Empirical Software Engineering (2023) 28:118
https://doi.org/10.1007/s10664-023-10354-3

Can the configuration of static analyses make resolving security vulnerabilities more effective? - A user study

Goran Piskachev[1,2] · **Matthias Becker**[1] · **Eric Bodden**[1,3]

Accepted: 5 June 2023 / Published online: 12 September 2023

Abstract
The use of static analysis security testing (SAST) tools has been increasing in recent years. However, previous studies have shown that, when shipped to end users such as development or security teams, the findings of these tools are often unsatisfying. Users report high numbers of false positives or long analysis times, making the tools unusable in the daily workflow. To address this, SAST tool creators provide a wide range of configuration options, such as customization of rules through domain-specific languages or specification of the application-specific analysis scope. In this paper, we study the configuration space of selected existing SAST tools when used within the integrated development environment (IDE). We focus on the configuration options that impact three dimensions, for which a trade-off is unavoidable, i.e., precision, recall, and analysis runtime. We perform a between-subjects user study with 40 users from multiple development and security teams - to our knowledge, the largest population for this kind of user study in the software engineering community. The results show that users who configure SAST tools are more effective in resolving security vulnerabilities detected by the tools than those using the default configuration. Based on post-study interviews, we identify common strategies that users have while configuring the SAST tools to provide further insights for tool creators. Finally, an evaluation of the configuration options of two commercial SAST tools, FORTIFY and CHECKMARX, reveals that a quarter of the users do not understand the configuration options provided. The configuration options that are found most useful relate to the analysis scope.

Keywords Static analysis · Security · User study · Empirical research · SAST

Communicated by: Jacques Klein

✉ Goran Piskachev
gpiskach@amazon.de

[1] Fraunhofer IEM, Paderborn, Germany

[2] Amazon Web Services, Berlin, Germany

[3] Department of Computer Science, Paderborn University, Paderborn, Germany

1 Introduction

Static application security testing (SAST) tools were introduced in the early 2000s. In recent years, their popularity has increased significantly, as many commercial (Checkmarx, 2021; Microfocus, 2021; Veracode, 2021; SonarSource, 2021; Github, 2021; Snyk, 2021) and open source (of Maryland U, 2021; Facebook, 2021; Krüger et al., 2018) tools became available. Thus, the research on these tools has strong industrial relevance. In particular, researchers have been studying the usability of static analysis tools. Christakis et al. reported developers' pain points when using program analysis tools such as, *"high number of false warnings"*, *"analysis running too long"*, *"bad warning messages"*, and *"wrong checks turned on by default"* (Christakis and Bird, 2016). Others have been studying different aspects, e.g. explainability (Nguyen Quang Do and Bodden, 2014; Nguyen Quang Do et al., 2020), integrated development environment (IDE) integration (Vassallo et al., 2018; Nguyen Quang Do et al., 2020), and continuous integration (Zampetti et al., 2017).

Most issues reported by developers can be handled by choosing the right configuration for the target program being analyzed. To enable this, tool vendors provide a wide range of configuration options. For example, developers can select rules or write custom ones using a domain specific language; they can set different thresholds for the analysis engine, or can select the scope of the target program. In this paper, we study the configuration options of SAST tools that are *integrated into an IDE*. In this scenario, a short analysis time is of particular importance (Johnson et al., 2013; Christakis and Bird, 2016). Within an IDE, users often have a specific context in which they work, e.g. the code last written or edited, or a vulnerability relevant to a specific component of the project that the user is responsible for. When this is the case, the SAST tool can be configured to provide the results quickly by focusing on such a limited scope (Nguyen Quang Do et al., 2017).

This paper studies the configuration options that impact precision, recall, and analysis time. We omit other options such as filtering and prioritization of warnings, as they are more relevant when a project is analyzed as a whole. This is often the case when the analysis time is not that critical, e.g. during nightly builds. We investigate two specific options: (1) selection of security rules and (2) selection of analysis scope. We chose these options based on previous studies (Nguyen Quang Do et al., 2020; Christakis and Bird, 2016) showing that users find them to be highly relevant. To evaluate the users' effectiveness in resolving the findings reported by a SAST tool, we performed a between-subjects (Charness G et al., 2012) user study. We divided the 40 participants/subjects into two groups of 20 participants each and gave them the task to resolve the findings reported by a SAST tool, by assessing each finding as *true positive*, *false positive*, or *don't know*. The first group (treatment group) was able to configure the SAST tool's options via a configuration page and trigger the analysis multiple times. The second group (control group) could run the analysis once in a default configuration. The results show that the treatment group was more effective in resolving the findings.

In a post-study interview, we asked participants to elaborate on the strategies they used while solving the given task. It showed that most participants selected all entry points of the project and iteratively ran the tool with a single security rule. Yet, when we asked whether they would use the same strategy if that was their own project, most of them told they would apply a different strategy, i.e., selecting specific entry points only with a subset of the security rules. Then, we conduced a questionnaire in which we asked participants to label the configuration options of two SAST tools, FORTIFY and CHECKMARX, with two labels, indicating whether the option is understandable, and the other indicating whether it is useful to them. We found

that, on average, one quarter of the users do not understand the existing configuration options in these tools. From those that can be understood, 78% of the participants find the options to be useful for their use case (i.e., the mean over all options with standard deviation σ=16). Finally, based on the data we collected, we formed a list of recommendations for future SAST tools. To name a few, the tools should provide details about the security vulnerability reported including description, examples, and possible fixes, and the rules of the tools should be available to the users for inspection to better evaluate if the findings are true or false positive.

This paper makes the following contributions:

- a user study showing that users of SAST tools can resolve security vulnerabilities more effectively when they configure the tool to their context,
- the identification of the common strategies that users have while configuring SAST tools to find and resolve security vulnerabilities,
- an evaluation of the existing configuration options of two SAST tools in terms of comprehensibility and usefulness for the end users, and
- recommendations for SAST tools creators based on our research described in this paper.

The dataset and all materials we created and used for this research, are available as an artifact via the link *https://research-sast-config.github.io/*.

In the following, we start by discussing the related work. In Section 3, we outline our methodology. The results of our study are discussed in Section 4. Section 5 outline the possible threats to validity, and we finally conclude in Section 6.

2 Related work

Static analysis has been used in industry for a long time. However, it started to gain popularity with the introduction of more sophisticated SAST tools for development teams such as (Microfocus, 2021; Checkmarx, 2021; Grammatech, 2021). With the increased number of new open-source and commercial tools on the market, researchers have studied different aspects, such as usability, incorporation into users' workflow, quality of results, etc. In the following, we discuss existing studies on static analysis tools. We categorize the studies into three groups: (1) studies that focus on usability of static analysis tools, (2) studies targeting the security aspect, and (3) studies on the quality of the results. Table 1 summarizes all studies by stating their goal, methodology, and scale in terms of number of participants, number of companies involved and/or size of data analyzed.

2.1 Usability of static analysis

Within this group of studies, we identified two subgroups: studies that focus on usability and studies that focus on the adaption and integration of the tools within the companies' processes.

(Nguyen Quang Do et al., 2020) performed a survey with developers from the German company *Software AG* and analyzed the usage data of (Checkmarx, 2021) for two of the company's projects. They identified needs and motivations that developers have while using static analysis tools. Based on that, they provide recommendations for new features and research ideas for future consideration. A similar study has been conducted earlier at *Microsoft* by (Christakis and Bird, 2016). They performed a survey and interviews with developers, and analyzed data from live site incidents, which are high-critical bugs handled by the on-call

engineers. They identified usability issues and functionalities that program analysis should provide for better usability. Similar results were reported also in the study by (Johnson et al., 2013), in which they performed semi-structured interviews with 16 developers from a single company and four graduate students. All these studies are complementing with our study and they have identified several overlapping results. Our study is the only one that includes a users study. It is also the only one that measures the effectiveness in resolving the vulnerabilities produced by a given tool.

(Vassallo et al., 2018) studied the different contexts in which static tools are used by developers, i.e., development environment, review, and continuous integration. Moreover, they studied the configuration options for these contexts and found out that most developers use the same configuration among all environments (IDE, continuous integration, or review). In comparison, our study targets only the IDE. They performed a survey with 42 participants. To confirm their findings, they interviewed eleven developers from six companies.

We discuss two studies that target the integration of static analysis tools into the development workflow and processes. (Sadowski et al., 2015) proposed the Tricoder platform that is used by *Google* to integrate different program analysis tools in one system and improve the user experience. They explain the requirements and how the system was deployed. They collected usage data from the deployed system to confirm the design decisions of the platform. Our user study is complementing this approach by collecting data from a survey instead of usage data from the tool. (Zampetti et al., 2017) studied how static analysis tools are integrated in the pipeline in open-source Java projects from GitHub. They used repository mining techniques which is different source of data and moreover, they answer different research questions compared to our work.

2.2 Studies on adaption of security tools

Next, we discuss studies with a particular focus on detecting security vulnerabilities with static analysis tools. (Thomas et al., 2016) performed an experiment with 13 developers in which the participants were given the task to use a tool that reports security vulnerabilities in the IDE. After interacting with the tool, the participants were interviewed. In a similar study, (Smith et al., 2015) invited 10 developers from the same project to solve four tasks when using an extended version of FindBugs. The participants were asked to orally explain their thoughts, which the authors used to formulate questions. Then, they used the card-sorting method to come to relevant conclusions. Compared to these studies, our work is the only one that focuses on specific configuration options. (Smith et al., 2020) evaluated the usability of the user interfaces of four SAST tools. They used heuristic walkthroughs and an experiment followed by an interview with twelve participants. In a survey study with multiple stages, (Witschey et al., 2015) identified factors that can predict the adoption of security tools by developers. (Patnaik et al., 2019) have mined 2,491 Stack Overflow questions to identify usability issues that developers face when using cryptography libraries. Compared to all these studies, our work does not target the usability aspect of the tool.

2.3 Taint analysis results and comparison

Several previous studies focus on the quality of the analysis results in terms of recall, precision, or runtime, reported by taint analysis tools. (Luo et al., 2019) performed a quantitative and qualitative analysis of the taint flows reported by (Arzt S et al., 2014), by analyzing 2,000 Android apps. They identified that the selection of sources and sinks was one of

Table 1 Related studies: goal, methodology, and scale

Study	Goal	Methodology	Scale
(Nguyen Quang Do et al., 2020)	identify developers' goals and motivations for using static analysis tools	a survey and an analysis of company's usage data of CheckMarx	87 participants and data from two internal projects at *Software AG*
(Christakis and Bird, 2016)	identify practitioners' needs from program analysis	a survey, interviews with group managers, and an analysis of live site incidents	375 participants and 256 live site incidents reports from 17 services at *Microsoft*
(Johnson et al., 2013)	identify developers' usability issues with static analysis tools	semi-structured interviews	16 developers from single company and 4 graduate students
(Vassallo et al., 2018)	study the developers' usage context of static analysis tools	a survey and semi-structured interviews	42 participants through open invitation and 11 interviewees from six companies
(Sadowski et al., 2015)	provide a set of principles for building and integrating program analysis tools in practice	case study of Tricoder as a platform for program analysis tools	Tricoder usage data at *Google*
(Zampetti et al., 2017)	study the CI and usage of static analysis tools	mining repository techniques	20 open source GitHub projects
(Thomas et al., 2016)	study the perceptions and actions taken by developers when they interact with static analysis tool in the IDE	experiment with an interactive tool in the IDE and an interview	13 participants from multiple companies
(Smith et al., 2015)	study the information need of developers while using static analysis tool for security vulnerabilities	an experiment and card sorting	10 developers working on the same project
(Smith et al., 2020)	study the user interface of 4 tools and propose areas for improvements	heuristic walkthroughs, an throughs, an experiment and interviews	12 participants
(Witschey et al., 2015)	quantify the relative importance of factors that predict security tool adoption	a multi-staged survey	119 participants from 14 companies and 61 participants from 5 mailing lists
(Patnaik et al., 2019)	identify usability issues of crypto libraries used by developers	mining techniques	2,491 Stack Overflow questions
(Luo et al., 2019)	identify important factors for imprecision in FlowDroid	a case study with manual inspection	2000 analysed apps and 146 manually inspected

Table 1 continued

Study	Goal	Methodology	Scale
(Qiu et al., 2018)	compare the results of Android taint analysis tools and identify strength and weaknesses	analysis and inspection of the results by three tools (FlowDroid, Amandroid, and DroidSafe)	collection of microbenchmarks
(Habib and Pradel, 2018)	compare the results of three static analysis tools	automatic and manual inspection of the bugs reported from three tools	collection of 15 Java application (Defects4J) with known bugs
(Zhang et al., 2017)	reduce false positives by proposing an interactive approach for resolving findings from static analysis	experiment with datarace analysis	evaluated on 8 Java projects and data with known bugs collected from Java developers hired from UpWork
(Lee et al., 2017)	reduce false positives via clustering algorithms	experiment with buffer overflow findings	evaluated on the findings from the Sparrow static analyzer on 14 C packages

the main factors for imprecision. Compared to our work, this methodology was performed manually by experts who evaluated the quality of the findings. (Qiu et al., 2018) compared the three Android taint analysis tools, FlowDroid, (Wei et al., 2018) and (Gordon et al., 2015). They ran all tools under same configuration in order to gain a fair comparison of the tools' capabilities. Their work on finding a common configuration among the three tools to make a fair comparison provides useful insights on the importance of the configuration for the quality of the findings. While they performed an automated experiment to compare the different options, they do not include the intended user as in our user study. (Habib and Pradel, 2018) investigated the quality of the findings from three static analysis tools Spotbugs, Infer, and Error Prone. They used the real-world Java applications from Defects4J with 594 known bugs and inspected the findings, both, automatically and manually. They find that the tools detect only 4.5% of the bugs and the types of findings they report are complementary. Compared to our work, they did not reason how users resolve the findings. (Zhang et al., 2017) proposed an interactive approach for resolving the findings from static analysis tools. They performed an experiment with datarace analysis to evaluate their approach and show a 74% reduction in the false positive rate. They used 8 real-world Java applications and a set of questions collected from Java developers that they hired from UpWork. This approach complements our work with interactive design. Finally, (Lee et al., 2017) proposed a clustering algorithm for static analysis findings. They compared several algorithms on the buffer overflow findings from 14 C packages and show 45% reduction in the false positive rate. Compared to our work, this is a fully automatic approach that can be used in combination with a tool like SECUCHECK.

3 Methodology

Aiming to find new insights about how users of SAST tools use different configuration options, we designed a user study that we explain in this section. To counter-act possible learning effects by study participants, we applied a between-subjects study design (Charness

G et al., 2012) consisting of a lab study and a semi-structured interview to find out to what extent the configuration options in SAST tools are helpful to end users. Next, we explain the recruiting of the participants, the study design, and the tools we used.

3.1 Participants

We recruited the participants through our contacts in seven companies with software development departments. We had a single point of contact that internally distributed our invitation to potential participants, either developers or security experts. As the number of security experts was insufficient, we additionally invited Ph.D. candidates doing research in security or program analysis. Previous studies have shown that graduate students are valid proxies for such studies (Naiakshina et al., 2020, 2018, 2019, 2017). The participation was voluntary and without incentive. In total, we had 40 participants (P01-P40), 24 from industry and 16 Ph.D. candidates from academia. 23 participants have mainly software development responsibilities, while 17 participants are security or program-analysis experts. Table 2 lists the profile of each participant. The columns *From* and *Role* were collected prior to the session with the participant, based on the information provided by the point of contact. The columns *Coding* and *Security* originate from a self-assessment by the participant, collected during the session. The column *Study Type* is the allocation of the two types of experiments, which we explain in the following section. This allocation was done at random, separately for each group, industry and academic participants.

3.2 Design

This subsection discusses the main aspects of the design of our user study.

3.2.1 Lab study

The goal of the study is to evaluate whether users that *do configure* the SAST tools within the IDE are more effective in resolving the findings than users running the default configuration only.

Usage scenario. The study focuses on users of SAST tools during development time. Our selected tool, SecuCheck, runs within the IDE in the background while the user can review or edit the code before checking it into the remote repository. Subsection 3.2.2 provides further details on SecuCheck and the reasons for our tool choice. The user has given time to use the tool and decide which of the findings from the tool are true positives and which are false positives. To make this scenario as realistic as possible, we provide each participant with a relative small project which still has a meaningful logic and is a complete Java application (see Subsection 3.2.3). Each participant was given time to familiarize itself well with the code before performing the tasks. The moderator explicitly asked the participants whether they are familiar enough with the code in order to be able to resolve the findings from SecuCheck.

Each participant was given a time of 15 minutes to run the tool, look into the findings, and resolve them. We chose this amount of time based on our test runs of the study where we observed that the test participants were able to resolve a portion of the findings, but not all. This is similar to a real-world scenario when the user has a limited time for such task and typically the tools will produce large amount of findings. (Nguyen Quang Do et al., 2020) explored this scenario and found out that typically developers spend only short amount of

Table 2 Participants' profile

	From	Role	Coding (years)	Security(experience)	Study Group
P01	industry	developer	3-5	beginner	control
P02	industry	developer	6-9	beginner	treatment
P03	industry	developer	10+	knowledgeable	control
P04	industry	developer	6-9	knowledgeable	control
P05	industry	developer	10+	knowledgeable	control
P06	industry	developer	10+	beginner	treatment
P07	industry	expert	10+	knowledgeable	treatment
P08	industry	developer	10+	knowledgeable	control
P09	industry	expert	10+	knowledgeable	control
P10	industry	developer	6-9	knowledgeable	treatment
P11	industry	developer	3-5	beginner	control
P12	industry	developer	10+	beginner	control
P13	industry	developer	6-9	beginner	treatment
P14	industry	developer	6-9	knowledgeable	control
P15	industry	developer	10+	beginner	control
P16	industry	developer	6-9	beginner	treatment
P17	academia	developer	6-9	knowledgeable	control
P18	academia	expert	6-9	knowledgeable	treatment
P19	industry	developer	10+	knowledgeable	treatment
P20	academia	developer	6-9	beginner	control
P21	academia	expert	6-9	beginner	treatment
P22	academia	expert	3-5	beginner	treatment
P23	academia	expert	10+	knowledgeable	control
P24	academia	expert	3-5	beginner	treatment
P25	academia	expert	10+	beginner	treatment
P26	academia	expert	10+	expert	treatment
P27	industry	developer	3-5	beginner	treatment
P28	academia	expert	6-9	beginner	control
P29	academia	expert	6-9	knowledgeable	control
P30	academia	expert	6-9	beginner	treatment
P31	industry	developer	10+	knowledgeable	treatment
P32	industry	expert	10+	beginner	control
P33	academia	developer	6-9	expert	control
P34	industry	expert	10+	knowledgeable	treatment
P35	industry	expert	3-5	expert	treatment
P36	academia	expert	3-5	knowledgeable	control
P37	industry	developer	6-9	knowledgeable	control
P38	academia	expert	10+	beginner	treatment
P39	academia	developer	3-5	beginner	control
P40	industry	developer	3-5	knowledgeable	treatment

time for such tasks, usually between larger tasks or only when they have extra time. The participant was asked to resolve the finding by stating if the finding is a true positive, false positive, or the participant cannot decide due to missing expertise or any other reason. We randomly divided the participants into two groups. The one group, the treatment group, was able to use the configuration page of our tool freely and run the tool multiple times with different configurations. They could use two configuration options, i.e., selection of security rules and selection of classes to be analyzed. Thus, depending on the selection chosen by the participant, not all possible findings were shown by the tool at once. These two options can be found in similar form in many tools, in particular commercial tools (e.g., both commercial tools chosen in Section 4, FORTIFY and CHECKMARX have configuration options that can limit the analysis scope, as well as options to select the analysis rules).

The control group used only the default configuration, in which all security rules and all classes are selected. Hence, the control group ran the tool only once and saw all possible findings at once. Our experience with SAST tools is that most default configurations will include all of the available security rules and will analyze the entire project. For simplification of the task, in our user study we only used taint-style vulnerabilities. Through the program statements, a taint analysis tracks values from so-called source statements (such as user input values from Http-request object) making them tainted values, to methods of interest so-called sink statements, e.g., performing security-relevant operations (such as writing to file or database). We chose taint analysis because it can be used to detect 17 (Piskachev et al., 2022) of the top 25 popular vulnerabilities by the SANS institute (Mitre CWE, 2021a).

The session was organized as follows. It starts with short introductions and clarification of what the study is about and what data is collected. Then, the moderator performs the first two parts of the interview (part one are general meta-questions and part two are general questions on SAST experience). Before giving the task, the moderator explains the main concepts of taint analysis and how it works to find an SQL injection as an example. The moderator also makes a walkthrough with SECUCHECK to demonstrate its features. Then, the participant is provided with the Eclipse IDE and the project under analysis available in the workspace. The participant is given time to familiarize herself/himself with the project after which the task is explained and given 15 minutes to work. After the task, the moderator starts with the second two parts of the interview: part three is discussion about the task, the strategies and feedback on the tool and part four is the questionnaire for the configuration options in FORTIFY and CHECKMARX.

3.2.2 SAST tool

Previous studies have shown that users prefer SAST tools that run within their IDE, using the existing features such as syntax highlighting, error view, hover messages, etc. Such integration enables the users to easily locate and understand the findings in the code. For our user study, we selected the existing tool, SECUCHECK (Piskachev et al., 2021) (version 1.0). It is a research tool implemented as a MagpieBridge Server (Luo et al., 2019), that uses the Language Server Protocol (LSP) to run within multiple IDEs, such as Eclipse, Visual Studio Code, IntelliJ IDEA, and others. SECUCHECK detects different taint-style vulnerabilities, which are among the most popular vulnerabilities, such as SQL injection (CWE-89) and Cross-site scripting (CWE-79). The taint analysis in SECUCHECK can be configured with different security rules via a Java-internal domain-specific (DSL) language, named *FLUENT*TQL. Compared to other existing DSLs which are designed for security and static analysis experts, *FLUENT*TQL is developer-centric and simple. We provided the specified rules to the participants in case they wanted to inspect them to understand the reported findings easier.

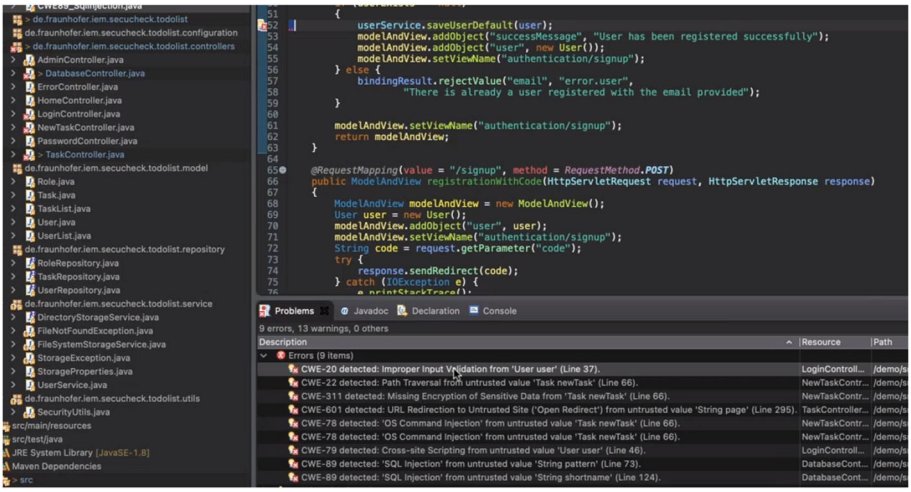

Fig. 1 IDE view of the workspace in which participants worked on the given task. Below are shown the findings that the tool reports in default configuration

For this study, we used the Eclipse client in which the analysis results are shown in the error view with click navigation to the relevant file. The standard markers on the sidebar from the editor mark relevant statements (sources and sinks) for each finding. For the configuration, SECUCHECK has custom page that is shown in the web browser via HTTP. Using this page, the participants can select the security rules and the classes to be analyzed via check boxes, and trigger the analysis by clicking a button. Figure 1 shows a screenshot of the view of the IDE and the project given to the participants while Figure 2 shows a screenshot of the configuration page. None of the participants have used SECUCHECK before the user study.

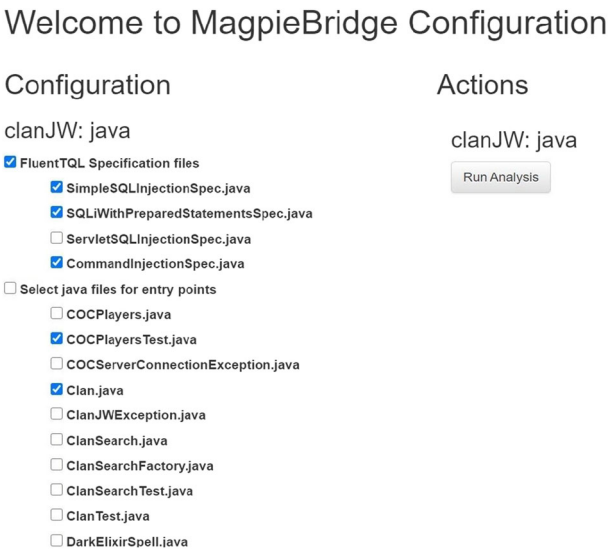

Fig. 2 Configuration page used in the tool for the user study

Table 3 List of findings reported by the SAST tool on the target project used in the user study

ID	CWE	Name	Location	TP/FP
F01	20	Improper input validation	LoginController.java	TP
F02	22	Path traversal	NewTaskController.java	TP
F03	311	Missing encryption	NewTaskController.java	TP
F04	601	Open redirect	TaskController.java	TP
F05	78	OS command injection	NewTaskController.java	FP
F06	78	OS command injection	NewTaskController.java	TP
F07	79	Cross-site scripting	LoginController.java	TP
F08	89	SQL injection	DatabaseController.java	TP
F09	89	SQL injection	DatabaseController.java	FP

3.2.3 Target project and built-in vulnerabilities

We considered several criteria for selecting the target project used in the user study. First, the project should be realistic such that participants can see clear functionalities and business logic implemented in the code. Second, it should be relatively small, so that participants can understand the code in the limited time they are given and be comfortable to resolve potential security vulnerabilities. Third, there should be multiple security vulnerabilities that SECUCHECK will report including true positives and false positives. These are all taint-style vulnerabilities as we manually checked that the tool uses the corresponding taint-flow rules. These vulnerabilities can be detected by taint analysis as shown in previous work (Piskachev et al., 2014), but not exclusively. We chose to use an existing Java Spring[1] project created earlier in our research group as a demonstrative project that showcases different security vulnerabilities. The project implements a simple task management tool where users can create, delete, and edit tasks, which are stored in MySQL database. The application uses the Spring MVC architecture. It does not contain any obfuscated code because the scenario considers the user to be familiar with the project, e.g., own project, or from the same team or organization. It consists of 35 classes and nine findings that SECUCHECK reports. The nine findings correspond to seven unique, real vulnerabilities and two false positives. Table 3 lists the findings reported by the tool when all rules and classes are selected, showing the vulnerabilities type (common weakness enumeration - CWE[2]), name of the class in which the vulnerability is located, and whether it is true or false positive. The taint analysis in SECUCHECK uses rules specified in Java fluent interface form. These files were also available to the participant for inspection. The participant was not required to do any changes in the rules or in the code.

3.2.4 Semi-structured interview

The interview consisted of four parts, (1) meta-questions, (2) questions on the experience with SAST tools, (3) discussion and feedback, and (4) a questionnaire. In part one, we asked three questions (**Q1-3**) on previous coding experience and security expertise. Part two comprised 11 questions (**Q4-14**) about previous experience with SAST tools, e.g. when and

[1] https://spring.io/

[2] https://cwe.mitre.org/data/index.html

how tools are used, who configures the tools, which tools are used, and any company-related regulations for using SAST tools. This part was asked before the task to give the participants more context and recall their own experience with SAST tools. The data collected in this part helps us understand the background of our population. Part three consisted of 9 questions (**Q15-23**). The moderator asked about the experience with the task and the tool to collect feedback. Additionally, the participants of the treatment group were asked to explain their strategies when using the configuration page. Finally, part four included 2 questions (**Q24-25**) that listed the configuration options available in two commercial SAST tools, FORTIFY and CHECKMARX and asked the participant to label each option if it is *understandable* and whether it is *useful* for her role. The questionnaire only listed the option names as they appear in the tools. The participants were encouraged to ask if they need explanation for that option, in which case we provided them with further description based on the official documentation of the tools. We selected all options that are relevant for precision, recall or analysis time. We went through the official documentation of the tools. The names of the tools were not revealed. **Q24** contains 18 options from FORTIFY and the **Q25** contains 11 options from CHECKMARX.

3.3 Calculating effectiveness

We explain how we calculate the effectiveness in resolving the findings from our SAST tool. In limited time of 15 minutes the participant was asked to use the tool, look into the findings, and based on the code, assess if that finding is *true positive*, *false positive*, or *cannot assess*. The maximum number of findings that the tool reports and each participant can resolve is nine. We count the ratio of correctly resolved findings out of the total number of findings that the participant resolved. We refer to this as *effectiveness in resolving the findings* (do the right thing). We prioritize quality over quantity. We do not look into the efficiency, i.e., more number of (correct) resolvings in less time, because security is not a property where users should compete, neither feel pressure for quantity. In other words, to do the task effectively, we care about the quality of each resolving. When the participant resolved X findings of which Y were resolved correctly as true or false positive, we report the effectiveness as the ratio Y/X, expressed in percent. Moreover, with our study design, calculating efficiency (resolving more findings in the given time) would give advantage to the control group, since these participants see all possible findings at the beginning while the treatment group may choose configurations that will not show all findings in the application.

3.4 Statistical tests

For the reported results in this study, we perform relevant statistical tests. The main treatment variable is the study group in which the participant was randomly allocated to, i.e., control vs. treatment group (nominal data). The background variables are: years of coding experience (ordinal data), role (nominal data), institution (nominal data), and security experience (nominal data). Finally, we have four effort variables, one for the outcome of each finding (nominal data), the strategy used during the task (nominal data), number of submitted configurations (ordinal data), and inspection of the rules (binary data). As most of the data is nominal and ordinal, we used only non-parametric statistical tests. We report individually each selected test and the results. We used the significance level $\alpha = 0.05$ for all tests.

3.5 Data collection

The user study was conducted during December 2020 and January 2021. After the initial contact with each participant via e-mail, which provided basic information about the study and the data we plan to collect, we arranged a virtual session over Microsoft Teams. On average, the sessions took 75 minutes. The participants did not need to prepare or install any software. All required tools were prepared by the moderator who shared the screen and when needed gave control to the participant to perform the task. All sessions were recorded for post-processing. We invited a researcher with experience in conducting user studies to moderate all sessions. To adapt and verify our design, we performed four test runs with students from our group. The recorded videos were analyzed by the first author who documented the raw data, which was further processed for the results reported in Section 4.

As an IDE, we used Eclipse, in which we installed our MagpieBridge Server (Luo et al., 2019). For the semi-structured interview, we used *Google Forms* in which the moderator collected answers. Most questions were of closed type. The answers of the few open questions (Q16-19, 22-23) which were in part three were collected in the post-processing phase.

Our dataset is available as an artifact and includes the questionnaire, the collected and processed data, the code used for the task, and tool source code and documentation.

3.6 Ethics

The participation in the study was voluntary. The participants in the interviews signed a consent form. For most questions, we provided an option for participants not wanting to give any details (i.e., "I don't know"). We aligned our study to the data protection laws in Germany and the EU. The user study design has been approved by the corresponding head of department at Fraunhofer IEM.

4 Results

In the following, we answer our main research questions:

RQ1 Can users resolve findings in SAST tools more effectively by configuring the tools' analysis scope and rule selection?
RQ2 What strategies do users of static analysis tools in the IDE use to resolve the findings?
RQ3 Which configuration options in the existing tools do users find useful?

4.1 Resolving findings in configurable tools effectively

We asked each participant during the task to clearly state which finding is *true positive*, *false positive*, or if the participant cannot answer (*do not know*). In Table 4, we summarize the data per finding and for each study group. The treatment group, which consisted of participants allowed to configure the tool, resolved 76% of SecuCheck's findings correctly, while the control group only resolved 61% correctly (effectiveness as described in Subsection 3.3).

Table 4 Number of resolved findings per study group and per finding (F01-F09). T = true (correctly resolved), F = false (incorrectly resolved), DN = do not know

	Control group			Treatment group		
	T	F	DN	T	F	DN
F01	12	5	2	16	1	0
F02	10	4	5	12	3	0
F03	8	7	3	9	3	2
F04	14	0	2	13	0	0
F05	8	7	1	9	4	0
F06	5	8	1	9	2	1
F07	11	3	1	8	3	3
F08	15	1	0	17	1	0
F09	6	7	0	9	7	1

Participants using the configuration page resolved the findings reported by SecuCheck more effectively than those participants using a single default configuration. Using the configuration page requires more engagement from the user. This may include understanding of the code and its architecture and/or understanding how the security vulnerabilities can occur.

Furthermore, we considered the participants' profiles (Table 2) to observe whether specific group of participants performed better in resolving the findings. We found out that the participants from academia resolved the findings slightly better than the participants from industry, 74% versus 68%. With respect to the coding experience, the participants with 3-5 years of experience were slightly less effective in resolving than the rest of the participants. To our surprise, there is no significant different when it comes to the role of the participant, i.e., developer versus expert. Finally, the participants labeled as experts with security knowledge, resolved the findings with 79% effectiveness, compared to the others with 60%. However, in these findings, we found no statistical significance (Subsection 3.4).

4.2 Strategies that users have when configuring static analysis tools

To answer **RQ2**, we collected the usage data of the configuration page from the participants in the treatment group. Additionally, we gathered qualitative data from the answers in the third part of the interview. We asked two questions about the strategies each participant used. First, *What was your strategy when you used the configuration page?* (**Q22**), and second, *Would you use the same strategy if this was your own project?* (**Q23**). Based on the answers, the moderator asked further questions to gain more details. Among all answers, we identified that there are four different strategies that the participants named. During the task, all participants used one or two of these strategies, i.e., some of them decided to change their strategy at some point. The first strategy, *AllEntriesSubsetRules*, is to use all possible entry points, and for each run of the analysis to select only a subset of the vulnerability rules. Most of the participants that used this strategy selected a single vulnerability rule per configuration. When asked why they decided to do this, there were only two reasons given: first, to avoid **being overwhelmed by a high number of findings**, and second, to **avoid long running times** of the analysis if everything is selected. One participant said *"I get overwhelmed when I get too many results to*

resolve" and another said *"I was not aware how long would it take to check the vulnerability so selected incrementally"*.

The second strategy, *PairingEntryAndRule*, is to select a combination of entry points and vulnerability rules. When asked why, one participant said *"I tried to match a vulnerability with entry point that makes sense based on the name"*. The third strategy, *SelectAll* is to select everything. The main reason, all participants mentioned, was to make sure that they do not miss any vulnerability. And the last strategy, *AllRulesSubsetEntries*, is to select all vulnerability rules but make different configurations by selecting a subset of the entry points.

Table 5 shows the strategies that each participant used or mentioned during the interview. The participants applied one or two strategies while solving the given task. With 13 participants using it, *AllEntriesSubsetRules* was the most used strategy among all participants. With six participants, the second-most used strategy was *SelectAll*. Based on the interview, we learned that the participants did not use *AllRulesSubsetEntries* during the task as most of them said that they were not familiar with the code given and therefore did now know how to choose which classes will be relevant as entry points. However, when asked if that was their own project whether they would use a different strategy, *AllRulesSubsetEntries* was chosen by most participants, with 13 answers, followed by *PairingEntryAndRule* with nine answers.

On average, the participants re-ran the tool, i.e., selected different configurations, 3.4 times (with σ =1.79), with maximally six times done by four participants. Three participants ran the tool only once (P26, P31, and P38) and they all used *SelectAll*. In doing so, these three participants performed the task the same as the control group, i.e., did not make use of the configuration option. Interestingly, they resolved 24 findings in total and correctly resolved 14 of them, yielding 58%, which is lowest in the group, and conforming to the result of **RQ1**. Participant P31 would always use *SelectAll*, saying *"All at a time all selected! Even if the analysis was much slower, I would leave it run over the night, and get everything. But I would like to have a lazy loading. I should see the relevant findings for the file that I open and not the rest"*.

Most participants selected all entry points at once and ran the configuration individually for each vulnerability rule. When asked if they would use the same strategy if they used the tool on their project, most of them would use the selection of entry points as they would know where to look for particular vulnerabilities. To avoid missing findings, some few participants would only use the default configuration, where all entry points and vulnerability rules are selected. With such strategy the tool will analyze everything, hence the participants have confidence that no findings will be missed by the tool. Yet, these participants were comparatively ineffective.

4.3 Configuration options in commercial tools

To answer **RQ3**, we use the data collected in part four of the questionnaire, i.e. **Q24-25**, where the participant was asked to evaluate the undersandability and the usfullness from the configuration options in FORTIFY and CHECKMARX. Figure 3 shows a screenshot how these options were presented to the participant. For the options we used the exact formulation as defined in the user documentation of the tools. We screened all configuration options that these tools provide and selected the options that impact the precision, recall, or the analysis time. In total, we selected 18 options from FORTIFY (F1-F18) and 11 options from CHECKMARX (C1-C11). See Appendix 6 for the full list.

Table 5 Strategies for using the configuration page

	Strategy used during the task	Strategy would use on own project	Number of configurations submitted	Inspected the rules
P02	*PairingEntryAndRule* then *SelectAll*	*PairingEntryAndRule* then *AllRulesSubsetEntries*	6	no
P06	*AllEntriesSubsetRules*	*AllRulesSubsetEntries*	6	no
P07	*PairingEntryAndRule* then *SelectAll*	*PairingEntryAndRule* then *SelectAll*	2	no
P10	*PairingEntryAndRule* then *AllEntriesSubsetRules*	*PairingEntryAndRule* then *AllRulesSubsetEntries*	2	yes
P13	*AllEntriesSubsetRules*	*AllEntriesSubsetRules* then *AllRulesSubsetEntries*	4	no
P16	*AllEntriesSubsetRules*	*AllEntriesSubsetRules* then *PairingEntryAndRule*	6	yes
P18	*AllEntriesSubsetRules*	*AllRulesSubsetEntries* then *PairingEntryAndRule*	2	no
P19	*AllEntriesSubsetRules*	*AllEntriesSubsetRules* then *SelectAll*	3	yes
P21	*AllEntriesSubsetRules* then *SelectAll*	*AllRulesSubsetEntries*	2	yes
P22	*AllEntriesSubsetRules* then *SelectAll*	*AllRulesSubsetEntries*	2	yes
P24	*AllEntriesSubsetRules*	*AllEntriesSubsetRules* then *AllRulesSubsetEntries*	3	yes
P25	*AllEntriesSubsetRules*	*PairingEntryAndRule*	5	yes
P26	*SelectAll*	*AllRulesSubsetEntries*	1	yes
P27	*AllEntriesSubsetRules* then *SelectAll*	*AllRulesSubsetEntries*	4	yes
P30	*AllEntriesSubsetRules*	*AllRulesSubsetEntries* then *PairingEntryAndRule*	5	yes
P31	*SelectAll*	*SelectAll*	1	yes
P33	*AllEntriesSubsetRules*	*AllRulesSubsetEntries* then *PairingEntryAndRule*	4	yes
P34	*AllEntriesSubsetRules*	*PairingEntryAndRule* then *SelectAll*	3	yes
P38	*SelectAll*	*AllEntriesSubsetRules* then *AllRulesSubsetEntries*	1	yes
P40	*AllEntriesSubsetRules* then *SelectAll*	*PairingEntryAndRule* then *SelectAll*	6	no

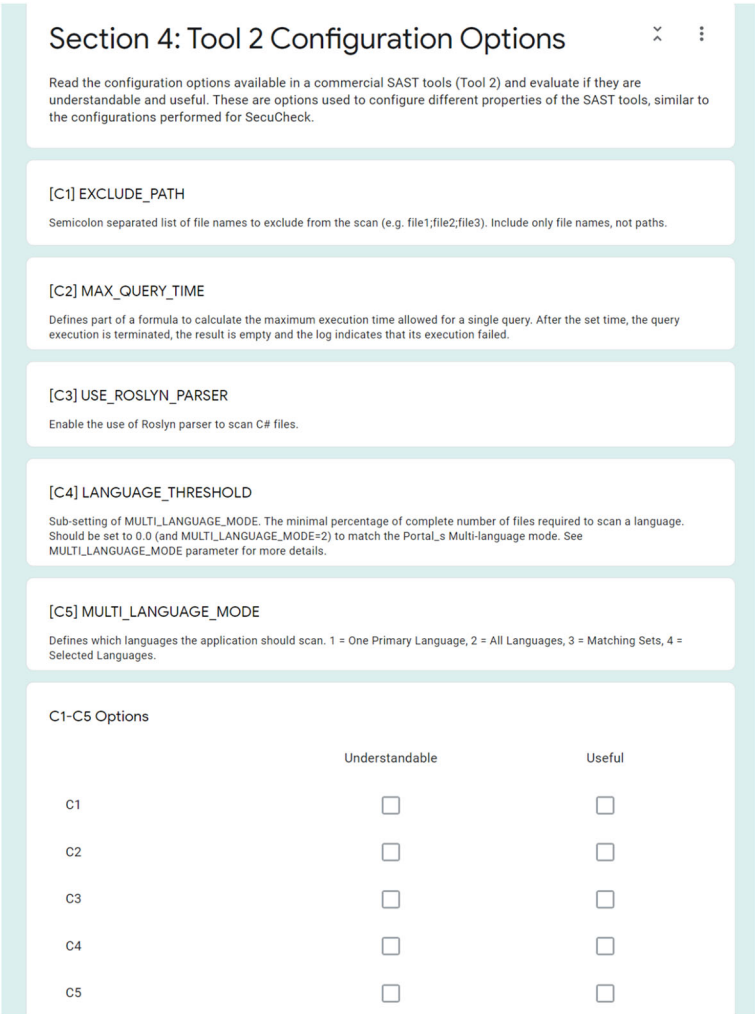

Fig. 3 Screenshot of **Q25** of the questionnaire

Figure 4 shows four boxplots, two for FORTIFY and two for CHECKMARX. Each FORTIFY option is a single data point in the first two boxplots (blue and yellow). Each CHECKMARX option is a single data point in the second two boxplots (green and red). The first boxplot (blue) shows the understandability of the FORTIFY options in percent. The mean is 73.53 (σ=19.69), i.e. 73.53% of the options provided to the users are found to be understandable by the participants. There are four outliers on the upper part, F7 (*noDefaultRules*), F9 (*rules*), F11 (*dataflowMaxFunctionTimeMinutes*), and F12 (*maxFunctionVisits*), which are options that are found understandable by most participants (over 85%). There are also four outliers on the lower part, F6 (*noDefaultIssueRules*), F10 (*enableInterproceduralConstantResulution*), F13 (*maxTaintDefForVar*), and F14 (*maxTaintDefForVarAbort*), which are options that were found least understandable (under 63%). The second boxplot (yellow) shows the understandability and usefulness of the FORTIFY options in percent. The mean is 78.12 (σ=13.4).

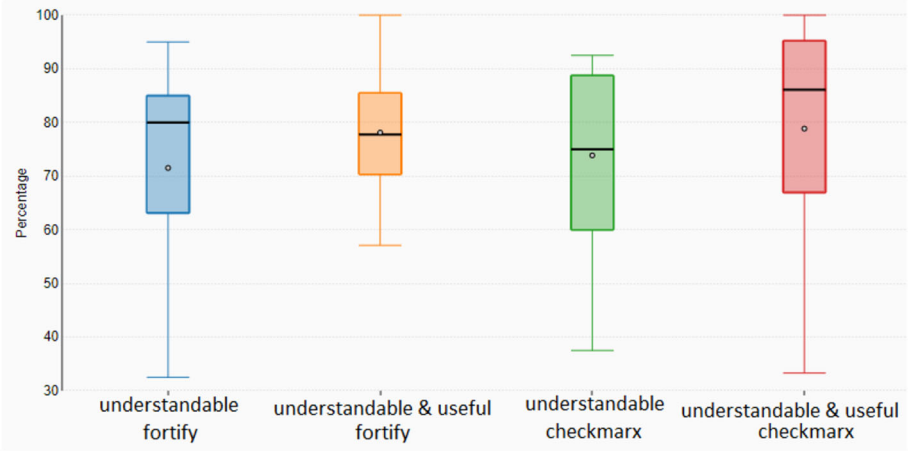

Fig. 4 BoxPlots for percentage of participants that marked each option as *understandable* and *useful* for each tool, FORTIFY (blue and yellow) and CHECKMARX (green and red)

There are four outliers on the upper part (over 85%), F1 (*filter*), F2 (*disableSourceBundling*), F4 (*analyzers*), and F9 (*rules*), and four in the lower part (under 70%), F3 (*disableLanguage*), F8 (*noDefaultSinkRules*), F10 (*enableInterproceduralConstantResolution*), and F14 (*maxTaintDefForVarAbort*). The third boxplot (green) shows the understandability of the CHECKMARX options in percent. The mean is 73.86 (σ=17.98). There are three outliers on the upper part, C2 (*maxQueryTime*), C6 (*scanBinaries*), and C11 (*maxQueryTimePer100K*), which are options that are found understandable by most participants (over 88%). There are also three outlies on the lower part, C2 (*maxQueryTime*), C3 (*useRoslynParser*), and C11 (*maxQueryTimePer100K*), which are options that were found least understandable (under 60%). The fourth boxplot (red) shows the understandability and usefulness of the CHECKMARX options in percent. The mean is 78.85 (σ=21.15). There are three outliers on the upper part (over 95%), C1 (*excludePath*), C5 (*maxQueryTime*), and C6 (*scanBinaries*), and three in the lower part (under 66%), C4 (*languageThreshold*), C9 (*maxFileSizeKb*), and C10 (*maxPathLength*). When compared the mean and standard deviation between both tools there are no significant differences.

Finally, we categorize the options into three categories. Category (1) includes options that are related to the analysis scope (F1-3, C1, C4-7), Category (2) includes options that impact the approximations done by the solver or set thresholds for analysis time (F4-5, F10-18, C2-3, C8-11), and Category (3) includes options that are related to rule selection (F6-9). The most understandable category is (2) with average score of 32.5, but with least percentage of usefulness 70%. Category (1) has average score of 29.25 and most usefulness with 85%, where as (3) has score of 28 and 74% usefulness.

A quarter of the participants find the provided options to be not understandable. The other three quarters find the options to be on average 78% useful. The most useful are the options related to the analysis scope.

4.4 Recommendations for building the future SAST tools

As introduced in Section 3.2, we used SecuCheck as a SAST tool for our user study. SecuCheck provides a configuration page with the options we needed for the study. Other commercial tools have many more configuration options that would have distracted the participants. After the participant had experienced the tool, in part three of the interview we also asked questions about the experience with the tool. In particular, we asked what features of the tool the participants liked and disliked, and what other features they would like to have when working on a similar tool within the IDE. Moreover, in part two we asked questions about their previous experience with SAST tools. In the following we present the results. Based on these results and the results from the previous sections we propose a list of recommendations for future SAST tools.

Twenty four of the 40 participants said that they use SAST tools in their everyday workflow. Out of these, 17 use SAST tools within the IDE. The tools that participants named are shown in Figure 5. When asked if they configure those tools, 30 participants said they do not, while only 10 said they do. Only 8 participants said that they have used a domain-specific language to configure the tools. During the task, we also made the vulnerability detection rules available to the users and let them know that they are available. Then we observed how many of them will actually open the files and inspect them. As seen in Table 5 most of them did. However, many said that they would not write the rules on their own but they like to have them available to help them decide if the findings reported by the tool are true or false positives. For example, one participant checked the rule to verify if the tool was aware of potential sanitizers. Another participant said, she prefers writing the rules and never uses default rules, as only then she is sure that the tool is doing the right job. 27 participants said there are regulations or policies by the company for using SAST tools. 22 participants said they are allowed to configure the tools, while 13 answered they are not.

Based on these results, we can make the first recommendation for tool developers and the future development of SAST tools.

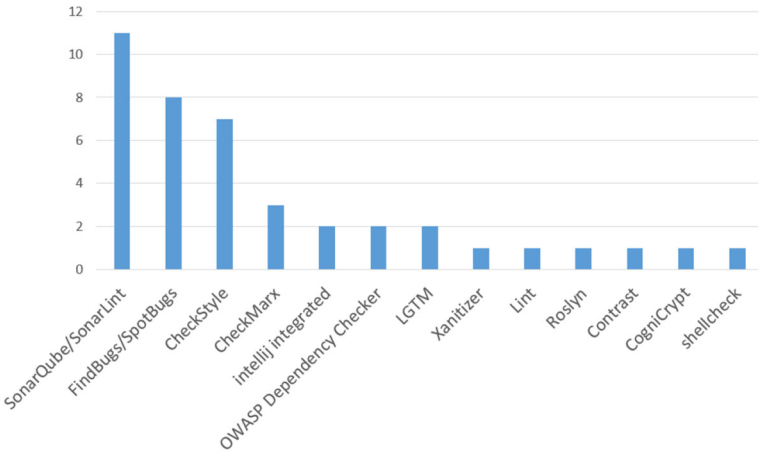

Fig. 5 Tools that participants use or used in the past and number of participants that named each tool

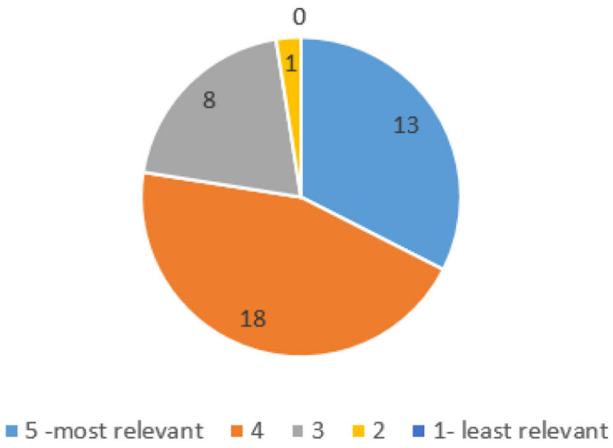

Fig. 6 How relevant is the following statement: *The issues reported by the tool should be relevant for me (the context I am currently focused on)*?

Recommendation 1: SAST tools vendors should provide the implementation of the rules of the analysis to the users for inspections or modification. With such strategy, it is expected to reach broader number of users. Additionally, these rule should be specified in a language that is developer-centric with good documentation. In particular, this will reach more specific group of users who express interest in the security domain. It is expected that these users will specify new rules that are specific to the codebase. A mechanism to share newly created rules among teams is a promising future work.

In part three of the interview, we asked the participants to name the things they liked, disliked, and missed about the tool they used to perform the given task. Most of the participants liked the integration of the analysis within the standard features of the IDE, such as error view list, error markers, clicking links to the findings, etc. The most disliked feature of the tool was that the configuration page was in the web-browser and not within the IDE. This was perceived as a usability issue due to context-switching. We have contacted the authors of MagpieBridge, which we used to build our tool, and they extended the framework to allow the page to be opened also within the IDE if the respective IDE supports this. Additionally, we pointed out some few other usability issues and cooperated with them to make the framework support more UI elements for the configuration page. Among the features that the participants missed, there were two which were mentioned frequently. First, the participants wanted to see a better visualization of the data-flow path between the source and the sink. Second, participants said that our tool had a limited description of the vulnerability reported in the error view. They prefer to see a feature where the description can be expanded to show further details, examples and possible fixes or proposals which validation libraries to use. Based on this data, we make the second recommendation for tool vendors.

Recommendation 2: The SAST tools should have full integration within the IDE, i.e., the tool should have native look within the IDE. Additionally, when reporting, the tools should have rich explainability mechanism for data-flow path and educational information for the vulnerabilities that are reported. This includes features like error markers and witness-path that explains the relevant statements in the code and the entry point for reaching the vulnerable code. A particular care should be put in the messages used to explain the vulnerability. Ideally, this should include examples for secure and insecure code.

In Section 4.2, based on the questions after the task we learned that many participants would use the entry points selection if they worked on their own project (Table 5). Additionally, in part two of the interview we asked the participants to rank the following statement *"The issues reported by the tool should be relevant for me, i.e., the context I am currently working on."*. Figure 6 shows the distribution of the responses showing that for most participants the reported issues should be relevant for the context. We also asked the participants about the granularity level of the entry points. In our study, we chose entry points to be on the class level. As answers, we offered method level, class level, package level, or other. Figure 7 shows the distributions of the answers. Under *Other*, we received 2 answers "all", 2 answers combination of two of the given options, 1 answer "annotations" and 1 answer "hierarchical starting at package level". Based on this data, we make the next recommendation. Related to the entry point granularity, we can make the next rexommendation.

Recommendation 3: The SAST tools in the IDE should provide options for the analysis scope in which the users can select which parts of the project should be analyzed. Popular choices for this are selection of methods or classes as entry points of the analysis. Next step for the future is to automate this task, by identifying the code that the user worked lately (e.g., from the IDE usage data, or recent commits) and making an initial recommendation based on that.

Finally, we refer to the results from our user study where we learned that the participants from the treatment group were able to resolve the findings more effectively than participants from the control group. From the data, we learned that most participants used the vulnerability rules as main selection criterion. They did not use the entry points as they were not very familiar with the code. This is a realistic scenario where security teams or quality assurance teams are performing the analysis for code that they have not written themselves. We also observed that all participants started with resolving the *SQL injection* (Mitre CWE, 2021b) findings. This is one of the most popular vulnerability nowadays. This shows that the users of SAST tools will probably focus on the vulnerabilities that they know and are more likely to solve. Finally, we make our fourth recommendation.

Recommendation 4: The tools should enable users to selectively enable or disable the vulnerability rules. With that users can focus on familiar vulnerabilities that they are confident to work on. Ideally, the vulnerabilities that are unfamiliar for the users, can be resolved by security experts in the team or the organization.

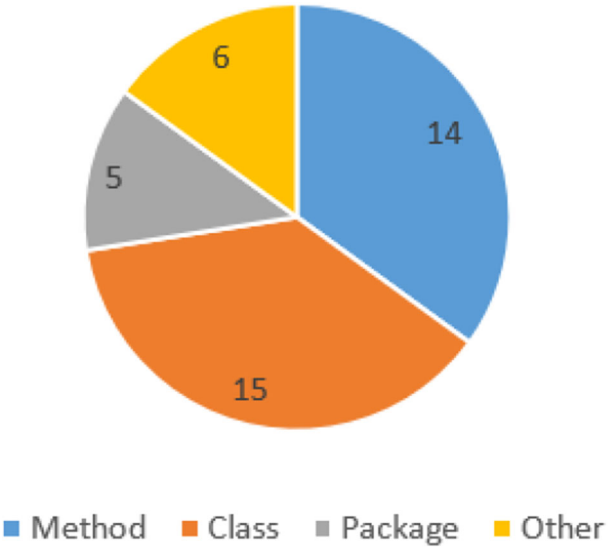

Fig. 7 On what level would you prefer the entry points selection option to be?

5 Threats to validity

We next discuss the most relevant threats to the validity of our study design and evaluation based on the threat types by (Cook and Campbell, 1979).

Construct validity A possible threat to the validity of the user study relates to its setup. All participants performed the study remotely by sharing voice, video, and screen. During the task, the moderator noted the outcome of the findings that the respective participant resolved. To mitigate the risk of human mistake, in the post-study processing phase the first author watched all videos to collect and confirm the results. We had a 100% inter-rater reliability score. Additionally, we collected notes of each session from which we created an artifact from our study.

Internal validity To avoid any random answers to the questionnaire, we asked the participant to share the screen and give further comments to some questions while guided by the moderator in an interview style. To confirm the clarity of the questions and the timing of our study, we ran four tests with students from our research department.

External validity The participation in the study was voluntary and without compensation. We asked our contacts from industry to invite their software developers and additionally, we invited researchers and students from Paderborn University. The invitation explained that the user will evaluate the configuration capabilities of SAST tools. Having this information, it is more likely that the participants have some interest security and static analysis. This might make our population biased towards SAST tools.

We consciously chose a study design that would yield high internal validity, at the necessary cost of limiting external validity (Siegmund et al., 2015). This is limited by the tool we used for the study and the choice of example project. The tool is limited to taint analysis, while other SAST tools also include other types of analyses. However, most popular vulnerabilities

are of taint-style and the core of most SAST tools is a taint analysis (Piskachev et al., 2014). The project we used is artificially created but it includes different components that modern web application would have. The vulnerabilities within the application are inserted based on existing vulnerabilities that we found in OWASP benchmark (Benchmark O, 2021) and OWASP Webgoat (WebGoat, 2021). We decided to use our own tool as this gave us control over what features to include and exclude.

The fact that several participants noted that the analysis was fast compared to what they expect from a SAST tool, is due to the reason that our example project is relative small compared to most real-world projects in industry. Since this may impact the strategies that the participants used and discussed in **Q22**, we additionally asked **Q23**, i.e., how they would have used the tool if that was their own project where the analysis time would be an important factor. This question allowed us to gain further insights relating to more real-world situations, strengthening external validity.

6 Conclusion

Static analysis' trade-off for precision, recall, and runtime is addressed through a large configuration space of SAST tools. However, many of the configuration options are made for static analysis experts. In this paper, we studied how the users of SAST tools, including developers, static analysis experts, and security experts, are able to use a subset of the configuration space. We focused on two configuration options that largely impact the precision, recall, and runtime: the selection of security rules and of the analysis scope (via selection of entry points). It proved that users exploiting these configuration options resolve the findings from the tool more effectively. From the quantitative and qualitative data we were able to identify the strategies that the users apply while using the configuration options. For projects that they are not familiar with, in each new configuration most participants would iteratively select a subset of the security rules. For their own projects which they are familiar with, they would use a subset of the entry points, fine-tuning the analysis scope. Additionally, we asked the participants to evaluate the usability of the related configuration options from two popular commercial SAST tools. We found out that the options need improved descriptions since quarter of the participants do not understand them. Moreover, the most useful options found by the participants are related to the analysis scope.

Finally, based on our results, we created a list of recommendations that SAST tools creators should consider for the future tool in order to address the expectations and needs of the different users.

Appendix

List of configuration options from FORTIFY used in the user study.

- F1 filter - Apply a filter using a filter file
- F2 disablesource-bundling - Exclude source files from the FPR file
- F3 disable-language - To disable specific languages
- F4 analyzers - To disable specific analyzers, include this option at scan time with a colon- or comma-separated list of analyzers you want to enable. The full list of analyzers is: buffer, content, configuration, control-flow, data-flow, findbugs, nullptr, semantic, and structural

F5 incremental-base - Specifies that this is the initial full scan of a project for which you plan to run subsequent incremental scans. Use this option for the first scan when you plan to run subsequent scans on the same project with the incremental option
F6 no-default-issue-rules - Disables rules in default Rulepacks that lead directly to issues. Still loads rules that characterize the behavior of functions. Note: This is equivalent to disabling the following rule types: Data-flow Sink, Semantic, Control-flow, Structural, Configuration, Content, Statistical, Internal, and Characterization:Issue
F7 no-default-rules - Specifies not to load rules from the default Rulepacks
F8 no-default-sink-rules - Disables sink rules in the default Rulepacks
F9 rules - Specifies a custom Rulepack or directory. You can use this option multiple times to specify multiple Rulepack files. If you specify a directory, includes all of the files in the directory with the .bin and .xml extensions
F10 enableInterproceduralConstantResolution - Use constant resolution
F11 dataflowMaxFunctionTimeMinutes - Set a threshold to limit the data-flow analysis time of a single function
F12 maxFunctionVisits - Set a threshold for the number of times a function is analyzed
F13 maxTaintDefForVar - Dimensionless value expressing the complexity of a function
F14 maxTaintDefForVarAbort- The upper bound for MaxTaintDefForVar
F15 maxChainDepth - Set a threshold for depth of functions chain
F16 alias.EnableInterprocedural - Enable interprocedural alias analysis.
F17 maxFieldDepth Set a threshold for the depth of the analyzed fields.
F18 maxPaths - Set a threshold for the number of analyzed paths

List of configuration options from CHECKMARX used in the user study.

C1 EXCLUDE_PATH - Semicolon separated list of file names to exclude from the scan (e.g. file1;file2;file3). Include only file names, not paths
C2 MAX_QUERY_TIME- Defines part of a formula to calculate the maximum execution time allowed for a single query. After the set time, the query execution is terminated, the result is empty and the log indicates that its execution failed
C3 USE_ROSLYN_PARSER - Enable the use of Roslyn parser to scan C# files
C4 LANGUAGE_THRESHOLD - Sub-setting of MULTI_LANGUAGE_MODE. The minimal percentage of complete number of files required to scan a language. Should be set to 0.0 (and MULTI_LANGUAGE_MODE=2) to match the Portal_s Multi-language mode. See MULTI_LANGUAGE_MODE parameter for more details
C5 MULTI_LANGUAGE_MODE - Defines which languages the application should scan. 1 = One Primary Language, 2 = All Languages, 3 = Matching Sets, 4 = Selected Languages
C6 SCAN_BINARIES - Whether or not to scan binary files (only available for .jar files - Java - and for .dll files - C#). *Note*: Requires Java to be installed on the machine
C7 SUPPORTED_LANGUAGES - Sub-setting of MULTI_LANGUAGE_MODE. If MULTI_LANGUAGE_MODE = 1 or 2 ignore/meaningless. If MULTI_LANGUAGE_MODE = 4 then languages are separated by commas. See MULTI_LANGUAGE_MODE parameter for more details
C8 TYPES_TO_DECOMPILE - When SCAN_BINARIES is set to true, this flag should be used to specify which packages/namespaces should be decompiled and then included in the scan. Format x.y.* can be used to specify that all the types under package/namespace x.y should be decompiled and scanned. The list of packages/namespaces should be separated by a semicolon (;).
C9 MAXFILESIZEKB - Files exceeding the set size (in KB) will not be scanned

C10 MAX_PATH_LENGTH - Defines the maximum amount of flow elements allowed in an influence flow calculation. Paths with length exceeding this number are ignored
C11 MAX_QUERY_TIME_PER_100K - Sub setting of MAX_QUERY_TIME. Defines part of formula to calculate the maximum execution time allowed for a single query. See MAX_QUERY_TIME parameter for more details

Acknowledgements We gratefully acknowledge the funding by the project "AppSecure.nrw - Security-by-Design of Java-based Applications" of the European Regional Development Fund (ERDF-0801379). We thank Ranjith Krishnamurthy and Oshando Johnson for their work on the user study

Author Contributions The first author is the main contributor to this research. The second and third author contributed with conceptual ideas and feedback

Funding Open Access funding enabled and organized by Projekt DEAL.

Data Availability https://research-sast-config.github.io/

Code Availability https://github.com/secure-software-engineering/secucheck

Declarations

Funding and/or Conflicts of interests/Competing interests Not applicable

Ethics approval The user study design has been approved for ethical correctness by one of the companies participated in the study as well as by the corresponding head of department at Fraunhofer IEM

Consent to participate and publication For the user study, all 40 participants signed a written consent form in which they agreed to participate voluntarily in the study. They also agreed that the collected data can be used for research publication. The written consent form was obtained from all participants before the study

References

Arzt S, Rasthofer S, Fritz C, Bodden E, Bartel A, Klein J, Le Traon Y, Octeau D, McDaniel P (2014) Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. Acm Sigplan Notices 49(6):259–269

Benchmark O (2021) Owasp. https://owasp.org/www-project-benchmark/, online; Accessed January 2021

Charness G, Gneezy U, Kuhn MA (2012) Experimental methods: Between-subject and within-subject design. J Econ Behav Organ 81(1):1–8. https://doi.org/10.1016/j.jebo.2011.08.00, https://ideas.repec.org/a/eee/jeborg/v81y2012i1p1-8.html

Checkmarx (2021) Checkmarx. Online; Accessed January 2021

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, ACM, New York, NY, USA, ASE 2016, pp 332–343

Cook TD, Campbell DT (1979) Quasi-Experimentation: Design and Analysis Issues for Field Settings. Houghton Mifflin, Boston, USA, Boston

Facebook (2021) Infer. Online; Accessed January 2021

Github S (2021) Lgtm. Online; Accessed January 2021
Gordon M, deokhwan K, Perkins J, Gilham L, Nguyen N, Rinard M (2015) Information-flow analysis of android applications in droidsafe. In: Network and Distributed System Security Symposium 2015, 10.14722/ndss.2015.23089
Grammatech (2021) Codesonar. Online; Accessed January 2021
Habib A, Pradel M (2018) How many of all bugs do we find? a study of static bug detectors. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, Association for Computing Machinery, New York, NY, USA, ASE 2018, p 317–328, https://doi.org/10.1145/3238147.3238213
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 2013 International Conference on Software Engineering, IEEE Press, ICSE '13, p 672–681
Krüger S, Späth J, Ali K, Bodden E, Mezini M (2018) CrySL: An Extensible Approach to Validating the Correct Usage of Cryptographic APIs. In: European Conference on Object-Oriented Programming (ECOOP), pp 10:1–10:27, https://bodden.de/pubs/ksa+18crysl.pdf
Lee W, Lee W, Kang D, Heo K, Oh H, Yi K (2017) Sound non-statistical clustering of static analysis alarms. ACM Trans Program Lang Syst 39(4), https://doi.org/10.1145/3095021
Luo L, Bodden E, Späth J (2019) A qualitative analysis of android taint-analysis results. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp 102–114, 10.1109/ASE.2019.00020
Luo L, Dolby J, Bodden E (2019) MagpieBridge: A General Approach to Integrating Static Analyses into IDEs and Editors (Tool Insights Paper). In: Donaldson AF (ed) 33rd European Conference on Object-Oriented Programming (ECOOP 2019), Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, Leibniz International Proceedings in Informatics (LIPIcs), vol 134, pp 21:1–21:25, https://doi.org/10.4230/LIPIcs.ECOOP.2019.21,http://drops.dagstuhl.de/opus/volltexte/2019/10813
of Maryland U (2021) Findbugs. Online; Accessed January 2021
Microfocus (2021) Fortify. Online; Accessed January 2021
Mitre CWE (2021a) 2011 cwe/sans top 25 most dangerous software errors. http://cwe.mitre.org/top25/, online; Accessed January 2021
Mitre CWE (2021b) Improper neutralization of special elements used in an sql command. https://cwe.mitre.org/data/definitions/89.html, online; Accessed January 2021
Naiakshina A, Danilova A, Tiefenau C, Herzog M, Dechand S, Smith M (2017) Why do developers get password storage wrong? a qualitative usability study. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, Association for Computing Machinery, New York, NY, USA, CCS 17, pp 311–328, https://doi.org/10.1145/3133956.3134082
Naiakshina A, Danilova A, Tiefenau C, Smith M (2018) Deception task design in developer password studies: Exploring a student sample. Proceedings of the Fourteenth USENIX Conference on Usable Privacy and Security, USENIX Association, USA, SOUPS 18:297–313
Naiakshina A, Danilova A, Gerlitz E, von Zezschwitz E, Smith M (2019) If you want, i can store the encrypted password: A password-storage field study with freelance developers. In: Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems, Association for Computing Machinery, New York, NY, USA, CHI 19, pp 1–12
Naiakshina A, Danilova A, Gerlitz E, Smith M (2020) On conducting security developer studies with cs students: Examining a password-storage study with cs students, freelancers, and company developers. In: Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems, Association for Computing Machinery, New York, NY, USA, CHI 20, pp 1–13
Nguyen Quang Do L, Bodden E (2020) Explaining static analysis with rule graphs. IEEE Transactions on Software Engineering pp 1–1, 10.1109/TSE.2020.3004525
Nguyen Quang Do L, Ali K, Livshits B, Bodden E, Smith J, Murphy-Hill E (2017) Just-in-time static analysis. In: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, ACM, New York, NY, USA, ISSTA 2017, pp 307–317, https://doi.org/10.1145/3092703.3092705
Nguyen Quang Do L, Wright JR, Ali K (2020) Why do software developers use static analysis tools? a user-centered study of developer needs and motivations. In: Proceedings of the Sixteenth Symposium on Usable Privacy and Security, 10.1109/TSE.2020.3004525
Patnaik N, Hallett J, Rashid A (2019) Usability smells: An analysis of developers' struggle with crypto libraries. In: Fifteenth Symposium on Usable Privacy and Security (SOUPS 2019), USENIX Association, Santa Clara, CA, https://www.usenix.org/conference/soups2019/presentation/patnaik
Piskachev G, Do LNQ, Bodden E (2019) Codebase-adaptive detection of security-relevant methods. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, Association for Computing Machinery, New York, NY, USA, ISSTA 2019, pp 181–191

Piskachev G, Krishnamurthy R, Bodden E (2021) Secucheck: Engineering configurable taint analysis for software developers. In: 21st IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2021), IEEE, Luxembourg

Piskachev G, Späth J, Budde I, Bodden E (2022) Fluently specifying taint-flow queries with fluenttql. Empirical Softw Eng 27(5), https://doi.org/10.1007/s10664-022-10165-y

Qiu L, Wang Y, Rubin J (2018) Analyzing the analyzers: Flowdroid/iccta, amandroid, and droidsafe. In: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, Association for Computing Machinery, New York, NY, USA, ISSTA 2018, pp 176–186, https://doi.org/10.1145/3213846.3213873

Sadowski C, van Gogh J, Jaspan C, Söderberg E, Winter C (2015) Tricorder: Building a program analysis ecosystem. In: Proceedings of the 37th International Conference on Software Engineering - Volume 1, IEEE Press, ICSE '15, pp 598–608

Siegmund J, Siegmund N, Apel S (2015) Views on internal and external validity in empirical software engineering. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, IEEE, vol 1, pp 9–19

Smith J, Johnson B, Murphy-Hill E, Chu B, Lipford H (2015) Questions developers ask while diagnosing potential security vulnerabilities with static analysis. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ACM, New York, NY, USA, ESEC/FSE 2015, pp 248–259

Smith J, Nguyen Quang Do L, Murphy-Hill E (2020) Why can't johnny fix vulnerabilities: A usability evaluation of static analysis tools for security. In: Proceedings of the Sixteenth Symposium on Usable Privacy and Security, SOUPS 2020

Snyk (2021) Deepcode. Online; Accessed January 2021

SonarSource (2021) Sonarqube. Online; Accessed January 2021

Thomas TW, Lipford H, Chu B, Smith J, Murphy-Hill E (2016) What questions remain? an examination of how developers understand an interactive static analysis tool. In: Twelfth Symposium on Usable Privacy and Security (SOUPS 2016), USENIX Association, Denver, CO, https://www.usenix.org/conference/soups2016/workshop-program/wsiw16/presentation/thomas

Vassallo C, Panichella S, Palomba F, Proksch S, Zaidman A, Gall HC (2018) Context is king: The developer perspective on the usage of static analysis tools. In: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp 38–49

Veracode (2021) Veracode. Online; accessed January 2021

WebGoat (2021) Owasp webgoat. https://owasp.org/www-project-webgoat/, online; Accessed January 2021

Wei F, Roy S, Ou X, Robby (2018) Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. ACM Trans Priv Secur 21(3),https://doi.org/10.1145/3183575

Witschey J, Zielinska O, Welk A, Murphy-Hill E, Mayhorn C, Zimmermann T (2015) Quantifying developers' adoption of security tools. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ACM, New York, NY, USA, ESEC/FSE 2015, pp 260–271, https://doi.org/10.1145/2786805.2786816

Zampetti F, Scalabrino S, Oliveto R, Canfora G, Di Penta M (2017) How open source projects use static code analysis tools in continuous integration pipelines. In: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), pp 334–344

Zhang X, Grigore R, Si X, Naik M (2017) Effective interactive resolution of static analysis alarms. Proc ACM Program Lang 1(OOPSLA), https://doi.org/10.1145/3133881

Goran Piskachev (Ph.D. 2022) is an applied scientist in the Privacy Engineering Team at Amazon Web Services in Berlin. Previously, he was a team manager for development tools for secure services and apps as well as a research associate in the same team at Fraunhofer IEM in Paderborn. He received his doctoral and master degree in computer science at Paderborn University. He completed an engineering degree at the Ss. Cyril and Methodius University in Skopje. His research interests include static code analysis, security testing, privacy engineering, domain specific languages, and machine learning for code analysis

Matthias Becker (Ph.D 2017) is head of the department *Secure Services and Apps* at the *Fraunhofer IEM* in Paderborn, Germany. He received his doctoral degree in 2017 in the area of software quality with a dissertation about scalability prediction of large-scale software systems. His research interests include software quality, especially software security, static code analysis, machine learning, and model-driven engineering. Dr. Becker received a postgraduate diploma in management at the European School of Management and Technology (ESMT), Berlin in 2022 and is member of the Bitkom Management Club

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 and IT Security at the Fraunhofer Institute for Engineering Mechatronic Systems Design. 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. He is an ACM Distinguished Member