Designing UIs for Static-Analysis Tools

EVALUATING TOOL DESIGN GUIDELINES WITH SWAN

DANIIL TIGANOV
LISA NGUYEN QUANG DO
KARIM ALI

Past research has shown that static-analysis tools suffer from common usability issues such as a high rate of false positives, lack of responsiveness, and unclear warning descriptions and classifications. Although these tools have grown more complex and their industry usage has spread, those issues have remained prominent.[6,7,9,11,13,15,19,20]

To address the usability issues of static-analysis tools, Lisa Nguyen Quang Do et al.[20] proposed a user-centered approach to designing these tools during the development of the analysis, as opposed to keeping the development of the analysis and its UI (user interface) separate. To this end, they defined 10 guidelines for designing the UI of an analysis tool. The authors extracted those guidelines from existing literature and a study that they have conducted across 17 static-analysis tools and 87 software developers at Software AG. The guidelines consider analysis engine requirements, user behavior, reporting platforms, and the

effects of company policies on the usage and adoption of static-analysis tools.[18]

This article explores the effect of applying this user-centered approach and the design guidelines to SWAN,[26] a security-focused static-analysis tool for the Swift programming language. SWAN is being actively developed to feature better integration into the Swift development workflow, a faster and more precise analysis engine, and a new UI. The purpose of this article is to evaluate the effectiveness of the approach and guidelines for improving the usability of the next version of SWAN.

SWAN is being created to address the lack of openly available static-analysis tools for Swift. It provides users with a CLI (command-line interface) and GUI (graphical user interface) to visualize the results of its static analyses. One of SWAN's goals is to provide immediate analysis results during development, as opposed to running the analysis overnight, as is often done in industry.[9,15]

Multiple elements of SWAN make it an interesting case study for exploring static-analysis tool usability. First, it has a large target audience: *all* Swift iOS developers (in the remainder of this article, the terms *Swift developers* and *iOS developers* are used interchangeably). While this user group spans different profiles, from large organizations to single developers, their varying requirements are tamed by SWAN's immediate, lightweight nature and tight integration into the Swift development workflow. Therefore, how the various types of developers resolve warnings is very similar, as are their UI requirements.

Second, because of its goal to provide immediate results, SWAN has the potential to integrate easily into

developers' workflows, allowing it to support more use cases and usability tests in the future. Third, SWAN is independent from existing analysis platforms. Therefore, its UI can be modified without the external constraints that those platforms would impose, allowing exploration of different UI designs.

ANALYZING SWIFT PROGRAMS

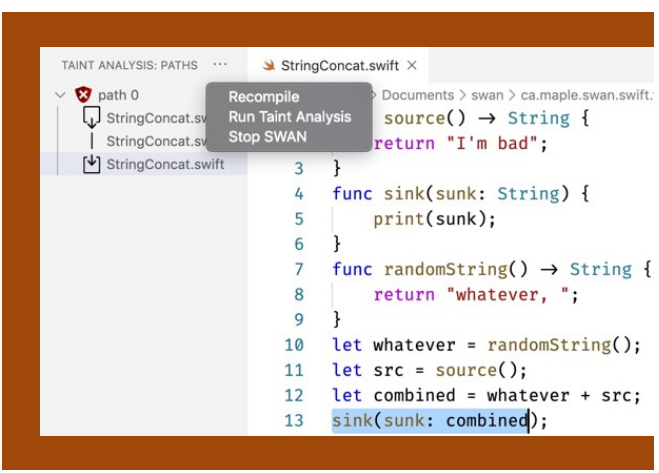
Swift[23] is an open-source programming language and Apple's recommended choice for development on both its mobile operating system, iOS,[3] and desktop operating system, macOS.[4] The web-traffic analysis tool StatCounter estimates that in 2019 iPhones and iPads made up 24.79 percent of mobile devices across the world,[22] and macOS devices accounted for 16.46 percent of desktop machines.[21] Trends also show that the popularity of both operating systems in 2020 has increased by 4.41 percent and 3.82 percent, respectively. Therefore, the ability to conduct static analysis of Swift applications has significant impact on millions of users around the world.

Although many static-analysis frameworks exist for Android devices (e.g., FlowDroid,[5] SCanDroid,[8] and DroidInfer[12]), there is a lack of comparable tools for Swift. While LLVM[16] and Clang[17] support some low-level analyses, they are unsuitable for deeper analyses of Swift applications, such as the precise detection of data leaks. This is because language-specific structures and information are typically lost during the compilation of Swift source code to low-level LLVM IR. Moreover, most of the currently available tools for Swift (e.g., SwiftLint[24] and Tailor[25]) only help enforce Swift best-coding practices.

SWAN bridges the gap between the increasing popularity of Swift and the lack of available analysis tools.[26] This open-source static-analysis framework for Swift primarily targets app developers. It offers both a CLI and a GUI. The CLI enables the developer to integrate SWAN into a continuous integration workflow, providing analysis results at major development milestones. Alternatively, developers can use the GUI to obtain on-demand analysis results directly in VSCode.[28] This relatively immediate feedback helps developers focus on the task at hand, which further helps them fix more bugs in less time.[14]

Figure 1 shows the main GUI elements of the SWAN VSCode extension. The user starts SWAN by selecting the SWAN tab and pressing `Run Taint Analysis` in the sidebar. The extension then automatically attaches to an

FIGURE 1: **THE MAIN GUI OF THE SWAN VSCODE EXTENSION**



existing SWAN instance, if one is running, or starts a new one if none is already running. For the code example in the figure, SWAN displays the results in the sidebar in file tree–like form. Each vulnerable path is an element in the tree marked with a red cross. Its children are the nodes in the path. The first child is the source, the last is the sink, and any nodes in between are intermediates. The user may select any path node, and the extension will open the file at the corresponding source location. In the figure, the user has selected the last node. Therefore, SWAN highlights line 13 in the source file, showing the user that tainted data reaches the function `sink()` as a parameter.

DESIGN GUIDELINES

In past research, Nguyen Quang Do et al.[20] studied the usage of static-analysis tools at Software AG, focusing on code developers' motivations for using the tools (e.g., working context, type of tool), their behavior when facing common usability issues, and the strategies created in-house to address those issues. Their research spans 17 analysis tools, 87 developers, and two large-scale projects, and includes a developer survey and a study of the analysis results and developer responses to those results for one of those projects.

That study found that time constraints are the main influence on how developers interact with static-analysis tools and therefore also influence the strategies they use to optimize their work. For example, to save time, 44 percent of the participants mark warnings as false positives based on the type of issue without investigating the warning further. Similarly, some participants ignore

Participants mostly use static-analysis tools during their spare time.

or suppress a warning if they do not understand its description. In both cases, a different UI might help developers make better decisions (e.g., by providing examples of similar warnings that were already resolved).

As a result, Nguyen Quang Do et al. define 10 guidelines (referred to as G1–G10 in this article) on how to create UIs for static-analysis tools that better support developers. Those guidelines echo usability issues reported in former studies, detailed later.

The following describes the guidelines and notes which of them are applicable to SWAN.

G1 – Consider time constraints when designing the UI processes

Nguyen Quang Do et al. found that participants mostly use static-analysis tools during their spare time (e.g., between meetings), a finding echoed by Carmine Vassallo et al.[27] Since time constraints are the main motivator for using static-analysis tools, all UI interactions must be designed within this context. For example, the tool could help developers determine how long a given warning would take to fix and help them plan their work. SWAN's immediate analysis approach and tight integration into the Swift development workflow may reduce the need for such time-based warning metrics.

G2 – The analysis responsiveness and the tool interface should be crafted to minimize wait times

Waiting for the warnings to update after submitting a fix is one of the main reasons reported by the participants for stopping their work with analysis tools. Because complex

analyses may take a long time to run, the fixing process may be interrupted, splitting a fix over multiple debugging sessions. This usability issue has also been reported as a cause for workflow disruption.[11,13,15] SWAN is being created with responsiveness in mind. Its UI could be designed to display and update immediate warnings, but this presents challenges.

G3 – Feed the developer knowledge back into the analysis

The studies by Nguyen Quang Do et al.[20] and Vassallo et al.[27] observed that programmers develop heuristics on how to interpret analysis results. For example, Software AG engineers built a knowledge of which API calls are not handled well by the analysis, and they dismiss warnings that contain such calls.[20] Istehad Chowdhury and Mohammad Zulkernine[10] have shown that such analysis weaknesses may help distinguish true from false positives. As a result, Nguyen Quang Do et al. advocate for developer knowledge to be fed back into the analysis so that it may profit all other users. SWAN's initial lack of API security modeling can produce many false positives and false negatives; there are steps developers can take in these cases.

G4 – Provide specific warning explanations

As reported in past research, warning descriptions are often generic and require domain-specific knowledge of security vulnerabilities or of the project being analyzed.[6,11,13,15] This can make understanding warnings and figuring out how to fix them a difficult task for developers. Nguyen Quang Do et al.[20] recommend making the warnings

as specific as possible to the particular piece of code that is analyzed, and providinge explanations of what it means in the context of the user (e.g., how difficult it would be to fix and what kind of knowledge is required). Since SWAN aims to accommodate all developers, including those with limited technical experience, providing understandable explanations will be essential for its adoption. SWAN will initially have traditional explanations that contain information such as category, severity, how the warning can affect the code and how it can be exploited, and how the warning can be fixed, at a high level. Later in this article is a proposal for evaluating the effectiveness of SWAN's explanations.

G5 – Assist developers with recommendations based on their knowledge

As mentioned in G4, fixing warnings requires specific domain knowledge. Since analysis tools have access to repair history, they are the ideal actor for recommending specific warnings to specific developers, or recommending past fixes for specific warnings, for example. SWAN first needs to be tested with a large user base to increase understanding how a knowledge-base system containing past warnings, fixes, and developer information should be developed and integrated. Therefore, this guideline falls under future plans.

G6 – Provide collaboration options

In relation to G3 and G5, Nguyen Quang Do et al.[20] recommend providing a platform for sharing developer knowledge and for recommending developers who might

have the knowledge required to fix certain bugs. Like G5, this guideline requires a knowledge base, discussed later in this article.

G7 – Encourage good developer strategies

In their study, Nguyen Quang Do et al. observed strategies in how they behave with regard to warnings. For example, they create heuristics to distinguish true from false positives, or they default to certain behavioral patterns with warnings that they do not understand (e.g., ignore, escalate, or suppress those warnings). The authors recommend using the analysis knowledge and the collective developer knowledge to encourage good strategies such as escalation and discourage more dangerous strategies such as warning suppression. SWAN first must be tested to determine whether the previous guidelines are insufficient for promoting good behavior. This guideline is discussed later in the section on future plans.

G8 – Use different types of analysis tools to cover different aspects

Since this is the responsibility of the analysis user, this guideline is considered outside the scope of SWAN.

G9 – Use a single reporting platform for all warnings

Like G8, integrating different analysis tools is the responsibility of the analysis user, and thus, G9 is also outside the scope of SWAN. To facilitate the integration work, however, the next iteration of SWAN may include an export option in SARIF (Static Analysis Results Interchange Format).[1]

A heuristic is needed to estimate the time required to fix a given warning.

G10 – Promote the usage of analysis tools via policy enforcement and by spreading awareness

SWAN can provide a smooth integration into git, building, and IDEs to make it easier to adopt, configure, and use. This should make developers more willing to adopt SWAN, which would also make enforcing its usage through company policy easier. SWAN could also be shipped with specific policy suggestions and advertising material to help developers understand what SWAN offers them. A tool cannot directly influence a company's policies, however, so G10 is outside the scope of this article.

SHORT-TERM DESIGN GOALS

Let's now examine the *immediately* applicable usability guidelines from the perspective of SWAN.

Time constraints

G1 – Developers have limited amounts of time. Importantly, they have small time spans available to them between tasks. This is when they are most willing to resolve warnings.

Nguyen Quang Do et al. suggest, "the [analysis] tool can propose suitable warnings to fix for a given time span [that the developer can select]." In other words, the developer selects a desired time span for fixing warnings (e.g., 15 or 30 minutes), and the analysis tool then provides a *batch* of warnings to be fixed within that time span.

To accomplish this, a heuristic is needed to estimate the time required to fix a given warning. This heuristic should take into account several factors such as warning complexity (e.g., how many files and methods it flows

through), warning type, and developer experience. To select the warnings efficiently, their priority (or severity) must be balanced with the time it takes to fix them. Developing this time-estimate heuristic would require tuning it with empirical developer trials, which may be included in future trials. Developing such a heuristic may not be a priority if the emphasis on time constraints is reduced by limiting upstream warnings through immediate fixes.

SWAN will promote immediate fixes and limit how many warnings appear upstream. The upstream state will still be made available to view outstanding warnings. Furthermore, SWAN is security focused, so any warnings that arise should not be ignored, as is often the case with general bugs or code style issues.

The nature of the Swift development workflow and how SWAN integrates into the workflow through the build system allow it to find warnings quickly while the developer is actively working on the app. The number of warnings that the developer has to fix while developing is expected to be small enough that they can be fixed immediately. Hence, the importance of addressing time constraints may be greatly reduced in this regard.

There are two major ways, however, in which many warnings can be introduced at once, and that may warrant having time estimates. First, when the developer runs the analysis on a codebase for the first time, SWAN will likely issue many warnings that the developer will not be able to resolve immediately. Second, whenever SWAN's analysis changes in a new release, it may produce many new warnings because of new warning support. Future plans

include exploring how to evaluate whether and to what extent SWAN will produce a backlog of warnings that the developer will have to resolve with limited time. This will help determine whether or not to pursue the development of time heuristics.

Furthermore, while developers are working, they may not want to resolve a warning immediately because they are preoccupied and focused on coding. In this case, a time estimate may also be helpful. A rough estimate would likely be sufficient here. The SWAN UI could also provide a convenient button that prompts a calendar event with the warning information and time estimate, so that developers can allocate time to resolve the warning later—hence, immediately handling the warning rather than ignoring it.

The next steps in this research area would be to study how developers use SWAN to determine which metrics to incorporate into the time estimates. These metrics can range from the developer's experience with a given type of warning, to the size of the codebase, to the time of day. The influence of such metrics on the time estimate could differ widely from one developer to another, but since SWAN focuses on very specific code warnings, those uncertainties could likely be restricted.

Responsiveness

G2 – Developers do not like waiting for analysis tools, especially when they have time constraints. The analysis responsiveness and the tool interface should be crafted to minimize wait times.

Swift applications follow a module-based design, where each user (non-package) module is usually small enough

that it can be quickly analyzed by SWAN. The user's code may be distributed across multiple modules, and all packages are included as modules.

Swift libraries are not distributed in binary form and can therefore be analyzed by SWAN along with the user code. SWAN's new infrastructure is being designed to handle these modules individually and to link them together automatically. Therefore, upon rebuilding, SWAN does not need to reanalyze all modules and will process only the changed module(s). SWAN will then link them with the cached modules from the previous build to have complete data flow information. With this design, SWAN provides fast iterative analysis for each subsequent build.

Regarding the UI, SWAN aims to provide immediate feedback while developers are working. The lightweight analysis will run every time they build their applications, and they should receive warning change notifications inside their IDEs. Not all IDEs provide plug-in support, however.[2] Therefore, plug-ins should not be considered a means of creating SWAN's primary UI, but the goal is to implement IDE integrations that at least notify the user of new warnings. Detailed warning information will be available in SWAN's dedicated (primary) UI, which will be either a desktop or a web-based application.

Developers should easily be able to tell what effect their changes have on the current list of warnings—that is, if their changes resolved or introduced warnings. SWAN can provide multiple lists in the dedicated UI to help developers understand their changes: warnings that differ from the latest commit (similar to git) and warnings that differ from the previous build, including the state

of warnings at each previous build. The IDE notifications mentioned earlier could similarly be configured to display changes from the latest commit or build. Part of the future work on SWAN's UI will endeavor to design and test different features that can display realtime updates without confusing the user. This system is discussed further in relation to encouraging good behavior and its evaluation in the section on future plans for SWAN.

Customizable heuristics

G3 – Developers should be able to configure the analysis tool with their own rules or heuristics to adapt it to their needs when necessary.

Swift developers use various packages (i.e., libraries or APIs). One major advantage of the Swift build system is that all non-Apple/proprietary packages are included with the source code. This means that basic data flow through these packages does not have to be modeled but their security behavior does. Because SWAN does not currently have the resources to support all common APIs, some security information is missing, such as which API methods are sources, sinks, or sanitizers. Before its release, however, SWAN will at least have modeled the Swift Standard Library.

Inevitably, developers will find that some warnings are false positives (e.g., when a sanitizer is missing) or false negatives (e.g., when a source or sink is missing). SWAN will provide two ways that developers can simultaneously adapt their local analysis to minimize those incorrect warnings and notify the developers of SWAN of incorrect behavior.

First, in the case of a false negative, developers can create a short report about missing behavior, including the API method in question, its correct behavior (source, sink, or sanitizer), and optionally a short description. This will automatically give that API method the desired behavior in the analysis engine and send a report to the SWAN developers.

Second, they can create a similar report about a specific warning, but in this case more report information will automatically be provided. No user code will be included in the reports that are sent to the SWAN developers. Through these options, developers may continue to use SWAN without suppressing warnings, and the SWAN team can adapt SWAN to have better support in the next release.

With this report data, the SWAN developers can build a substantial knowledge base of the developers' heuristics. Once those heuristics are categorized, SWAN's initial bug-reporting system can be improved with a more detailed UI for software engineers to input different types of heuristics, and for those heuristics to be automatically fed back into the analysis engine. This will require further research in how and when the different types of heuristics should influence the analysis engine.

DISCUSSION AND FUTURE PLANS

While some design guidelines are immediately applicable to SWAN, others are not. This section addresses the latter set, as well as future plans for evaluating SWAN's usability.

Warning backlog

SWAN focuses on minimizing the number of warnings that developers do not resolve immediately by tightly integrating into their workflow and providing easy ways to understand the warnings their changes cause. Many warnings, however, may suddenly be introduced that the developer will have to work through eventually. The SWAN team needs to evaluate if and to what extend this is the case. It may turn out that once a developer has fixed the initial set of warnings, there is rarely a need to use time estimates (G1), recommender (G5), or collaboration (G6) systems. This would reduce the relevance of these guidelines to SWAN. Warning explanations (G4) can also play a big part in helping developers immediately resolve warnings. Therefore, in future trials the SWAN developers plan to analyze how many warnings are left unresolved and what part the warning explanations play in this outcome.

Encouraging good behavior

The previous section on G2 discussed IDE warning notifications that appear as the developer is working in order to increase responsiveness. This is a critical moment when the developer decides whether or not to address a warning. Therefore, it is essential that developers do not ignore important warnings (we think all security warnings are important) and have clear information to help them resolve the warnings. Time estimates and good warning explanations can help achieve this, but this may not be sufficient, so other ways to encourage good behavior (G7) should be explored. For example, SWAN focuses on security and will produce many warnings that can quickly

be resolved by passing data through a sanitizing API. "Quick fix" options could be possible for these warnings, although lack of IDE plug-in support could make this difficult. SWAN trials could help evaluate if and why developers do not immediately resolve warnings and lead to determining which strategies can be improved or added to encourage good behavior.

A warning message should suggest people who can help the developer resolve the warning.

Knowledge base

The following guidelines require SWAN to have a knowledge base that contains information such as past warnings, fixes, and developer information. Before such a system is developed, SWAN must be tested with a significant user base to find what data is available to record and use.

G5 – Warning messages and recommender systems should consider developer knowledge and time available.

Personalized warning messages may help developers resolve warnings more efficiently. As previously mentioned, developer knowledge should be considered when calculating warning-time estimates, which are included in warning messages. Furthermore, a warning message should suggest people who can help the developer resolve the warning, such as teammates. The message should recommend developers who have experience resolving similar warnings, have worked on the sections of the codebase through which the warning flows, or have self-identified as experts in a particular warning type or category.

G6 – Users should have methods of collaborating to resolve warnings, such as through a messaging system where they can communicate about specific warnings.

The UI should provide a means of communicating about warnings, such as a chat system. It should provide a way to start a conversation about a warning, and the developer should then be able to tag suggested developers for help. An archive of these conversations should be made available to all developers so they can read through previous conversations about a warning they have been assigned before seeking help. A conversation archive could also later provide insight into what warnings developers most struggle with (seek help for), what specifically developers do not understand, and how quickly they receive help, in order to improve the tool's usability. A quick link to related conversations should be included in the warning message.

Future user studies

To evaluate the next generation of the SWAN UI, the team plans to conduct several user studies. In particular, identifying integration points in developer workflows will help in understanding how to automate analysis tools and minimize interference. Additionally, exploring UI design and layout in depth, with specific examples from industry static-analysis tool UIs, will help in designing the new SWAN UI. Combined with Nguyen Quang Do et al.'s work, future usability studies will provide a well-rounded understanding of SWAN.

References

1. Anderson, P. 2018. Static analysis results: a format and a protocol: SARIF & SASP. GrammaTech Blog; https://blogs.grammatech.com/static-analysis-results-a-format-and-a-protocol-sarif-sasp.
2. Apple Developer. 2021. Xcode; https://developer.apple.com/xcode/.
3. Apple iOS Team. 2007. iOS 14; https://www.apple.com/ca/ios/.
4. Apple macOS Team. 2001. macOS Big Sur; https://www.apple.com/ca/macos/mojave/.
5. Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., McDaniel, P. D. 2014. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 259–269; https://doi.org/10.1145/2594291.2594299.
6. Ayewah, N., Pugh, W., Hovemeyer, D., Morgenthaler, J. D., Penix, J.,2008. Using static analysis to find bugs. *IEEE Software* 25(5), 22–29; https://doi.org/10.1109/MS.2008.130.
7. Ayewah, N., Pugh, W. 2008. A report on a survey and study of static analysis users. In *Proceedings of the Workshop on Defects in Large Software Systems*; https://doi.org/10.1145/1390817.1390819.
8. Azim, T., Neamtiu, J. 2013. Targeted and depth-first exploration for systematic testing of Android apps. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, ed. A. L. Hosking, P.

Th. Eugster, and C. V. Lopes, 641–660; https://doi.org/10.1145/2509136.2509549.

9. Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Henri-Gros, C., Kamsky, A., McPeak, S., Engler, D. 2010. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM* 53(2), 66–75; https://doi.org/10.1145/1646353.1646374.
10. Chowdhury, I., Zulkernine, M. 2010. Can complexity, coupling, and cohesion metrics be used as early indicators of vulnerabilities? In *Proceedings of the ACM Symposium on Applied Computing*, 1963–1969; https://doi.org/10.1145/1774088.1774504.
11. Christakis, M., Bird, C. 2016. What developers want and need from program analysis: an empirical study. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 332–343; https://doi.org/10.1145/2970276.2970347.
12. Huang, W., Dong, Y., Milanova, A., Dolby, J. 2015. Scalable and precise taint analysis for Android. In *Proceedings of the International Symposium on Software Testing and Analysis*, ed. M. Young and T. Xi, 106–117; https://doi.org/10.1145/2771783.2771803.
13. Johnson, B., Song, Y., Murphy-Hill, E., Bowdidge, R. 2013. Why don't software developers use static analysis tools to find bugs? In *Proceedings of the International Conference on Software Engineering*, 672–681; https://dl.acm.org/doi/10.5555/2486788.2486877.
14. Kersten, M., Murphy, G. C. 2006. Using task context to improve programmer productivity. In *Proceedings of the 14th ACM SIGSOFT International Symposium on*

Foundations of Software Engineering, 1–11; https://doi.org/10.1145/1181775.1181777.

15. Lewis, C., Lin, Z., Sadowski, C., Zhu, X., Ou, R., Whitehead, E. J. 2013. Does bug prediction support human developers? Findings from a Google case study. In *35th International Conference on Software Engineering*, 372–381. IEEE; https://doi.org/10.1109/ICSE.2013.6606583.
16. LLVM Developer Group. 2003. The LLVM compiler infrastructure; https://llvm.org/.
17. LLVM Developer Group. 2007. Clang: a C language family front end for LLVM; https://clang.llvm.org/.
18. Nguyen Quang Do, L. 2019. User-centered tool design for data-flow analysis. Ph.D. dissertation. Paderborn University; https://doi.org/10.17619/UNIPB/1-820.
19. Nguyen Quang Do, L., Ali, K., Livshits, B., Bodden, E., Smith, J., Murphy-Hill, E. R. 2017. Just-in-time static analysis. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 307–317; https://doi.org/10.1145/3092703.3092705.
20. Nguyen Quang Do, L., Wright, J. R., Ali, K. 2020. Why do software developers use static analysis tools? A user-centered study of developer needs and motivations. *IEEE Transactions on Software Engineering* (June 24). IEEE; https://ieeexplore.ieee.org/document/9124719.
21. StatCounter GlobalStats. 2019. Desktop operating system market share worldwide; https://gs.statcounter.com/os-market-share/desktop/worldwide/#monthly-201901-201912.
22. StatCounter GlobalStats. 2019. Mobile operating system market share worldwide; https://gs.statcounter.

com/os-market-share/mobile/worldwide/#monthly-201901-201912.
23. Swift. 2015. The Swift Programming Language; https://swift.org/.
24. SwiftLint. 2015. A tool to enforce Swift style and conventions. GitHub; https://github.com/realm/SwiftLint.
25. Tailor. 2015. Cross-platform static analyzer and linter for Swift. GitHub; https://github.com/sleekbyte/tailor.
26. Tiganov, D., Cho, J., Ali, K., Dolby, J. 2020. SWAN: A static analysis framework for Swift. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 1640–1644; https://doi.org/10.1145/3368089.3417924
27. Vassallo, C., Panichella, S., Palomba, F., Proksch, S., Zaidman, A., Gall, H. C. 2018. Context is king: the developer perspective on the usage of static analysis tools. In *IEEE 25th International Conference on Software Analysis, Evolution and Reengineering*, 38–49. IEEE; https://doi.org/10.1109/SANER.2018.8330195.
28. Visual Studio. 2015. Visual Studio Code – Code editing. Redefined; https://code.visualstudio.com.

Daniil Tiganov *is a computing science student at the University of Alberta. He is the main contributor to the SWAN project.*

Lisa Nguyen Quang Do *is a software engineer at Google Zurich. She received her PhD degree at Paderborn University in 2019. Her research focuses on improving the usability of*

analysis tools for code developers and analysis developers through different aspects ranging from the optimization of the analysis algorithm to the implementation of its framework to the usability of its interface.

Karim Ali *is an Assistant Professor in the Department of Computing Science at the University of Alberta. Prior to that, he was a postdoctoral researcher at Technische Universität Darmstadt, Germany within the Secure Software Engineering (SSE) Group led by Eric Bodden. He finished his Ph.D. studies under Ondrej Lhoták in the Programming Languages Group at the University of Waterloo.*