask for missing knowledge. A similar questioning system is used by Dillig et al. [29], where the authors use abductive inference to determine the minimal set of questions to ask the user. Eugene, by Mangal et al. [70] proposes a user-guided approach in which developers can "like" or "dislike" warnings, allowing the analysis to learn about warnings of interest. The JIT analysis concept is closer to the first system, in which the developer can directly customize the analysis. It however simplifies their input from writing a full analysis to simpler prioritization rules, as presented in Section 6.2.2.

6.2 The Just-in-Time Analysis Concept

In this section, we introduce the concept of *Just-in-Time* (JIT) analysis that allows software developers to specify prioritization information into the analysis. We illustrate it through three analysis examples, and show how to apply it to existing data-flow analysis solvers.

6.2.1 Overview

Despite years of work on eliminating false positives in static analysis tools, end-user experience tends to be overwhelming, even for the unsound (or optimistic) commercial tools; this is sometimes called the "wall of bugs" effect [14, 22, 47, 65]. Observing how developers interact with static analysis tools, we highlight that: (1) warnings that are not relevant to the software developers are unlikely to be addressed, (2) the analysis does not necessarily have the knowledge of what "relevance" means to a particular developer, and (3) interrupting the development process with the fixing process is an undesirable trait for static analysis tools.

Building on those observations, we define the following requirements for a sensible analysis:

- **Prioritization:** The analysis must report the results most relevant to the user first. We expand on the definition of relevance later in this section.

- **Responsiveness:** To provide the users with immediate feedback on their changes, the analysis should report the earliest results quickly. In particular, analyses that run in the background of the IDE can also report the results *earlier* or *later* in time, allowing developers to focus on a subset of warnings while the analysis computes further results. This approach of *interleaving* analysis and developer activities reduces the perceived analysis latency, improving the overall usability of an integrated analysis tool.

- **Monotonicity:** To avoid confusing developers with disappearing warnings, a reported issue cannot be refuted until the developer has fixed it: the analysis only adds warnings over time. Therefore, refining an imprecise pre-analysis is not possible.

In Section 6.2.2, we present the concept of JIT analysis which addresses those requirements.

Relevance Examples

The relevance of a warning can only be defined by the developer since it depends on how the analysis is intended to be used. We now outline three concrete examples for expressing various relevance metrics using Listing 6.1, Listing 6.2, and Listing 6.3.

Listing 6.1 presents a Java snippet containing three data leaks. A taint analysis would report them from the source line 16 to the sinks line 21 ⓐ, line 25 ⓑ, and line 31 ⓖ.

```
13  public class A {
14
15    void main(B b)
16      String s = telephonyManager.getDeviceId();
17      String t = s;
18      String u = s;
19      sendDeviceId(s);
20      b.sendDeviceId(t);
21      smsManager.sendTextMessage("+49 1234", null, u, null, null); ⓐ
22    }
23
24    void sendDeviceId(String x) {
25      smsManager.sendTextMessage("+49 1234", null, x, null, null); Ⓑ
26    }
27  }
28
29  public class B {
30    void sendDeviceId(String y) {
31      smsManager.sendTextMessage("+49 1234", null, y, null, null); Ⓖ
32    }
33  }
```

Listing 6.1: Running example illustrating a JIT taint analysis.

To ensure correct API usage, analyses verify that programs follow a certain usage protocol [60]. API misuses are often detected with typestate analyses. In Listing 6.2, a typestate analysis would verify that a cryptographic cipher is always initialized with $init()$ before a call to $encrypt()$. Result $\textcircled{\delta}$ is harder to detect than $\textcircled{\epsilon}$, because the call to $init()$ on line 35 may resolve to either of the two implementations of the method: line 44 and line 52, with the latter not initializing the cipher.

A nullness analysis searches for null dereferences to avoid runtime errors. In Listing 6.3, a nullness analysis reports three warnings: $\textcircled{\zeta}$ because f points to $null$, $\textcircled{\eta}$ because f and g must-alias after the assignment statement on line 61, and $\textcircled{\theta}$ due to the may-alias on line 63. The latter alias is a may-alias because depending on the condition line 63, the alias may or may not happen, which the analysis cannot evaluate.

Encoding Relevance in the Prioritization System

Depending on the use case, developers may encode different factors into the priority system to obtain different warning orderings for the same analysis. For example, when writing code, developer attention is focused on the particular parts of the code that are being edited. Hence, it is sensible to prioritize warnings by *locality*, i.e., report in priority warnings that are closest to the developer's edit point. For example, if the developer is editing the $main()$ method in Listing 6.1, $\textcircled{\alpha}$ should be reported first, because it is located in the same method as the edit point. $\textcircled{\beta}$ should be reported later, because it is in the same class, but not in the same method, and $\textcircled{\gamma}$ should be reported last, because it is located in a different class.

Prioritizing by locality allows the analysis to provide quick updates for the warnings that are in the direct line of sight of the developer, leaving the more distant ones for later, while the developer fixes the first ones. This prioritization system also adds a secondary dimension to the warning ordering: as early warnings are contained in smaller parts of the codebase, they are more likely to be more understandable than warnings with larger traces and spanning different classes that the developer may not be familiar with, making the fixing process easier.

```
34  void encrypt(MyCipher myCipher, DESedeCipher desCipher, byte[] plainText) {
35    myCipher.init();
36    myCipher.encrypt(plainText); δ
37
38    desCipher.init();
39    desCipher.encrypt(plainText); ε
40  }
41
42  public class AESCipher extends MyCipher {
43    Cipher cipher;
44    void init() {
45      cipher = Cipher.getInstance("AES/GCM/PKCS5Padding");
46      cipher.init(Cipher.ENCRYPT-MODE, SecretKeyGen.getSecretKey());
47    }
48  }
49
50  public class DESedeCipher extends MyCipher {
51    Cipher cipher;
52    void init() {
53      cipher = Cipher.getInstance("DESede/CBC/PKCS5Padding");
54    }
55  }
```

Listing 6.2: Example illustrating JIT API misuse detection.

Another prioritization strategy based on *confidence* can prioritize monomorphic calls over polymorphic calls, because the latter are more likely to yield false positives. This way, ε is reported before δ, displaying first the warning that has the highest chance of being correct. Prioritizing warnings by confidence reduces the perceived number of false positives at the beginning, and gives the developer a better experience with the analysis. Another rule that can be encoded to order warnings by confidence is through "must" or "may" information: must information can be prioritized over may information, to ensure that warnings with the highest confidence are reported first. For example, ζ and η are reported before θ and the warnings derived from its data-flow facts.

A last example is prioritizing by *computational resources*. Depending on the available resources at a given time, an analysis could delay the computation of costly operations such as polymorphic calls, because they create more data flows than monomorphic calls, thus reporting ε before δ, for example. The analysis could also delay the computation of alias information: while ζ takes minimal computation to find, η and θ require additional alias information. In real-world programs, such flows may become exponentially more complex, and take minutes of computation to be reported, holding back the delivery of other simpler results that could be fixed in the meantime. With an ordering strategy by confidence, ζ can be reported quickly, while alias information is computed in the background to find η and then θ.

An advantage derived from warning prioritization is that fixing early warnings potentially also fixes others as well. For example, setting a sanitizer before line 19 would fix β and γ at the same time as α. Similarly, fixing ζ, may also fix η and θ at the same time, reducing the total number of warnings that are presented to the developer. Fixing ε does not fix δ, but the similarity of the two warnings provides the developer with the know-how of how to fix the more complex warning δ.

```
56  void main() {
57    F g = new F();
58    F h = new F();
59    F f = null;
60
61    g = f;
62
63    if(...) h = f;
64
65    x = f.a;  ⓏΞ ζ
66    y = g.a;  Ξ η
67    z = h.a;  Ξ θ
68  }
```

Listing 6.3: Example illustrating JIT nullness analysis.

6.2.2   JIT Analysis through Layering

We now present the concept of Just-in-Time analysis that addresses the requirements shown in Section 6.2.1. We use the example of Listing 6.1 with a taint analysis prioritized by locality to illustrate the JIT concept.

Locality-Based Layering System

Implementing a JIT locality-based system means dividing the program in locality-based *layers*. The goal is to immediately report the results closest to the user's working set. Lower analysis layers run first, yielding the first results in a few seconds. The following layers enrich the analysis by computing increasingly complex results.

We propose a layered analysis that computes warnings by gradually increasing the analysis scope, i.e., by incrementally taking more code into consideration, starting at the current edit point. Table 6.1 shows the set of layers for this strategy. The general idea is to propagate data-flow facts along the program, to stop at method calls, and to resolve them only at the corresponding layer. For example, at **L2**, the analysis would not propagate into a method that is out of the class of the current edit point, keeping it for when **L3** is reached.

The *prioritization* property comes by design, since the prioritization strategy is based on locality. *Responsiveness* is ensured as lower layers require minimal class loading and computational resources. For example, only one class is loaded to compute results up to **L3**. *Monotonicity* is guaranteed by the internal implementation of each layer: if a layer cannot confirm a result, it delegates its computation to later layers.

Table 6.2 shows the warnings that the analyses described in Section 6.2.1 would report using the layering system of Table 6.1 for the examples in Listing 6.1, Listing 6.2, and Listing 6.3. The JIT taint analysis reports the direct leak Ⓐ at **L1**, and Ⓑ at **L2**, after having resolved the call on line 19. Supposing that classes *A* and *B* are in the same file, Ⓖ is reported after the resolution of the call on line 20, at **L4**. The JIT API misuse detection reports Ⓓ and Ⓔ after the two calls to *init*() on line 35 and line 38, respectively. Assuming that the calls are not in the same package as the *encrypt* method, Ⓓ is reported in **L7** and Ⓔ in **L6**. Since the layering system does not include alias-specific information, the three null dereferences Ⓩ, Ⓗ, and Ⓣ are reported at **L1**.

To turn an existing analysis into a JIT analysis, a software developer must thus define the following three factors:

74

Table 6.1: Layers of a JIT analysis for Android applications. **L3** and **L8** model the lifecycle of the event-based Android framework.

Layer	*The layer propagates the dataflows...*
L1 Method	... in the same method as the edit point.
L2 Class	... along calls to methods in the same class as the edit point.
L3 Class Lifecycle	... along lifecycle methods in the same class as the edit point.
L4 File	... along calls to methods in the same file as the edit point.
L5 Package	... along calls to methods in the same package as the edit point.
L6 Project Monomorphic	... along the monomorphic calls in the project.
L7 Project Polymorphic	... along the polymorphic calls in the project.
L8 Android Lifecycle	... along lifecycle methods in the project, to handle interactions between the application components.

Table 6.2: JIT analysis results for Listing 6.1, Listing 6.2, and Listing 6.3, for the respective starting points: *main*, *encrypt*, and *main*.

	L1	**L2**	**L3**	**L4**	**L5**	**L6**	**L7**	**L8**
Taint	α	β		γ				
API						ϵ	δ	
Nullness	ζ η θ							

- *Triggers*: Statements at which the JIT analysis pauses the propagation of certain data-flow facts to prioritize others. In Listing 6.1, the triggers are the two calls to *sendDeviceId* on line 19 and line 20. At those triggers, the JIT analysis propagates u before propagating s and t to prioritize α, because it is in the same method as the starting point *main()*.

- *Priority layers*: At triggers, the choice of propagating certain data-flow facts depends on the *priority layers*. Data-flow facts created at a trigger create a *task* in the underlying analysis. In Listing 6.1, two tasks are created: one with the initial set $\{s\}$ with priority **L2**, because the call to *sendDeviceId* on line 19 resolves to a call in the same class, and one with the set $\{t\}$ with priority **L4**, because the call to *sendDeviceId* on line 20 resolves to a call in the same file. The analysis executes the first task because its layer has a lower priority, propagating s until the next trigger or the end of the program (reporting β on its way), and then executes the second task to report γ.

- *Initial task*: A first task from which all other tasks are created is used to initialize the analysis. In traditional analyses, this would correspond to the analysis' entry points such as the *main()* method.

Layering an Existing Analysis

We now present Algorithm 2 as a general recipe to transform a distributive data-flow analysis into a JIT analysis. While the algorithm requires a few changes to the analysis solver (highlighted in gray in the algorithm), the definition of the data-flow problem remains entirely unmodified.

The procedure *analyze()*, excluding line 14 to line 17, represents a standard fixed-point iteration for a traditional data-flow analysis that applies the flow function f_s to the statements of a program (line 21) until the *OUT* sets remain unchanged. The transformation to a JIT analysis divides this large fixed-point iteration into smaller ones (tasks). At trigger points, the

---

Algorithm 2 Formalization of a JIT analysis. The modifications made to the fixed-point algorithm are shown in gray.

1: **procedure** MAIN
2:     $PriorityQueue := \{initialTask()\}$ // initialize the priority queue
3:     $computedTasks = \emptyset$
4:     **while** $PriorityQueue \neq \emptyset$ **do**
5:         pop $task$ off priority queue $PriorityQueue$
6:         **if** $task \notin computedTasks$ **then**
7:             ANALYZE($task$)
8:             $computedTasks \cup = \{task\}$
9: **procedure** ANALYZE($\langle l, s_t, in \rangle$)
10:     $worklist := \{s_t\}$
11:     $IN[s_t] = in$
12:     **while** $worklist \neq \emptyset$ **do**
13:         pop $statement$ off $worklist$
14:         **if** $isTrigger(statement)$ and $s_t \neq statement$ **then**
15:             **for** $l' \in \{1..|layers|\}$ **do**
16:                 $in' := \{i \mid i \in IN[s], layer(statement, i, l) = l'\}$
17:                 add new task $\langle l', statement, in' \rangle$ to $PriorityQueue$
18:         **else**
19:             $OLD := OUT[statement]$
20:             $IN[statement] := \sqcap\{OUT[pred] \mid pred \in predecessors(statement)\}$
21:             $OUT[statement] := f_{statement}(IN[statement])$
22:             **if** $OLD \neq OUT[statement]$ **then**
23:                 $worklist \cup = successors(statement)$

The priority queue pops tasks with the lowest priority layers first. $initialTask()$, $isTrigger()$, and $layer()$ are specified by the developer.

76

JIT analysis forces an intermediate fixed-point by not modifying the *OUT* set (line 14), stopping the current analysis task prematurely. Non-trigger statements are handled in the same way that the traditional analysis does (line 18 to line 23).

To eventually compute the same results as the traditional analysis, the JIT analysis creates new tasks at triggers, and adds them to the priority queue to be executed later (line 15 to line 17). When a task is executed, the JIT analysis pops the next highest-priority task from the queue. It then creates a new instance of the traditional analysis, and initializes it with the appropriate *IN* set, to continue the propagation where the previous task stopped. The role of the priority queue is to prioritize task propagation to report certain warnings first. This is determined by *layer(s,i,l)*, that returns the priority layer *l'* of the new task that will continue propagating the fact *i* at statement *s*, knowing that it was paused at layer *l*.

While there are multiple ways of instantiating the JIT concept, Algorithm 2 requires minimal changes to adapt existing analyses: (1) a priority queue is added to the solver, (2) no changes are introduced in the original flow function $f_s()$, leaving the definition of the data-flow problem entirely unmodified, and (3) different priority systems can be instantiated independently from the solver and the flow functions through *initialTask*(), *isTrigger*(), and *layer*().

In summary, in order to create a JIT analysis according to Algorithm 2, the analysis developer must ensure the following requirements:

- The base analysis must terminate.

- The analysis problem must be distributive.

- The priority layers must provide a complete and disjoint partitioning of the *IN* set.

Layering by locality as described in Table 6.1 fulfills these requirements by using method calls as triggers, and partitioning *IN* sets according to their callees. Other layering strategies can be used to fit other problems, e.g., by confidence or computational resources.

Termination Algorithm 2 extends an existing traditional analysis. If the traditional analysis terminates, the inner loop (line 12) is guaranteed to terminate for all analysis instances, because the algorithm does not modify the *IN* and *OUT* sets. The outer loop (line 4) also terminates, because the number of tasks is bounded. Since tasks depend on their associated set of facts, if the data-flow lattice of the traditional analysis is bounded, the number of facts—and therefore of tasks—is also bounded. At line 6, we check that no task is computed twice, ensuring termination and improving efficiency.

Soundness To be as sound as the base traditional analysis, the JIT analysis checks that every data-flow fact created by the flow functions of the traditional analysis is assigned to at least one layer. Algorithm 2 partitions the *IN* set of a statement into smaller sets (line 16). For this operation to be safe, the data-flow facts should be separable so that data-flow facts can be independently distributed between the layers. This is automatically ensured if the analysis is distributive. We further improve efficiency by assigning each data-flow fact of an *IN* set to exactly one layer.

6.3   Cheetah, a JIT Taint Analysis for Android Applications

We present Cheetah, a JIT taint analysis for Android applications implemented as an Eclipse plugin. Cheetah instantiates the JIT concept with a prioritization system based on proximity to the current edit point of the developer in the code. We first detail the implementation of the

tool and how it instantiates the JIT framework. Then, we describe Cheetah's GUI, showing its features and their relationship to the requirements identified in Chapter 5. The source code of Cheetah and a video demonstration are available online [86].

6.3.1 Implementation

Following the layered approach from Section 6.2.1, Cheetah instantiates the JIT concept by extending a batch-style taint analysis, which we refer to as Batch. Batch is implemented on top of the IFDS framework and its flow functions are defined in Section 2.3. In this section, we detail the implementation of Cheetah.

Layered Taint Analysis

Using the layers in Table 6.1 and Algorithm 2 to transform a traditional IFDS taint analysis into Cheetah, we define:

$$initialTask() = \{\textbf{L1}, startPoint, \{0\}\}$$
$$isTrigger(s) = s.containsMethodCall()$$
$$layer(s,i,l) = distance(s.callee, startPoint)$$

Cheetah marks all call sites as triggers, meaning that the data-flow propagations at call sites are paused and continued in subsequent tasks. The layer that is assigned to a fact at a call site is determined by the distance (in terms of the priority layers) between the callee and the start point of the analysis, which is the method containing the current edit point. For example, if Cheetah encounters a call to a method that is in the same file but not the same class as the starting point, the new task is assigned **L4**. As a result, one task creates as many tasks as the number of call sites it contains. New tasks are added to the priority queue, and are executed in order of distance from the starting point. To adapt Algorithm 2 to the IFDS framework, we apply the following changes:

- Every time a task is executed (line 7), Cheetah creates a new IFDS instance starting at the task's start statement (s_t), and initializes it with the facts contained in its *in*-set. To reuse previously computed results, the state of the IFDS solver is carried over from one instance to the next.

- The priority queue is initialized with the task $\{\textbf{L1}, stmt, \{0\}\}$, where *stmt* is the first statement of the currently edited method, and 0 is the initial fact for a standard IFDS propagation.

- To pause the analysis at call sites and create a new task, we extend Batch's call flow function:

$$\langle stmt \rangle'(\alpha) = \begin{cases} \langle stmt \rangle(\alpha) & \text{if } stmt = task.startStmt \\ \emptyset & \text{otherwise} \end{cases}$$

  Returning $\emptyset$ ensures that the propagation of the data-flow facts is stopped at all call sites, except for when the call is the start statement of the current task. In this case, Cheetah collects the variables that need to be propagated further (i.e., the parameters of the call, the static variables, and the receiver of the call) in an *inSet*. A new task $\{layer(stmt), stmt, inSet\}$ is then added to the priority queue to be executed later. This change corresponds to line 14–line 17 in Algorithm 2.
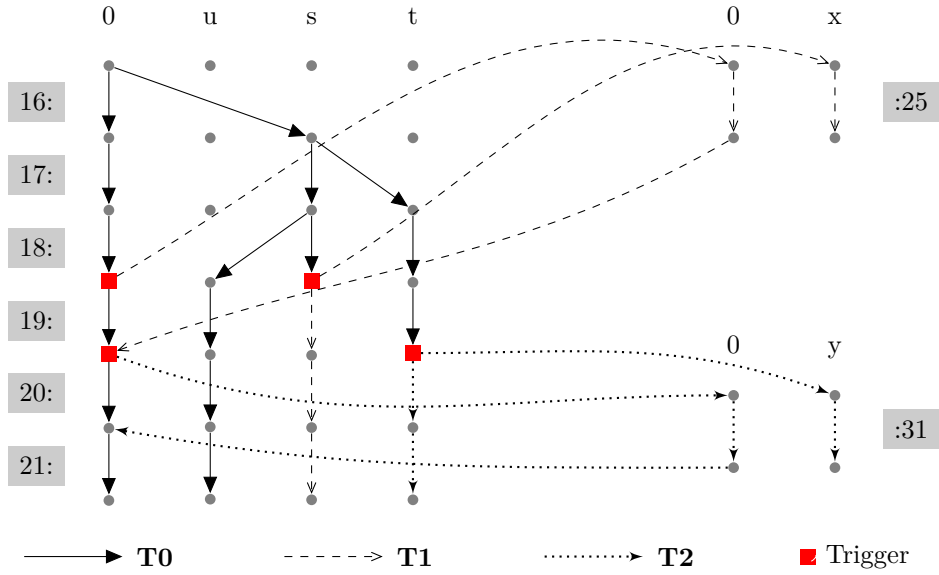
Figure 6.1: Exploded Super Graph (ESG) of the example Listing 2.1 for a forward taint analysis. The statements are represented by their line numbers (highlighted in gray).

- The normal, return, and call-to-return flow functions remain the same as Batch's.

Applying Cheetah to the example in Listing 6.1 results in the following steps (also shown in Figure 6.1):

1. The user triggers the analysis at the *main()* method. Task **T0** = {**L1**, *line* 16, {0}} is enqueued.

2. **T0** is executed, resulting in the solid edges labeled **T0**.

   (a) (α) is found and reported.

   (b) Task **T1** = {**L2**, *line* 19, {0, s}} is created.

   (c) Task **T2** = {**L4**, *line* 20, {0, t}} is created.

3. Task **T1** is executed, resulting in the dashed edges labeled **T1**, and (β) is reported.

4. Task **T2** is executed, resulting in the dotted edges labeled **T2**, and (γ) is reported.

Call Graph-Related Challenges

Traditionally, an IFDS-based analysis requires access to all classes in a given program. It also requires a call graph as input, which is typically constructed before the analysis starts. Implementing Cheetah using this traditional approach would result in an unnecessary initial cost for class loading and call-graph construction, which would negatively affect Cheetah's responsiveness. To address this issue, Cheetah only loads the classes that are necessary to execute the current task. To construct the call graph, Cheetah uses Soot's *OnTheFlyJimpleBasedICFG*, an on-demand algorithm that uses the class hierarchy to resolve calls, similar to the Class Hierarchy Analysis approach (CHA) [12]. This approach for class loading and call graph construction enables Cheetah to quickly deliver the results that are computed in the early layers. Since Cheetah's call graph is based on CHA, we also use a CHA call graph for Batch.

To emulate the Android lifecycle and callbacks, Arzt et al. [9] introduce a dummy main method that explicitly calls all registered callbacks in all possible orders. However, this approach is not useful in the context of a JIT analysis such as Cheetah, because resolving callbacks in a class requires loading it. Therefore, creating the dummy main method for the whole application means loading all classes at the beginning, contradicting the class-loading and call-graph construction strategy that Cheetah uses. Cheetah thus models the Android lifecycle on a per-class basis by distributing the dummy $main()$ over the layering system, at **L3** and **L8**.

Full Code Coverage

Unlike traditional analyses, Cheetah aims at supporting software developers who may be working on unreleased features that are not yet reachable in the codebase. While traditional analyses typically ignore unreachable code, Cheetah analyzes the full codebase, including unreachable code, to report warnings about those unreleased features to the developer. This *full code coverage* is a useful feature for development scenarios where developers work on incomplete programs or programs that may mot even have a main method, which a traditional IFDS-based taint analysis does not provide.

This feature is implemented by artificially creating tasks to cover the whole codebase. Each task instance implements the methods *requiredTasks* and *nextTask*, that respectively create tasks within the same scope as the current layer with a lower priority value, and the same task as the current task with a higher layer value, thus ensuring that the entire codebase is analyzed. An extra check ensures that no task is executed twice. This process is transparent to the developer.

Reporting Results

To report results, Cheetah provides a witness for each reported warning in its *Detail view*. To track tainted paths, we augment our data-flow facts with more information, similar to Lerch et al.'s method in FlowTwist [63]. Each data-flow fact holds its predecessor, its source statement, and a list of neighbors. A loop-aware depth-first search then provides one or more witnesses that show the path causing the warning reported by Cheetah. We also use those witnesses to compute locality metrics for our empirical evaluation.

6.3.2 User Interface

We detail Cheetah's GUI (Figure 6.2) and how it addresses **RE-S1**–**RE-S4**.

Integration with Eclipse

To enable a smooth integration with Eclipse, Cheetah is used like the Eclipse incremental builder: when the user saves the file, the project is rebuilt and compilation errors appear on the left gutter and in the Problem view at the bottom. Similarly, Cheetah hooks into the Eclipse incremental builder and is rerun when the project is saved, starting from the method that currently holds the focus. Every run of Cheetah kills any previous analysis instances, invalidating previous results until the current instance of Cheetah confirms or refutes them.

In the GUI, Cheetah warnings are shown in the left gutter (Ⓓ) and in the *Overview view* at the bottom (Ⓐ). The gutter icons show sources and sinks, and tooltips appear on hover to provide additional information about the data leak. To provide quick feedback to the developer, the icons can have two states: confirmed (in blue) and pending (in gray). The latter is used for warnings reported on the later layers of the previous analysis run, to notify the user that the current run of analysis is still in the process of computing them. The icons also have tooltips to provide additional information about each statement in the trace of the selected warning.
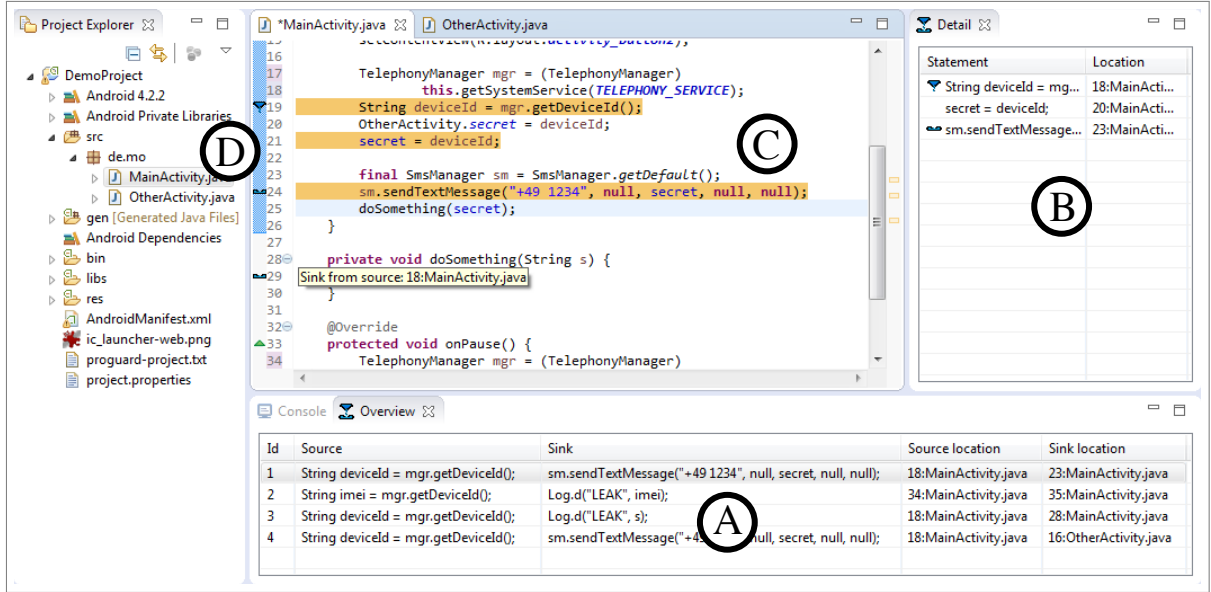
Figure 6.2: The GUI of CHEETAH. Features (A)–(D) are detailed in Section 6.3.2.

This integration limits the disruption of the developer workflow, addressing **RE-S1** and **RE-S2** with a responsive analysis (**F1**), a responsive GUI (**F2**), and visualizing the warnings in the codebase (**F10**).

Decluttered Views

To declutter the *Overview view*, CHEETAH uses two views to show warnings. The *Overview view* ((A)) provides a list of all reported warnings, and the *Detail view* ((B)) displays a trace of the selected warning, offloading the amount of information contained in the *Overview view*. This trace represents an evidence that there exists a path from the source to the sink, ensuring the validity of the warning.

Interleaving developer activities and fixing warnings requires careful reporting. Otherwise, warnings can literally become moving targets in result lists as new ones are constantly found and others are fixed, which confuses the developer. To address this issue, we introduce color-coding in the *Overview view*. Warnings in CHEETAH have three states: *confirmed* (confirmed by the latest analysis run), *pending* (found by the previous analysis run, but not yet confirmed by the current run), and *fixed*. CHEETAH displays pending warnings in gray in the left gutter and in the *Overview view*. Fixed warnings are grayed out for one run of the analysis, and removed from the view at the next run. This feature provides a light history of fixed leaks, allowing users to quickly check if a fix was effective.

The CHEETAH perspective aims at helping developers understand the warning list (**RE-S4**) by providing them with a responsive GUI (**F2**), a detailed trace of the data leak in the code (**F4**), and a graying system that allows users to track what has been achieved (**F15**). In addition, it displays the warnings in the *Overview view* in a precise order defined by the prioritization algorithm (**F13**).

81

Other features

To clarify the presentation of its results, CHEETAH also highlights the trace of the selected warning in the code (ⓒ) and provides navigation functionalities to jump between the different views and the code editor.

6.4 Evaluation

We evaluate CHEETAH through an empirical evaluation with 14 Android applications, and a user study with 18 developers. In both cases, we compare CHEETAH to the batch-style taint analysis it was derived from: BATCH. CHEETAH and BATCH have the same flow functions, except for the prioritization modifications that enable CHEETAH to be Just-in-Time.

BATCH's GUI is designed to be identical to CHEETAH's, with the exception of the quick updates system. Instead of triggering the analysis on a save action, the developer must click on a button to trigger the analysis. While the analysis runs, the GUI is blocked and displays a waiting popup that prevents the developer from modifying the codebase. When the analysis finishes, the warnings are displayed in the *Overview view* in the order in which they were detected.

We aim to evaluate the prioritization system of our JIT analysis. Thus, we verify if CHEETAH's layers are a sensible choice, if the analysis is able to report initial warnings quickly, if it reports warnings of interest (e.g., warning with shorter traces) early, and if it is sound. As a result, we ask the following research questions for our empirical evaluation:

- **RQ13**: How responsive is CHEETAH compared to BATCH?

- **RQ14**: Which layers of CHEETAH are most used?

- **RQ15**: Are the initial findings of CHEETAH easier to interpret than later ones?

- **RQ16**: What are the precision and recall of CHEETAH compared to BATCH?

In a second part, we evaluate the usability of CHEETAH as an IDE tool, and verify if the features presented in Chapter 5 and evaluated in the empirical evaluation help developers in practice. In our user study, we aim to answer the following research questions:

- **RQ17**: Can CHEETAH help developers fix data leaks faster than BATCH?

- **RQ18**: Which features of CHEETAH are most useful to the participants?

- **RQ19**: Does CHEETAH's GUI help debug data leaks better than BATCH's?

6.4.1 Empirical Evaluation: Responsiveness, Understandability, Precision

In an empirical evaluation, we compare CHEETAH and BATCH's responsiveness, precision, and recall. We discuss the layering strategy used for CHEETAH, and the ease of understanding its first warnings.

We ran the experiments on a benchmark suite composed of 14 Android applications selected from the most recent 100 applications in the F-Droid repository [35] (at the time of the study, in 2016), such that each application has a GitHub repository with more than one commit and is available for mining in Boa [31]. The 14 applications are available online [86].

The analyses run with BATCH have one starting point: the dummy *main*() method that acts as the entry point to the Android application. For CHEETAH, we consider two types of starting points: the methods modified in the commit history of the applications' repositories,
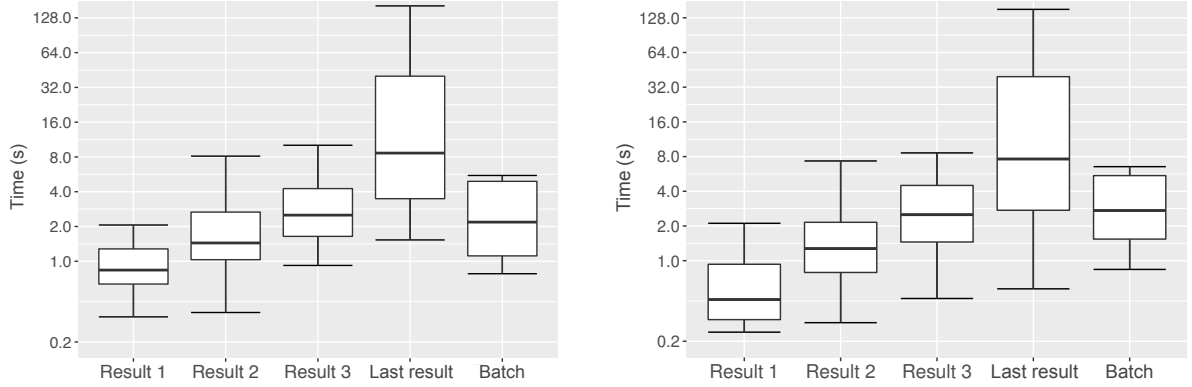
Figure 6.3: Time to report warnings (in log scale) for Cheetah and Batch, starting at SPB (left) and SPS (right).

which we refer to as SPB (Starting Points Boa), and the sources of known data leaks in the applications, which we name SPS (Starting Points Sources). SPB model the cases when the user is in active code development and does not use Cheetah, and SPS, the cases where the developer investigates a particular bug. The SPS were collected after running Cheetah once on the applications, and the SPB were collected using Boa. Each application has at least 26 unique SPB (min: 26, max: 316, median: 127).

We ran two sets of experiments: one with SPS and one with SPB. In each of the experiments, for each of the 14 applications, we ran the full Batch analyses 20 times, and Cheetah with 20 randomly selected methods among either SPS or SPB. We ran our experiments on a 64-bit Windows 7 machine with one dual-core Intel Core i7 2.6 GHz CPU running Java 1.8.0_102, and limited the Java heap space to 1 GB.

6.4.2 Evaluation Results

In this section, we detail the results of our empirical evaluation, answering **RQ13**–**RQ16**.

Responsiveness (RQ13)

We have measured the time that Cheetah takes to report the first, second, third, and last result when it starts at SPB and at SPS. We compare those times to the time that Batch takes to report its final results. Figure 6.3 shows, in log scale, the response times for those quantities.

Across our benchmark, Cheetah reports the first result in a median time of less than 1 second when it starts at SPB and a median of less than 0.5 seconds when it starts at SPS. These results are below Nielsen's 1 second recommended threshold for interactive user interfaces, suggesting that Cheetah usually allows the "user's flow of thought to stay uninterrupted" [95].

Batch's run times are consistent between the two experiments, with median times of 1.85 seconds and 2.13 seconds, which are significantly smaller than the median time of 9.03 seconds Cheetah takes to complete the analysis with SPB and 7.79 seconds with SPS. We attribute this to the full code coverage feature of Cheetah, which creates more data-flow facts along paths that are unrealizable from the dummy $main()$. Any analysis imprecision in those parts propagates to the other computations, making the analysis perform more work than strictly necessary. Nevertheless, such a feature is desirable for real-life code development scenarios.

We also note that Cheetah returns warnings generally faster with SPS than with SPB, showing that in scenarios where the user is focused on particular analysis warnings, Cheetah is able to report those relevant warnings faster.
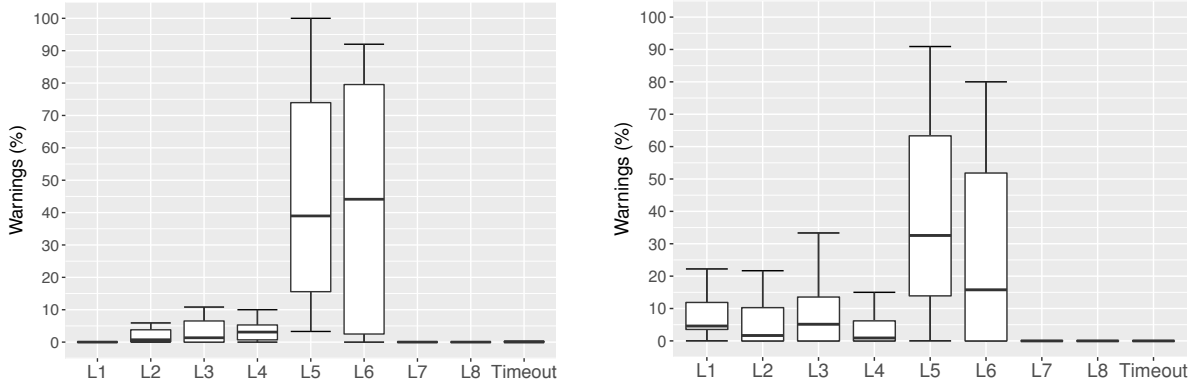
Figure 6.4: Percentage of warnings reported at each layer in Cheetah with SPB (left) and SPS (right).

Layering System (RQ14)

To evaluate the layering system chosen for Cheetah (Table 6.1), we count the number of warnings reported at each layer. Figure 6.4 presents those numbers for SPB and SPS. We see that across our benchmark, when Cheetah starts at SPB, a median of 38.97% of the warnings is reported in **L5** and a median of 44.12% in **L6**. Starting at SPS, Cheetah reports a median of 32.56% warnings in **L5** and a median of 15.77% in **L6**. Thus, with SPS, Cheetah reports more warnings in earlier layers: a median of 4.58% in **L1** and a median of 5.13% in **L3**. Unlike SPB, SPS simulates scenarios where users are interested in the analysis results. In those cases, 33.3% of the warnings are reported at **L1-L4** on average, against 11.6% for SPB.

We observe that if Cheetah is guided towards precise data leaks, more warnings are reported at earlier layers, because the scope of the analysis now focuses around known tainted paths. Tracking those taints at lower layers also reduces the time spent to report the corresponding leaks, since a smaller program scope means loading fewer classes and propagating data-flow facts through a smaller portion of the program. The proximity of the starting point to the source could thus explain why, in Figure 6.3, early warnings are returned faster with SPS than with SPB. Therefore, starting at SPS is optimal when the user requires analysis updates while fixing a particular warning.

After Cheetah reports a data leak, a separate module retrieves the paths between the leak's source and sink to provide the user with more information. The process of retrieving those paths times out in less than 1% of all the cases (average: 0.81%, median: 0%). It is important to note that, for those timeouts, Cheetah itself does not time out, but the path-finding module does.

For our benchmark, no results are reported in **L7**, because none of the applications pass sensitive information through polymorphic calls. Similarly, no warnings are reported in **L8**, because Cheetah does not support inter-component flows.

Warning Understandability (RQ15)

The quick response time of Cheetah is most useful when the first warnings that it reports are easier to interpret by the developers. Otherwise, they spend more time trying to trace their way through the program to understand more complex warnings.

We approximate the ease of interpretation of a warning with the length of its trace: the number of statements between the source and the sink. Figure 6.5 shows the trace lengths for the warnings that appear in each layer of Cheetah. When Cheetah starts at SPB, the median length of the traces for the initial layers **L1**–**L4** is 0, 1, 4, and 4 statements, respectively. For later layers, Cheetah reports more complex warnings with longer trace lengths: medians of
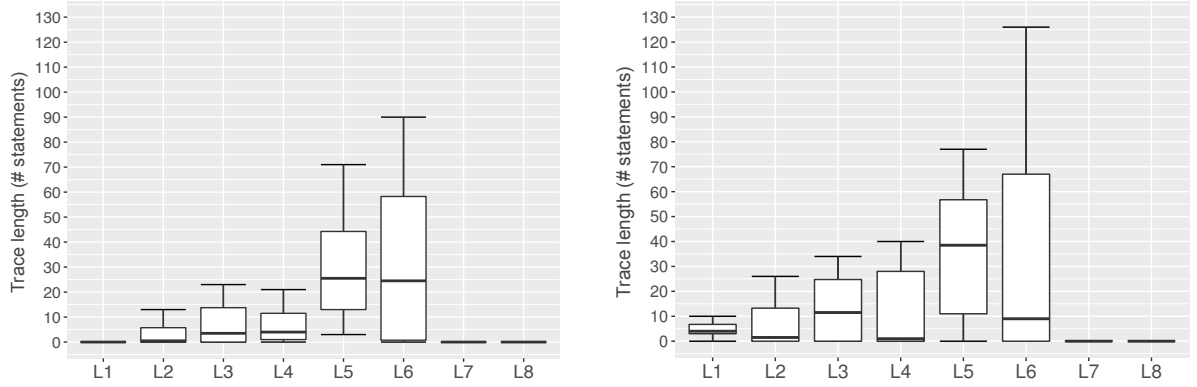
Figure 6.5: Trace length of the warnings reported at each layer in Cheetah with SPB (left) and SPS (right).
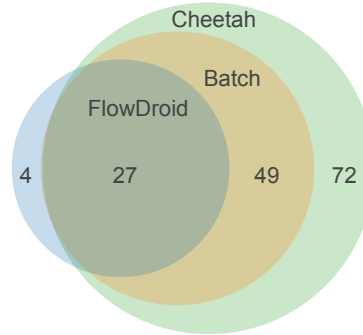
Figure 6.6: Number of data leaks found by FlowDroid, Batch, and Cheetah as a Venn diagram.

26 and 25 statements for **L5** and **L6**, respectively. Starting at SPS, Cheetah reports more warnings in earlier layers. In such a case, the median length of the traces that Cheetah reports for the initial layers **L1**–**L4** have a median length of 4, 2, 12, and 1 statements, respectively.

We see that with both SPB and SPS, Cheetah tends to report warnings with shorter traces in early layers (**L1**–**L4**), making them easier to interpret than warnings with equivalent longer traces. This confirms the assumption made in Section 6.2.1, that a second effect of prioritizing by location is that early warnings span a smaller scope of the codebase, and should have smaller traces. As a result, we see that the warnings that are reported fast by Cheetah are also the easiest to understand.

Most of the warnings are reported on **L5** and **L6**, which both require class loading on a large scope: a package or the entire project. More granularity could be added to those layers to minimize the loading time, e.g., handling aliases at a later stage.

Precision and Recall (RQ16)

Figure 6.6 presents a Venn diagram that shows the number of warnings reported by Cheetah, Batch, and FlowDroid [9] across all 14 Android applications. FlowDroid is a reputable taint analysis optimized for Android, which we use to illustrate the applicability of Batch to real-world applications. We see that Batch reports all warnings found by FlowDroid, except four. Those warnings require the analysis' flow functions to handle threads and model application layout, which are currently not supported in Batch. Batch also reports more warnings than FlowDroid, which is due to its lower precision. For example, Batch uses a CHA call graph, while FlowDroid uses a more precise call graph. FlowDroid also performs type checking to eliminate

85

spurious taints. Overall, while BATCH is less precise than FlowDroid, it still approximates its precision and soundness. Since CHEETAH shares the same flow functions as BATCH, we can also extend this reasoning to CHEETAH, showing that by construction, both analyses have the same precision and recall when compared to FlowDroid. We can conclude that that both CHEETAH and BATCH are representative of taint analyses that can be used on real-life applications.

We observe that CHEETAH reports 2.1× more warnings compared to BATCH (min: 1×, max: 10×, geometric mean: 2.06×). CHEETAH reports those additional warnings because it analyzes the whole program, including unreachable code. In real life, developers may be working on code that is still unreachable from the program's entry points. CHEETAH analyzes those parts of the codebase, generating data-flow facts that are propagated further, into new parts of the code, or back into already analyzed ones, which then require a new partial analysis. This system helps provide the user with a more relevant result set than traditional analyses such as BATCH, but also increases the number of reported warnings: the cascading effect of a single imprecision in the analysis generates more data-flows with CHEETAH than with BATCH, which explains why CHEETAH takes significantly longer than BATCH to report its last warnings (Figure 6.3).

6.4.3 User Study: Usability of Cheetah

In a comparative user study between CHEETAH and BATCH, we evaluate how a JIT analysis integrates into the development workflow compared to a batch-style analysis.

Experimental Setup

To make the conditions of the study as close to day-to-day development activities as possible, the participants were asked to perform a development task: removing code duplicates in an Android application. At the same time, we asked them to keep the number of data leaks to a minimum. To help detect potential data leaks, CHEETAH and BATCH were provided as Eclipse plugins. To fix the leaks, we provided the participants with data sanitization libraries. Each task was limited to 10 minutes.

In a within-subjects study, each participant performed one task with each tool (BATCH and CHEETAH). Before each task, the participants were primed with a small Android application to get used to the tool. The order of the tools was randomized, so that half of the participants started with CHEETAH, and the other half with BATCH, in a simple latin-square design. Afterwards, the participants filled a comparative questionnaire and were interviewed in person. The test applications, questionnaire, responses, and the interview protocol are available online [86].

The priming was performed on a small, artificial Android application that contains 6 simple data leaks. The two main tasks were performed on a real-life application from F-Droid: Bites [34], a basic cookbook app. Because each task had to be completed in a limited amount of time (10 minutes), we have modified Bites to add data leaks around code duplications, resulting in a total of 106 more complex data leaks. This ensured that participants encountered data leaks while conducting their duplication removal task. In the pilot study, some participants had spent most of their time handling code duplicates not related to any data leaks.

We first conducted a pilot study with 11 participants of mixed skill and experience levels in both Android development and static analysis tools (including professional developers and Android security analysts). The outcomes of the pilot study were mixed, because most participants were distracted the initial GUI of CHEETAH. One specific issue was that the *Overview view* kept changing as new results were computed, making it impossible to track a particular warning. We introduced the graying system presented in Section 6.3.2 to address this problem and fixed other issues reported by the initial batch of participants.

Questionnaire and Interviews

The post-task questionnaire is comprised of 48 questions, which we refer to as **Q1**–**Q48**. The questions are detailed in Appendix D.

1. **Participant information:** **Q1**–**Q4** gather information about the participants. **Q1**–**Q3** ask for their professional experience with Java development, Android development, and with using static analysis tools, in years. **Q4** asks them which analysis tools they have used in the past.

2. **System Usability Scale:** **Q5**–**Q13** are questions from the System Usability Scale (SUS) [18] for the first tool used by the participants (Batch or Cheetah). Each question is rated on a Likert scale from 1 to 5. **Q14**–**Q22** are the same SUS questions for the second tool participants used.

3. **Workflow integration:** **Q23**–**Q33** asks the participants how they perceived the integration of both tools in their development workflow. **Q23** asks them for which tool blocked the UI, and **Q24** if this system made it easier to correct data leaks, on a Likert scale from 1 (Strongly disagree) to 5 (Strongly agree). **Q26** and **Q27** ask which tool was triggered with a button and on build, respectively, and **Q28** asks which system they preferred. **Q30** asks participants with which tool they had to wait longer to obtain updates and gave three choices: the first tool, the second one, or neither. With the same choices, **Q31** asks participants with which tool they noticed changes in the ordering of the warnings throughout the tasks, and **Q32** asks them if they were comfortable with this system on a Likert scale from 1 (Strongly disagree) to 5 (Strongly agree).

4. **Perceived usefulness:** In **Q34**, **Q35** and **Q38**, participants noted which tool was more useful for understanding warnings, correcting warnings, and understanding the application code. Participants also marked with which tool they thought warnings were faster to understand (**Q36**) and to correct (**Q37**). Finally, they noted which tool they would rather use to correct data leaks (**Q39**). For those questions, participants could choose between the first tool, the second tool, or neither.

5. **Net Promoter Score:** Using the Net Promoter Score (NPS) [109] in **Q40**–**Q41**, participants rated on a Likert scale from 0 to 10 how likely they would recommend a tool environment above the other.

6. **Open questions:** **Q42**–**Q48** are used to gather general comments about the tools, and capture the usability issues participants encountered. Those full-text questions serve as guidelines for the short interviews. In **Q46**–**Q47**, we ask participants which tasks they would rather use each of the tools for. In **Q42**–**Q45**, we ask for the positive and negative features of both tools, and in **Q48**, what they would change in Cheetah or Batch.

During the individual interviews, we collected qualitative information on the participants' experience of the tools, focusing on the perceived differences, in particular waiting times, integration of the tools in the IDE, and warning ordering. The interviews lasted 14 minutes on average (min: 10 minutes, max: 23 minutes).

Participants

Our study includes 18 participants of varying backgrounds (9 academics and 9 professional developers), with different skill levels in terms of Android development and knowledge of taint
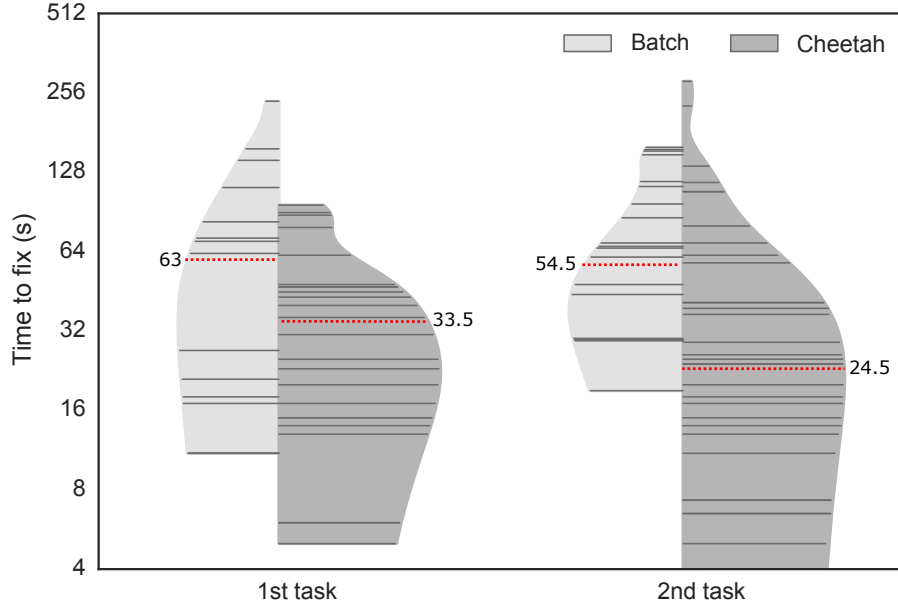
Figure 6.7: Violin plot representing the distribution of the times to fix leaks across all participants, by task and tool used during the task. Each horizontal line represents data leaks fixed in the corresponding time. The length of a line represents the number of leaks fixed in that time. Dashed lines are medians.

analysis. In the following, we identify them as **P1**–**P18**. We discard **P17**'s data, because they encountered a user interface bug and was unable to properly perform the tasks. The 18 participants of the main study are disjoint from the 11 participants of the pilot study.

While only 17.6% of the participants had professional development experience with Android, 47.1% had more than 10 years of such experience with Java. In addition, 29.4% had between one and five years of development experience with Java, 17.6% had between five and ten years, and 5.9%, under one year. The majority of the participants had little to no professional experience with using static analysis tools (88.2%), the remainder having between three and four years of experience with such tools. This low experience of analysis tools reduces the bias participants would have towards a particular workflow they may be familiar with, as noted by **P18** who had four years of professional experience using static analysis tools: "Due to [my] past experience as a code auditor, I prefer to separate [the] tasks [of developing and fixing warnings]."

6.4.4 Study Results

We now present the results of the comparative user study, answering **RQ17**–**RQ19**.

Fixing Data Leaks (RQ17)

Figure 6.7 shows, in log scale, the distribution of the time taken to fix a data leak for the two tasks. The reported numbers take into account the time taken to fix a leak, discarding the time needed by the participant to understand the code, the tool, or to discuss the solution before implementing it. Across both tasks, participants using CHEETAH took half as long as participants using BATCH to fix a data leak ($0.53\times$ and $0.45\times$, respectively). For Task 1, the median time to fix a data leak using BATCH was 63 seconds, compared to 33.5 seconds per leak for CHEETAH users. The times reported for Task 2 are lower (54.5 seconds per leak for BATCH and 24.5 seconds per leak for CHEETAH, which we attribute to the participants getting used to

the application and the tasks. We also note that across both tasks, participants were significantly faster when using Cheetah compared to Batch ($p < .01$, Wilcoxon Rank-Sum test), regardless of whether they used Batch first and Cheetah second ($2.6\times$ faster), or Cheetah first and Batch second ($1.6\times$ faster).

In Figure 6.7, we see that the plots for Cheetah contain more horizontal lines than the plots for Batch. This shows that in the 10 minutes allocated to each task, the participants using Cheetah fixed more leaks than the participants using Batch. Overall, Cheetah users fixed more data leaks than Batch users, with a median of two data leaks with Batch and four data leaks with Cheetah in Task 1. For Task 2, the medians are 3 data leaks for Batch users and 4 data leaks for Cheetah users. The Wilcoxon Rank-Sum test failed to detect significant differences in the number of leaks fixed ($p = .31$).

We can thus conclude that using Cheetah enables software developers to fix leaks twice as fast compared to using Batch.

Most Useful Features of Cheetah (RQ18)

In the questionnaire and interviews, we asked the participants which features of the tools were most and least useful to them. Three particular features stood out: quick updates, ordering, and the graying system in the *Overview view.*

Quick Updates: In total, 12 participants found Cheetah's quick updates useful, noting this feature as the main advantage of the tool. In particular, professional developers noted that this system is "much more comfortable, and what I would expect in the Eclipse environment" (**P7**). **P2** noted that Cheetah "integrates well into the Eclipse build-on-save paradigm". Cheetah is reported to have a "more fluent workflow" (**P9**), as opposed to Batch, which proved more interruptive to the participants: "having to wait interrupts the coding and thinking process" (**P6**). **P4** explained from their personal experience with UI-blocking compilation tools that they "do a context switch in your head. [...] When you are back to the actual work, you might have forgotten what you wanted to do". In summary, participants felt that for code development, Cheetah was less interruptive, because it allowed them to deviate less from their coding tasks.

Ordering: While only two participants noticed that Cheetah applied an ordering strategy, seven participants complained about the random ordering offered by Batch, saying that they would like to be able to "sort the leaks by source, sink, line number" (**P6**). No such complaints were made for Cheetah, regardless of the order in which the participants used the tools, which further validates the choice of layers in Cheetah. **P8**, a professional Android developer, noticed the ordering in Cheetah, and commented, when using Batch: "When I'm in one class, I get familiar with it, and when I click on a warning, it takes me to a completely different class, and I have to get used to it again". **P18** did not notice Cheetah's ordering, but handled the leaks in the order in which they were presented. They fixed all encountered leaks when using Cheetah, but skipped most of the first warnings when using Batch after deeming their traces "too long". We see that reporting warnings following Cheetah's layers positively affects participant performance and integrates more discretely into their workflow.

However, Cheetah's ordering only helps developers if they follow it. **P13** wanted to have a full overview of all warnings at each code change, and thus waited until Cheetah had completed before resuming development, which took longer than Batch's run time. **P15** started fixing data leaks starting from the bottom of the list. Such warnings were reported by the later layers, and thus took a long time to update, which led to longer waiting times than with Batch.

Graying system: The graying system was deemed useful by some participants: "I had the impression I did something" (**P16**), "History is useful" (**P9**), but it proved confusing to others, as they still had trouble finding out if a fix was successful. Some participants noted that the coloring was "disturbing because results changed" (**P6**). The shifting colors prompted them to wait until the *Overview view* stabilized, showing that despite the graying system, that view still proves complex to understand. **P1** noted that the feedback given by the tool on a change "should be more prominent". To see if a leak was fixed, some participants relied on the gutter icons instead: "The sink [icon] has now disappeared, that's a good thing" (**P5**). Seven participants requested clickable icons. Having such a feature would help users focus on the code more than the *Overview view*, integrating the tool better into the IDE.

Other features: Participants reported other bugs and usability features that we leave for the next design iteration. In particular, two expert participants expressed performance concerns about CHEETAH running too often on big projects: "if the analysis affects the performance, I would like to have a button to control it" (**P13**), "if it has a big impact on the CPU, it might be annoying and I might not be as productive" (**P4**). This concern led some participants to think that "the analysis was slowing the IDE down" (**P13**), as it took more time to compute all warnings than BATCH. We note that CHEETAH runs in the background, and was given enough memory to not interfere with the UI thread. This issue could be mitigated by for example introducing incremental analysis into CHEETAH to further improve its performance.

Debugging Data Leaks with Cheetah and Batch (RQ19)

Overall, the participants responded positively to CHEETAH. Its NPS score is of 6, denoting a positive score. In comparison, BATCH received a score of -94. According to the aggregated SUS scores, 12 participants rated CHEETAH higher than BATCH, and 4 rated BATCH higher. Using a Wilcoxon Signed-Rank test [145], we observed significant ($p < 0.05$) differences between these aggregated scores and participant's responses on four of the individual SUS questions: "In general, I think that I would use tool X frequently", "I found tool X unnecessarily complex", "I found the various functions of tool X well integrated", and "I found tool X very cumbersome to use". With those questions, we see that compared to BATCH, participants less likely found CHEETAH unnecessarily complex or cumbersome (-0.6 mean response each). Moreover, the participants responded that they were more likely to use CHEETAH frequently (+0.7 mean response), and more likely found its functions well-integrated (+0.5 mean response).

When asked in which cases they would use one tool rather than the other, twelve participants reported CHEETAH is best suited for code development. **P9**, in particular, noted that it would make the development task slightly harder, but it would "force me to write better code from scratch". Eleven participants noted that BATCH should be used infrequently or in situations where debugging and coding are separated: "after a milestone" (**P9**), "creating reports for software" (**P7**). No participants reported they would use BATCH for code development. This observation confirms that the responsiveness created by the JIT system allows CHEETAH to be used in an IDE less disruptively than a more traditional batch-style analysis.

6.5 Limitations and Threats to Validity

The user study with Bites was conducted on a closed setting, with tasks limited to ten minutes. Because of the limited time, we added data leaks in locations where the tasks were to be performed, ensuring that participants would run into them while removing code duplications. We ensured that the added data leaks are as close as possible to existing data leaks in the code,

by reusing the same source and sink API methods, and emulating the same trace length and scope. While this setup allowed us to eliminate external factors when comparing Cheetah and Batch, it would be interesting to complement this study with one in a practical setting.

As a within-subjects study, the user study created a learning effect between the tools: participants tended to perform better with the second tool. We applied a latin-square design to reduce this effect, with which half of the participants using Cheetah first, and the other half, Batch first, and reported on the aggregated results. We also primed the participants before each task.

Our particular implementation of the JIT analysis concept determines priority based on the distance to the current edit point. As a result, Cheetah reports significantly more warnings than its non JIT counterpart, because a traditional whole-program analysis usually starts from a main method and propagates through the code that is reachable from there. Because Cheetah can start from anywhere in the code, it also covers the unreachable parts of the code. We argue that this approach is better suited for the scenario of code development, where developers often work on new features in incomplete programs that may not even have a main method. Cheetah provides full code coverage by artificially creating tasks that are not naturally induced by the codebase. Since Batch and Cheetah have the same flow functions, they have the same soundness and precision by construction. By covering more of the codebase, Cheetah provides the code developer with a more relevant result set than traditional analyses such as Batch.

Cheetah is an instantiation of the JIT prioritization system that targets responsiveness. While it is able to return warnings quick enough to not disrupt the developer workflow, it still reanalyzes the entire application code instead of just the changeset, like an incremental analysis would do. It would be interesting to merge the two methods together, and obtain an incremental analysis that end-users are able to guide.

6.6   Summary

In this chapter, we address a few of the requirements identified in Chapter 5, focusing on the responsiveness of the analysis tools (**RE-S2**). Through the Just-in-Time analysis concept, we allow software developers to guide the analysis, using their knowledge of the analyzed codebase (**RE-S3**). We also provide a general recipe to create a JIT analysis by modifying a base distributive data-flow analysis with minimal changes to the analysis code or the analysis solver. We illustrate the JIT concept through Cheetah, a JIT taint analysis for finding data leaks in Android applications. Cheetah is designed with the developer's needs in mind, integrating a long-running analysis in the IDE with minimal disruption of the developer workflow (**RE-S1**), and returning more comprehensive warnings first, improving the explainability of the analysis warnings (**RE-S4**). A JIT analysis's layering system can support different layering schemes. As discussed in Section 6.2, an analysis could prioritize by location, performance, confidence, understandability, or combinations of those properties. Cheetah prioritizes warnings by code location, but other schemes can be implemented. The source code of Cheetah is available online [86], and open to contributions.

In an empirical evaluation on 14 real-world Android applications, we show that Cheetah is able to return its first warnings in under a second, that it returns the most understandable warnings first, and that the layering system chosen for Cheetah could be improved, especially towards the later layers. Through a comparative user study with 18 developers, we also show that Cheetah's non-blocking system allows developers to fix warnings twice as quickly than with an equivalent batch-style analysis. Its quick updates and ordering strategy make it particularly well-suited for integrating bug fixing within the natural flow of code development.

Through this study, we confirm the validity of **RE-S1**–**RE-S4** from Chapter 5 and advocate for a better integration of static analysis in the development environment of analysis developers, to assist them when fixing analysis warnings. Traditional static analyses written with precision and scalability in mind often overlook user-experience issues such as responsiveness, causing analysis tools to be used in specific ways, e.g., as part of nightly builds. Through a user-centered design process, we address this issue and explore a different usage scenario that is more comfortable to software developers: direct integration in an IDE.

7

Rule Graphs for Analysis Configuration

In past years, researchers and practitioners have improved the capabilities of static analyses, enabling them to find increasingly complex bugs [9,60], support more languages [118], and report more accurate warnings [128,129] in less time [116]. As analyses grow more complex, using them becomes more difficult. In practice, analyses are known to report many false positives. Some are due to over-approximations (e.g., for collections or arrays), others to missing knowledge about the particular codebase (e.g., specific libraries or coding constructs). As a result, analysis tools are typically configured by end-users (dedicated teams or software developers themselves, depending on available resources) before they are deployed in a company. Such teams typically configure the analysis options, how analysis results are displayed (e.g., how to group warnings together, or decide which ones are more important than others), and edit the analysis rules to customize them to the codebase.

To help developers configure out-of-the-box analyses, allowing the customization of analysis rules is a core requirement. As seen in Chapter 5 (**RE-S3**), developers have external knowledge that the analysis does not possess, and the contribution of such heuristics could help direct the analysis in yielding more accurate warnings. To enable developers to understand and add to the analysis rules, a core notion is warning explainability (**RE-S4**). An analysis interprets the source code and builds its own understanding of how it works. Sometimes, this understanding may not match the developer's, which results in uncertainties, a wrong treatment of critical warnings, wrong tool configurations, or even tool abandonment [47,65].

Static analyses are typically used as black boxes, their warnings being post-processed using information that is external to the analysis rules to provide developers with more complete warnings. In an effort to bridge the understandability gap, we instead propose to make use of *internal* information: how the analysis interprets the analyzed code. Focusing on data-flow analysis, we introduce the novel concept of *rule graphs* that encode internal analysis information, and explain how to use them to give developers more insight into the analysis' reasoning. In our evaluation, we show that the use of rule graphs can improve the developers' understanding of analysis warnings, assist them in classifying warnings, and help identify weak or missing analysis patterns and rules that can be corrected in the analysis' rulesets.

In this chapter, we describe rule graphs, present a general method for transforming static data-flow analyses in order to support them, and show how to use them for four tasks: understanding and classifying warnings, and identifying weak or missing analysis patterns. We illustrate those four uses through a taint analysis and an IntelliJ plugin: MUDARRI. Our user study with 22 software developers shows that the use of rule graphs significantly improves warning understandability. Through an empirical evaluation on Android applications, we illustrate

```
69  protected void doGet(HttpServletRequest request, HttpServletResponse
        response) throws ServletException, IOException {
70    String [] s = new String[5];
71    String userId = request.getParameter("userId");
72    s[0] = userId;
73    s[1] = "safe";
74
75    Statement st = conn.createStatement();
76    String q1 = "SELECT * FROM  User WHERE userId='" + s[1] + "';";
77    ResultSet res = st.executeQuery(q1);
78
79    String url = "https://" + s[1] + ".com";
80    response.sendRedirect(url);
81
82    String q2 = "UPDATE User SET ct = ct + 1 WHERE userId='" + userId + "';";
83    ResultSet res = st.executeQuery(q2);
84  }
```

Listing 7.1: Potential SQL injections (line 77, line 83) and open redirect (line 80) from line 71. The first SQL injection and the open redirect are false positives.

that applying machine learning to rule graphs can be used to classify data leaks in Android, and can help developers discover weak and missing analysis rules.

The work presented in this chapter is currently under submission [90].

7.1 Motivating Example

In this section, we show how exposing an analysis' internal rules can assist the developer in four different tasks: **T1**–**T4**. To illustrate them, we use Listing 7.1 containing an SQL injection [79] from the source at line 71 to the sink at line 83. The potential SQL injection at line 77 and the open redirect [74] (line 80) do not occur, because $q1$ and url only contain the safe string "$safe$" from $s[1]$.

T1: Understand a Warning

To configure an analysis, developers must first evaluate if the warnings it yields are correct, and of interest to their particular situation. To do so, they have to gain a full understanding of the warnings, and in particular, of why the analysis thinks they are to be reported. Current tools generally provide external information that can range from vulnerability descriptions to more complex data such as warning severity, detailed traces, exploit examples, or even fix suggestions. An often overlooked aspect is that the analysis algorithm can be faulty or approximative and can misinterpret certain parts of the code. For example, for the sake of scalability, static analyses often over-approximate arrays: instead of tracking the individual elements, the entire array is considered insecure if one of its elements is. For Listing 7.1, such an analysis would consider all elements in s as dangerous after line 72, and mistakenly report the SQL injection line 77 and the open redirect. Even for a developer who knows how the vulnerabilities occur, figuring out why the analysis reports those two false positives is not straightforward, because the array over-approximation is internal to the analysis and thus, completely hidden from the developer. Analysis shortcomings may come from mishandling varied coding concepts: collections, aliasing, multithreading, etc. We argue that making the analysis' internal rules more explicit to the developer might address this issue and improve the understandability of analysis results.

T2: Classify Warnings

The end-users of the analysis tool (the developers that fix the warnings once the tool is deployed) need to select the warnings they focus on first [47,65]. To help them with this decision, analyses often classify warnings using factors that are external to the analysis such as severity, confidence, code location, bug type, etc. Such classifications can have their limitations. For example, a legitimately hardcoded password may slip through the mass of hardcoded strings falsely reported as hardcoded passwords because they are all classified under the same bug type. As a result, many commercial tools allow classification rules to be customized. To assist developers in setting up those rules, we argue that warning *similarity* can be computed based on *internal* analysis information instead of external factors: similarity is thus determined by how the analysis understands the warnings, and how similarly they can be fixed or refuted. In the example in Listing 7.1, the two warnings closest to each other are not the two SQL injections, but the SQL injection line 77 and the open redirect line 80: they both undergo the same array assignment line 72, and are both false positives for that reason. They could thus be classified together, in an "Array assignment" category.

T3: Identify Weak Analysis Patterns and T4: Identify Missing Analysis Patterns

Because of its necessity to over-approximate, static analysis often reports false positives [47,65]. To help developers differentiate between true and false positives, many analysis tools calculate a confidence metric (how confident the analysis is that a warning is a true positive) generally based on external features, such as the bug type (e.g., SQL injection) [70,114]. Since false positives are mainly due to the analysis' weaknesses, we argue that using internal analysis information would help identify those weaknesses and their resulting warnings. Discovering which combinations of analysis rules are at fault or missing allows developers to modify or add those rules to avoid such warnings in the future, as a typical source of false positives. A weak pattern (**T3**) that matches the array over-approximation in Listing 7.1 could be the write-access to the array at line 72, thus marking the warnings line 77 and line 80 as rather likely false positives. Similarly, missing analysis patterns (**T4**) can also yield false positives or false negatives. For example if arrays are not handled by the analysis, they would be ignored and potential warnings resulting from the tainted element $s[0]$ would not be reported.

7.2 Related Work

In this section, we present past research about improving warning explainability and classification, and the integration of developer-specific knowledge in the analysis rules to find weak or missing analysis rules.

7.2.1 Warning Explainability and Classification

Past studies have highlighted warning explainability as one of the major issues of static analysis tools. Fourteen of the twenty developers interviewed by Johnson et al. [47] reveal that poorly presented output is one of the main reasons for tool underuse. With a study on Microsoft developers, Christakis et al. [22] show that the second most popular pain point of static analyzers is the bad warning messages, the lack of fix suggestions coming fifth on the list of 15 pain points. In a study at Google, Lewis et al. [65] cite "obvious reasoning" as a desirable feature of an analysis tool, where they advocate for the analysis tool to provide clear proof of why a warning is reported. In a study of Fortify SCA [37], Ayewah et al. [10] find that badly explained traces

harm the understanding of the bug and the confidence of the developer in the tool. Those studies motivate the need for better developer support in understanding analysis warnings.

Similarly, for warning understanding, current approaches base themselves on external information. Phang et al. [105] only focus on the code when visualizing warning traces. Nanda et al. [85] visualize warnings based on warning and code information, like modern commercial tools such as Checkmarx [20] or CodeSonar [41]. With our approach, we argue that internal analysis-specific information would also give more insight into the warning.

Approaches for classifying warnings—in particular, between true and false positives—mainly use external information. Sherriff et al. [124] compute alert signatures based on common semantic errors in the source code and use them to identify false positives. Shen et al. [123] calculate the likelihood of bug categories in FindBugs to be false positives. Heckman et al. [44] use the history of the warning to detect false positives. Kremenek et al. focus on warning correlation and code locality [58] or code semantics [59], while Ruthruff et al. [115] use warning types, code location, and code history. While some approaches may integrate semantic information on the analyzed code that aim for known analysis weaknesses (e.g., arrays), no approach we know of proposes the direct use of the analysis rules for **T1**–**T4**.

7.2.2 Usage of Internal Analysis Rules

The idea of integrating analysis-specific knowledge to assist developers has been mentioned by Jung et al. [48]. They use a statistical analyzer to help triage false positives from true positives, based on "symptoms". In their conclusion, they advance the idea that using the analysis' weaknesses as symptoms would yield better results than external symptoms such as coding features for example. Past work on nano-patterns has proven data-flow and control-flow features useful to characterize Java methods [125] and detect software vulnerabilities [21, 28, 131], but those features are not used for warning classification, understanding, and analysis adjustment. In our approach, we use internal information to classify true and false positives in static analysis, aid developer understand and fix warnings, and help them identify analysis weaknesses.

7.3 Rule Graphs

To address **T1**–**T4**, we propose the concept of *rule graphs* that exposes the analysis' internal information to the end-user.

7.3.1 Definition

Let us consider the running example of Listing 7.1 and a *taint analysis* that over-approximates array elements to the entire array. The analysis would report the two SQL injections and the open redirect by tracking the variables containing the tainted data from the source *getParameter*() to the sinks *executeQuery*() and *sendRedirect*(). Figure 7.1 shows the three rule graphs representing each of the analysis warnings. For example, following the edges of the middle graph (open redirect), we see that the data is first assigned to *userId* at line 71, then to *s* at line 72 (because of the over-approximation of *s*[0] to *s*), and finally to *url* at line 79 (over-approximation of *s*[1] to *s*) before being reported at line 80. In addition to the code-related information found in the edge labels, the nodes encode the internal analysis rules. For the open redirect, the root node is the source, creating the taint to *userId*. The taint is then assigned to *s* because it is an array, and then transferred to the local variable *url* by a part of the analysis which handles locals and is different from the one which handles array assignments. Sinks are handled separately, which
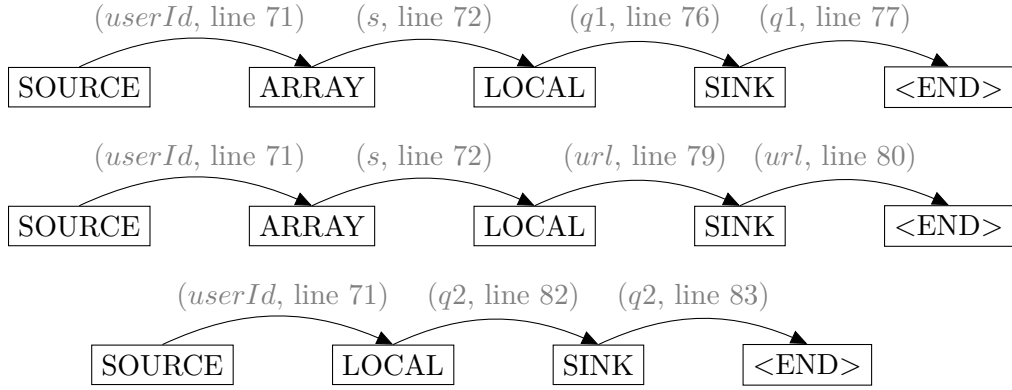
Figure 7.1: Simplified rule graphs for the three vulnerabilities in Listing 7.1: SQL injection line 77 (top), open redirect (middle), SQL injection line 83 (bottom).

is represented by the SINK node. The warning stops at an artificial node <END>, introduced so that its edge from the sink stores the last step of the trace.

The nodes of the rule graphs mark the different rules of the analysis that handle the different constructs of the analyzed code, as highlighted in gray in the flow function of our taint analysis, in Algorithm 3. For a given statement s and a set of variables tainted before the statement in, it generates, transfers, and kills taints, yielding an updated set of tainted variables after the statement: out. Five analysis rules are encoded in this flow function: the generation of taints when a source is detected (denoted with the SOURCE marker, line 6), the taint transfers for arrays (ARRAY, line 11) and local variables (LOCAL, line 13), the taint transfer and reporting at sinks (SINK, line 15), and the transfer of existing taints (ID, line 3). The graph nodes thus correspond to *rule markers* and give insight into *how* the analysis interprets the code. Note that we have removed the ID nodes from Figure 7.1 to simplify the example.

We define a rule graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ as a set of nodes $\mathcal{V}$ (the set of markers in the analysis) and edges $\mathcal{E}$, which can carry location labels.

Size of the rule graphs: Depending on which rules need to be tracked, different markers can be chosen. As a result, rule graphs can approximate a traditional warning trace, or be vastly different. The flow function shown in Algorithm 3 is a minimal example working for Listing 7.1. Real-world analyses have more rules, and more complex rules, to handle aliasing, multithreading, sanitizers, etc. As a result, rule graphs can become quite large, but remain at worst in the size range of traditional warning traces ($\mathcal{O}(tm)$ with t the size of the traditional trace, and m the number of markers).

Edge labels are only needed for **T1** and **T4**, where code-specific information is needed. Otherwise, code location is not needed for grouping warnings based on internal information (**T2**) and determining weak analysis patterns (**T3**), since this information is completely analysis-specific. We show this in our evaluation in Section 7.6.4. As a result, the edge labels (in gray in Figure 7.1) can be dropped, making the rule graphs much smaller and simpler to handle. In the example Figure 7.1, the top two graphs would then become identical.

7.3.2 Generating Rule Graphs

We formalize how rule graphs can be obtained from a data-flow analysis expressed in the monotone framework [50]. This is done in two phases: when the analysis runs, it stores rule marker information, and after it terminates, it uses it to extract the graph.

Algorithm 3 Flow function for a taint analysis for Listing 7.1, with an over-approximation of arrays at line 11. The marker information is shown in gray.

 1: **procedure** $f_{statement}(\langle in \rangle)$
 2:     $out := \emptyset$
 3:     **for** d in in **do**
 4:         $d' := newDataFlowFact(d, statement)$
 5:         $out = out \cup (d', (\{d\}, \mathrm{ID}))$
 6:     **if** $isAssignStatement(statement) \wedge rightIsSource(statement)$ **then**
 7:         $d' := newDataFlowFact(leftSide(statement), statement)$
 8:         $out = out \cup (d', (\{\langle \mathrm{ZERO} \rangle\}, \mathrm{SOURCE}))$
 9:     **if** $isAssignStatement(statement) \wedge (rightSide(statement) \cap in \neq \emptyset)$ **then**
10:         $d' := newDataFlowFact(leftSide(statement), statement)$
11:         **if** $leftIsArray(statement)$ **then**
12:             $out = out \cup (d', (\{d\}, \mathrm{ARRAY}))$
13:         **else**
14:             $out = out \cup (d', (\{d\}, \mathrm{LOCAL}))$
15:     **if** $isSink(statement) \wedge sinkParameter(statement) \in in$ **then**
16:         $d' := newDataFlowFact(sinkParameter(statement), statement)$
17:         $out = out \cup (d', (\{d\}, \mathrm{SINK}))$
18:         $report(d')$
19:     **return** out

Collecting Rule Marker Information

Algorithm 4 presents the traditional fixed-point iteration algorithm of data-flow analyses that applies the flow function f_s to the statements of the program until the out-sets stabilize, and the modifications made to support rule graphs. The main change is the introduction of rule marker and predecessor information, highlighted in gray. As shown in Figure 7.2, the modified flow function does not only report data-flow facts (tainted variables in terms of taint analysis) in its out-set: it encapsulates each data-flow fact with *marker information* (*mi*) containing the data-flow fact's predecessors and a rule marker explaining the reason why the data-flow information was transferred from the predecessors to the current fact. For example, in Algorithm 3, the rule at line 11 states that if the right side of an assignment statement is tainted (if the variable is in the *in* set), the array on the left side of the assignment should be tainted. As a result, at line 12, d', the data-flow fact representing the left side of the assignment is marked with its predecessor d and the rule marker ARRAY. This notation means that the taint from d is transferred to d' at statement s because of an assignment to an array. The merge operator $\sqcap$ (line 6) should also be adapted to handle marker information.

Soundness and termination: The modifications added to the analysis do not affect its termination and soundness since the marker information piggybacks on the data-flow facts without influencing them. Thus, Algorithm 4 is as sound as the original analysis and terminates in as many iterations.

Extracting Rule Graphs

Once the marker information is computed, we run Algorithm 5 for each warning, to retrieve its rule graph. The algorithm implements a modified depth-first search (DFS). It recursively recon-

---

Algorithm 4 Fixed-point algorithm for a data-flow analysis. The modifications to support rule graphs are shown in gray.

1: **procedure** ANALYZE
2:     $wl := entrypoints()$
3:     **while** $wl \neq \emptyset$ **do**
4:         pop *statement* off *wl*
5:         OLD := { $d \in$ OUT[*statement*] }
6:         IN[*statement*] := $\sqcap'$ { $(d, mi) \in$ OUT[r] | $r \in predecessors(statement)$ }
7:         OUT[*statement*] := $f'_{statement}$(IN[*statement*])
8:         NEW := { $d \in$ OUT[*statement*] }
9:         **if** OLD $\neq$ NEW **then**
10:             $wl \cup = successors(statement)$

$f'_{statement}$ is a modified flow function, defined in Figure 7.2.
$\sqcap'$ is the merge function, adapted to handle marker information.

$$f'_{statement}(in) = f'_{statement}(\{(d_1, mi_1), ..., (d_n, mi_n)\})$$
$$= \{(e_1, ni_1), ..., (e_m, ni_m)\}$$

with $d_i \in D$,

$e_i \in D$ such that $f_{statement}(d_1, ..., d_n) = \{e_1, ..., e_m\}$,

$f_{statement}$ the original flow function,

$mi_i = (D_i, rule\_marker_i)$ such that $D_i \subseteq D$,

$ni_i = (D_i, rule\_marker_i)$ such that $D_i \subseteq \{d_1, ..., d_n\}$.

Figure 7.2: The modified flow function.

structs the graph from the <END> node to the different sources, using predecessor information for the edges, and rule markers for the nodes. The stopping condition (line 2) holds if a source is reached when invalid paths are visited (where returns do not match calls), or when the DFS runs into a loop (causing unnecessarily complex patterns). In the latter cases, we then clean up the graph, removing data from the invalid path.

Soundness and termination: With a DFS, the algorithm steps back into all possible traces tracked by the marker information, including invalid ones. However, as long as the analysis' merge operator keeps the correct predecessors in the marker information, the DFS finds all correct traces, along with potential false positives. Since the number of generated data-flow facts is finite and the stopping condition keeps track of loops, the algorithm must terminate. Retrieving all paths between two nodes with a DFS has a complexity of $\mathcal{O}(se)$ with s the number of nodes and e the number of edges, so we apply optimizations to reduce the number of explored paths, which we detail in Section 7.5.

7.4   Applications of Rule Graphs

In this section, we discuss how to use rule graphs for the four tasks presented in Section 7.1.

Algorithm 5 Graph reconstruction.

1: **procedure** WALKBACK(⟨*df, next_rule_marker, a*⟩)
2: **if** *stoppingCondition()* **then**
3:    *pruneInfeasiblePaths(a)* **return**
4: *s := getStatement(df)*
5: $(D', rule\_marker) := mi \in \text{OUT}[s] \mid d = df$
6: **for** *pred* in D' **do**
7:    *a.addEdge(rule_marker, next_rule_marker, df)*
8:    *walkBack(pred, rule_marker, a)*

7.4.1 Warning Understandability

The internal rule information contained in the rule graphs can be used to enhance the explainability of analysis warnings (**T1**). Displaying rule information to the developer can help them better understand how the analysis handles the analyzed code from one line of code to the next. With a rule graph, the analysis tool can thus provide a detailed analysis trace from the source to the sink that explains the analysis' reasoning step by step. Each edge e represents a step of the trace from the point of view of the analysis, and gives access to the following step information:

- The data-flow fact of interest: in e's label.

- Why it is of interest to the analysis: e's origin node.

- The location of the step in the code: in e's label.

- The next step: the edges departing from e's destination node.

- The previous step: the edges arriving at e's origin node.

Looking up the first three points has a complexity of $\mathcal{O}(1)$. The lookup of the last two points has a worst-case complexity of $\mathcal{O}(n)$, with n the number of edges in the graph.

For the example in Listing 7.1, the enhanced information about the open redirect can be read from the middle graph and reported to the developer as follows:

- **line 71**: SOURCE statement: *userId* is tainted.

- **line 72**: Assignment to an ARRAY: s is tainted from *userId*.

- **line 79**: Assignment to a LOCAL: *url* is tainted from s.

- **line 80**: SINK statement: *url* is reported.

Unlike traditional traces, this gives developers insight into the inner-workings of the analysis and allows them to pinpoint points of uncertainty: here for example, the over-approximation of the array assignment. Note that this analysis trace approximates a traditional trace, but more targeted sets of markers can be used, reporting only on the SOURCE, SINK, and ARRAY markers if LOCAL is of no interest.

7.4.2 Warning Classification

Rule graphs can also be used to *classify* warnings (**T2**). For example, when grouping warnings by *confidence*, analysis weaknesses such as array assignments can denote a lower confidence.

Another example consists in using rule graph similarity to group warnings by potential fix (thus grouping the open redirect and the SQL injection line 77 together in Listing 7.1).

Classifying warnings based on how the analysis computes them is equivalent to classifying warnings based on the different combinations of either the graph nodes or its unlabeled edges (a combination of two nodes). Given a *training set* of classified warnings, it is thus possible to predict in which class another warning is likely to belong, based on which nodes (or edges) the graph contains. We achieve this prediction by applying supervised machine learning to the training set, using either the nodes or the unlabeled edges of a rule graph as features. Let us consider the example Figure 7.1, with the learning features being graph nodes, the classes being *true positive* and *false positive*, the training set being composed of the open redirect (*false positive*) and the SQL injection line 83 (*true positive*), and the test set being composed of the SQL injection line 77. From the training set, the features SOURCE, LOCAL, SINK, and <END> are found in both true and false positives. The feature ARRAY is only found in the false positive. As a result, the classifier uses the ARRAY feature to differentiate between true and false positives. Since the SQL injection line 77 contains that feature, it is classified as a *false positive*. Using edges instead of nodes, three features can be used to determine the same choice: SOURCE → ARRAY, ARRAY → LOCAL, and SOURCE → LOCAL. We note the importance of the quality of the training set in machine learning: using a different one could result in a different or wrong classification. We discuss this further in Section 7.6.4.

7.4.3   Identification of Weak Analysis Patterns

For **T1**, we expose the analysis' internals to the developer, allowing them to pinpoint its shortcomings. We can take that a step further and semi-automate the detection of patterns in the analysis rules that can lead to wrong results (e.g., over-approximation of arrays in Listing 7.1).

Given a *training set* of labeled true and false positives, we can use their rule graphs to find the most likely causes for the false positives and thus detect weak analysis patterns (**T3**). Those patterns are combinations of analysis rules that lead to false positives. In terms of rule graphs, those are subsets of the edges (or the nodes). Weak patterns are retrieved from a machine-learning classifier, such as presented in Section 7.4.2, initialized with a training set labeled with the classes *false positive* and *true positive*. Once the classifier learns which combinations of features are more likely to lead to false positives, we use its decision rules to determine weak patterns. This method requires the classifier to be a rule-based classifier, such as a decision tree.

Following the example of Figure 7.1 in Section 7.4.2, a rule-based classifier would learn that the presence of the node ARRAY leads to false positives, thus bringing attention to the weak ARRAY rule from Algorithm 7.2. In the case of unlabeled edges, the classifier can choose any of the three edges, either pointing us to the ARRAY rule or to its absence. We discuss the use of nodes compared to edges in Section 7.6.4.

7.4.4   Identification of Missing Analysis Patterns

As mentioned in **T4**, another cause for wrong analysis results is missing analysis rules (e.g., assignments to field variables are ignored by the flow function in Listing 7.1). Missing patterns not handled by the analysis are difficult to identify automatically, as a missing analysis rule could depend on anything. In that case, it is more efficient to show a potentially problematic warning to the developer in a fashion similar to **T1**, so they can manually determine where the analysis does not behave as expected. We can use rule graphs to pick good candidates to show the developer. For example, if two rule graphs are similar (showing that the analysis handles the corresponding warnings similarly), but one is a false positive and one is a true positive, the

analysis could be missing rules to properly handle the false positive, and the difference between the two warnings may indicate what that missing rule could be.

To calculate the similarity between two warnings, we define the following *similarity coefficient* based on the number of edges their rule graphs have in common. The greater the coefficient, the more similar the warnings.

$$similarity(\mathcal{G}_1, \mathcal{G}_2) = \frac{1}{2} \cdot \left(\frac{\#(e_{\mathcal{G}_1} \cap e_{\mathcal{G}_2})}{\#e_{\mathcal{G}_1} + \#e_{\mathcal{G}_2}} + \frac{\#(el_{\mathcal{G}_1} \cap el_{\mathcal{G}_2})}{\#el_{\mathcal{G}_1} + \#el_{\mathcal{G}_2}} \right)$$

with $\mathcal{G}_i$ the warning graph, $e_{\mathcal{G}}$ the set of unique edges in $\mathcal{G}$ ignoring the edge labels, and $el_{\mathcal{G}}$ the set of unique edges in $\mathcal{G}$ taking the edge labels into account. The first part of the equation encodes the analysis' treatment of the warning regardless of the particular lines of code (edge labels). The second part is used when two warnings are semantically similar: the coefficient differentiates them by code location, using the edge labels. The $1/2$ normalizes the coefficient to a maximum value of 1. Applied to Figure 7.1, we see that the warnings that are most similar are the SQL injection line 77 and the open redirect, with a similarity coefficient of 0.75 (all eight edges in common, four labeled edges in common), compared to 0.14 for the other two relationships (two of seven edges in common, no labeled edges in common).

7.5 Implementation Details

We now detail the implementation of the rule graphs and of the modules that use them for **T1**–**T4**: the classification module of **T2**, the pattern detection modules of **T3** and **T4**, and for **T1**, MUDARRI, an IntelliJ plugin that displays warning traces augmented with marker information. The source code is available online [86].

7.5.1 Rule Graphs

We have implemented the rule graphs on top of a taint analysis for Java and Android applications in the Soot-based [139, 140] IFDS [110] solver HEROS [15]. Its flow functions are detailed in Section 2.3. To compute the rule marker information (Section 7.3.2), we use a modified version of the path-reconstruction approach from FlowTwist [63]: we encapsulate the data-flow facts with additional information containing the original source statement of the current data-flow fact, the data-flow fact's predecessors and their corresponding rule markers, and the data-flow fact's neighbors, used for trace reconstruction. We also adapt the analysis solver to propagate data-flow facts along neighbors.

Over the 650 LOC of our analysis' flow functions, we use a total of 39 rule markers. Thirteen are related to calls to library APIs, eight are used for taint transfers at call and return sites, six detail the different types of taint transfer at assignment statements, three refer to sources and sinks, eight cover different manipulations of aliases, and one is a marker used for the trace lookup algorithm. To provide more or less granularity in the warning explanations, we do not always generate exactly one level of markers per call to the flow function, as shown in Section 7.3.2. When rules are not of interest, we do not generate new data-flow facts. When we need more detailed explanations, we generate a chain of data-flow facts in the same flow function. For example, if a variable is tainted at a source and that variable is already aliased to another variable, a chain of two data-flow facts is produced at once: SOURCE and ALIAS. The longest chain in our analysis is of length 4.

For scalability, we limit the graph extraction (Section 7.3.2) to one trace per warning. We also reduce the number of visited paths by checking for loops and invalid paths (i.e., paths with non-matching call stacks), and by using FlowTwist's neighbor system.
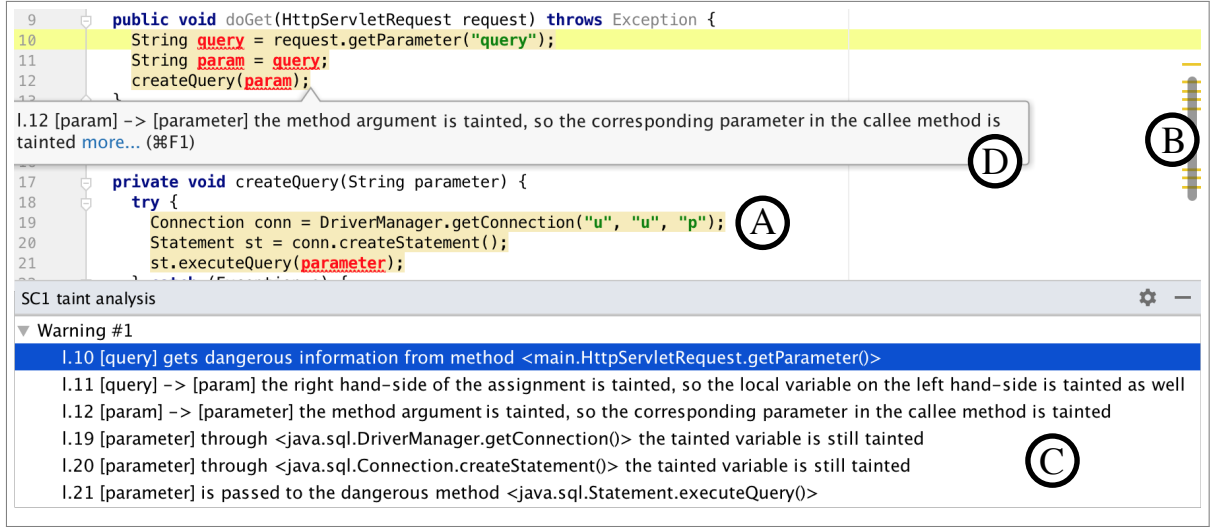
```
 9  ⊖    public void doGet(HttpServletRequest request) throws Exception {
10           String query = request.getParameter("query");
11           String param = query;
12           createQuery(param);
```

l.12 [param] –> [parameter] the method argument is tainted, so the corresponding parameter in the callee method is tainted more... (⌘F1)

```
17  ⊖    private void createQuery(String parameter) {
18  ⊖       try {
19              Connection conn = DriverManager.getConnection("u", "u", "p");
20              Statement st = conn.createStatement();
21              st.executeQuery(parameter);
```

Ⓐ Ⓑ Ⓓ

SC1 taint analysis   ⚙ —

▼ Warning #1

    l.10 [query] gets dangerous information from method <main.HttpServletRequest.getParameter()>

    l.11 [query] –> [param] the right hand–side of the assignment is tainted, so the local variable on the left hand–side is tainted as well

    l.12 [param] –> [parameter] the method argument is tainted, so the corresponding parameter in the callee method is tainted

    l.19 [parameter] through <java.sql.DriverManager.getConnection()> the tainted variable is still tainted

    l.20 [parameter] through <java.sql.Connection.createStatement()> the tainted variable is still tainted

    l.21 [parameter] is passed to the dangerous method <java.sql.Statement.executeQuery()>

Ⓒ

Figure 7.3: The GUI of MUDARRI. Features Ⓐ–Ⓓ are detailed in Section 7.5.

7.5.2 Graphical User Interface

The taint analysis can run as a standalone command-line tool, but we have also integrated it in an IntelliJ plugin [46] we name MUDARRI, to illustrate **T1**. Figure 7.3 shows MUDARRI's Graphical User Interface (GUI). The code contains an SQL injection highlighted in the editor (Ⓐ) and in the right gutter (Ⓑ). The bottom view (Ⓒ) details the warnings found in the application. For each step of the warning, it provides the line of code, the tainted variable(s), and an explanation on the rule marker. For example, the first line in the view shows that the variable *query* is tainted because of the source method *getParameter*, and the second line explains that the taint is transferred from *query* to *param* because the tainted *query* is assigned to *param*. Selecting a warning in the bottom view highlights the corresponding lines of code in the editor and marks the tainted variables in a bold red font. Those details also appear in a tooltip (Ⓓ) when hovering over a highlighted line of code. Double-clicking on a step in the bottom view opens the corresponding file in the editor.

MUDARRI addresses **RE-S4** by improving the explainability of the analysis warnings through explanations of the bug and how it is detected in the code (**F3**–**F5**), and visualizing the bug in the interface (**F10**).

7.5.3 Offline Functionalities

Given a training set of labeled warnings, the warning classification module predicts which class an unlabeled warning belongs to (**T2**). From a set of warnings marked as true or false positives, the pattern detection module retrieves weak analysis rules from the rules of a rule-based classifier, as shown in Section 7.4.3 (**T3**). This module can also compute the similarity coefficient between warning pairs and yield a list of similar warnings to explore, allowing the user to identify missing analysis rules as in Section 7.4.4 (**T4**).

The machine learning modules used for **T2**–**T3** are implemented using the WEKA machine learning framework [147]. Both graph nodes and edges can be used as learning features, and their presence or absence in a particular rule graph provides binary inputs to help classify the graph. We discuss the particular classifiers and the choice of nodes or edges in Section 7.6.2. Since the warning classification module and the pattern detection module depend on machine learning, they run into the scalability issues known to that domain [137]. As a result, a full

learning/classification cycle cannot be used at runtime in the developer's Integrated Development Environment (IDE). Instead, we run them offline, as part of post-processing modules after the analysis terminates.

By allowing developers to adjust the analysis rules with the offline modules, we encourage the integration of developer knowledge in the analysis rules (**RE-S3**) through **F16** (configurability of the analysis) and **F18** (customization of the analysis rules).

7.6 Evaluation

In this section, we evaluate the use of rule graphs for **T1**–**T4**. We test warning understandability (**T1**) through a user study with 22 participants, in which we compare Mudarri to NoMarkers, a modified version of Mudarri that does not contain analysis information. The two tools are visually identical, except for the details of the trace in the bottom view: NoMarkers also contains the tainted variables and the lines of code, but the rule marker information is replaced by the Java statement at that line of code, showing a traditional trace.

For the other three tasks (**T2**–**T4**), we conduct an empirical evaluation over the warnings yielded by a taint analysis on 1098 real-life Android applications from F-Droid [35]. Like for the user study, we use the taint analysis defined in Section 2.3. For **T2**, we assume the use of the classification *true / false positives*, meaning that we evaluate how well rule graphs can be used to distinguish true positives from false positives.

Thus, we ask the following research questions:

- **RQ20**: Can rule graphs help software developers understand warnings?

- **RQ21**: Can rule graphs help software developers classify warnings?

- **RQ22**: Can rule graphs help software developers find weak analysis rule patterns?

- **RQ23**: Can rule graphs help software developers find missing analysis rule patterns?

7.6.1 User Study: Warning Understandability

Through a user study, we evaluate how internal analysis information assists the developer in understanding warnings (**RQ20**). We ran a comparative, within-subjects user study between Mudarri and NoMarkers with 22 participants, referred to as **P1**–**P22** (13 students, 9 researchers). Of all participants, 13.6% have between one and two years of experience as professional software developers, 9.1% of the have more than five years of experience, 22.7% have between three and five years of experience, 13.6% have between two and three years of experience, and 40.9%, a year or under.

The participants were given a 15-minutes task: to go through a list of data leaks reported by the taint analysis, decide if they are true positives or false positives, and explain their reasoning. In those tasks, false positives are due to weak or missing analysis rules, so we verified if they have understood a warning if they could name the source and the sink of the data leak, and correctly detail the taint transfers for each step between the source and the sink.

The participants performed the task twice: once with Mudarri, and once with NoMarkers. A latin-square design was used to counter the learning effects: half of the participants used Mudarri first, the other half used NoMarkers first. For the two tasks, we used two real-world Android applications from F-Droid: Balance [33] and Sparkleshare [36] containing respectively 8 and 16 data leaks over their respective 1,000 and 1,700 LOC. All participants performed the first task with Balance and the second with Sparkleshare.
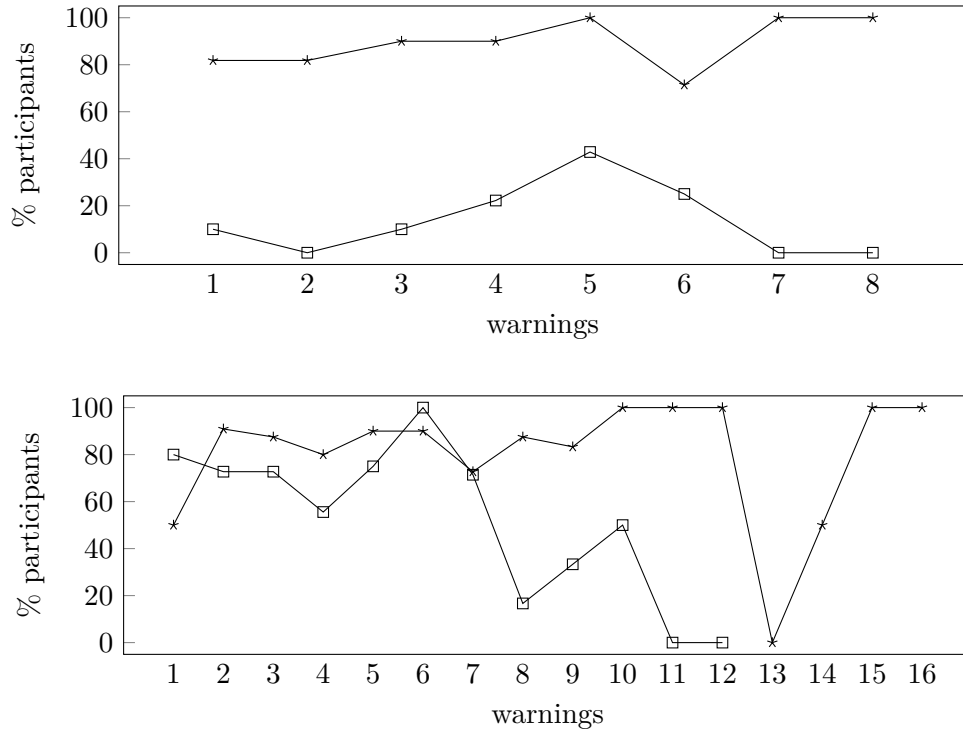
Figure 7.4: Percentage of participants correctly understanding the 8 warnings of Balance (top) and the 16 warnings of Sparkleshare (bottom) with Mudarri ($\rightarrow\!\!*\!\!-$) and NoMarkers ($-\square-$).

During the tasks, we asked the participants to think aloud, allowing us to determine which warnings they understood correctly. We also obtained information about the features of Mudarri and NoMarkers that were perceived as most or least useful. After each task, the participants also graded the usefulness of the tool they used on a Likert scale from 0 to 10.

The study results and test applications are available online [86].

7.6.2 Study Results

Figure 7.4 presents the percentage of participants who could correctly explain each warning. For the first task, we see that with Mudarri, participants understood Balance's warnings better than with NoMarkers: on average, 86.67% of the participants understood the warnings correctly using Mudarri, but only 14.81% using NoMarkers. This is mostly due to Mudarri's clarification of the source and sink methods (**P12** "[With NoMarkers] I don't know if it is a true sink") and the taint transfers in cases of complex code constructs. For example, the first warning in Balance had an Intent constructor as a source, which was not explicitly stated in No-Markers. Since the constructor does not look like a typical source method, most participants overlooked it and misidentified the source for another method. Mudarri explicitly names the source method, allowing participants to easily identify it. In another example, a wrong analysis rule mistakenly transferred a taint in a listener object. Most Mudarri users immediately identified the problem, because Mudarri reported a taint transfer due to an assignment on a line with no assignment. NoMarkers users spent more time speculating on how the taint entered the listener.

For the second task, the warnings of Sparkleshare were of two kinds: warnings 1 to 7 were similar to a few warnings already seen in Balance, and warnings 8 to 16 were new types of warnings. We see that NoMarkers users perform much better on warnings they are familiar

with (74.60%) than not (26.67%), while Mudarri users perform equally well over both groups (respectively 82.81% and 82.61%). With new warnings, we again see the general trend of Mudarri helping participants understand warnings better than NoMarkers. For warnings 1 to 7, the similar performance of both participant groups illustrates a learning effect: NoMarkers users had used Mudarri for their first task, and the knowledge gained then allowed them to perform almost as well as Mudarri users for warnings that they had already seen before (**P8** "If I didn't [already] know what it was, I could not find out what it is about"). With new types of warnings, their performance decreased, showing that they were not given the necessary knowledge to understand and assess them. Over both tasks, a two-tailed Wilcoxon Rank-Sum test shows that the participants understand significantly more warnings with Mudarri ($p = 0.00008 < 0.05$).

Mudarri's NPS score is of 13.6, denoting a positive score. In comparison, NoMarkers received a score of -50. On the NPS scale from 0 to 10, the participants gave Mudarri an average grade of 7.48, and NoMarkers, 6.01. We attribute the grades' closeness to the tools' visual similarity. Mudarri and NoMarkers were designed to be visually identical. In particular, participants noted the highlighting and navigation capabilities of both tools as very useful. Since participants were not told when they interpreted the warnings incorrectly, their impression of the two tools depended entirely on the GUI. Based on the grades, 16 of the 22 participants preferred Mudarri, which we attribute to the analysis details provided by the tool (**P6** "I can recognize the problem quickly", **P15** "It helps me know why the tool thinks that"). Despite their better performance when using Mudarri, three participants preferred NoMarkers. We attribute this discrepancy to the large amount of text in Mudarri, which can be time-consuming and tedious to read, especially when the participant already knows how the warning works (**P13** "There is too much information [...] it breaks my workflow.").

Answering **RQ20**, we can conclude that analysis-based information helps developers understand warnings significantly better than code-based traces for warnings that they have not seen before. It also helps build up a reusable knowledge of the tool and warning types.

7.6.3 Empirical Evaluation: Warning Classification and Pattern Detection

In an empirical evaluation, we evaluate the classification and the pattern extraction modules, using the classes *true / false positive* to answer **RQ21**–**RQ23**.

As a training set, we use DroidBench [9], an open-source benchmark for taint analysis in Android applications. Our taint analysis finds 202 warnings in DroidBench, 14 of which are false positives detailed in Table 7.1. We refer to them as **FP1**–**FP14**. **FP1**–**FP5** are caused by weak analysis rules, **FP6**–**FP14** are due to missing analysis rules. Our test set consists of 1098 real-world applications from F-Droid [35]. Our ground truth consists in 200 warnings we manually classified out of the 11,148 found by our taint analysis, 54 of which were classified as false positives. For the sake of representativeness, the 200 warnings were selected to ensure a complete coverage of all existing rule graph nodes and unlabeled edges.

The test applications are available online [86]. We ran our experiments on a 64-bit 10.13 Mac OS X laptop with an Intel Core i5 2,9 GHz CPU running Java 1.8, with a Java heap space of 8 GB.

7.6.4 Evaluation Results

Warning Classification (RQ21)

To test the usefulness of rule graphs in classifying true and false positives, we use eight different classifiers from WEKA. Four are rule-based (J48, DecisionTable, RandomForest, and JRip), as required by **T3** to extract rule patterns. For the sake of comparison, we add probabilistic

Table 7.1: List of false positives reported by the taint analysis in DroidBench, and their root causes. **FP1**–**FP5** are caused by weak analysis rules, **FP6**–**FP14**, by missing analysis rules.

ID	Test case	Cause
FP1	ArrayAccess2	Array over-approximation
FP2	ArrayAccess5	Array over-approximation
FP3	IntentSink2	Android lifecycle not modeled correctly
FP4	IntentSink2	Android lifecycle not modeled correctly
FP5	Merge1	Incomplete handling of aliases
FP6	Button2	Type of callback not handled
FP7	Exceptions3	Type of exception not handled
FP8	Exceptions7	Type of exception not handled
FP9	HashMapAccess1	Collection over-approximation
FP10	SimpleUnreachable1	No check for unreachable code
FP11	UnreachableBoth	No check for unreachable code
FP12	UnreachableSource1	No check for unreachable code
FP13	UnreachableSink1	No check for unreachable code
FP14	Unregister1	Callback behavior not handled

classifiers (NaiveBayes and BayesNet) and function-based classifiers (SMO and Logistic, using WEKA's default hyper-parameters). Table 7.2 presents the precision and recall of each classifier for a ten-fold cross validation on the DroidBench training set. The table reports the median precision and recall over ten runs for two types of machine learning features: rule graph nodes and unlabeled edges. Similarly, Table 7.3 shows the precision and recall when running the full classification on the test set of 200 F-Droid warnings.

On the cross validation, the precision and recall do not vary significantly between the different classifiers. When using nodes, we reach a median precision of 0.876 and recall of 0.912. With edges, the median precision and recall are comparable: respectively 0.876 and 0.887. However, for the full classification, we observe a drop in precision when using edges: the median precision and recall are of respectively 0.712 and 0.733 for nodes, and respectively 0.648 and 0.696 for edges. This drop in precision is attributed to the low number of false positives in DroidBench causing overfitting in the classification: the additional information carried by the edges makes

Table 7.2: Precision and recall when using graph nodes or unlabeled edges as features, for the 10-fold cross validation on DroidBench.

	Nodes		Edges	
	Precision	Recall	Precision	Recall
J48	0.865	0.916	0.865	0.906
JRip	0.866	**0.926**	0.866	**0.926**
DecisionTable	0.866	0.921	0.865	0.916
RandomForest	**0.902**	**0.926**	0.866	**0.926**
SMO	0.866	0.931	0.887	0.921
Logistic	0.888	0.906	**0.894**	0.876
NaiveBayes	0.881	0.881	0.879	0.871
BayesNet	0.882	0.886	0.885	0.861

Table 7.3: Precision and recall when using graph nodes or unlabeled edges as features, on the F-Droid applications.

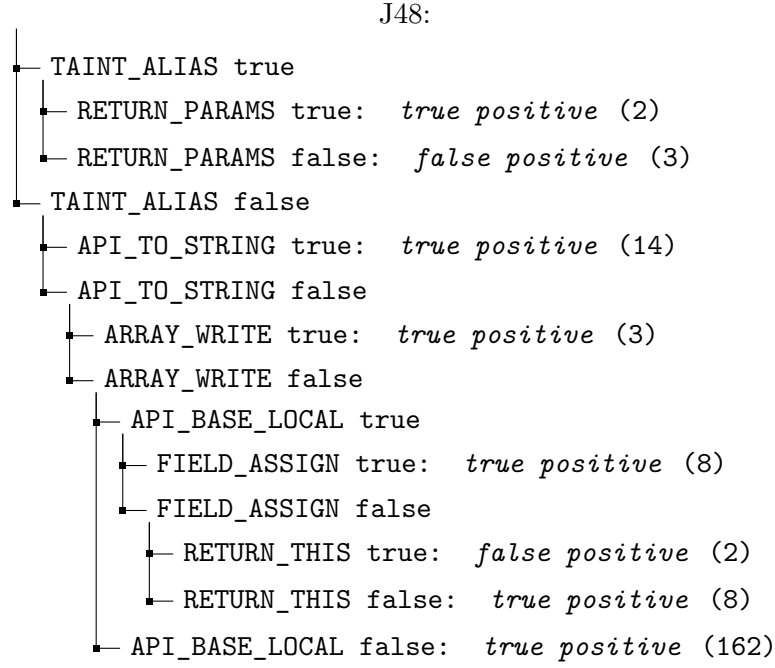|	Nodes		Edges	
	Precision	*Recall*	*Precision*	*Recall*
J48	**0.774**	**0.790**	0.678	0.685
JRip	0.737	0.760	0.712	0.740
DecisionTable	0.526	0.725	0.526	0.725
RandomForest	0.758	0.765	0.525	0.720
SMO	0.710	0.730	0.714	**0.745**
Logistic	0.743	0.765	0.615	0.645
NaiveBayes	0.735	0.680	0.693	0.675
BayesNet	0.715	0.655	**0.720**	0.630

the classification rules too specific to the particular use case of DroidBench, and causes the precision to drop on test sets which false positives do not exactly match DroidBench's.

The precision drop may be avoided by introducing a training set with more varied false positives, or by using nodes as learning features. The latter option makes a more sensible choice. Not only does it compensate for the overfitting, but it also reduces the complexity of the classification: in a program with v variables and l lines of code, there may be up to $s \times v \times l$ transitions, with s the number of states. The DroidBench warnings contain a total of 27 states and 146 unique unlabeled transitions.

Using nodes, we note that in the case of the cross validation, the precision and recall are similar over all classifiers ($0.014 \leq \sigma \leq 0.019$). But for the full classification, the precision and recall vary depending on the classifier ($0.078 \leq \sigma \leq 0.083$). The rule-based classifiers have a generally better precision and recall—with the exception of DecisionTable, making rule-based classifiers more suited to handle them. The particularly low precision of DecisionTable is explained by the low number of false positives in DroidBench, which generates a single imprecise rule: classify everything as a true positive. Although rule-based classifiers are more suited for handling rule graphs, it is necessary to have a good training set.

To discuss the limitations of the classification, let us consider notable misclassified warnings in DroidBench and F-Droid. Array and collection accesses are misclassified by all classifiers: because the analysis over-approximates arrays and collections and treats all such warnings the same, the classifier is unable to distinguish between a true and a false positive and puts them all in the same class. Similarly, mishandled exceptions (warnings that cannot happen because the correct exception is not triggered, but that the analysis reports because it does not distinguish between different types of exceptions) are also often misclassified. Those misclassifications show that despite yielding good precision and recall, analysis-specific information is not enough to accurately classify warnings. Additional features are needed to supplement the analysis' weaknesses, for example the specific exception type, which would help distinguish between the different classes.

We see that rule graphs can be used to classify warnings with good precision (~0.712) and recall (~0.733), and that using nodes rather than edges as learning features is more precise and scalable. The precision and recall of our approach can be improved with a training set containing varied true and false positives, or by using code-specific features on top of internal analysis information in the classification.

J48:

```
├── TAINT_ALIAS true
│   ├── RETURN_PARAMS true:   true positive (2)
│   └── RETURN_PARAMS false:  false positive (3)
├── TAINT_ALIAS false
    ├── API_TO_STRING true:   true positive (14)
    └── API_TO_STRING false
        ├── ARRAY_WRITE true:   true positive (3)
        └── ARRAY_WRITE false
            ├── API_BASE_LOCAL true
            │   ├── FIELD_ASSIGN true:   true positive (8)
            │   └── FIELD_ASSIGN false
            │       ├── RETURN_THIS true:   false positive (2)
            │       └── RETURN_THIS false:  true positive (8)
            └── API_BASE_LOCAL false:  true positive (162)
```

JRip:

$$ID \wedge TAINT\_ALIAS \wedge \neg RETURN\_PARAMS \implies \text{false positive (2)}$$
$$ID \wedge \neg LOCAL \wedge RETURN\_THIS \implies \text{false positive (2)}$$
$$\implies \text{true positive (198)}$$

Figure 7.5: Decision tree of J48 and rules of JRip on DroidBench, and the number of warnings matching each rule.

Identification of Weak Analysis Patterns (RQ22)

To evaluate the pattern detection module for the detection of weak analysis rules, let us consider the classification rules generated by the four rule-based classifiers used in Section 7.6.4 (J48, DecisionTable, RandomForest, and JRip), using nodes as features.

Figure 7.5 presents the decision tree created by J48. We note two main false positive patterns: first, when an alias is tainted, the corresponding warning is likely to be a false positive if the taint does not return to a caller through its parameter. This corresponds to **FP5**. Second, the conjunction of API_BASE_LOCAL (API calls tainting the base object if one of the parameters is tainted), and RETURN_THIS (tainting the base object of a caller if the *this* variable is tainted in the callee) is also a false positive pattern. This corresponds to **FP3** and **FP4** and happens because of the incomplete modeling of the Android lifecycle, which causes an entire Activity to be tainted if one of its static attributes are. For true positive patterns, we also see that warnings using the rule ARRAY_WRITE (corresponding to **FP1** and **FP2**) are considered true positives. This is due to DroidBench's five array test cases, three of which are true positives. Using true positive patterns in conjunction with false positive warnings matching those patterns allows us to locate the analysis' weaknesses. From the classifiers' rules and classification results, we can see which analysis patterns match the DroidBench false positives that are due to weak analysis rules: use of arrays, and mishandling aliases and Android Intents, which can then be used to fix the weak analysis rules.
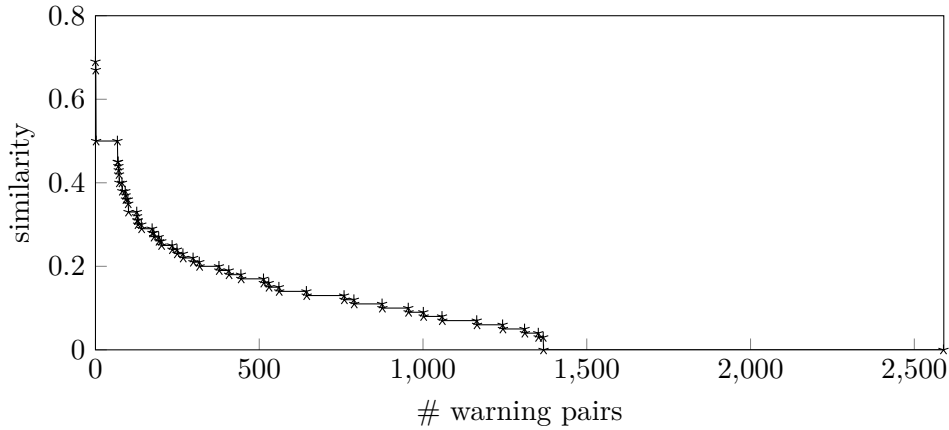
109

Figure 7.6: Distribution of the similarity coefficient between each pair of warnings.

```
85  try {
86    setImeiAsSourceAndCreateInvalidCastException();
87  } catch (ArrayIndexOutOfBoundsException ex) {
88    sms.sendTextMessage(num, null, imei, null, null);
89  }
```

Listing 7.2: Simplified source code of the DroidBench Exceptions7 test case. The Exceptions3 test case is similar, but has a RuntimeException at line 87.

Although JRip contains fewer rules than J48 (Figure 7.5), its rules still confirm the main rules from Figure 7.5: the combination of TAINT_ALIAS and ¬RETURN_PARAMETERS and the RETURN_THIS denoting false positives, while the rest is classified as true positives. We attribute the better performance of J48 to the more efficient pruning of JRip, which is implemented as part of RIPPER [24], an improvement of the REP method used by J48. Combined with the low number of false positives in the training set, JRip's more efficient pruning policy yields simpler rules. This effect is also observed with DecisionTable's unique rule, which classifies all warnings as true positives: since DecisionTable is a majority classifier, the false positives are considered as noise, and the resulting rule selects the majority class: true positives. RandomForest goes in the opposite direction— generating 100 trees of a larger size (from 29 leaf nodes to 69) most of which also contain the three rules discussed above, which we attribute to RandomForest's known overfitting behavior in the case of noisy datasets [119].

With this experiment, we see that classifier rules obtained from rule graphs can help determine weak analysis rules. In the case of DroidBench's weak analysis patterns, J48 provides the best tradeoff between pattern size and precision. Integrating a weak rule detection module in an analysis tool would help software developers determine which analysis rules should be adjusted when configuring or using the tool.

Identification of Missing Analysis Patterns (RQ23)

To assist developers in the detection of missing analysis rules, we generate pairs of similar true / false positives from the DroidBench training set, using the similarity coefficient. As mentioned in Section 7.4.4, the difference between very similar pairs can be indicative of missing rules. So, we point the developer towards the pairs with the highest similarity coefficient. We obtain a total of 2,590 pairs, with similarity coefficients ranging from 0 to 0.691 ($avg = 0.085$, $\sigma = 0.115$). Figure 7.6 shows the similarity coefficient for each of the pairs in the list. Only two pairs of

warnings have a coefficient higher than 0.5, 956 are at 0.1 or more, and 1,222 have a similarity coefficient of 0.

The top two pairs (with coefficients of 0.691 and 0.669) report on **FP6**. In this test case, the true positives share a large part of their traces with the false positive. The difference between the traces mainly consists in calls to the *dummyMain* method, which we use to model the Android lifecycle [9]. This observation reveals the missing analysis support for the particular type of callback found in the false positive. Two pairs concerning exception handling (**FP7** and **FP8**) with coefficients of 0.5 and 0.421, also provide ideal comparisons: the source code, illustrated in Listing 7.2, is almost identical but for the exception types. This similarity leads to almost identical rule graphs (with unlabeled edges), and allows the user to easily identify the missing rule: the analysis does not distinguish between different types of exceptions.

However, other pairs provide less explicit explanations. **FP10** and **FP12** are involved in 64 pairs, all with a similarity coefficient of 0.5. All of those warnings have a simple SOURCE → SINK graph. As a result, it is not easy to infer which rule is problematic since the true positives are so different from the false positives. Other pairs share this issue, namely those concerning **FP9**–**FP14**. Better pairs would require the true positive to be semantically close to the false positive (like in the exception examples), but DroidBench does not contain such test cases. A possible solution to this problem would be to add code-specific information in the similarity coefficient, to differentiate between two identical traces such as the SOURCE → SINK edges of **FP10** and **FP12**.

We see that rule graphs can be used to compute warning similarities and to point the developer towards missing analysis rules, helping them add missing rules in the analysis' ruleset. Code-specific features can also be used in the similarity computation to increase the precision of the approach.

7.7 Limitations and Threats to Validity

We ran the user study in a controlled environment in which participants only had 20 minutes to work on small-scale applications. While running the experiments in a controlled environment allowed us to remove external threats to validity, it would be interesting to also evaluate Mudarri in a real-life environment.

The user study was a within-subjects study, so we observed a learning effect in which participants performed better on the second task. We addressed this issue with a latin-square design, in which half of the participants used Mudarri first, and the other half, NoMarkers first, and reported on the aggregated results of both halves.

It is subjective to decide whether a warning is a true or a false positive, because it depends on the rater's definition of a valid bug. To limit the subjectivity of the rater's judgement for the classification of DroidBench and F-Droid warnings, we defined a true positive as a pair of source-sink methods with a valid data flow between them. Subjective factors such as whether the leaked information contains sensitive data, or whether the warning has a high impact or not were thus kept out of the decision. Similarly, in the user study, we judged that a participant understood a warning only if they could explain the valid data flow between its source and sink, regardless of whether they marked it as a true or false positive.

As seen in Section 7.6.3, the use of analysis rules in classification and pattern extraction is not enough to completely distinguish all warning classes. External code and warning-specific features could improve the classification.

The quality of the patterns found for **T3** and **T4**—and therefore also the quality of the prediction algorithm in **T2**—heavily depend on the quality of the training set. Like all machine learning approaches, a bad training set introduces uncertainties and yields inaccurate results. In

our evaluation, we used DroidBench, a benchmark of minimal examples for taint analysis with both true and false positives as our training set. While not complete, this benchmark covers the most common use cases for Android. To apply our approach to other kinds of applications or different languages, it is important to define a good training set first.

A drawback of the methods for **T2** and **T3** is their scalability, which we mitigate by running them offline. It would be possible to incrementally apply those methods, to use them interactively in the IDE.

7.8 Summary

Developer support in analysis tools often focus on the task of fixing bugs. However, the tasks related to configuring the analysis are often overlooked. With the user in mind, we explore this use case, addressing requirements **RE-S3** and **RE-S4**, by proposing novel tool features for analysis configuration, thus illustrating the value of the user-centered process.

In this chapter, we present the concept of rule graphs, with the goal of using the internal rules of the analysis to assist developers in four specific tasks: understanding and classifying warnings, and identifying weak and missing analysis patterns. With our implementation of rule graphs and of the modules illustrating the tasks such as MUDARRI, we demonstrate how to apply the concept of rule graphs for taint analysis.

Through a user study with 22 developers and an empirical evaluation on 1098 real-world Android applications, we show that with more understandable reporting and assistance in classifying warnings and evaluating the analysis' rules, we can provide a better user experience of static analysis configuration to software developers. With this research, we advocate for a more transparent use of analysis rules in static analysis tools and encourage researchers to explore other applications of rule graphs.

8

Conclusion and Future Work

We now summarize the contributions made in this thesis and present avenues for future work opened with our research.

Looking at the design of tools for static analysis from the end-user's point of view, we conducted a survey in Chapter 3 to understand the motivations and needs of static analysis developers with regards to their coding environment. Through that survey, we determined that soundness and precision are the main goal of analysis developers, and identified the main causes for errors in static analysis code. We extracted nine design recommendations for building a coding environment for static analysis, such as clear visualizations, omniscient debugging, or breakpoints that can be used in both the analysis code and the analyzed code.

In Chapter 5, we apply the same user-centered methodology to a second user group: software developers. While many usability studies have been conducted on static analysis tools to find user-experience issues, we focus on developer motivations and strategies when using those tools. Through a survey, a study of analysis reports, and a small cognitive walkthrough, we discovered new aspects to designing static analysis tools, and identified eleven recommendations for designing and using such tools in practice. Our study, shows that developers make decisions focusing on time constraints, creating heuristics to help them decide which warnings to fix, and how. Those time constraints make responsiveness and the treatment of such heuristics (e.g., encourage and integrate the good ones, and discourage the bad ones) the main points of concern for the design of static analysis tools.

The main motivation for using tools for static analysis differs between the two user groups. Analysis developers, whose main goal is to ensure the correctness of their analysis, give a high importance to understanding what happens under the hood of the analysis. Features such as exposing internal analysis information and providing full control when navigating this information (with two sets of breakpoints for example) are of prime interest to analysis developers. On the other hand, the primary goal of a software developer is to fix warnings in a limited time. Considering the additional dimension of time, software developers favor features that help them optimize their work, such as tool responsiveness, the possibility to contribute to the analysis in order to reduce false positives, or quick fixes, for example. While some of those features are specific to their target user group (e.g., collaboration options for software developers), others can be useful to both analysis developers and software developers, such as better warning explainability, or analysis responsiveness. However, the way such features should be designed differs according to the use case. For example, when explaining warnings, analysis developers require extensive disclosure of the analysis' internal information, but such details might not be as useful for software developers, who prefer to only see minimal relevant information.

In Chapter 4, we focus on the case of the analysis developer, and explore how to provide them with extensive control on the analysis' internal information. We address six of the recommendations we identified for analysis developers, and present VISUFLOW, a debugging environment for data-flow static analysis. Through a user study, we observed that VISUFLOW's features allow analysis developers to debug static analysis faster than with the Eclipse debugging environment. This work shows that adopting a user-centered approach yields useful tools. However, VISU-FLOW is far from complete. While it is a good first iteration of the user-centered design process, it still contains user-experience issues and lacks features like full support for inter-procedural analyses. Further iterations are needed to refine the tool, and the adaption of certain features require further research, such as automated testing for static analysis, quick updates of the analysis results as the code is edited, or even synthesizing flow-functions directly from the graph visualizations to give the analysis developer quicker and simpler control over the analysis' internal rules.

Turning our attention to the software developer, we research a different aspect of controlling the analysis by integrating developer knowledge in the analysis. In Chapter 6 we study how to use this aspect to enhance responsiveness and address four of the recommendations found in Chapter 5. To ensure the responsiveness of static analysis tools, we introduce the concept of Just-in-Time static analysis, allowing developers to express priorities with respect what the analysis should explore first. While this concept can be used for different purposes such as resource management, we illustrate it through CHEETAH, a taint analysis prioritized by code location, allowing developers to see results in their IDE as they code. In an empirical evaluation and a user study, we show that CHEETAH is able to return its first results in less than a second, and allows developers to fix data leaks twice as fast as with a traditional taint analysis.

In Chapter 7, we study the integration of developer knowledge in static analysis at the configuration stage. To support developers in understanding and classifying warnings, and in detecting weak or missing analysis rules, we introduce rule graphs that model the relationships between the analysis rules at runtime and that can be used to reason about how the analysis interprets the analyzed code. By doing so, we expose internal analysis information to the software developers—in a simpler and more limited way than with VISUFLOW, and allow them to tap into this new source of information to bridge the understandability gap between the analysis and the developers' understanding of the analysis. Through a user study and an empirical evaluation, we show that rule graphs can effectively support developers in their configuration tasks, addressing two recommendations of Chapter 5.

The tools developed in Chapter 6 and Chapter 7 both enhance the usability of analysis tools for the code developer. They show that when designing tools with the users' goals in mind, user needs can lead to new tool concepts such as a configuration helper (Chapter 7), or the integration of developer priorities in the analysis (Chapter 6). The tools written in those two chapters can be improved and merged together, especially to integrate more use cases for configuring or prioritizing analyses. More of the recommendations from Chapter 5 can be explored, keeping in mind that developer motivation is a key factor in end-user experience. In particular, research in recommender systems, persuasive technologies, and collaborative problem-solving may result in more usable analysis systems customized for particular developers in particular situations.

In conclusion, this thesis illustrates the need for careful user research in the design of static analysis tools. Different user groups, motivations, and use cases require different sets of tool features, shaping the final design of analysis tools in very different directions. As a result, we advocate for a more user-centered approach of tool design for tools for static analysis, and illustrate the following thesis statement:

We can build more usable tools for data-flow analysis by
putting the user at the center of the design process.

Bibliography

[1] AGARAWALA, A., AND BALAKRISHNAN, R. Keepin' it real: Pushing the desktop metaphor with physics, piles and the pen. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 2006), CHI '06, ACM, pp. 1283–1292. (referenced in p. 12)

[2] AMANN, S., NGUYEN, H. A., NADI, S., NGUYEN, T. N., AND MEZINI, M. A systematic evaluation of static api-misuse detectors. *IEEE Transactions on Software Engineering* (2018). (referenced in p. 1)

[3] ANDREASEN, E. S., MØLLER, A., AND NIELSEN, B. B. Systematic approaches for increasing soundness and precision of static analyzers. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis* (2017), ACM, pp. 31–36. (referenced in p. 16)

[4] ANDROID DEVELOPERS BLOG. How we fought bad apps and malicious developers in 2017. https://android-developers.googleblog.com/2018/01/how-we-fought-bad-apps-and-malicious.html, Accessed in 2019. (referenced in p. 15)

[5] APPBRAIN. Number of android applications. https://www.appbrain.com/stats/number-of-android-apps, Accessed in 2019. (referenced in p. 15)

[6] ARAI, S., SAKAMOTO, K., WASHIZAKI, H., AND FUKAZAWA, Y. A gamified tool for motivating developers to remove warnings of bug pattern tools. In *2014 6th International Workshop on Empirical Software Engineering in Practice* (Nov 2014), pp. 37–42. (referenced in p. 49)

[7] ARZT, S., AND BODDEN, E. Reviser: Efficiently updating ide-/ifds-based data-flow analyses in response to incremental program changes. In *Proceedings of the 36th International Conference on Software Engineering* (New York, NY, USA, 2014), ICSE 2014, ACM, pp. 288–298. (referenced in p. 70)

[8] ARZT, S., RASTHOFER, S., AND BODDEN, E. Susi: A tool for the fully automated classification and categorization of android sources and sinks susi: A tool for the fully automated classification and categorization of android sources and sinks. In *The Network and Distributed System Security Symposium (NDSS)* (2013). (referenced in p. 6)

[9] ARZT, S., RASTHOFER, S., FRITZ, C., BODDEN, E., BARTEL, A., KLEIN, J., LE TRAON, Y., OCTEAU, D., AND MCDANIEL, P. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of*

115

the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (New York, NY, USA, 2014), PLDI '14, ACM, pp. 259–269. (referenced in p. 1, 62, 80, 85, 93, 106, 111)

[10] AYEWAH, N., HOVEMEYER, D., MORGENTHALER, J. D., PENIX, J., AND PUGH, W. Using static analysis to find bugs. *IEEE Software 25*, 5 (Sep. 2008), 22–29. (referenced in p. 1, 46, 47, 62, 95)

[11] AYEWAH, N., AND PUGH, W. A report on a survey and study of static analysis users. In *Proceedings of the 2008 Workshop on Defects in Large Software Systems* (New York, NY, USA, 2008), DEFECTS '08, ACM, pp. 1–5. (referenced in p. 1, 46)

[12] BACON, D. F., AND SWEENEY, P. F. Fast static analysis of c++ virtual function calls. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications* (New York, NY, USA, 1996), OOPSLA '96, ACM, pp. 324–341. (referenced in p. 79)

[13] BALSAMIQ. Balsamiq home page. https://balsamiq.com/, Accessed in 2019. (referenced in p. 49)

[14] BESSEY, A., BLOCK, K., CHELF, B., CHOU, A., FULTON, B., HALLEM, S., HENRI-GROS, C., KAMSKY, A., MCPEAK, S., AND ENGLER, D. A few billion lines of code later: Using static analysis to find bugs in the real world. *Commun. ACM 53*, 2 (Feb. 2010), 66–75. (referenced in p. 1, 46, 55, 61, 69, 71)

[15] BODDEN, E. Inter-procedural data-flow analysis with ifds/ide and soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis* (New York, NY, USA, 2012), SOAP '12, ACM, pp. 3–8. (referenced in p. 5, 10, 36, 102)

[16] BRAM MOOLENAAR. Vim - the ubiquitous text editor. http://www.vim.org/, Accessed in 2019. (referenced in p. 24)

[17] BRITTON, T., JENG, L., CARVER, G., AND CHEAK, P. Reversible debugging software: Quantify the time and cost saved using reversible debuggers, 2013. (referenced in p. 28)

[18] BROOKE, J., ET AL. SUS-A quick and dirty usability scale. *Usability Evaluation in Industry 189*, 194 (1996), 4–7. (referenced in p. 37, 87, 132, 146)

[19] CERN COMPUTER SECURITY. Static code analysis tools. https://security.web.cern.ch/security/recommendations/en/code_tools.shtml, Accessed in 2019. (referenced in p. 46)

[20] CHECKMARX. Checkmarx home page. https://www.checkmarx.com/, Accessed in 2019. (referenced in p. 19, 45, 46, 47, 62, 70, 96)

[21] CHOWDHURY, I., AND ZULKERNINE, M. Can complexity, coupling, and cohesion metrics be used as early indicators of vulnerabilities? In *Proceedings of the 2010 ACM Symposium on Applied Computing* (New York, NY, USA, 2010), SAC '10, ACM, pp. 1963–1969. (referenced in p. 96)

[22] CHRISTAKIS, M., AND BIRD, C. What developers want and need from program analysis: An empirical study. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (New York, NY, USA, 2016), ASE 2016, ACM, pp. 332–343. (referenced in p. 1, 46, 55, 62, 69, 71, 95)

[23] Cifuentes, C., Keynes, N., Li, L., Hawes, N., and Valdiviezo, M. Transitioning parfait into a development tool. *IEEE Security Privacy 10*, 3 (May 2012), 16–23. (referenced in p. 70)

[24] Cohen, W. W. Fast effective rule induction. In *Machine Learning Proceedings 1995*. Elsevier, 1995, pp. 115–123. (referenced in p. 110)

[25] Cooper, A., Reimann, R., and Cronin, D. *About face 3: the essentials of interaction design*. John Wiley & Sons, 2007. (referenced in p. 1)

[26] Coverity. Coverity home page. https://scan.coverity.com/, Accessed in 2019. (referenced in p. 19, 46, 69)

[27] Deering, T., Kothari, S., Sauceda, J., and Mathews, J. Atlas: A new way to explore software, build analysis tools. In *Companion Proceedings of the 36th International Conference on Software Engineering* (New York, NY, USA, 2014), ICSE Companion 2014, ACM, pp. 588–591. (referenced in p. 16)

[28] Destefanis, G., Tonelli, R., Tempero, E., Concas, G., and Marchesi, M. Micro pattern fault-proneness. In *2012 38th Euromicro Conference on Software Engineering and Advanced Applications* (Sep. 2012), pp. 302–306. (referenced in p. 96)

[29] Dillig, I., Dillig, T., and Aiken, A. Automated error diagnosis using abductive inference. *SIGPLAN Not. 47*, 6 (June 2012), 181–192. (referenced in p. 71)

[30] Dodge, Y. *Latin Square Designs*. Springer New York, New York, NY, 2008, pp. 297–297. (referenced in p. 13)

[31] Dyer, R., Nguyen, H. A., Rajan, H., and Nguyen, T. N. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *Proceedings of the 2013 International Conference on Software Engineering* (Piscataway, NJ, USA, 2013), ICSE '13, IEEE Press, pp. 422–431. (referenced in p. 82)

[32] Eclipse Foundation. Eclipse home page. https://eclipse.org/, Accessed in 2019. (referenced in p. 15, 24, 31)

[33] F-Droid. Balance. https://f-droid.org/en/packages/de.mangelow.balance/, Accessed in 2019. (referenced in p. 104)

[34] F-Droid. Bites. https://f-droid.org/wiki/page/caldwell.ben.bites, Accessed in 2019. (referenced in p. 86)

[35] F-Droid. Free and Open Source Android App Repository. https://f-droid.org, Accessed in 2019. (referenced in p. 82, 104, 106)

[36] F-Droid. Sparkleshare. https://f-droid.org/en/packages/org.sparkleshare.android/, Accessed in 2019. (referenced in p. 104)

[37] Fortify Software. Fortify home page. http://www8.hp.com/us/en/software-solutions/static-code-analysis-sast/, Accesed in 2019. (referenced in p. 46, 69, 70, 95)

[38] Free Software Foundation. Gnu emacs. https://www.gnu.org/software/emacs/, Accessed in 2019. (referenced in p. 24)

[39] GARCA, F., PEDREIRA, O., PIATTINI, M., CERDEIRA-PENA, A., AND PENABAD, M. A framework for gamification in software engineering. *J. Syst. Softw. 132*, C (Oct. 2017), 21–40. (referenced in p. 49)

[40] GÖRANSSON, B. *User-centred systems design: designing usable interactive systems in practice.* PhD thesis, Acta Universitatis Upsaliensis, 2004. (referenced in p. 1)

[41] GRAMMATECH. Codesonar home page. https://www.grammatech.com/products/codesonar, Accessed in 2019. (referenced in p. 62, 96)

[42] GRANT, S., AND BETTS, B. Encouraging user behaviour with achievements: An empirical study. In *Proceedings of the 10th Working Conference on Mining Software Repositories* (Piscataway, NJ, USA, 2013), MSR '13, IEEE Press, pp. 65–68. (referenced in p. 49)

[43] GRAPHSTREAM TEAM. Graphstream a dynamic graph library. http://graphstream-project.org/, Accessed in 2019. (referenced in p. 32)

[44] HECKMAN, S., AND WILLIAMS, L. A model building process for identifying actionable static analysis alerts. In *2009 International Conference on Software Testing Verification and Validation* (April 2009), pp. 161–170. (referenced in p. 70, 96)

[45] HUDSON, W. The encyclopedia of human-computer interaction, 22. card sorting. https://www.interaction-design.org/literature/book/the-encyclopedia-of-human-computer-interaction-2nd-ed/card-sorting, 2014. (referenced in p. 18, 29, 38)

[46] JETBRAINS. Intellij home page. https://www.jetbrains.com/idea/, Accessed in 2019. (referenced in p. 15, 24, 62, 103)

[47] JOHNSON, B., SONG, Y., MURPHY-HILL, E., AND BOWDIDGE, R. Why don't software developers use static analysis tools to find bugs? In *Proceedings of the 2013 International Conference on Software Engineering* (Piscataway, NJ, USA, 2013), ICSE '13, IEEE Press, pp. 672–681. (referenced in p. 1, 46, 55, 61, 62, 69, 71, 93, 95)

[48] JUNG, Y., KIM, J., SHIN, J., AND YI, K. Taming false alarms from a domain-unaware c analyzer by a bayesian statistical post analysis. In *Proceedings of the 12th International Conference on Static Analysis* (Berlin, Heidelberg, 2005), SAS'05, Springer-Verlag, pp. 203–217. (referenced in p. 96)

[49] KAHLON, V., YANG, Y., SANKARANARAYANAN, S., AND GUPTA, A. Fast and accurate static data-race detection for concurrent programs. In *Proceedings of the 19th International Conference on Computer Aided Verification* (Berlin, Heidelberg, 2007), CAV'07, Springer-Verlag, pp. 226–239. (referenced in p. 1)

[50] KAM, J. B., AND ULLMAN, J. D. Monotone data flow analysis frameworks. *Acta Inf. 7*, 3 (Sept. 1977), 305–317. (referenced in p. 5, 6, 8, 97)

[51] KHEDKER, U. Compiler analysis and optimizations: What is new. (referenced in p. 1)

[52] KHEDKER, U., SANYAL, A., AND KARKARE, B. *Data Flow Analysis: Theory and Practice*, 1st ed. CRC Press, Inc., Boca Raton, FL, USA, 2009. (referenced in p. 1)

[53] KHOO, Y. P. *User-centered Program Analysis Tools*. PhD thesis, University of Maryland, College Park, MD, USA, 2013. (referenced in p. 14)

[54] Khoo, Y. P., Foster, J. S., Hicks, M., and Sazawal, V. Path projection for user-centered static analysis tools. In *Proceedings of the 8th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering* (New York, NY, USA, 2008), PASTE '08, ACM, pp. 57–63. (referenced in p. 16)

[55] Kildall, G. A. A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (New York, NY, USA, 1973), POPL '73, ACM, pp. 194–206. (referenced in p. 5, 6)

[56] Kim, S., and Ernst, M. D. Which warnings should i fix first? In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering* (New York, NY, USA, 2007), ESEC-FSE '07, ACM, pp. 45–54. (referenced in p. 70)

[57] Ko, A. J., and Myers, B. A. Designing the whyline: a debugging interface for asking questions about program behavior. In *Proceedings of the 2004 Conference on Human Factors in Computing Systems, CHI 2004, Vienna, Austria, April 24 - 29, 2004* (2004), pp. 151–158. (referenced in p. 16)

[58] Kremenek, T., Ashcraft, K., Yang, J., and Engler, D. Correlation exploitation in error ranking. *SIGSOFT Softw. Eng. Notes 29*, 6 (Oct. 2004), 83–93. (referenced in p. 96)

[59] Kremenek, T., and Engler, D. Z-ranking: Using statistical analysis to counter the impact of static analysis approximations. In *Proceedings of the 10th International Conference on Static Analysis* (Berlin, Heidelberg, 2003), SAS'03, Springer-Verlag, pp. 295–315. (referenced in p. 96)

[60] Krüger, S., Späth, J., Ali, K., Bodden, E., and Mezini, M. Crysl: An extensible approach to validating the correct usage of cryptographic apis. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018)* (2018), Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. (referenced in p. 1, 72, 93)

[61] Landis, J. R., and Koch, G. G. The measurement of observer agreement for categorical data. In *Biometrics* (1977), pp. 159–174. (referenced in p. 19, 38)

[62] Layman, L., Williams, L., and Amant, R. S. Toward reducing fault fix time: Understanding developer behavior for the design of automated fault detection tools. In *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)* (Sep. 2007), pp. 176–185. (referenced in p. 46)

[63] Lerch, J., Hermann, B., Bodden, E., and Mezini, M. Flowtwist: Efficient context-sensitive inside-out taint analysis for large codebases. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (New York, NY, USA, 2014), FSE 2014, ACM, pp. 98–108. (referenced in p. 80, 102)

[64] Lewis, B. Debugging backwards in time. *CoRR cs.SE/0310016* (2003). (referenced in p. 16, 36)

[65] Lewis, C., Lin, Z., Sadowski, C., Zhu, X., Ou, R., and Whitehead, E. J. Does bug prediction support human developers? findings from a google case study. In *2013 35th International Conference on Software Engineering (ICSE)* (May 2013), pp. 372–381. (referenced in p. 1, 47, 61, 69, 71, 93, 95)

[66] LIANG, G., WU, Q., WANG, Q., AND MEI, H. An effective defect detection and warning prioritization approach for resource leaks. In *2012 IEEE 36th Annual Computer Software and Applications Conference* (July 2012), pp. 119–128. (referenced in p. 70)

[67] LIVSHITS, B., SRIDHARAN, M., SMARAGDAKIS, Y., LHOTÁK, O., AMARAL, J. N., CHANG, B.-Y. E., GUYER, S. Z., KHEDKER, U. P., MØLLER, A., AND VARDOULAKIS, D. In defense of soundiness: A manifesto. *Commun. ACM 58*, 2 (Jan. 2015), 44–46. (referenced in p. 9)

[68] LOWDERMILK, T. *User-Centered Design A Developer's Guide to Building User-Friendly Applications.* O'Reilly Media, 2013. (referenced in p. 12)

[69] LU, Y., SHANG, L., XIE, X., AND XUE, J. An incremental points-to analysis with cfl-reachability. In *Proceedings of the 22Nd International Conference on Compiler Construction* (Berlin, Heidelberg, 2013), CC'13, Springer-Verlag, pp. 61–81. (referenced in p. 70)

[70] MANGAL, R., ZHANG, X., NORI, A. V., AND NAIK, M. A user-guided approach to program analysis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (New York, NY, USA, 2015), ESEC/FSE 2015, ACM, pp. 462–473. (referenced in p. 71, 95)

[71] MANN, C., AND STAROSTIN, A. A framework for static detection of privacy leaks in android applications. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing* (New York, NY, USA, 2012), SAC '12, ACM, pp. 1457–1462. (referenced in p. 1)

[72] MICROSOFT. Prefast analysis tool. https://msdn.microsoft.com/en-us/library/ms933794.aspx, Accessed in 2019. (referenced in p. 69)

[73] MICROSOFT. Using sal annotations to reduce c/c++ code defects. https://docs.microsoft.com/en-us/visualstudio/code-quality/using-sal-annotations-to-reduce-c-cpp-code-defects?view=vs-2019, Accessed in 2019. (referenced in p. 71)

[74] MITRE. Cwe-601: Url redirection to untrusted site ('open redirect'). https://cwe.mitre.org/data/definitions/601.html, Accessed in 2019. (referenced in p. 94)

[75] MITRE CORPORATION. Cwe-200: Information exposure. https://cwe.mitre.org/data/definitions/200.html, Accessed in 2019. (referenced in p. 6)

[76] MITRE CORPORATION. Cwe-327: Use of a broken or risky cryptographic algorithm. https://cwe.mitre.org/data/definitions/327.html, Accessed in 2019. (referenced in p. 6)

[77] MITRE CORPORATION. Cwe-476: Null pointer dereference. https://cwe.mitre.org/data/definitions/476.html, Accessed in 2019. (referenced in p. 6)

[78] MITRE CORPORATION. Cwe-798: Use of hard-coded credentials. https://cwe.mitre.org/data/definitions/798.html, Accessed in 2019. (referenced in p. 6)

[79] Mitre Corporation. Cwe-89: Improper neutralization of special elements used in an sql command ('sql injection'). https://cwe.mitre.org/data/definitions/89.html, Accessed in 2019. (referenced in p. 5, 6, 94)

[80] Mitre Corporation. Cwe: Common weakness enumeration. http://cwe.mitre.org/top25/, Accessed in 2019. (referenced in p. 6)

[81] Møller, A., and Schwartzbach, M. I. Static program analysis. https://cs.au.dk/ amoeller/spa/spa.pdf, Accessed in 2019. (referenced in p. 46)

[82] Muske, T., and Serebrenik, A. Survey of approaches for handling static analysis alarms. In *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)* (Oct 2016), pp. 157–166. (referenced in p. 70)

[83] Muslu, K., Brun, Y., Ernst, M. D., and Notkin, D. Making offline analyses continuous. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (New York, NY, USA, 2013), ESEC/FSE 2013, ACM, pp. 323–333. (referenced in p. 70)

[84] Naik, M., Aiken, A., and Whaley, J. Effective static race detection for java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2006), PLDI '06, ACM, pp. 308–319. (referenced in p. 1)

[85] Nanda, M. G., Gupta, M., Sinha, S., Chandra, S., Schmidt, D., and Balachandran, P. Making defect-finding tools work for you. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2* (New York, NY, USA, 2010), ICSE '10, ACM, pp. 99–108. (referenced in p. 96)

[86] Nguyen Quang Do, L. Doctoral Thesis Artifact "User-Centered Tool Design for Data-Flow Analysis". https://doi.org/10.5281/zenodo.3267324, July 2019. (referenced in p. 4, 19, 29, 31, 38, 43, 48, 50, 66, 78, 82, 86, 91, 102, 105, 106, 127, 132, 136, 146)

[87] Nguyen Quang Do, L., Ali, K., Livshits, B., Bodden, E., Smith, J., and Murphy-Hill, E. Cheetah: Just-in-time taint analysis for android apps. In *Proceedings of the 39th International Conference on Software Engineering Companion* (Piscataway, NJ, USA, 2017), ICSE-C '17, IEEE Press, pp. 39–42. (referenced in p. 3, 69)

[88] Nguyen Quang Do, L., Ali, K., Livshits, B., Bodden, E., Smith, J., and Murphy-Hill, E. Just-in-time static analysis. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis* (New York, NY, USA, 2017), ISSTA 2017, ACM, pp. 307–317. (referenced in p. 3, 42, 62, 69)

[89] Nguyen Quang Do, L., and Bodden, E. Gamifying static analysis. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (New York, NY, USA, 2018), ESEC/FSE 2018, ACM, pp. 714–718. (referenced in p. 3, 45, 49, 62, 65)

[90] Nguyen Quang Do, L., and Bodden, E. Lifting the burden of static analysis tool configuration with rule graphs. Under submission, 2019. (referenced in p. 3, 94)

[91] Nguyen Quang Do, L., Krüger, S., Hill, P., Ali, K., and Bodden, E. Debugging static analysis. *IEEE Transactions on Software Engineering* (2018), 1–1. (referenced in p. 3, 15, 31)

[92] NGUYEN QUANG DO, L., KRÜGER, S., HILL, P., ALI, K., AND BODDEN, E. Visuflow: A debugging environment for static analyses. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings* (New York, NY, USA, 2018), ICSE '18, ACM, pp. 89–92. (referenced in p. 3, 30, 31)

[93] NGUYEN QUANG DO, L., WRIGHT, J., AND ALI, K. Why do software developers use static analysis tools? a user-centered study of developer needs and motivations. Under submission, 2019. (referenced in p. 3, 45)

[94] NICHOLSON, S. A user-centered theoretical framework for meaningful gamification. In *Games+Learning+Society 8.0* (2012). (referenced in p. 66)

[95] NIELSEN, J. *Usability Engineering.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993. (referenced in p. 83)

[96] NIELSEN NORMAL GROUP. Between-subjects vs. within-subjects study design. https://www.nngroup.com/articles/between-within-subjects/, Accessed in 2019. (referenced in p. 13)

[97] NIELSEN NORMAL GROUP. Priming and user interfaces. https://www.nngroup.com/articles/priming/, Accessed in 2019. (referenced in p. 13)

[98] NIELSON, F., NIELSON, H. R., AND HANKIN, C. *Principles of Program Analysis.* Springer-Verlag, Berlin, Heidelberg, 1999. (referenced in p. 1)

[99] OPEN WEB APPLICATION SECURITY PROJECT (OWASP). Source code analysis tools. https://www.owasp.org/index.php/Source_Code_Analysis_Tools, Accessed in 2019. (referenced in p. 46)

[100] OPEN WEB APPLICATION SECURITY PROJECT (OWASP). Top 10 2007. https://www.owasp.org/index.php/Top_10_2007, Accessed in 2019. (referenced in p. 6)

[101] OPEN WEB APPLICATION SECURITY PROJECT (OWASP). Top 10 2007-information leakage and improper error handling. https://www.owasp.org/index.php/Top_10_2007-Information_Leakage_and_Improper_Error_Handling, Accessed in 2019. (referenced in p. 6)

[102] OPEN WEB APPLICATION SECURITY PROJECT (OWASP). Top 10-2017 a1-injection. https://www.owasp.org/index.php/Top_10-2017_A1-Injection, Accessed in 2019. (referenced in p. 5, 6)

[103] OPEN WEB APPLICATION SECURITY PROJECT (OWASP). Top 10-2017 top 10. https://www.owasp.org/index.php/Top_10-2017_Top_10, Accessed in 2019. (referenced in p. 5, 6)

[104] PEARCE, D. J., KELLY, P. H., AND HANKIN, C. Efficient field-sensitive pointer analysis of c. *ACM Trans. Program. Lang. Syst. 30*, 1 (Nov. 2007). (referenced in p. 9)

[105] PHANG, Y., FOSTER, J. S., HICKS, M., AND SAZAWAL, V. Triaging checklists: a substitute for a phd in static analysis. In *Evaluation and Usability of Programming Languages and Tools (PLATEAU)* (2009), PLATEAU'05. (referenced in p. 96)

[106] Piskachev, G., Nguyen, L., and Bodden, E. Codebase-adaptive detection of security-relevant methods. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (New York, NY, USA, 2019), ISSTA 2019, ACM. (referenced in p. 6, 20)

[107] Porkoláb, Z., Mihalicza, J., and Sipos, A. Debugging c++ template metaprograms. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering* (New York, NY, USA, 2006), GPCE '06, ACM, pp. 255–264. (referenced in p. 16)

[108] Preece, J., Rogers, Y., and Sharp, H. *Interaction design: beyond human-computer interaction.* John Wiley & Sons, 2015. (referenced in p. 1)

[109] Reichheld, F. F. The one number you need to grow. *Harvard Business Review 81*, 12 (2003), 46–55. (referenced in p. 38, 87, 132, 146)

[110] Reps, T., Horwitz, S., and Sagiv, M. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 1995), POPL '95, ACM, pp. 49–61. (referenced in p. 5, 9, 102)

[111] Research Triangle Institute. The economic impacts of inadequate infrastructure for software testing, 2002. (referenced in p. 46)

[112] Rogers, Y., Sharp, H., and Preece, J. *Interaction Design: Beyond Human - Computer Interaction*, 3rd ed. Wiley Publishing, 2011. (referenced in p. 12)

[113] Rushby, J. Verified software: Theories, tools, experiments. In *Automated Test Generation and Verified Software*, B. Meyer and J. Woodcock, Eds. Springer-Verlag, Berlin, Heidelberg, 2008, pp. 161–172. (referenced in p. 36)

[114] Ruthruff, J. R., Penix, J., Morgenthaler, J. D., Elbaum, S., and Rothermel, G. Predicting accurate and actionable static analysis warnings: An experimental approach. In *Proceedings of the 30th International Conference on Software Engineering* (New York, NY, USA, 2008), ICSE '08, ACM, pp. 341–350. (referenced in p. 95)

[115] Ruthruff, J. R., Penix, J., Morgenthaler, J. D., Elbaum, S., and Rothermel, G. Predicting accurate and actionable static analysis warnings: An experimental approach. In *Proceedings of the 30th International Conference on Software Engineering* (New York, NY, USA, 2008), ICSE '08, ACM, pp. 341–350. (referenced in p. 96)

[116] Ryder, B. G. Incremental data flow analysis. In *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages* (1983), ACM, pp. 167–176. (referenced in p. 36, 42, 70, 93)

[117] SANS Institute. Cwe/sans top 25 most dangerous software errors. https://www.sans.org/top25-software-errors, Accessed in 2019. (referenced in p. 20)

[118] Schubert, P. D., Hermann, B., and Bodden, E. Phasar: An inter-procedural static analysis framework for c/c++. In *Tools and Algorithms for the Construction and Analysis of Systems* (2019), TACAS, Springer, Cham. (referenced in p. 93)

[119] Segal, M. R. Machine learning benchmarks and random forest regression. *UCSF: Center for Bioinformatics and Molecular Biostatistics* (2004). (referenced in p. 110)

[120] SHARIR, M., AND PNUELI, A. *Two approaches to interprocedural data flow analysis.* New York Univ. Comput. Sci. Dept., New York, NY, 1978. (referenced in p. 9)

[121] SHEN, H., FANG, J., AND ZHAO, J. Efindbugs: Effective error ranking for findbugs. In *ICST* (2011), pp. 299–308. (referenced in p. 19, 46, 62, 70)

[122] SHEN, H., FANG, J., AND ZHAO, J. Efindbugs: Effective error ranking for findbugs. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation* (March 2011), pp. 299–308. (referenced in p. 70)

[123] SHEN, H., FANG, J., AND ZHAO, J. Efindbugs: Effective error ranking for findbugs. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation* (March 2011), pp. 299–308. (referenced in p. 96)

[124] SHERRIFF, M., HECKMAN, S. S., LAKE, J. M., AND WILLIAMS, L. A. Using groupings of static analysis alerts to identify files likely to contain field failures. In *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007* (2007), pp. 565–568. (referenced in p. 96)

[125] SINGER, J., BROWN, G., LUJÁN, M., POCOCK, A., AND YIAPANIS, P. Fundamental nano-patterns to characterize and classify java methods. *Electron. Notes Theor. Comput. Sci. 253*, 7 (Sept. 2010), 191–204. (referenced in p. 96)

[126] SMARAGDAKIS, Y., BRAVENBOER, M., AND LHOTÁK, O. Pick your contexts well: Understanding object-sensitivity. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2011), POPL '11, ACM, pp. 17–30. (referenced in p. 9)

[127] SOFTWARE AG. Software ag. https://www.softwareag.com, Accessed in 2019. (referenced in p. 45)

[128] SPÄTH, J., ALI, K., AND BODDEN, E. Ideal: Efficient and precise alias-aware dataflow analysis. *Proc. ACM Program. Lang. 1*, OOPSLA (Oct. 2017), 99:1–99:27. (referenced in p. 36, 93)

[129] SPÄTH, J., DO, L. N. Q., ALI, K., AND BODDEN, E. Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java. In *30th European Conference on Object-Oriented Programming (ECOOP 2016)* (Dagstuhl, Germany, 2016), S. Krishnamurthi and B. S. Lerner, Eds., vol. 56 of *Leibniz International Proceedings in Informatics (LIPIcs)*, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 22:1–22:26. (referenced in p. 9, 93)

[130] SUGIYAMA, K., TAGAWA, S., AND TODA, M. Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man & Cybernetics 11*, 2 (1981), 109–125. (referenced in p. 32)

[131] SULTANA, K. Z., DEO, A., AND WILLIAMS, B. J. Correlation analysis among java nano-patterns and software vulnerabilities. In *2017 IEEE 18th International Symposium on High Assurance Systems Engineering (HASE)* (Jan 2017), pp. 69–76. (referenced in p. 96)

[132] THE GDB DEVELOPERS. Gdb: The gnu project debugger. https://www.gnu.org/software/gdb/, Accessed in 2019. (referenced in p. 15)

[133] The LLVM admin team. The llvm compiler infrastructure. http://llvm.org/, Accessed in 2019. (referenced in p. 20)

[134] Tillmann, N., De Halleux, J., Xie, T., Gulwani, S., and Bishop, J. Teaching and learning programming and software engineering via interactive gaming. In *Proceedings of the 2013 International Conference on Software Engineering* (Piscataway, NJ, USA, 2013), ICSE '13, IEEE Press, pp. 1117–1126. (referenced in p. 49)

[135] T.J. Watson. Wala. http://wala.sourceforge.net/wiki/index.php, Accessed in 2019. (referenced in p. 20, 36)

[136] Tsipenyuk, K., Chess, B., and McGraw, G. Seven pernicious kingdoms: A taxonomy of software security errors. *IEEE Security and Privacy 3*, 6 (Nov. 2005), 81–84. (referenced in p. 1)

[137] Ulanov, A., Simanovsky, A., and Marwah, M. Modeling scalability of distributed machine learning. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)* (April 2017), pp. 1249–1254. (referenced in p. 103)

[138] U.S. Department of Health & Human Services. User-centered design basics. https://www.usability.gov/what-and-why/user-centered-design.html, Accessed in 2019. (referenced in p. 12)

[139] Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., and Sundaresan, V. Soot: A java bytecode optimization framework. In *CASCON First Decade High Impact Papers* (2010), IBM Corp., pp. 214–224. (referenced in p. 8, 102)

[140] Vallée-Rai, R., Gagnon, E., Hendren, L., Lam, P., Pominville, P., and Sundaresan, V. Optimizing java bytecode using the soot framework: Is it feasible? In *Compiler Construction* (Berlin, Heidelberg, 2000), D. A. Watt, Ed., Springer Berlin Heidelberg, pp. 18–34. (referenced in p. 1, 8, 20, 31, 102)

[141] Viswanathan, S., and Shrikant, I. B. Observer agreement paradoxes in 2x2 tables: Comparison of agreement measures. In *BMC Medical Research Methodology* (2014). (referenced in p. 19)

[142] Vorobyov, K., and Krishnan, P. Comparing model checking and static program analysis: A case study in error detection approaches. *SSV* (2010). (referenced in p. 46)

[143] Vredenberg, K., Isensee, S., and Righi, C. *User-Centered Design: An Integrated Approach with Cdrom.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001. (referenced in p. 12)

[144] Wharton, C., Rieman, J., Lewis, C., and Polson, P. The cognitive walkthrough method: A practitioner's guide. In *Usability Inspection Methods*, J. Nielsen and R. L. Mack, Eds. John Wiley & Sons, Inc., New York, NY, USA, 1994, pp. 105–140. (referenced in p. 50)

[145] Wilcoxon, F. Individual comparisons by ranking methods. *Biometrics Bulletin 1*, 6 (1945), 80–83. (referenced in p. 90)

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

[147] Witten, I. H., Frank, E., Hall, M. A., and Pal, C. J. *Data Mining, Fourth Edition: Practical Machine Learning Tools and Techniques*, 4th ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2016. (referenced in p. 103)

[148] Xiao, S., Witschey, J., and Murphy-Hill, E. Social influences on secure development tool adoption: Why security tools spread. In *Proceedings of the 17th ACM Conference on Computer Supported Cooperative Work & Social Computing* (New York, NY, USA, 2014), CSCW '14, ACM, pp. 1095–1106. (referenced in p. 69)

[149] Xie, J., Lipford, H., and Chu, B.-T. Evaluating interactive support for secure programming. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 2012), CHI '12, ACM, pp. 2707–2716. (referenced in p. 71)

[150] Zeller, A., and Hildebrandt, R. Simplifying and isolating failure-inducing input. *IEEE Trans. Software Eng. 28*, 2 (2002), 183–200. (referenced in p. 16)

[151] Zhan, S., and Huang, J. Echo: Instantaneous in situ race detection in the ide. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (New York, NY, USA, 2016), FSE 2016, ACM, pp. 775–786. (referenced in p. 70)

Appendices

A Survey: Debugging Tools for Static Analysis

In this appendix, we detail the survey questions from Section 3.2. The anonymized answers are available online [86]. The survey refers to the analyzed code as *application code*, which was explained to the participants beforehand.

Q1 Are you...

- Multiple choice question with an "Other..." free-text field.
- Choices:
    * A Bachelor/Master student?
    * A researcher in academia (incl. Ph.D. student)?
    * A researcher in industry?
    * Other...

Q2 Your research / study topic.

- Free-text field.
- Mandatory.

Q3 How long have you been writing static code analysis?

- Single-answer question.
- Choices:
    * I have never written static analysis
    * < 1 years
    * 1-2 years
    * 2-5 years
    * 5-10 years
    * > 10 years
- Mandatory.

Q4 Which languages have you written static analysis for?

- Multiple choice question with an "Other..." field.
- Choices:

- ∗ Java
- ∗ JavaScript
- ∗ C/C++
- ∗ .NET
- ∗ Objective C/Objective C++
- ∗ Perl
- ∗ PL/SQL
- ∗ Python
- ∗ Other...
- – Mandatory.

Q5 Which applications have you written static analyses for?

- – Multiple choice question with an "Other..." field.
- – Choices:
  - ∗ Security and privacy
  - ∗ Functional correctness
  - ∗ Program understanding
  - ∗ Automated performance optimization
  - ∗ Other...
- – Mandatory.

Q6 Which branches of static analysis have you written for?

- – Multiple choice question with an "Other..." field.
- – Choices:
  - ∗ Symbolic execution
  - ∗ Model checking
  - ∗ Data-flow analysis
  - ∗ Abstract interpretation
  - ∗ Other...
- – Mandatory.

Q7 On which program level do you usually write static analysis for?

- – Multiple choice question with an "Other..." field.
- – Choices:
  - ∗ Line-based
  - ∗ Function-based
  - ∗ Module-based
  - ∗ Program-based
  - ∗ System-based
  - ∗ Other...

Q8 Have you written static analysis for industrial tools? Which ones?

- – Multiple choice question with an "Other..." field.

- Choices:
  * I haven't written analyses for industrial tools.
  * Fortify
  * AppScan
  * CheckMarx
  * Klocwork
  * Coverity
  * FindBugs
  * VeraCode
  * Other...

Q9 Do you use frameworks to write your analyses?

- Multiple choice question with an "Other..." field.
- Choices:
  * Soot
  * OPAL
  * WALA
  * Doop
  * Chord
  * Crystal
  * PMD
  * FindBugs
  * Other...

Q10 Can you list a few examples of analyses you have written?

- Free-text field.

Q11 Do you find it easier to debug application code or static analysis code?

- Likert scale from 1 (application code is harder to debug) to 10 (analysis code is harder to debug).
- Mandatory.

Q12 Why?

- Free-text field.
- Mandatory.

Q13 When developing a static analysis, how long do you usually spend writing vs debugging your code?

- Likert scale from 0 (100% of my time coding, 0% of my time debugging) to 10 (0% of my time coding, 100% of my time debugging).
- Mandatory.

Q14 In which cases have you debugged static-analysis code?

- Free-text field.

Q15 Give a few examples of bugs you typically encounter when debugging static-analysis code.

- Free-text field.

Q16 Which features of your coding environment do you use most when getting rid of those bugs?

- Free-text field.
- Mandatory.

Q17 List the top 3 features of your coding environment that you particularly *like* when debugging static analysis.

- Free-text field.
- Mandatory.

Q18 List the top 3 features of your coding environment that you particularly *dislike* when debugging static analysis.

- Free-text field.
- Mandatory.

Q19 Which features would you like to have (to support debugging static analysis) that your coding environment does not provide?

- Free-text field.

Q20 When writing application code, how long do you usually spend writing vs debugging your code?

- Likert scale from 0 (100% of my time coding, 0% of my time debugging) to 10 (0% of my time coding, 100% of my time debugging).
- Mandatory.

Q21 Give a few examples of bugs you typically encounter when debugging application code.

- Free-text field.

Q22 Which features of your coding environment do you use most when getting rid of those bugs?

- Free-text field.
- Mandatory.

Q23 List the top 3 features of your coding environment that you particularly *like* when debugging application code.

- Free-text field.
- Mandatory.

Q24 List the top 3 features of your coding environment that you particularly *dislike* when debugging application code.

- Free-text field.

– Mandatory.

Q25 Which features would you like to have (to support debugging application code) that your coding environment does not provide?

– Free-text field.

Q26 Rate how important the following features are to you when debugging static analysis.

– Multiple-choice grid.
– Categories:
  * Good visuals of graphs (Control-flow graphs, call-graphs, etc.)
  * Good visuals (not graphs)
  * Showing the intermediate representation on which the analysis is based
  * Providing default test cases
  * Quick updates
  * Breakpoints
  * Stepping functionalities through both the analysis code and the test cases
– Choices for each categories:
  * Not important
  * Neutral
  * Important
  * Very important
  * Not applicable

Q27 Are there features that you would like to add and how important are them to you?

– Free-text field.

Q28 Which development environment do you use most often when writing static analysis?

– Single-answer question.
– Choices:
  * A simple text editor (vim, emacs...)
  * An IDE (Eclipse, NetBeans, ...)
  * Other solutions
– Mandatory.

Q29 Which editor(s) do you use?

– Free-text field.
– Mandatory

Q30 Why?

– Multiple choice question with an "Other..." field.
– Choices:
  * It is fast
  * It is lightweight

* It is highly customizable
* The UI is simple
* it is just what I need
* Other tools have too many functionalities, it is confusing
* It is installed by default on most platforms
* It provides coding support (error checking, code navigation and generation, etc.)
* It provides build and run support (automated and incremental building, dependency importing, run configurations, etc.)
* It is specifically designed for my use case
* Other...

Q31 Would you be willing to participate in a user study on a prototype supporting the debugging of static analysis?

– Single-answer question.
– Choices:
  * Yes
  * No
  * Maybe
– Mandatory.

Q32 If yes or maybe, please provide a contact email. This email will only be used to contact you for a subsequent user study. It will be removed from the rest of the survey to keep the data anonymous.

– Free-text field.

B Questionnaire: VisuFlow User Study

In this appendix, we detail the questionnaire from Section 4.2.1. The anonymized answers are available online [86].

In the questionnaire, we refer to as CE1 (Coding Environment 1) the first coding environment used by the participants (Eclipse or VisuFlow). CE2 refers to the other coding environment. The questionnaire refers to the analyzed code as *application code*, which was explained to the participants beforehand. **Q6** to **Q15** are questions from the System Usability Likert scale from [18] for CE1. **Q16** to **Q25** are the same questions for CE2. **Q32** to **Q35** are Net Promoter Scores [109].

Q1 How long have you been writing static code analysis?

– Single-answer question.
– Choices:
  * 1 years
  * $1 - 2$ years
  * 2-5 years
  * $5 - 10$ years
  * > 10 years

    – Mandatory.

Q2 Which coding environment do you primarily use (name of the IDE or text editor)?

    – Free-text field.

    – Mandatory.

Q3 How familiar are you with Eclipse?

    – Likert scale from 0 (Never heard of) to 10 (Expert).

    – Mandatory.

Q4 How familiar are you with data-flow analysis?

    – Likert scale from 0 (Never heard of) to 10 (Expert).

    – Mandatory.

Q5 How familiar are you with Soot?

    – Likert scale from 0 (Never heard of) to 10 (Expert).

    – Mandatory.

Q6 If I had to do the task frequently, I think that I would use CE1 frequently.

    – Likert scale from 1 (Strongly disagree) to 5 (Strongly agree).

    – Mandatory.

Q7 I found CE1 unnecessarily complex.

    – Likert scale from 1 (Strongly disagree) to 5 (Strongly agree).

    – Mandatory.

Q8 I found CE1 easy to use.

    – Likert scale from 1 (Strongly disagree) to 5 (Strongly agree).

    – Mandatory.

Q9 I would need the support of a technical person to use CE1.

    – Likert scale from 1 (Strongly disagree) to 5 (Strongly agree).

    – Mandatory.

Q10 I found the various functions of CE1 well integrated.

    – Likert scale from 1 (Strongly disagree) to 5 (Strongly agree).

    – Mandatory.

Q11 I thought there was too much inconsistency in CE1.

    – Likert scale from 1 (Strongly disagree) to 5 (Strongly agree).

    – Mandatory.

Q12 I would imagine that most people would learn to use CE1 very quickly.

    – Likert scale from 1 (Strongly disagree) to 5 (Strongly agree).

– Mandatory.

Q13 I found CE1 very cumbersome to use.

– Likert scale from 1 (Strongly disagree) to 5 (Strongly agree).
– Mandatory.

Q14 I felt confident using CE1.

– Likert scale from 1 (Strongly disagree) to 5 (Strongly agree).
– Mandatory.

Q15 I needed to learn a lot of things before I could get going with CE1.

– Likert scale from 1 (Strongly disagree) to 5 (Strongly agree).
– Mandatory.

Q16 – **Q25** Same questions as **Q6** – **Q15**, for CE2.

Q26 In general, errors were easier to understand with:

– Single-answer question.
– Choices:
    * CE1
    * CE2
    * Neutral
– Mandatory.

Q27 In general, errors were easier to fix with:

– Single-answer question.
– Choices:
    * CE1
    * CE2
    * Neutral
– Mandatory.

Q28 I fixed more errors with:

– Single-answer question.
– Choices:
    * CE1
    * CE2
    * Neutral
– Mandatory.

Q29 I fixed errors faster with:

– Single-answer question.
– Choices:

* CE1
* CE2
* Neutral

– Mandatory.

Q30 In general, I had a better grasp on the analysis code with:

– Single-answer question.
– Choices:
  * CE1
  * CE2
  * Neutral
– Mandatory.

Q31 In general, I had a better grasp on the application code with:

– Single-answer question.
– Choices:
  * CE1
  * CE2
  * Neutral
– Mandatory.

Q32 How much would you recommend CE1 over CE2 to a friend for the type of task you performed?

– Likert scale from 0 (Not at all) to 10 (Go for it!).
– Mandatory.

Q33 How much would you recommend CE2 over CE1 to a friend for the type of task you performed?

– Likert scale from 0 (Not at all) to 10 (Go for it!).
– Mandatory.

Q34 How much would you recommend CE1 over your own coding environment to a friend for the type of task you performed?

– Likert scale from 0 (Not at all) to 10 (Go for it!).
– Mandatory.

Q35 How much would you recommend CE2 over your own coding environment to a friend for the type of task you performed?

– Likert scale from 0 (Not at all) to 10 (Go for it!).
– Mandatory.

Q36 Describe your coding environment when writing static analysis. What do you typically use (software, favorite features)?

– Free-text field.

– Mandatory.

Q37 Which features of CE1 would you like to have in your current coding environment?

– Free-text field.
– Mandatory.

Q38 Which features of CE2 would you like to have in your current coding environment?

– Free-text field.
– Mandatory.

Q39 In your opinion, which task(s) would CE1 be best suited for?

– Free-text field.
– Mandatory.

Q40 In your opinion, which task(s) would CE2 be best suited for?

– Free-text field.
– Mandatory.

Q41 Would you change anything about CE1 and CE2? Other comments?

– Free-text field.

C   Survey: Developer Behavior and Motivation

We detail the survey questions from Section 5.2. The anonymized answers are available online [86].

Q1 How long have you worked as a software developer?

– Single-answer question with an "Other..." free-text field.
– Choices:
  * < 1 year
  * $1 - 2$ years
  * 2-5 years
  * $5 - 10$ years
  * > 10 years
  * Other...
– Mandatory.

Q2 At the moment, which programming languages do you develop with?

– Multiple choice question with an "Other..." free-text field.
– Choices:
  * Java / Android
  * C / C++
  * C# / .NET

- ∗ Perl
- ∗ JavaScript / TypeScript / NodeJS
- ∗ PHP
- ∗ Python
- ∗ Ruby
- ∗ Objective C
- ∗ Other...
    – Mandatory.

Q3 Would you be willing to participate in a later interview? If so, please provide your email below.

- – Free-text field.

Q4 Which code analysis tools do you use in your current projects?

- – Multiple choice question with an "Other..." free-text field.
- – Choices:
    - ∗ IDE notifications (e.g., dead code in Eclipse)
    - ∗ FindBugs
    - ∗ Fortify
    - ∗ Checkmarx
    - ∗ CodeSonar
    - ∗ Coverity
    - ∗ VeraCode
    - ∗ AppScan
    - ∗ Klocwork
    - ∗ SonarQube
    - ∗ Linters
    - ∗ Other...
- – Mandatory.

Q5 At which points of your projects are code analysis tools typically run?

- – Multiple choice question with an "Other..." free-text field.
- – Choices:
    - ∗ During coding (in the editor)
    - ∗ During nightly builds
    - ∗ At commit time
    - ∗ At major milestones in the projects
    - ∗ Other...
- – Mandatory.

Q6 Who usually configures the analysis tools?

- – Multiple choice question with an "Other..." free-text field.
- – Choices:

    ∗ Yourself

    ∗ A dedicated team

    ∗ No one. I use the default settings of the tools.

    ∗ My manager

    ∗ Other...

  – Mandatory.

Q7 What kind of issues are typically detected by code analysis tools on your projects?

  – Multiple choice question with an "Other..." free-text field.

  – Choices:

    ∗ Security vulnerabilities

    ∗ Functional bugs

    ∗ Coding style

    ∗ Memory consumption

    ∗ Concurrency

    ∗ Performance

    ∗ Other...

  – Mandatory.

Q8 In your opinion, what kind of issues *should* code analysis tools detect in your projects?

  – Multiple choice question with an "Other..." free-text field.

  – Choices:

    ∗ Security vulnerabilities

    ∗ Functional bugs

    ∗ Coding style

    ∗ Memory consumption

    ∗ Concurrency

    ∗ Performance

    ∗ Other...

  – Mandatory.

Q9 Do you usually review the analysis results yourself?

  – Single-answer question.

  – Choices:

    ∗ Yes

    ∗ No

  – Mandatory.

Q10 Where are the analysis results typically reported in your projects?

  – Multiple choice question with an "Other..." free-text field.

  – Choices:

    ∗ In my code editor

    ∗ In the build output

* In the code review
* In a dedicated tool
* In a PDF report
* By email
* Other...

– Mandatory.

Q11 Where would you prefer analysis results to be reported?

– Multiple choice question with an "Other..." free-text field.
– Choices:
    * In my code editor
    * In the build output
    * In the code review
    * In a dedicated tool
    * In a PDF report
    * By email
    * Other...
– Mandatory.

Q12 If you are using multiple analysis tools, would you prefer all analysis results to be reported in a single interface or in multiple ones?

– Single-answer question.
– Choices:
    * One single tool
    * Multiple tools
    * I am only using one analysis tool
– Mandatory.

Q13 How long does it usually take you to completely fix an analysis warning?

– Single-answer question with an "Other..." free-text field.
– Choices:
    * A few minutes
    * < 1 hour
    * < 1 day
    * < 1 week
    * < 1 month
    * < 6 months
    * > 6 months
    * Other...
– Mandatory.

Q14 After you have modified your code in response to an analysis warning, how long would you be willing to wait for the analysis to verify your change?

- Single-answer question with an "Other..." free-text field.
- Choices:
  * < 1 second
  * < 1 minute
  * A few minutes
  * < 1 hour
  * A few hours
  * < 1 day
  * A few days
  * > 1 week
  * Other...
- Mandatory.

Q15 Any additional comments you would like to share about reporting tools?

- Free-text field.

Q16 Which analysis tool do you use most?

- Free-text field.
- Mandatory.

Q17 Why do you use this tool?

- Multiple choice question with an "Other..." free-text field.
- Choices:
  * Company policy
  * It helps me code faster
  * It helps me code better
  * Other...
- Mandatory.

Q18 How often do you use the tool?

- Single-answer question with an "Other..." free-text field.
- Choices:
  * Once a month or less
  * 1 − 3 times a week
  * Once a day
  * 2 − 5 times a day
  * > 5 times a day
  * Other...
- Mandatory.

Q19 When do you usually use the tool?

- Multiple choice question with an "Other..." free-text field.
- Choices:

* In the morning
* In the afternoon
* In the evening
* At night
* When I have a few minutes here and there
* During the work week
* During week-ends
* Other...
    – Mandatory.

Q20 Where are you when you usually use the tool?

– Multiple choice question with an "Other..." free-text field.
– Choices:
* At work, at my desk
* At work, with other colleagues
* At work, during meetings
* At home
* In the transports
* Other...
– Mandatory.

Q21 How long per week do you spend interacting with this tool?

– Single-answer question with an "Other..." free-text field.
– Choices:
* < 1 hour
* < 1 – 5 hours
* < 5 – 10 hours
* < 10 – 30 hours
* > 30 hours
* Other...
– Mandatory.

Q22 Typically, how long do you use the tool in one working session?

– Single-answer question with an "Other..." free-text field.
– Choices:
* < 10 minutes
* 10 - 30 minutes
* > 30 minutes
* Hours
* Other...
– Mandatory.

Q23 What is your goal when you open the tool?

    – Multiple choice question with an "Other..." free-text field.

    – Choices:

         * To fix all of the warnings it reports

         * To fix all the warnings I can in the time I have

         * To fix a set number of warnings

         * To consult the list of warnings

         * Other...

    – Mandatory.

Q24 Why do you stop using the tool?

    – Multiple choice question with an "Other..." free-text field.

    – Choices:

         * I cannot fix an issue

         * I have to go to a professional obligation (meeting, etc.)

         * I am distracted by office events (calls, coffees, etc.)

         * I am waiting for the tool to update

         * I finished fixing all issues

         * Other...

    – Mandatory.

Q25 Do you use the default layout of the analysis tool? (i.e., do you change the position of any component in the user interface?)

    – Single-answer question with an "Other..." free-text field.

    – Choices:

         * I use the default layout of the tool

         * I am using a layout customized by the company

         * I have customized the interface for my own use

         * I change the interface regularly as I use the tool

         * Other...

    – Mandatory.

Q26 When you open the tool, what is the first thing in the interface that you look for?

    – Single-answer question with an "Other..." free-text field.

    – Choices:

         * The list of warnings

         * A dashboard

         * Some code

         * Other...

    – Mandatory.

Q27 When you close the tool, what is the last thing in the interface that you look at?

    – Single-answer question with an "Other..." free-text field.

    – Choices:

* The list of warnings
* A dashboard
* Some code
* Other...

– Mandatory.

Q28 What are the components in the interface that you use the most?

– Multiple choice question with an "Other..." free-text field.
– Choices:

* The list of warnings
* A dashboard
* Some code
* Other...

– Mandatory.

Q29 Once you finish fixing the issues, does anyone review your fixes?

– Multiple choice question with an "Other..." free-text field.
– Choices:

* A colleague
* My manager
* A separate team
* No one reviews my fixes
* Other...

– Mandatory.

Q30 In the perfect analysis tool, how important are the following features to you?

– Multiple-choice grid.
– Categories:

* Responsiveness of the analysis (the time taken by the analysis to process my fix)
* Responsiveness of the UI (does the User Interface lag?)
* Bug information: how a bug works and what it is about
* Severity information: how severely the bug can affect your code
* Execution information: how the bug can be executed in your code
* Interaction information: how the bug is related to other bugs in the code base
* Fix information 1: how the bug can be fixed, on a high level
* Fix information 2: how the bug can be fixed in your code
* Quick fixes: fixes generated by the tools itself
* Visualizations: the tools provide you with a detailed visualization of the warnings in the code
* Visualizations: the tools provide you with a general dashboard of the issues
* Sorting: the tools allow you to sort through warnings and search for them
* Prioritization: the tools allow you to select which bugs to fix first
* List: the tools allow you to keep a list of 'your' bugs

- * Tracking progress: the tools give you a clear view of what you have achieved so far
  * Analysis configurations: configurations of the analysis before it runs
  * Feedback in the reporting tool: ability to tell if a warning is correct or not in the reporting tool
  * Customization of the analysis rules: writing my own rules for the analysis
  * Collaboration options: the tools allow you to collaborate with other colleagues to fix issues
- – Choices for each categories:
  * This should not be in an analysis tool
  * Neutral
  * Low importance
  * Important
  * Very important
  * Indispensable
- – Mandatory.

Q31 Which other features have we missed, and how important are they to you?

- – Free-text field.

Q32 In a typical analysis report, what is the usual proportion of warnings that you personally investigate?

- – Likert scale from 0 (0%) to 10 (100%).
- – Mandatory.

Q33 How do you usually differentiate between a false positive and a real issue?

- – Multiple choice question with an "Other..." free-text field.
- – Choices:
  * False positives occur in particular places in the code I know are never executed
  * Certain categories of issues are more likely to be false positives
  * The path shown by the analysis tool is not executable
  * The conditions along the path are never true
  * Some constructs of the source code are not well handled by the analysis (e.g., static constructs, loops, etc.)
  * Other...
- – Mandatory.

Q34 How do you select which warnings to investigate first?

- – Multiple choice question with an "Other..." free-text field.
- – Choices:
  * I follow the list of warnings from the top down
  * I look at warnings affecting my code first
  * I prioritize warnings which I know I can fix

144

        * I prioritize warnings with the most impact

        * Other...

    – Mandatory.

Q35 Among the warnings that you investigate, what is the usual proportion of warnings that you cannot understand / explain?

    – Likert scale from 0 (0%) to 10 (100%).

    – Mandatory.

Q36 What makes such warnings difficult to understand / explain?

    – Multiple choice question with an "Other..." free-text field.

    – Choices:

        * They span over too much of the code base

        * They are issues that I rarely encounter

        * I do not understand the explanation given by the analysis tool

        * I do not understand the code base

        * Other...

    – Mandatory.

Q37 What do you typically do with warnings that you cannot understand / explain?

    – Multiple choice question with an "Other..." free-text field.

    – Choices:

        * I ignore them

        * I suppress them

        * I mark them and leave them for later

        * I ask colleagues for help

        * I escalate them

        * Other...

    – Mandatory.

Q38 In a typical analysis report, what is the usual proportion of warnings that you ask others to help you fix?

    – Likert scale from 0 (0%) to 10 (100%).

    – Mandatory.

Q39 What is your main motivation to ask others to help you fix warnings?

    – Multiple choice question with an "Other..." free-text field.

    – Choices:

        * They have more experience with the code base

        * They have experience in this type of issues

        * They have more experience with the analysis tool

        * I don't ask others to help me fix warnings

        * Other...

– Mandatory.

Q40 Any additional comments you would like to share about how you review analysis warnings?

– Free-text field.

D   Questionnaire: Cheetah User Study

In this appendix, we detail the questionnaire from Section 6.4.3. The anonymized answers are available online [86].

The questionnaire refers to as *tool 1* the first analysis tool used by the participants (Cheetah or Batch). *Tool 2* refers to the other analysis tool. **Q5** to **Q13** are questions from the System Usability Likert scale from [18] for tool 1. **Q14** to **Q22** are the same questions for tool 2. **Q40**–**Q41** are Net Promoter Scores [109].

Q1 How much experience do you have with Java development? (in years)

– Free-text field.
– Mandatory.

Q2 How much experience do you have with Android development? (in years)

– Free-text field.
– Mandatory.

Q3 How much experience do you have using static analysis tools? (in years)

– Free-text field.
– Mandatory.

Q4 If you have already used static analysis tools, which ones?

– Free-text field.
– Mandatory.

Q5 If I had to do the task frequently, I think that I would use tool 1 frequently.

– Likert scale from 1 (Strongly disagree) to 5 (Strongly agree).
– Mandatory.

Q6 I found tool 1 unnecessarily complex.

– Likert scale from 1 (Strongly disagree) to 5 (Strongly agree).
– Mandatory.

Q7 I found tool 1 easy to use.

– Likert scale from 1 (Strongly disagree) to 5 (Strongly agree).
– Mandatory.

Q8 I would need the support of a technical person to use tool 1.

– Likert scale from 1 (Strongly disagree) to 5 (Strongly agree).

146

– Mandatory.

Q9 I found the various functions of tool 1 well integrated.

– Likert scale from 1 (Strongly disagree) to 5 (Strongly agree).
– Mandatory.

Q10 I thought there was too much inconsistency in tool 1.

– Likert scale from 1 (Strongly disagree) to 5 (Strongly agree).
– Mandatory.

Q11 I would imagine that most people would learn to use tool 1 very quickly.

– Likert scale from 1 (Strongly disagree) to 5 (Strongly agree).
– Mandatory.

Q12 I found tool 1 very cumbersome to use.

– Likert scale from 1 (Strongly disagree) to 5 (Strongly agree).
– Mandatory.

Q13 I felt confident using tool 1.

– Likert scale from 1 (Strongly disagree) to 5 (Strongly agree).
– Mandatory.

Q14 – **Q22** Same questions as **Q5** – **Q13**, for tool 2.

Q23 For which tool was the UI blocked when the analysis was running?

– Single-answer question.
– Choices:
  * Tool 1
  * Tool 2
– Mandatory.

Q24 This made it easier to correct data leaks.

– Likert scale from 1 (Strongly disagree) to 5 (Strongly agree).
– Mandatory.

Q25 Why?

– Free-text field.
– Mandatory.

Q26 For which tool was the analysis was triggered with a button?

– Single-answer question.
– Choices:
  * Tool 1
  * Tool 2

   – Mandatory.

Q27 For which tool was the analysis was triggered on build?

   – Single-answer question.

   – Choices:

      ∗ Tool 1

      ∗ Tool 2

   – Mandatory.

Q28 I preferred the tool to be triggered...

   – Single-answer question.

   – Choices:

      ∗ With a button

      ∗ On build

   – Mandatory.

Q29 Why?

   – Free-text field.

   – Mandatory.

Q30 After I corrected a warning, I had to wait longer to get an update with:

   – Single-answer question.

   – Choices:

      ∗ Tool 1

      ∗ Tool 2

   – Mandatory.

Q31 With which tool did the warnings change as I was editing the code?

   – Single-answer question.

   – Choices:

      ∗ Tool 1

      ∗ Tool 2

      ∗ Neither of the tools

   – Mandatory.

Q32 I felt comfortable with this system:

   – Likert scale from 1 (Strongly disagree) to 5 (Strongly agree).

   – Mandatory.

Q33 Why?

   – Free-text field.

   – Mandatory.

Q34 With which tool were the warnings easier to understand?

- – Single-answer question.
- – Choices:
  - ∗ Tool 1
  - ∗ Tool 2
  - ∗ Neutral
- – Mandatory.

Q35 With which tool were the warnings easier to correct?

- – Single-answer question.
- – Choices:
  - ∗ Tool 1
  - ∗ Tool 2
  - ∗ Neutral
- – Mandatory.

Q36 With which tool were the warnings faster to understand?

- – Single-answer question.
- – Choices:
  - ∗ Tool 1
  - ∗ Tool 2
  - ∗ Neutral
- – Mandatory.

Q37 With which tool were the warnings faster to correct?

- – Single-answer question.
- – Choices:
  - ∗ Tool 1
  - ∗ Tool 2
  - ∗ Neutral
- – Mandatory.

Q38 With which tool did you have a better grasp on the application code?

- – Single-answer question.
- – Choices:
  - ∗ Tool 1
  - ∗ Tool 2
  - ∗ Neutral
- – Mandatory.

Q39 Which tool would you rather use to correct data leaks?

- – Single-answer question.

- Choices:
  * Tool 1
  * Tool 2
  * Neutral
- Mandatory.

Q40 How likely is it you would recommend tool 1 over tool 2 to a friend?

- Likert scale from 0 (Not at all) to 10 (Go for it!).
- Mandatory.

Q41 How likely is it you would recommend tool 2 over tool 1 to a friend?

- Likert scale from 0 (Not at all) to 10 (Go for it!).
- Mandatory.

Q42 What are tool 1's positive points?

- Free-text field.
- Mandatory.

Q43 What are tool 1's negative points?

- Free-text field.
- Mandatory.

Q44 What are tool 2's positive points?

- Free-text field.
- Mandatory.

Q45 What are tool 2's negative points?

- Free-text field.
- Mandatory.

Q46 Which tasks would tool 1 be best suited for?

- Free-text field.
- Mandatory.

Q47 Which tasks would tool 2 be best suited for?

- Free-text field.
- Mandatory.

Q48 Would you change anything about the tools? Other comments?

- Free-text field.
- Mandatory.