Debugging Static Analysis

Lisa Nguyen Quang Do, Stefan Krüger, Patrick Hill, Karim Ali, and Eric Bodden

Abstract—Static analysis is increasingly used by companies and individual code developers to detect and fix bugs and security vulnerabilities. As programs grow more complex, the analyses have to support new code concepts, frameworks and libraries. However, static-analysis code itself is also prone to bugs. While more complex analyses are written and used in production systems every day, the cost of debugging and fixing them also increases tremendously.
To understand the difficulties of debugging static analysis, we surveyed 115 static-analysis writers. From their responses, we determined the core requirements to build a debugger for static analyses, which revolve around two main issues: abstracting from both the analysis code and the code it analyses at the same time, and tracking the analysis internal state throughout both code bases. Most tools used by our survey participants lack the capabilities to address both issues.
Focusing on those requirements, we introduce VISUFLOW, a debugging environment for static data-flow analysis. VISUFLOW features graph visualizations and custom breakpoints that enable users to view the state of an analysis at any time. In a user study on 20 static-analysis writers, VISUFLOW helped identify 25% and fix 50% more errors in the analysis code compared to the standard Eclipse debugging environment.

Index Terms—Testing and Debugging, Program analysis, Development tools, Integrated environments, Graphical environments, Usability testing.

1 Introduction

In 2017, Google took down over 700,000 malicious applications that were submitted to their Play Store [1], 99% of which were removed before anyone could install them. One of the techniques used in the vetting process of app stores –and by some third-party code developers is static code analysis, a method of automatically reasoning about the runtime behaviour of a program without running it. As more complex applications are produced, more complex analyses are also created to efficiently detect bugs and security vulnerabilities in those applications, hereafter referred to as the *analyzed code*. An error in the code of the analyses 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 [2].

Prior static-analysis research has yielded many novel algorithms [3], [4], [5], analyses [6], [7], and analysis tools [8], [9] to better support code developers and app stores. However, standard debugging tools [10], [11], [12] are often ill-suited to help static analysis writers debug their analyses. Debugging static analysis comes with its own set of challenges, e.g. abstracting how the analysis code interprets the code it analyses and how the two code bases interact with each other. Analysis writers have to handle specific corner cases in two different code bases while also ensuring soundness and precision. Such tasks can be hard and time consuming, making the development of new analyses cumbersome in academia and industry.

• *Lisa, Nguyen Quang Do, and Stefan Krüger, and Patrick Hill are with Paderborn University. Emails: lisa.nguyen@upb.de, stefan.krueger@upb.de, pahill@campus.uni-paderborn.de*
• *Karim Ali is with University of Alberta. Email: karim.ali@ualberta.ca*
• *Eric Bodden is with Paderborn University & Fraunhofer IEM. Email: eric.bodden@upb.de*

To provide better tools for developing and debugging static analyses, we conducted a large-scale survey aimed at identifying the specificities of debugging static-analysis code. In this survey, we determine the particular debugging features needed to debug static analysis. The survey identifies (1) common types of static analysis, (2) common bugs in static-analysis code, (3) popular debugging tools used by analysis writers, (4) the limitations of those tools with respect to debugging static-analysis code, and (5) desirable features for a static-analysis debugger.

We have implemented some of the debugging features found in the survey for the most popular development environment —Eclipse, analysis framework —Soot [13]— and analyzed language —Java— from the survey. We present VISUFLOW, a debugging environment for Soot-based static analysis that helps analysis writers better visualize and understand their analysis code while debugging it. A user study with 20 participants shows that the debugging features of VISUFLOW help analysis writers identify 25% and fix 50% more errors in static-analysis code compared to using the standard Eclipse debugging environment. The participants found the debugging features in VISUFLOW more useful than their own tools for debugging static analysis.

This article makes the following contributions:

- A comprehensive survey to motivate the need for better tools to debug static analyses, and identify desirable features that such tooling should provide.
- VISUFLOW, a debugging environment integrated in Eclipse for Soot-based static analysis.
- A user study to evaluate the usefulness of VISUFLOW for debugging static analyses and determine which of the desirable features that we have extracted from the survey are, in fact, useful for debugging static analysis.

VISUFLOW is available online, along with a video demo, and the anonymized survey and user study datasets [14].

2 SURVEY

To identify useful debugging features for static analysis, we conducted a large-scale survey of 115 static-analysis writers. The goal of this survey is to understand the specific requirements of debugging static-analysis code compared to debugging any kind of code that is not static-analysis code. We ask participants to contrast those two categories, referred to as *analysis code* and *application code*, which can range from small test cases to large, complex systems.

We aim to answer the following research questions:

RQ1: Which types of analysis are most commonly written?
RQ2: Do analysis writers think that analysis code is harder/easier to debug than application code, and why?
RQ3: Which errors are most frequently debugged in analysis code and application code?
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?

2.1 Survey Design

The survey contains 32 questions that we refer to as **Q1**–**Q32**, in the order in which they were presented to participants. In this section, we omit questions that are not relevant to the study presented in this article. We group the survey questions into the following 8 sections:

1) **Participant information (RQ1):** Through multiple-choice questions, we asked participants for their affiliation (academia/industry) (**Q1**), how long they have been writing analyses (**Q3**), for which languages (**Q4**), and with which branches (**Q6**) and frameworks (**Q9**) of static analysis that they have worked with.
2) **Debugging analysis code compared to application code (RQ2): Q11** asks participants which type of code is easier to debug on a scale from 1 (application code) to 10 (analysis code). **Q12** asks them why, in free text.
3) **Debugging analysis code (RQ2–RQ3): Q13** asks participants how long they spend on writing analysis code compared to debugging it on a scale from 0 (100% coding, 0% debugging) to 10 (0% coding, 100% debugging). In free text, **Q15** asks for the typical causes of bugs that they find in analysis code.
4) **Tools for debugging analysis code (RQ4–RQ5):** In free text, we asked participants which features of their coding environments they like (**Q17**), dislike (**Q18**), and would like to have (**Q19**) to debug analysis code.
5) **Debugging application code (RQ2–RQ3): Q20** and **Q21** mirror **Q13** and **Q15**, for application code.
6) **Tools for debugging application code (RQ4–RQ5): Q23**–**Q25** mirror **Q17**–**Q19**, for application code.
7) **Specific debugging features (RQ5): Q26** asks participants to rate the importance of some debugging features on the following scale: Not important - Neutral - Important - Very important - Not applicable.
8) **Coding environment (RQ4): Q28** asks participants if they primarily use a text editor (e.g., Vim, Emacs) or an IDE (e.g., Eclipse, IntelliJ). **Q29** asks for the specific software, in free text.

Pilot Survey: We sent a pilot survey to 10 participants and asked them for feedback about length, quality, and understandability. As a result we shortened the survey from 48 to 32 questions and grouped together the questions about analysis code (becoming sections 3–4 in the survey) and application code (becoming sections 5–6 in the survey).

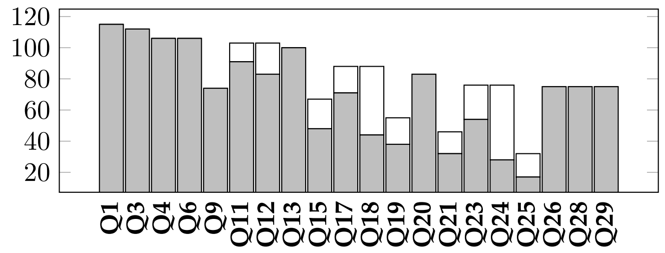


Fig. 1: Number of valid answers (gray) and "Others" answers (white) per question.

2.2 Result Extraction

We manually classified the answers to the free-text questions using an open card sort [15]. Two authors classified the answers into various categories, which were derived during the classification process. Responses that do not answer the question were classified in an "Others" category (e.g., "n/a"). We received a higher rate of answers that we classified into "Others" in the second half of our survey, due to the similarities of survey sections 3–4 and 5–6. In the latter sections, some participants were confused by the familiar questions and did not notice that they now pertained to application code. Answers such as "Already answered the question earlier" were thus classified into the "Others" category.

To verify the validity of our classification, another author –who had not been part of the classification phase– sorted the answers in the categories derived during the first classification. We then compared the agreement between the two classifications. Since one answer could match multiple categories (e.g., "I use breakpoints and stepping." matches both "Breakpoint" and "Stepping"), we calculated a percent agreement for each category of each question. The average percent agreement over all categories for all questions is 96.3% (median = 98%, min = 65.2%, max = 100%, standard deviation $\sigma = 0.05$). Because of the imbalance in the distribution of the answers, we ran into a paradox of inter-rater agreement [16], making the Cohen's Kappa [17] an unreliable statistic for this survey (average $\kappa = 0.66$, median $\kappa = 0.7$, min = -0.08, max = 1, $\sigma = 0.33$).

Due to optional questions and participants who did not finish the survey, some questions received fewer answers than others. Figure 1 reports the number of valid (gray) and "Others" (white) answers per question. In the following sections, the percentages reported for each question are based on the number of valid answers for the particular question and not on all 115 answers. Participants could choose multiple answers to the same multiple-choice question, and an answer to a free-text question could match multiple categories. Therefore, the percentages for each question may add up to more than 100%. Although in this article, we only report the most popular or relevant answers to the survey questions, we included all answers in our statistical tests, and report on all significant results. All questions and anonymized answers are available online [14].

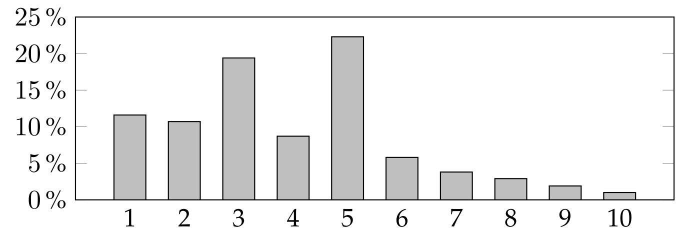


Fig. 2: Ranking the difficulty of debugging static analysis code compared to application code on a scale from 1 (static analysis is harder) to 10 (application code is harder). (**Q11**)

2.3 Participants

We contacted 450 authors of static-analysis papers published between 2014 and 2016 at the following conferences and their co-located workshops: ICSE, FSE, ASE, OOPSLA, ECOOP, PLDI, POPL, SAS. We received responses from 115 researchers, 85.2% from academia and 15.7% from industry (**Q1**). Most participants are experienced static-analysis writers. Approximately 31.3% of the participants have 2–5 years of experience writing static analysis, 22.3% have 5–10 years of experience, 26.8% have more than 10 years of experience, and only 9.8% have less than 2 years of experience (**Q3**).

2.4 Results

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

We asked participants which programming languages they analyze the most (**Q4**), and received 3 main answers: Java (62.3%), C/C++ (59.4%), and JavaScript (23.6%). Less than 8% analyze each of the other 34 named languages.

We also asked participants which branches of static analysis they write for (**Q6**). Data-flow analysis is the most popular (74.5%), followed by abstract interpretation (65.1%), symbolic execution (36.8%), and model checking (21.7%). Less than 4% use each of the 9 remaining categories.

The most popular framework used to write static analysis (**Q9**) is Soot [13] (55.4%), followed by WALA [18] and LLVM [19] as second and third (31.1% and 21.6%, respectively). Less than 10% use each of the 32 other frameworks.

RQ1: Java is the most analyzed programming language. Data-flow analyses are the most common type of static analysis. Soot is the most popular framework.

2.4.2 ***RQ2***: *Do analysis writers think that analysis code is harder/easier to debug than application code, and why?*

Q11 asks participants to rate how hard debugging analysis code is compared to debugging application code on a scale from 1 (analysis code is harder to debug) to 10 (application code is harder to debug). The average ranking is 4.0 (standard deviation $\sigma = 2.1$). Figure 2 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 is confirmed in **Q13** and **Q20** where participants reported that they spent more time debugging a piece of static-analysis code (53.2% of their time) than writing it (46.8%), and the contrary for a piece of application code (57.5% 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).

TABLE 1: Reasons why static analysis is harder to debug than application code (SA) and vice-versa (AC). EQ denotes the reasons why both are equally difficult to debug. (**Q12**)

Harder	**Reason**	**%**
SA	Abstracting two types of code	15.67%
	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%
EQ	Both are application code	13.3%
	They cannot be compared	7.2%
	No opinion	3.6%
AC	Used to developing static analysis	6.0%
	Application code is more complex	2.4%

Table 1 classifies the reasons that participants gave when asked why they found one type of code harder to debug than the other (**Q12**). In this article, we only report the reasons mentioned by more than one participant. The main reason that participants find analysis code harder to debug is the complexity of handling two code bases (i.e., the analysis code and the application code that is being analyzed) at the same time: "Static Analysis requires to switch between your analysis code and the Intermediate Representation which you actually analyse". This complexity creates more corner cases that the analysis writer must handle. Another reason is that correctness is harder to define for a static analysis. To quote a participant: "'correct' is better defined [in application code]". The final reason is that intermediate results of the analysis are not directly verifiable in contrast to the output of application code that can be directly validated: "Static-analysis code usually deals with massive amounts of data. [...] It is harder to see where a certain state is computed, or even worse, why it is not computed."

Participants who find analysis code and application code equally hard to debug have two main arguments. First, both are application code: "a static analyzer is an application, albeit a sophisticated one". Second, they are so different that they cannot be compared: "These two difficulties are qualitatively different and hence incomparable."

Participants who find application code more difficult to debug argue that it is more complex than static-analysis code, and thus contains numerous corner cases: "Static-analysis code usually includes very limited number of possible cases." Some participants also wrote that the reason why they find application code harder to debug is that they are used to developing static analysis.

RQ2: 5.3× more participants found analysis code harder to debug than application code. This is due to three main reasons: handling two code bases simultaneously, correctness requirements for static analysis, and the lack of support for debugging analysis code.

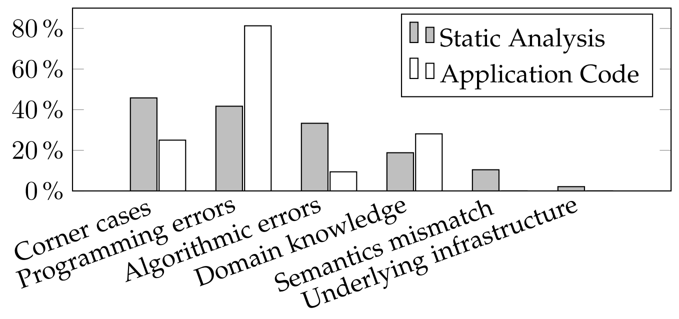


Fig. 3: The root causes of errors found when debugging static analysis and application code. (**Q15** and **Q21**)

2.4.3 ***RQ3**: Which errors are most frequently debugged in analysis code and application code?*

We asked participants for the typical root causes of errors they find when debugging analysis code (**Q15**) and application code (**Q21**), and classified them in the six categories shown in Figure 3. When debugging analysis code, the main cause of errors is handling *corner cases*. This category includes overlooked cases that the developer normally knows of (e.g., "Forgot to consider the effect of certain, rare instructions"). *Domain knowledge* refers to code behaviour that the developer is unaware of (e.g., "Unexpected values returned by an API"). *Programming errors* occur twice as often in application code than in static-analysis code. This category includes implementation errors such as "wrong conditions, wrong loops statements". *Algorithmic errors* contains errors due to a wrong design decision in the program's algorithm (e.g., a "non-convergence" of the analysis), which happens 3.5× more often in analysis code than in application code. *Semantics mismatch* and *underlying infrastructure* are specific to static analysis. The former refers to how the analysis interprets the analyzed code (e.g., "The code does not take [into] account the abstract semantics correctly"). The latter is similar to *domain knowledge*, but instead of the knowledge of the analyzed code, it is about the analysis framework (e.g., "Can't load classes/methods successfully.").

While bugs in application code are mainly due to programming errors, static-analysis bugs are distributed over multiple categories. We attribute this to the heightened interest of analysis writers to produce correct analyses. Testing functional correctness typically requires validating input/output relationships. For analysis code, those relationships are always imperfect due to necessary approximations. Hence, it is hard to define functional correctness for static analysis. Moreover, handling two code bases is also the cause of analysis-specific errors: more corner cases, semantics mismatch and underlying infrastructure. Because of the specific requirements of static analyses, the bugs that developers investigate in application code have different causes compared to analysis code, thus motivating the need for proper support of those specific categories of errors.

RQ3: Analysis code is more often debugged than application code for corner cases, algorithmic errors, semantics mismatch, and unhandled cases in the underlying analysis infrastructure.

TABLE 2: Useful features for debugging static analysis (SA) and application code (AC) for IDE users (IDE) and text editor users (TE). (**Q17** and **Q23**)

	SA/IDE	**SA/TE**	**AC/IDE**	**AC/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	✓			

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

In **Q28** and **Q29**, 56% of the participants answered that to write analysis code, they use an Integrated Development Environment (IDE) such as Eclipse [10] (used by 28%) or IntelliJ [11] (17.3%), while 42.7% use text editors such as Vim [20] (33.3%) or Emacs [21] (21.3%). Each of the other 21 tools is used by less than 10% of the participants.

We asked participants about the most useful features of their coding environments when debugging analysis code (**Q17**) and application code (**Q23**). Table 2 shows the features mentioned by more than one participant. The most popular debugging feature is *Breakpoints*, used by 35.2% of participants when debugging application code and 28.2% for analysis code. *Coding support* (e.g., auto-completion) is appreciated by 29.6% when writing analysis code, and 20.4% for application code. *Variable inspection* is used by 27.8% when writing application code and 19.7% when writing analysis code. *Debugging tools* (e.g., "GDB/JDB") are used by 20.4% when writing application code, and 16.9% for analysis code. 21.1% of the participants *print intermediate results* when debugging analysis code, compared to 13.0% for application code. IDE users highlighted IDE-specific features such as *type checkers*, *stepping*, and *hot-code replacement*.

A $\mathcal{X}^2$ test of independence shows a strong correlation between the type of editor used (IDE or text editor) and the most useful features of the debugging environment ($p = 0.01 \leq 0.05$) for application code. The test does not find such a correlation for analysis code, indicating that the debugging features used when writing analysis code are the same in all types of coding environments.

RQ4: Regardless of the coding environment, analysis writers use the same debugging features to debug analysis code and application code, e.g., breakpoints, variable inspection, coding support, and printing intermediate results.

TABLE 3: Unsatisfactory features when debugging static analysis (SA) and application code (AC) for IDE users (IDE) and text editor users (TE). (**Q18** and **Q24**)

	SA/IDE	**SA/TE**	**AC/IDE**	**AC/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				✗

TABLE 4: Requested features when debugging static analysis (SA) and application code (AC) for IDE users (IDE) and text editor users (TE). (**Q19** and **Q25**)

	SA/IDE	**SA/TE**	**AC/IDE**	**AC/TE**
Graph visualizations	✓	✓		
Omniscient debugging	✓	✓		
Visualizations	✓	✓		
Hot-code replacement	✓		✓	
Coding support			✓	✓
Test generation		✓		
Debugging tools		✓		
Intermediate results	✓			
Conditional breakpoints	✓			
Handling data structures			✓	

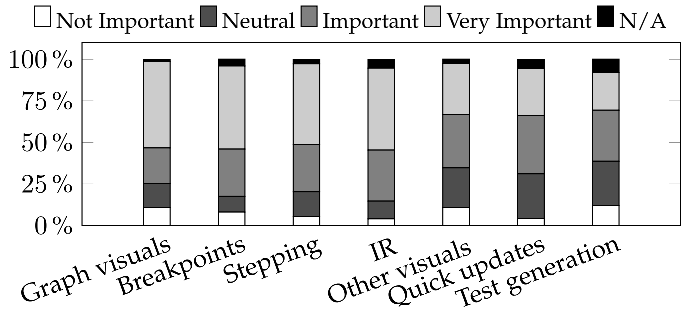


Fig. 4: Ranking the importance of features for debugging static analysis. IR is "Intermediate representation". (**Q26**)

2.4.5 ***RQ5***: *What are the limitations of the existing debugging tools and which features are needed to debug analysis code?*

Q18 and **Q24** ask participants about the features of their coding environments they dislike when debugging analysis code and application code, respectively. Features mentioned by more than one participant are shown in Table 3.

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. This suggests that although current tools are useful, analysis writers 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 along that path.” Therefore, debugging tools for static analysis could be improved by showing more of the *intermediate results* of the analysis. For application code, participants requested more support for handling *different systems* and environments. Participants complained about the “manual work to setup complex build/test systems” and “Dealing with an external dependency [...] that I cannot control”. 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* (e.g., “Missing an easy way to add a breakpoint when the analysis reaches a certain line in the input program (hence having to re-run an analysis)”).

RQ5-1: Current static-analysis debugging tools lack important features such as showing intermediate results, providing clear visualizations of the analysis, and special breakpoints.

To identify which debugging features would best support static-analysis writers, we asked participants to suggest useful features for debugging analysis code (**Q19**) and application code (**Q25**). Table 4 shows the features that are mentioned more than once. The requested debugging features for application code and analysis code are quite different. To write application code, participants requested better *hot-code replacement* and *coding support* (e.g., “better support to record complex data coming from external services”). For analysis code, 18.4% of participants asked for better *visualizations* of the analysis constructs, and 23.7%, for *graph visualizations*: “Easier way to inspect ‘intermediate’ result of an analysis, easier way to produce state graphs and inspect them with tools.” *Omniscient debugging* was 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 requested better *test generation tools* and *special breakpoints* (**RQ5-1**).

A $\mathcal{X}^2$ test on the features of Table 4 shows a correlation between the features requested by participants and the type of code (analysis code or application code) ($p = 0.04 \leq 0.05$), motivating the need for specific tooling for debugging static analysis code in particular. The same test did not yield significant p-values in the case of debugging features the users liked/disliked, indicating that they use the same debugging features for analysis code and application code.

In **Q26**, participants evaluate the importance of the desirable debugging features. Figure 4 shows that *graph visuals*, access to the *intermediate representation* count as very important features, along with *breakpoints* and *stepping* functionalities that consider both the analysis code and the analyzed code. *Other types of visuals*, better *test generation*, and *quick updates* are considered important.

RQ5-2: The most important features for debugging analysis code are (graph) visualizations, access to the intermediate representation, omniscient debugging, and special breakpoints.

2.5 Summary and Discussion

Our survey shows that writing static analysis entails specific requirements on the writer. Handling two code bases and defining soundness makes analysis code harder to debug than general application code (**RQ2**). Those requirements cause different types of bugs to be of interest to static-analysis writers when debugging (**RQ3**). To debug their code, analysis writers mainly use the traditional debugging features included in their coding environments such as breakpoints and variable inspection (**RQ4**). While those tools are helpful, they are not sufficient to fully support them: debugging features such as simple breakpoints fall short and force analysis writers to handle parts of the debugging process manually (**RQ5-1**).

Table 2 shows that the debugging tools that analysis writers currently use are adapted for more general application code. Table 4 shows that the features needed to debug analysis code are quite different from the features needed to debug application code. Those features revolve around improving the visibility of how the analysis code represents and analyses the analyzed code. The desirable features for debugging analysis code are (**RQ5-2**):

- *Graph visuals* to represent the analysis' constructs.
- Access to the *intermediate results* generated by the analysis at any point in the analyzed code.
- Access to the *intermediate representation* of the analyzed code, i.e., the representation of the analyzed code that is manipulated by the analysis code.
- *Other visualizations* that give visibility of how the analysis handles the analyzed code.
- *Breakpoints* and *stepping* which can be controlled from both code bases so that the execution can be stopped at particular points of the analysis code for particular statements of the analyzed code.
- *Omniscient debugging* to be able to track the intermediate results of the analysis back-in-time.
- *Automated test generation* to verify the results of a change in the analysis.
- *Quick updates* to recompute the analysis results on-the-fly when a modification in the analysis code or the analyzed code occurs.

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) ($p = 0.02 \leq 0.05$), which suggests that the desirable features are generalizable to the different types of static analysis and analysis writers represented by our participants, and that the set of desirable features only differs according to whether the analysis writer uses a text editor or an IDE.

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 code bases that interact with each other, e.g., testing or meta-programming. However, those branches have their own specificities and their own challenges in 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 those features can be applied to other fields of software engineering.

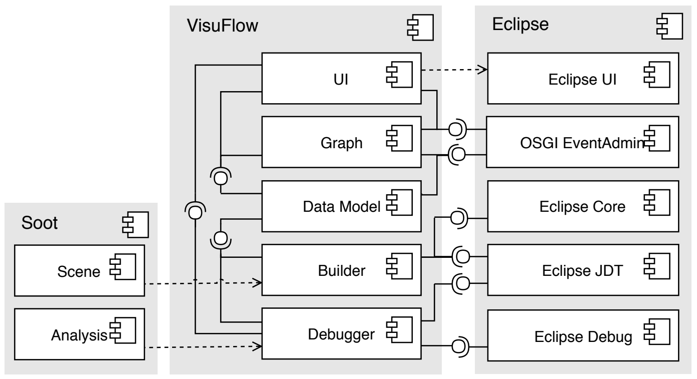


Fig. 5: Component diagram of VISUFLOW.

3 VISUFLOW: VISUAL SUPPORT FOR DEBUGGING DATA-FLOW ANALYSIS IN ECLIPSE

We have implemented VISUFLOW, a debugging tool for static analysis following the features we described in Section 2.5. To reach the largest possible target audience, the implementation is made specific to the most popular use case: debugging data-flow analysis of Java programs using the Soot analysis framework [13] (**RQ1**) in the Eclipse IDE (**RQ4**). A video demo of VISUFLOW is available online (https://www.youtube.com/watch?v=BkEfBDwiuH4).

3.1 Architecture of VisuFlow

In this section, we discuss how we adapted the desirable features for debugging static analysis to our use case. Figure 5 illustrates the component diagram of our implementation. The three main components are the Soot framework, the Eclipse platform, and VISUFLOW, an Eclipse plugin that acts as a buffer between the IDE and the analysis framework to provide debugging support to the analysis writer 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` (a `Unit` is a statement in the analyzed code). `VFEdge` and `VFNode` 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).

The data model (except the in-sets and out-sets) is updated every time the analyzed project is built: when Eclipse Core's `IncrementalProjectBuilder` is called, it 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 is available from Soot's own data model: the Soot `Scene`. VISUFLOW taps into the `Scene` and populates its own data model by wrapping around the Soot constructs with its own. The Soot model is generated from the Jimple intermediate representation, which is then propagated into the VISUFLOW data model.

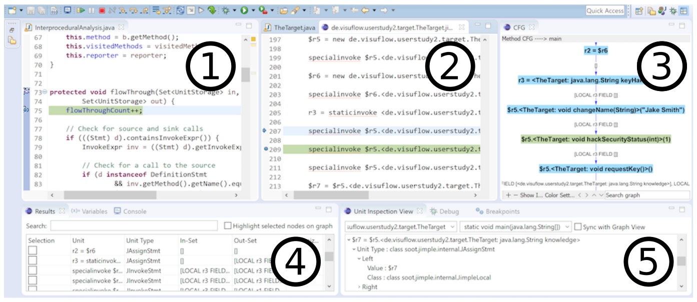
The in-sets and out-sets are populated at run-time, when the analysis writer runs the analysis on the analyzed code. Eclipse's debug component is run, which calls VISUFLOW's debug component through `ILaunchConfigurationDelegate2`. VISUFLOW then runs the analysis using Soot, and retrieves the in-sets and out-sets by instrumenting the transfer function of the analysis. As the analysis is executed, 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 is used to instrument the transfer function of the analysis such that, at the beginning of the transfer function, it sends the values present in the in-set and, at the end, 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 writer. 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 visuals* (CFGs and call graphs), *access to intermediate results* (in-sets and out-sets), *access to the intermediate representation* (`VFUnit` in Jimple format).

Dual breakpoints and stepping: Traditional debuggers allow static-analysis writers to set breakpoints in the analysis code only. When a user needs to debug an analysis at a specific statement of the analyzed code, they have to use conditional breakpoints to suspend the analysis for that specific statement. This is quite limiting, as it requires the user to know in advance which statements need inspection. VISUFLOW allows analysis writers to set breakpoints in both the analysis code and the analyzed code, allowing them to stop the execution for specific statements of the analyzed code. This feature is achieved by extending Eclipse's breakpoints to simulate a second set of breakpoints in the analyzed code. Breakpoints in the analyzed code are transformed into conditional breakpoints in the transfer function of the analysis code. If the user had previously set breakpoints in the transfer function, a condition is added 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 will create one at the start of the transfer 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 code bases. In the analyzed code, this is achieved by adding breakpoints at the successor statements of the currently examined statement and removing them later.

Dual breakpoints and *stepping* events are set through the Eclipse JDT, which transmits an `IJavaModel` of the project under execution to the VISUFLOW debugger and builder. When new in-set and out-set information are generated at execution time, VISUFLOW populates the data model and –if breakpoints are set– updates its UI views to highlight the statements which the user steps through.

Unimplemented features: VISUFLOW implements the debugging features that the survey participants marked as most important (**RQ5-2**) for a use in an IDE (Figure 4). Other features such as *omniscient debugging*, *automated test generation* and *quick updates* constitute significant bodies of work which have been widely researched in the past [4], [7], [22], [23], and whose integration we keep for future work.


InterproceduralAnalysis.java
TheTarget.java
de.visuflow.userstudy2.target.TheTarget.ji...
CFG
Method CFG ----> main
r2 = $r6
r3 = <TheTarget: java.lang.String keyHa
[LOCAL r3 FIELD []]
$r5.<TheTarget: void changeName(String)>("Jake Smith")
[LOCAL r3 FIELD []]
$r5.<TheTarget: void hackSecurityStatus(int)>(1)
[LOCAL r3 FIELD []]
$r5.<TheTarget: void requestKey()>()
IELD [<de.visuflow.userstudy2.target.TheTarget: java.lang.String knowledge>], LOCAL
Show I... Color Sett...
Search graph
Quick Access
Results
Variables
Console
Search:
Highlight selected nodes on graph
Unit Inspection View
Debug
Breakpoints
suflow.userstudy2.target.TheTarget
static void main(java.lang.String[])
Sync with Graph View
1
2
3
4
5

```
    this.method = b.getMethod();
    this.visitedMethods = visitedMet
    this.reporter = reporter;
}


protected void flowThrough(Set<UnitStorage> in,
        Set<UnitStorage> out) {
    flowThroughCount++;

    // Check for source and sink calls
    if (((Stmt) d).containsInvokeExpr()) {
        InvokeExpr inv = ((Stmt) d).getInvokeExp

        // Check for a call to the source
        if (d instanceof DefinitionStmt
                && inv.getMethod().getName().equ
```

```
$r5 = new de.visuflow.userstudy2.target.The

specialinvoke $r5.<de.visuflow.userstudy2.t

$r6 = new de.visuflow.userstudy2.target.The

specialinvoke $r6.<de.visuflow.userstudy2.t

r3 = staticinvoke <de.visuflow.userstudy2.t

specialinvoke $r5.<de.visuflow.userstudy2.t

specialinvoke $r5.<de.visuflow.userstudy2.t

specialinvoke $r5.<de.visuflow.userstudy2.t

$r7 = $r5.<de.visuflow.userstudy2.target.Th
```

Selection	Unit	Unit Type	In-Set	Out-Set
☐	r2 = $r6	JAssignStmt	[]	[]
☐	r3 = staticinvok...	JAssignStmt	[]	[LOCAL r3 FIE
☐	specialinvoke $r...	JInvokeStmt	[LOCAL r3 FIELD...	[LOCAL r3 FIE
☐	specialinvoke $r...	JInvokeStmt	[LOCAL r3 FIELD...	[LOCAL r3 FIELD...

```
˅ $r7 = $r5.<de.visuflow.userstudy2.target.TheTarget: java.lang.String knowledge>
  ˅ Unit Type : class soot.jimple.internal.JAssignStmt
    ˅ Left
        Value : $r7
        Class : class soot.jimple.internal.JimpleLocal
    › Right
```

Fig. 6: The Graphical User Interface of VISUFLOW. Features 1–5 are detailed in Section 3.2.

3.2 Graphical User Interface of VisuFlow

Figure 6 presents VISUFLOW's Graphical User Interface (GUI). Two small-scale studies with three and five Soot users were run during the implementation of VISUFLOW to test its usability and integration into the Eclipse IDE.

Code: VISUFLOW shows the analysis code in a *Java Editor* ① and the analyzed code in the *Jimple View* ② side by side, to help manipulating both code bases at the same time. The analyzed code is shown in both Java and the Jimple intermediate representation.

Graph visuals: To help the user better visualise the structure of the analyzed code, VISUFLOW presents a *Graph View* ③ 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 [24], to add zooming and panning features, along with a search bar that allows the user to locate a particular statement, and tooltip information. The graphs can be customized through node drag-and-drop and coloring. To draw the graphs, we use the GraphStream framework [25], because it scales up to a large number of nodes and edges.

Other visuals: A set of different views displays various information that can be used to help reason about the analysis code. The *Results View* ④ provides a compact summary of the *Graph View*, with search and filter options: it details the intermediate information for each application-code statement and also allows users to mark specific statements with custom tags. The *Unit Inspection View* ⑤ 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 are updated using the data model and have been enhanced with search and filtering options.

Integration with Eclipse: During early evaluations 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 navigation menus and highlighting features to allow users to switch between views more smoothly.

The top toolbar provides easy access to VISUFLOW's functionalities, such as: re-building 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 code bases. To stay consistent with Eclipse's usability paradigms, the dual 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.

3.3 Generalization of the approach

Based on the implementation of VISUFLOW, we describe a high-level framework for applying the main desirable debugging features described in Section 2.5 to different types of static analysis for different environments.

1) *Define a data model:* the data model represents the internal state of the analysis that is typically not exposed to the analysis writer. 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 transfer 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 dual breakpoint system.

3.4 Reusing VISUFLOW

The constructs of VISUFLOW's data model (call graph, CFGs, in-sets and out-sets) are specific to data-flow analysis. 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 [18], the IFDS/IDE solver Heros [3] or IDEal [26], if it can expose its data model and transfer functions to the VISUFLOW builder and debugger components.

4 USER STUDY

We conducted a user study to evaluate how useful VISUFLOW is as a debugging tool for static analysis through answering the following research questions:

RQ6: Which features of VISUFLOW are most useful?
RQ7: Does VISUFLOW help identify and fix more errors compared to Eclipse's debugging environment?
RQ8: Does VISUFLOW help understand and debug analyses better than other debugging environments?

4.1 Setup

We evaluate how users interact with VISUFLOW compared to the standard Eclipse debugging environment [10] (hereafter, referred to as Eclipse). Each participant performed two tasks: they debugged one static analysis with VISUFLOW and one with Eclipse. In the latter case, participants had access to the Eclipse debugging functionalities such as: breakpoints, variable view, and stack frame view. We also provided 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 given analyzed programs does not compute the correct results. For each task, a participant had 20 minutes to identify and fix as many errors as possible in the analysis code. 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, participants performed training tasks with a demo analysis to familiarize themselves with the tools. We calibrated the difficulty of the tasks through a pilot study conducted on 6 participants.

During the two tasks, we counted the number of errors that participants identified and fixed. We also logged how long the mouse focus was for each view of the coding environment to measure the time spent using each view. Afterwards, participants were asked to fill a comparative questionnaire of the two debugging environments, followed by a short interview of their impressions of the tools. The full-text answers presented in Section 4.3.3 were categorized by two authors in an open card sort [15]. 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 [17]. The questionnaire, the anonymized answers, and the user study results are available online [14].

4.2 Participants

The twenty participants of our user study (referred to as **P1–P20**) are of diverse backgrounds: researchers in academia (65%), researchers in industry (5%), and students (30%). Eleven participants have less than a year experience writing static analysis, 6 have 1–5 years of experience, and 3 have more than 5 years of experience.

Participants rated their familiarity of data-flow analysis, Eclipse, and Soot on a scale from 0 (novice) to 10 (expert). The average score is 5.7 (min: 1, max: 9) for data-flow analysis, 5.9 (min: 2, max: 8) for Eclipse, and 3.3 (min: 0, max:

TABLE 5: Main features of VISUFLOW and Eclipse that were used, and the average time spent using each feature.

	VISUFLOW		Eclipse	
	#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

7) for Soot. We thus gathered a variety of both novice and expert users in data-flow analysis and Eclipse. However, expert Soot users are rare. Only **P7** and **P8** participated in our initial survey whose results motivate the main features of VISUFLOW. We have not observed any significant difference between their results and other participants' results.

4.3 Results

4.3.1 ***RQ6***: *Which features of* VISUFLOW *are most useful?*

Table 5 shows the number of participants who used the features of VISUFLOW and Eclipse, and the median time 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* was also often used, showing that access to the intermediate representation is helpful when debugging static analysis. 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 code. Instead, they spent this time using the *Graph View*, the *Results View*, and the *Variable Inspection View*. This 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. We attribute this to VISUFLOW's dual breakpoints which allow users to step through both code bases simultaneously, sparing them the effort of writing conditional breakpoints in the *Breakpoints View*.

The *Unit View* was only used by 3 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, their Net Promoter Scores (NPS) [27], and the tool features that they used the most, suggesting that those results are generalizable to all user groups represented by our participants.

TABLE 6: Number of errors identified (I) and fixed (F) with Eclipse (E) and VISUFLOW (V) by each participant.

	Task 1 (E)		Task 2 (V)			Task 1 (V)		Task 2 (E)	
	I	F	I	F		I	F	I	F
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		17	11	10	7

RQ6: Graphs, access to the intermediate results and to the intermediate representation, and dual breakpoints, are the most used features in VISUFLOW (in this order).

4.3.2 ***RQ7***: *Does* VISUFLOW *help identify and fix more errors compared to Eclipse's debugging environment?*

Table 6 reports the number of errors identified and fixed by each participant. An error is *identified* when a participant could explain why it occurred in the analysis code. For Task 1, participants identified and fixed 1.4× more errors with VISUFLOW than with Eclipse. In particular, 17 errors were identified and 11 were fixed 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. 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 20 participants are Eclipse users, making the learning curve for the Eclipse tool less steep than for VISUFLOW. Despite this factor, 7 of those 12 participants found and fixed more errors with VISUFLOW than with their original debugging environment.

RQ7: Using VISUFLOW, participants identified 25% and fixed 50% more errors than with Eclipse.

4.3.3 ***RQ8***: *Does* VISUFLOW *help understand and debug analyses better than other debugging environments?*

After performing the tasks, participants filled out a comparative questionnaire to assess the perceived usefulness of the two debugging environments. They rated VISUFLOW, Eclipse, and their own debugging environment through an NPS [27], evaluated how VISUFLOW and Eclipse helped them perform the required tasks, and identified their preferred debugging features.

Overall, VISUFLOW was positively received. In the NPS questions, the 20 participants rated their likelihood of recommending a debugging environment over another one to a friend. VISUFLOW has a mean NPS score of 9.1 out of 10 (standard deviation $\sigma = 1.1$) compared to Eclipse, and 8.3 ($\sigma = 1.7$) compared to the participant's own debugging environment. Eclipse has a mean score of 1.4 ($\sigma = 1.6$) compared to VISUFLOW, and 3.4 ($\sigma = 3.3$) compared to the participant's own debugging environment.

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 4 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 1 participant preferred Eclipse, and 2 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 static analysis.

When asked what they would use both debugging environments for, 16 participants wrote they would use VISUFLOW to write and debug static analysis ("[I would use VISUFLOW for] visualising 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".

Participants were asked which features of VISUFLOW and Eclipse they would like to have in their own debugging environments. Three participants liked Eclipse's integrated debugger, which echoes our survey findings (Table 2). Ten participants asked for the *Graph View* ("visualising 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 find useful the most confirm our survey findings (**RQ5-2**), and match the participants' behaviour (**RQ6**).

Novice analysis writers 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."

RQ8: Participants find VISUFLOW more useful than Eclipse and than their own tools 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.

5 THREATS TO VALIDITY

In our survey, we reached out to 450 authors of static-analysis papers published between 2014 and 2016. While we covered a large chunk of active experts in the field, it would be interesting to contact other static-analysis writers whom we missed.

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.

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 times reported in Table 5 represent the use times of the different views of VISUFLOW and Eclipse. They are not exact, since participant attention may be divided between multiple views while the mouse can focus on only one of them. We argue that, for 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, the relative difference between the times spent in each view is still a reliable metric.

6 DISCUSSION AND FUTURE WORK

Our survey collects extensive data that we have not used to its full extent in this article, not only about debugging features for static analysis, but also about debugging features for general application code, motivations for writing static analysis (**Q5**), types of analysis written by participants (**Q2**, **Q7**, **Q8**), detailed analysis examples (**Q10**), reasons why participants debug static analysis (**Q14**), and why participants use a particular debugging environment (**Q30**). We have made our data available online for others to use [14].

Some of the debugging features identified in our survey can also be applied to debug other fields of software engineering that handle two interacting code bases. It would be interesting to explore how to support such use cases.

We designed VISUFLOW as a proof of concept to confirm the usefulness of some of the features identified in our survey. However, its current implementation does not gracefully scale up for two reasons: large graphs clutter the interface, and long update times when re-running complex analyses. We plan to address both issues in future work. It would also be interesting to integrate more of the debugging features found in the survey (e.g., omniscient debugging and quick updates) in debugging tools like VISUFLOW. VISUFLOW is available online [14], and we encourage contributions by other researchers and practitioners.

7 Related Work

Debugging static analysis has not been a major topic in the software engineering community. In this section, we discuss prior work on visualizing static-analysis information, existing debugging tools and techniques, and the usability of static-analysis tools.

7.1 Debugging Static Analysis

We are not aware of any tool that is tailored to address issues specific to debugging static analysis. In past work, Andreasen et al. [28] 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. Other tools also provide a subset of the features in VISUFLOW, especially in terms of visualization of information and data flows. For example, Atlas [29] visualizes data-flow paths based on the abstract syntax tree (AST) of a given program. To improve user understanding and evaluating error reports, Phang et al. [30] present a tool that visualizes program paths to help the user track where an error originates from. Unlike VISUFLOW, none of these tools enable static-analysis writers to debug their own analyses, but are rather tailored to the users of static-analysis tools (e.g., code developers), and therefore focus more on visualization features than debugging features.

7.2 Standard Debugging Tools

Most programming languages' runtime environments are shipped with debuggers provided by the language maintainers (e.g., GDB [12]). Many IDEs, such as Eclipse [10] and IntelliJ [11], integrate debugging functionalities for major programming languages natively in their tool sets. Since VISUFLOW is integrated into Eclipse, it uses all available features such as breakpoints and stepping. As our survey and user study have shown, such tools are designed for general application code, and do not have specific support for static analysis. There exist more complex debugging techniques such as delta debugging [31], omniscient debugging [22], and interrogative debugging [32]. However, they are not integrated into the most commonly used debugging tools.

7.3 Usability of Static-Analysis Tools

To our knowledge, this article presents the first large-scale survey of static-analysis developers. Most of the prior surveys of developers are targeted towards static-analysis users instead of writers. Ayewah et al. [33] present a survey of FindBugs [9] users to determine their usage habits and how they deal with the displayed warnings. The authors conclude that static-analysis tools have been widely adopted by their participants, but are not used regularly, and without customization. Christakis and Bird [34] asked 375 developers within Microsoft about their attitudes towards static-analysis tools. The features that participants deemed most important include: better usability, better responsiveness, and pre-configured prioritization of security and best-practice aspects when it comes to error reporting. Johnson et al. [35] investigated the warning and error reports of the static analyzer FindBugs, the Eclipse Java Compiler, and the code-coverage tool Eclemma [36]. Nguyen Quang Do et al. [7] evaluated the impact of their Just-in-Time static analysis on the workflow of developers who use such tools to detect errors in their code. Phang et al. [30] also tested their program-path visualization tool through a user study. However, those surveys and user studies aim to assess the usability of static analyzers from the end-user perspective. This article—based on our survey—presents requirements for a debugging tool for static-analysis writers, and then assesses the usability of such a tool.

8 Conclusion

Writing and debugging static analysis is a difficult task. We surveyed 115 static-analysis writers from different backgrounds and show that current debugging tools are not always sufficient to properly support the development of static analysis. In this article, we report the main causes of bugs in static analysis, show the major tool features used by analysis writers to debug their analyses, discuss their limitations, and identify features that would best support debugging static analysis.

We present VISUFLOW, a debugging environment designed specifically for debugging static analysis, including features that we identified in our survey. 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, confirming our survey's findings, and validating the usefulness of its debugging features.

This work can be used to design better tool support for debugging static analysis and help analysis writers to secure application code.

Acknowledgments

We thank the participants of our surveys and user studies for their invaluable input. We would also like to thank Michael Schlichtig, Henrik Niehaus, Shashank Subramanya, Kaarthik Radhakrishna, Zafar Syed, Nishitha Shivegowda, Yannick Kouotang Signe, and Ram Muthiah Bose Muthian for their work on VISUFLOW. This research was supported by a Fraunhofer Attract grant as well as the Heinz Nixdorf Foundation. This work has also been partially funded by the DFG as part of project E1 within the CRC 1119 CROSSING, and was supported by the Natural Sciences and Engineering Research Council of Canada.

References

[1] A. D. Blog, "How we fought bad apps and malicious developers in 2017," https://www.appbrain.com/stats/number-of-android-apps, 2018.

[2] Appbrain, "Number of android applications," https://www.appbrain.com/stats/number-of-android-apps, 2018.

[3] E. Bodden, "Inter-procedural data-flow analysis with IFDS/IDE and Soot," in *International Workshop on State of the Art in Java Program Analysis (SOAP)*, 2012, pp. 3–8. [Online]. Available: http://doi.acm.org/10.1145/2259051.2259052

[4] B. G. Ryder, "Incremental data flow analysis," in *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '83. New York, NY, USA: ACM, 1983, pp. 167–176. [Online]. Available: http://doi.acm.org/10.1145/567067.567084

[5] G. Liang, Q. Wu, Q. Wang, and H. Mei, "An effective defect detection and warning prioritization approach for resource leaks," in *Computer Software and Applications Conference (COMPSAC)*, 2012, pp. 119–128.

[6] J. Xie, H. Lipford, and B.-T. Chu, "Evaluating interactive support for secure programming," in *Conference on Human Factors in Computing Systems (CHI)*, 2012, pp. 2707–2716. [Online]. Available: http://doi.acm.org/10.1145/2207676.2208665

[7] N. Q. D. Lisa, A. Karim, L. Benjamin, B. Eric, S. Justin, and M.-H. Emerson, "Just-in-time static analysis," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2017. New York, NY, USA: ACM, 2017, pp. 307–317. [Online]. Available: http://doi.acm.org/10.1145/3092703.3092705

[8] A. J. Ko and B. A. Myers, "Barista: An implementation framework for enabling new tools, interaction techniques and views in code editors," in *Conference on Human Factors in Computing Systems (CHI)*, 2006, pp. 387–396.

[9] H. Shen, J. Fang, and J. Zhao, "Efindbugs: Effective error ranking for findbugs," in *International Conference on Software Testing, Verification and Validation (ICST)*, 2011, pp. 299–308.

[10] "Eclipse," https://eclipse.org/, 2018.

[11] "Intellij," https://www.jetbrains.com/idea/, 2018.

[12] "Gdb: The gnu project debugger," https://www.gnu.org/software/gdb/, 2018.

[13] R. Vallée-Rai, E. Gagnon, L. J. Hendren, P. Lam, P. Pominville, and V. Sundaresan, "Optimizing Java bytecode using the Soot framework: Is it feasible?" in *CC*, 2000, pp. 18–34. [Online]. Available: http://dx.doi.org/10.1007/3-540-46423-9_2

[14] N. Q. D. Lisa, K. Stefan, H. Patrick, A. Karim, and B. Eric, "Debugging static analysis," https://arxiv.org/abs/1801.04894, 2018.

[15] W. Hudson, "Card sorting," 2013.

[16] S. Viswanathan and I. B. Shrikant, "Observer agreement paradoxes in 2x2 tables: Comparison of agreement measures," in *BMC Medical Research Methodology*, 2014. [Online]. Available: https://bmcmedresmethodol.biomedcentral.com/articles/10.1186/1471-2288-14-100

[17] J. R. Landis and G. G. Koch, "The measurement of observer agreement for categorical data," in *Biometrics*, 1977, pp. 159–174.

[18] "Wala," http://wala.sourceforge.net/wiki/index.php, 2018.

[19] "Llvm," http://llvm.org/, 2018.

[20] "Vim," http://www.vim.org/, 2018.

[21] "Emacs," https://www.gnu.org/software/emacs/, 2018.

[22] B. Lewis, "Debugging backwards in time," *CoRR*, vol. cs.SE/0310016, 2003. [Online]. Available: http://arxiv.org/abs/cs.SE/0310016

[23] J. Rushby, "Verified software: Theories, tools, experiments," B. Meyer and J. Woodcock, Eds. Berlin, Heidelberg: Springer-Verlag, 2008, ch. Automated Test Generation and Verified Software, pp. 161–172. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-69149-5_18

[24] K. Sugiyama, S. Tagawa, and M. Toda, "Methods for visual understanding of hierarchical system structures," *IEEE Transactions on Systems, Man & Cybernetics*, vol. 11, no. 2, pp. 109–125, 1981.

[25] "Graphstream," http://graphstream-project.org/, 2018.

[26] J. Späth, K. Ali, and E. Bodden, "Ideal: Efficient and precise alias-aware dataflow analysis," in *2017 International Conference on Object-Oriented Programming, Languages and Applications (OOPSLA/SPLASH)*. ACM Press, Oct. 2017. [Online]. Available: http://bodden.de/pubs/sab17ideal.pdf

[27] F. F. Reichheld, "The one number you need to grow," *Harvard Business Review*, vol. 81, no. 12, pp. 46–55, 2003.

[28] E. S. Andreasen, A. Møller, and B. B. Nielsen, "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*. ACM, 2017, pp. 31–36.

[29] T. Deering, S. Kothari, J. Sauceda, and J. Mathews, "Atlas: A new way to explore software, build analysis tools," in *Companion Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE Companion 2014. New York, NY, USA: ACM, 2014, pp. 588–591. [Online]. Available: http://doi.acm.org/10.1145/2591062.2591065

[30] Y. P. Khoo, J. S. Foster, M. Hicks, and V. Sazawal, "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*, ser. PASTE '08. New York, NY, USA: ACM, 2008, pp. 57–63. [Online]. Available: http://doi.acm.org/10.1145/1512475.1512488

[31] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Trans. Software Eng.*, vol. 28, no. 2, pp. 183–200, 2002.

[32] A. J. Ko and B. A. Myers, "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.

[33] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh, "Experiences using static analysis to find bugs," *IEEE Software*, vol. 25, pp. 22–29, 2008, special issue on software development tools, September/October (25:5).

[34] M. Christakis and C. Bird, "What developers want and need from program analysis: An empirical study," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. New York, NY, USA: ACM, 2016, pp. 332–343. [Online]. Available: http://doi.acm.org/10.1145/2970276.2970347

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

[36] "Eclemma," http://www.eclemma.org/, 2018.

Lisa Nguyen Quang Do is a doctoral researcher in the 'Secure Software Engineering' group at Paderborn University. She has received her MSc. in Computer Science from EPFL in 2014. Her research focuses on improving the usability of analysis tools for code developers through different aspects that range from the optimization of the analysis algorithm to the implementation of its framework to the usability of its interface.

Stefan Krüger received a Master's Degree in Computer Science at Otto-von-Guericke University in Magdeburg and is a doctoral researcher at Paderborn University, and a member of the collaborative research center CROSSING. CROSSING aims to devise future-proof cryptography that is sound and easy to use even for non-experts. In that vein, Krüger's main research interests are API usability, DSLs for the specification of security properties of programs, and detection of cryptographic-API misuses.

Patrick Hill received his B.Sc. in Computer Science from Technische Universität Braunschweig. He currently studies software engineering at Paderborn University. His main interest is IT security.

Karim Ali is an Assistant Professor in the Department of Computing Science at the University of Alberta. His research interests are in programming languages and software engineering, particularly in scalability, precision, and usability of program analysis tools. His work ranges from developing new theories for scalable and precise program analyses to applications of program analysis in security, Just-in-Time compilers, and high-performance computing.

Eric Bodden is a full professor for Secure Software Engineering at the Heinz Nixdorf Institute of Paderborn University, Germany. He is further the director for Software Engineering at the Fraunhofer Institute for Engineering Mechatronic Systems. Prof. Bodden has been recognized several times for his research on program analysis and software security, most notably with the German IT-Security Price and the Heinz Maier-Leibnitz Price of the German Research Foundation, as well as with several distinguished paper and distinguished reviewer awards.