What Developers Want and Need from Program Analysis: An Empirical Study

Maria Christakis Christian Bird
Microsoft Research, Redmond, USA
{mchri, cbird}@microsoft.com

ABSTRACT

Program Analysis has been a rich and fruitful field of research for many decades, and countless high quality program analysis tools have been produced by academia. Though there are some well-known examples of tools that have found their way into routine use by practitioners, a common challenge faced by researchers is knowing how to achieve broad and lasting adoption of their tools. In an effort to understand what makes a program analyzer most attractive to developers, we mounted a multi-method investigation at Microsoft. Through interviews and surveys of developers as well as analysis of defect data, we provide insight and answers to four high level research questions that can help researchers design program analyzers meeting the needs of software developers.

First, we explore what barriers hinder the adoption of program analyzers, like poorly expressed warning messages. Second, we shed light on what functionality developers want from analyzers, including the types of code issues that developers care about. Next, we answer what non-functional characteristics an analyzer should have to be widely used, how the analyzer should fit into the development process, and how its results should be reported. Finally, we investigate defects in one of Microsoft's flagship software services, to understand what types of code issues are most important to minimize, potentially through program analysis.

CCS Concepts

•General and reference → Empirical studies;
•Software and its engineering → Software defect analysis;

Keywords

program analysis, code defects

1. INTRODUCTION

Large software companies have recently started building program analysis ecosystems, like Google's Tricorder [49] or Microsoft's CloudBuild [31]. These ecosystems allow for distributively running several analyzers, each with its own attributes, like speed of the analysis, type of detected code issues, or number of true or false positives. Designers of such ecosystems need to decide which analyzers should run and when, e.g., in the editor, as part of the build, or during code review. But how should the decisions be made? Which kinds of program analyzers are valuable to software engineers, rather than a waste of time? How do they fit in the development process? How should their results be reported?

So far, much research and many studies on program analysis tools have focused on the completeness of these tools (do they report spurious warnings?), their soundness (do they miss bugs?), automation, performance, annotation overhead, and modularity. However, as companies integrate program analyzers as part of their development process, more investigation is needed into how these tools are used in practice and if practitioners' needs are being met. We posit that for research in this area to be impactful, our community must understand the practices and needs of software developers with regard to program analysis.

In an effort to improve this understanding, our paper contains an empirical investigation at Microsoft to answer the following high level research questions.

1. What barriers hinder the adoption of program analyzers by practitioners?

2. What functionality do practitioners want from program analyzers?

3. What non-functional characteristics should a program analyzer have to be widely used?

4. What code issues occur most in practice that program analyzers should try to detect?

For our purposes, we define *program analysis* as the process of automatically analyzing the behavior of a program without running it, that is, we are only considering static program analysis. Program analysis detects potential issues in the code and gives feedback. Feedback is in the form of warnings that are either true or false positives. *True positives* flag real issues in the code, whereas *false positives* warn about code issues that do not occur in practice. We do not consider the compiler to be a program analyzer to only focus on tools whose primary functionality is program analysis and that are not by default part of the software development process.

Our study comprises a number of investigative techniques. We interviewed and surveyed developers from a diverse group

of products to understand their needs and how program analyzers can or do fit into their process. We also examined many corrected defects to understand what types of issues occur most and least often.

We expect the empirical results that we present here to shed light on many aspects of program analysis, specifically, on what tools should be integrated in the development process, where tool designers should focus their efforts, what developers like and dislike in analyzers, what types of code issues are most often encountered, and what project managers should expect from different bug-finding techniques.

We assert that by understanding the above, the program analysis research community can focus on analyzers that are most amenable to real world use. For researchers, our findings also provide a view into today's industrial realities with respect to program analysis.

2. SURVEY

In an effort to understand developers' perspectives on program analyzers, we deployed a broad survey across Microsoft. Surveys are beneficial because they allow researchers to elicit answers to the same set of questions from a large sample of some population. In our case, we are interested in industrial software developers. Our goal is to obtain a large enough sample such that responses are representative of the population and that quantitative analysis can find results with statistical significance (if indeed there are signals in the responses). Surveys have been used in empirical software engineering investigations many times in the past to provide insight [48].

2.1 Data and Methodology

We used Kitchenham and Pfleeger's guidelines for personal opinion surveys in software engineering research when designing and deploying our survey [41]. We followed a pilot and beta protocol when developing the survey. We started by identifying the high level goals for our investigation:

- Uncover any obstacles in the adoption of program analyzers by developers.
- Understand how practitioners use program analyzers today and what functionality they find desirable.
- Identify the non-functional characteristics that developers want in a program analyzer.
- Determine how program analyzers should fit into developers' current practices.

From these goals, we derived an initial set of survey questions. To pilot our questions, we scheduled interviews with five developers across Microsoft and administered our survey questions in person. This allowed us to gauge if each question was clear enough or should be altered, if the terms we used were familiar to developers, and if the questions we asked were actually eliciting answers that helped us achieve our goals for the survey.

After updating the questions following these interviews, we created a beta of our survey that we deployed to 100 developers randomly selected across the company. This initial survey included additional questions at the end, asking participants if they found any portion of the survey difficult to understand or answer, and asking if they had any other relevant information to share about the topic. We received 20 responses to this survey. These responses were solely used to improve the survey itself and were not included in subsequent data analysis presented in this paper.

We then made improvements to the survey based on responses to the beta. An example of such changes included defining terms such as "aliasing" and "purity" more clearly. In another case, we had a question with check boxes that asked developers which types of code issues they would like program analyzers to detect. This question was changed so that developers had to create a ranking of the types of issues; in the beta, some developers checked almost all of the boxes, making the answers less informative. A few of our questions were open ended (for example, "Why did you stop using program analyzers?"), and the responses to the beta showed that there were clear categories in the answers. For these, we changed the question to a multiple choice format that included each of the categories, and we added a write-in option if the respondents' answer did not fit into one of these categories. Such changes allow analysis to scale with large numbers of responses. We also made changes to the survey to ensure that it did not take too long to answer, as long surveys may deter participation. Our goal was for the survey to take a respondent approximately 15 minutes to complete.

After finalizing the questions, we sent invitations to answer the survey to 2,000 developers selected at random across all of Microsoft. The survey was anonymous as this increases response rates [52] and leads to more candid responses. As incentives have been shown to increase participation [51], respondents could enter themselves into a raffle to win four $50 Amazon gift cards. We received 375 responses to the final survey, yielding a 19% response rate. Other online surveys in software engineering have reported response rates from 14% to 20% [48]. The median time to complete the survey was 16 and a half minutes, quite close to our goal of 15 minutes. We report the median rather than average because there were some outliers that skew the average (one person had a completion time of just under five days!). The range of years of development experience was from zero to 43, with a median of nine (mean of 10.86).

The survey questions, responses, and analysis scripts can be accessed at `https://github.com/cabird/ProgramAnalysisSurvey`.

2.2 Results

We break our results down into three categories. First, we look at the barriers to using program analyzers and the reasons why developers stop using them.

Second, we examine the functionality that the developers' answers indicate they want in program analyzers. This functionality includes the types of issues that program analyzers catch, the types of programming languages they can analyze, whether the analyzer examines a whole program or changes, and if the developer can direct the program analyzer toward parts of the code.

Third, we look at the non-functional characteristics that a program analyzer should have. This includes attributes such as the time required for analysis, how many false positives it should yield, when it should run, where the output should be and what form it should take, and where the analysis should fit into the development process.

In addition, for most questions, we break down our answers by attributes of the respondents. From our interviews and based on anecdotal evidence, we believe that developers

who have at least a basic understanding of program analysis may have different views about the topic than those who are not familiar with it. For the context of this paper, we label these developers *experts*. 74% of respondents were at least familiar with program analysis. In addition, security issues are especially important to software companies, and security is often given high priority by development teams. In the research community, security is a significant subarea in program analysis that receives a large amount of attention. We refer to developers who indicate that security is a top concern to them as *security developers*. 40% of respondents indicated that they are security developers. For many questions, we examine the answers provided by developers who are familiar with program analysis and also by those who indicate that security is a top concern for them. We report cases where there is a statistically significant difference between these groups and the answers of the rest of the sample. In cases where there are only two alternatives (e.g., using program analysis versus not using it), we use a Fisher's exact test [30]. When there are more than two choices, such as the frequency of running program analysis, we use a χ^2 test to assess the difference in distributions between these groups.

Some of the questions on our survey asked developers to select and rank items from a list. For example, we asked developers to rank the pain points they encountered using program analysis as well as the code issues that they would like program analyzers to detect. To analyze the answers, for each option o, we compute the sum of the reciprocals of the rank given to that option for each developer d that responded ($d \in D$):

$$Weight(o) = \sum_{d \in D} \frac{1}{Rank_d(o)}$$

Ranks start at one (the option with the greatest importance) and go up from there. If an option is not added to the ranked list by a developer, the option is given a weight of zero for that developer.

In Section 5, we also give an overview of the program analyzers that the survey respondents use the most.

2.2.1 What makes program analyzers difficult to use?

In our beta survey, we asked developers what pain points, obstacles, and challenges they encountered when using program analyzers. We then examined their responses to create a closed response list of options. In the final survey, we asked developers to select and rank up to five of the options from the list. Figure 1 shows their responses and gives insight into what developers care about most when using program analyzers. Many of our findings, such as the fact that false positives and poor warning messages are large factors, are similar to those of Johnson et al. [39]; their work investigates why software engineers do not use static analysis tools to find bugs through a series of 20 interviews (see Section 6).

The largest pain point is that the default rules or checks that are enabled in a program analyzer do not match what the developer wants. Developers mentioned that some default program analysis rules, such as enforcing a specific convention (for instance, Hungarian Notation) to name variables or detecting spelling mistakes in the code or comments, are not useful, and on the contrary, they are actually quite annoying. Mitigations to this problem may include identifying a small key set of rules that should be enabled (rather than having all rules enabled, which is often the case), or making the process of selecting the rules and checks that are enabled easy for developers. Just as helpful is knowing the pain points at the bottom of the list. Developers care much more about too many false positives than about too many false negatives ("Misses too many issues"). One developer wrote of their team's program analyzer *"so many people ignore it because it can have a lot of false positives"*. Also, the ability to write custom rules does not appear important to many, unlike in the investigation by Johnson et al. [39].

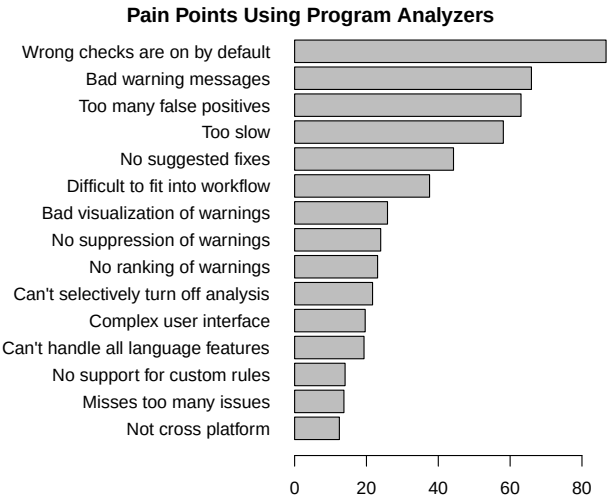

Figure 1: Pain points reported by developers when using program analyzers.

We also asked developers if they had used program analysis but stopped at some point. Only 9% of respondents indicated that they fell into this category. When asked why they stopped, there were three main reasons. 24% indicated that the reason was because the team policy regarding program analysis changed so that it was no longer required. Similarly, 18% indicated that they moved from a company or team that used program analysis to one that did not. Another 21% reported that they could not find a program analyzer that fit their needs; about half said this was due to the programming language they were using. This highlights one aspect of adoption of program analyzers that we also observed in discussions with developers: often, their use of analyzers (or lack thereof) is related to decisions and policies of the team they are on.

> Program analysis should not have all rules on by default.
>
> High false positive rates lead to disuse.
>
> Team policy is often the driving factor behind use of program analyzers.

2.2.2 What functionality should analyzers have?

One of the primary reasons why a program analyzer may or may not be used by a developer is whether the analyzer supports the programming language (or languages) that the developer uses. We therefore asked developers what languages they use in their work. Because the list was quite long, we aggregated responses into programming language cate-

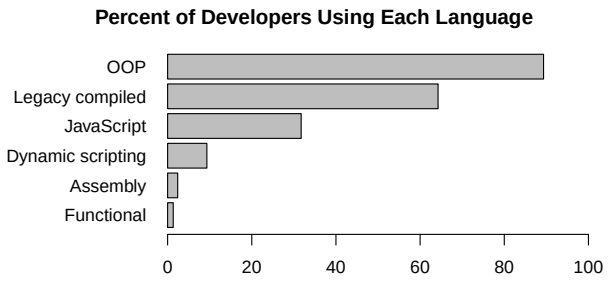

Figure 2: Languages used by developers.

gories, as shown in Figure 2. The primary languages in the object-oriented programming (OOP) category include C# and Java. Legacy compiled languages comprise C, C++, and Objective-C. Dynamic scripting languages include Python, Ruby, and Perl. We break out JavaScript (and variants such as TypeScript) because it is unique and one of the fastest growing languages. Our categorization is based on our perceptions and experiences as well as observations of Microsoft development. As such, it is one of many reasonable categorizations and is somewhat subjective (e.g., C++ is technically an OOP language, but it has been around much longer than Java or C# and is used for more low level functionality).

Next, we examine the types of code issues that developers consider to be important and that they would like program analyzers to detect. In our initial beta survey, we allowed developers to write in any type of issue that they deemed important. We then grouped the responses from the beta, leading to the types shown in Figure 3, and asked about the importance of each type of issue in the final survey. (Here and throughout the paper, we only present analysis and data from the final survey.) Note that these types of issues can be overlapping and their definitions flexible. For instance, many security issues could also be reliability issues, but the way to interpret Figure 3 for this particular example is that developers care much more about security issues than general reliability issues. In other words, this question was answered based on the cognitive knowledge developers have about each of these types. The results indicate that security issues are the most important, followed by violations of best practices. Interestingly, in an era where a non-trivial amount of software runs on mobile devices, developers do not consider it important to have program analysis detect power consumption issues.

Related to the types of issues that developers consider to be important are the potential sources of unsoundness in program analysis that can affect the detection of such issues. We listed the most common sources of unsoundness from program analysis research [28] and asked developers to rank up to five of them. During our initial interviews and the beta survey, we found that some developers were unfamiliar with the terminology used (though most were aware of the concepts). We therefore provided a brief explanation of each source of unsoundness in the survey. Figure 4 shows the results. As can be seen, exceptional control flow and aliasing top the list, while purity and dealing with floating point numbers are not considered critical.

Exceptions add a large number of control-flow transitions that complicate program analysis. To avoid losing efficiency and precision due to these transitions, many program analyzers choose to ignore exceptional control flow. Consequently,

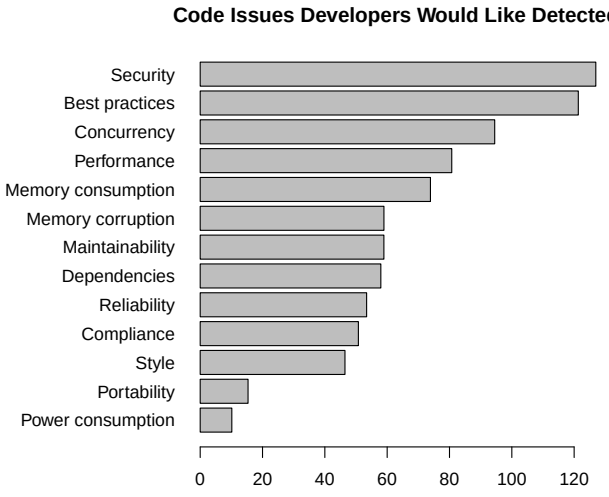

Figure 3: Ranking of types of code issues developers would like program analyzers to detect.

users who would like analyzers to soundly check exceptional control flow should be willing to sacrifice speed and false positive rates of the analysis. Ignoring certain side-effects due to aliasing avoids the performance overhead of precise heap analysis, so developers who do not want aliasing to be overlooked should be willing to wait longer for the analysis.

In practice, developers may not want program analysis to always examine all of the code in an application. When asked if developers would like the ability to direct the program analyzer toward certain parts of the code, 10% indicated that they have that functionality and are using it. Another 49% indicated that they do not use an analyzer that can do that, but it would be important to them that a program analyzer could be directed in such a way. Interestingly, of developers that are using a program analyzer with the ability to be directed to particular parts of the code, both experts and security developers use this functionality more than other developers to a statistically significant degree. When asked to what level of granularity developers would like to be able to direct a program analyzer, the overwhelming majority said the method level (46%) or the file level (35%).

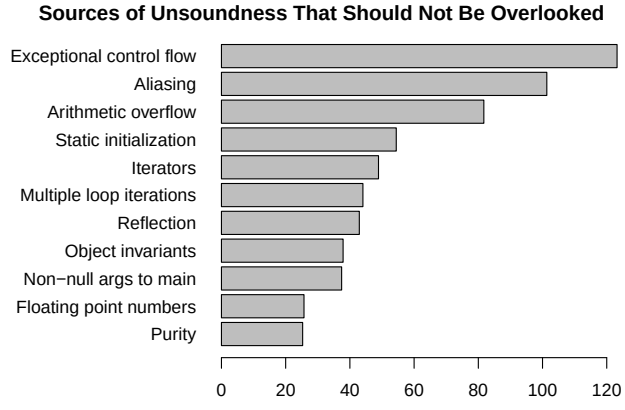

Figure 4: Ranking of the sources of unsoundness in program analysis that developers indicated should not be overlooked (i.e., considering them during analysis would be most helpful to developers).

A related functionality is the ability to analyze a changelist (also known as a commit) rather than the entire codebase. This type of functionality can help developers assess the quality and impact of a change before it is checked into the source code repository. 16% of developers indicated that they have and use this functionality in the program analyzer they use. Another 56% said that they do not have this functionality, but it would be an important factor in adopting a program analyzer. In sum, this 72% of developers use or would like this ability. When looking at experts, this value jumps to 77% (no change for security developers).

Other functionality is less attractive. Many analyzers provide the ability for developers to write their own program analysis rules. However, sometimes the learning curve can be steep or the background required may be deep in order to write custom analysis rules. When asked about the ability to write custom rules, 8% said that they have the functionality and use it and 26% said it is an important factor, while the rest said they either do not use it or do not care about having it. 66% of experts and 61% of security developers also indicated that they do not use or care about this functionality.

We postulated that the reason why developers are not interested in the ability to write custom program analysis rules is because they want to be able to select an analyzer and start using it without much effort. In fact, this is not the case. We asked developers whether they would be willing to add assertions, pre-, postconditions, and/or invariants to their code if this would improve the analysis results. Fully 79% of developers said they would add at least one of these types of specifications to their code, and 35% indicated that they would be willing to write all of them. This provides evidence that developers may be willing to provide additional information to program analyzers in return for better results (e.g., better precision). When asked about the form that such code specifications should take, an overwhelming majority (86%) of developers said that they would be more willing to annotate their code with specifications if these were part of the language, for example taking the form of nun-nullable reference types or an `assert` keyword.

One feature of program analyzers that developers use heavily is the ability to suppress warnings. 46% of developers indicated that they use some mechanism to suppress warnings. The primary methods are through a global configuration file, source code annotations (i.e., not in comments), annotations in source code comments, an external suppression file, and by comparing the code to a previous (baseline) version of it [45]. When asked which of these methods they like and dislike, 76% of those that use source code annotations like them, followed by using a global configuration file (63%) and providing annotations in code comments (56%).

Program analyzers should prioritize security and best practices and deal with exceptional control flow and aliasing.

Developers want the ability to guide program analyzers to particular parts of the code and analyze changelists.

While most are not interested in writing custom rules, developers are willing to add specifications in their code to help program analyzers.

Suppressing warnings is important, preferably through code annotations.

2.2.3 *What should the non-functional characteristics of program analyzers be?*

In the previous section, we focused on the functionality that developers indicate they want in program analyzers. When examining characteristics, we investigate non-functional aspects of program analyzers, such as how long they should take to perform the analysis, how often their warnings should be correct, and how they should fit into the development process. In many cases, there is a trade-off between characteristics (e.g., an analysis that has fewer false positives may include more complex techniques, such as alias analysis, which would require longer to complete). In these trade-off situations, we asked developers to indicate what characteristic they would sacrifice in order to improve another.

The time taken by a program analyzer is an important characteristic to developers because it can affect how often and where the analyzer can be run, which directly influences the utility of the analyzer to the developer. When asked how long a developer would be willing to wait for results from a program analyzer, 21% of developers said that it should run on the order of seconds, and 53% said they would be willing to wait multiple minutes. Thus, long running analyzers that exceed a few minutes would not be considered by nearly three quarters of developers.

The time required for an analysis dictates where it fits into the development process. When asked where in their development process they would like to use program analyzers, 25% of developers said every time they compile, 24% said once their change was complete but before sending out a code review request, 10% said during the nightly builds, 8% said every time unit tests were run, and 23% said they would like to run it at every stage of development.

Related to how a program analyzer should fit into the development process is how the results of the analyzer should be shown to the developer. The top four answers from developers are shown in Figure 5. The preferred location by a wide margin is in the code editor followed by the build output. This is in line with the findings of the interviews by Johnson et al. [39]: all 20 participants in their interviews wanted to be notified of issues in their code either in the IDE or at build/compile time. Moreover, one of the main lessons learned from the FindBugs experiences at Google [21] was that developers pay attention to warnings only if they appear seamlessly within the workflow.

Warnings that are false positives are cumbersome and time consuming. We asked developers the largest false positive rate that they would tolerate. We show the results as a reverse cumulative distribution curve in Figure 6, with the acceptable false positive rate on the x-axis and the percent of developers that find that rate acceptable on the y-axis. From the graph, 90% of developers are willing to accept up

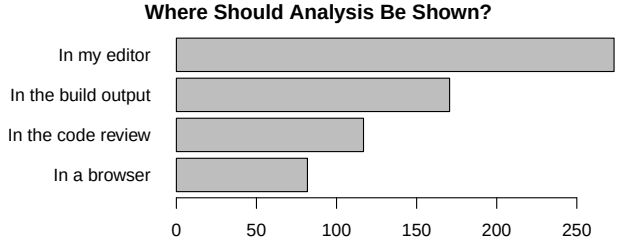

Figure 5: Where developers would like to have the output of program analyzers.

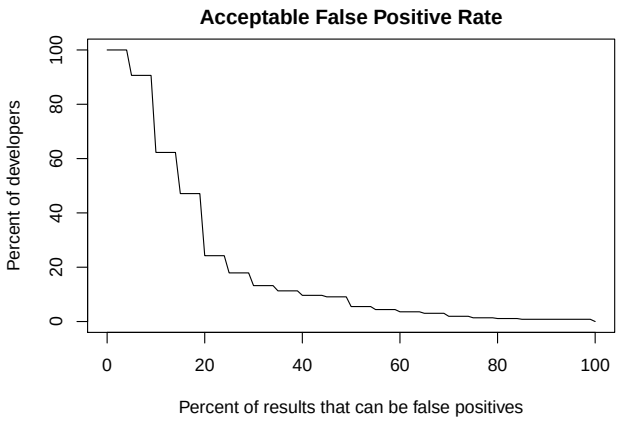

Figure 6: The largest false positive rate of analyzers that developers would tolerate.

to a 5% false positive rate, 47% of developers are willing to accept a false positive rate up to 15%, and only 24% percent of developers can handle a false positive rate as high as 20%. The designers of Coverity [23] confirm this aim for below 20% of false positives for stable checkers. When forced to choose between more bugs or fewer false positives, they typically choose the latter. We also asked developers when they are most willing to deal with false positives and gave them two extremes: (1) when it is easy to determine if a warning is a false positive, and (2) when analyzing critical code. In the latter case, finding issues is so important that developers are willing to deal with false positives in exchange for making sure no issues are missed. 55% of developers preferred to get false positives when they are easy to sift through.

Interestingly, developers are willing to give up more time if the quality of the results is higher. 57% said that they would prefer an analyzer that was slow, but found more intricate code issues to an analyzer that was fast, but was only able to identify superficial issues. Similarly, 57% also said they would prefer a slower analysis that yielded fewer false positives to a faster approach that was not as accurate. 60% reported that they would accept a slower analyzer if it captured more issues (fewer false negatives). While these all show that the majority of developers are willing to give up some speed for improved results, note that the majority is slight. Still 40–43% are willing to deal with more false positives, more false negatives, or superficial issues if meant the analysis was faster. These numbers do not change significantly when looking at just experts or security developers.

Much of the feedback from developers discussed the idea of having two kinds of analyzers, one fast and running in the editor, and another slow and running overnight. One developer put it quite succinctly *"Give me what you can give, fast and accurate (no false positives). Give me the slow stuff later in an hour (it is too good and cheap to not have it). No reasonable change is going to be checked in less than half a day but I do want that style check for that one line fix right away."* Another developer made a comparison of this to unit versus integration testing.

There is even less agreement when we compare the trade-off of reporting false positives versus missing real issues (false negatives). 49.3% developers would prefer fewer false positives even if it meant some real issues were missed, and 50.7% felt that finding more real issues was worth the cost of dealing with false positives.

Some program analyzers can provide lists of possible fixes for the warnings that they identify. We asked developers if they would prefer to sift through lists of potential fixes to identify the correct fix or if they would rather spend that time coming up with a fix on their own. 54% indicated they would be willing to sift through up to 10 potential fixes, while 45% felt that that time would be better spent designing a fix themselves.

> Program analysis should take a two-stage approach, with one analysis stage running in real time providing fast, easy feedback in the editor and another overnight finding more intricate issues.
>
> Program analysis designers should aim for a false positive rate no higher than 15–20%.
>
> Developers are willing to trade analysis time for higher-quality results (fewer false positives, fewer false negatives, more intricate issues).

2.2.4 Additional developer feedback

In our survey, we asked developers if they would like to share any additional opinions, thoughts, or feedback that they felt were not covered by our questions. 73 developers (19%) answered this question and we inspected and organized their responses. A number of key themes emerged from this analysis and we share those that are useful to program analysis researchers here.

Developers indicated that determinism of the program analysis is important. FxCop [11] (described in Section 5) is not deterministic because it uses heuristics about which parts of the code to analyze, and the Code Contracts analyzer [32] is not because it uses timeouts. If a program analyzer outputs different results each time it is run, it can be difficult to tell if an issue has been fixed. The Coverity designers also stress that randomization is forbidden, timeouts are also bad and sometimes used as a last resort but never encouraged [23].

When a developer makes a change to fix a program analysis warning, he or she would like an easy and quick way to check whether the warning is indeed fixed. A developer does not want to re-build and re-analyze everything for each warning. *"Supporting quickly re-checking whether a specific analysis error is fixed would significantly help the test-fix-test cycle."*

Many developers indicated that regardless of the analyzer they run, they would like a way to see and track their warnings, e.g., in SonarQube [18]. SonarQube is a web-based application that leverages its database to show and combine metrics as well as to mix them with historical measures.

Having a standard way of doing program analysis and a standard format of warnings across the organization is important as it can lessen the learning curve and decrease heterogeneity of tools and processes between teams.

It would be beneficial if analysis could help engineers understand how to properly use a programming language. Some engineers learn just enough about a language to do the work, and having an analyzer that teaches them which idioms, libraries, or best practices to use would be helpful.

2.2.5 Implications

In this section, we highlight the main implications of our survey findings for the program analysis community.

Expertise. When asked how frequently developers run program analyzers (daily, weekly, monthly, yearly, never), 37% said they run them daily and another 16% run them on a weekly basis, while 30% indicated they do not use program analyzers at all. There is a strong relationship between both familiarity with program analysis and a focus on security with frequency of use. 44% of security developers and 43% of experts use program analyzers daily (17% weekly for both groups). χ^2 tests showed that the differences in frequency of use of program analyzers between experts and non-experts and between security developers and non-security developers are both statistically significant ($p \ll 0.01$). This relationship may imply that if a developer has a deeper understanding of program analysis, then he or she will be more likely to use it. However, it may also be the case that in the process of using program analysis frequently, a developer develops an understanding of program analysis.

We also asked our survey participants which of the types of code issues that they encounter they estimate could have been caught by a program analyzer. By and large, for every type of code issue the majority believe that the issue could not have been caught. However, for reliability errors and maintainability issues, experts have more faith in program analysis to a statistically significant degree. 45% of experts think maintainability issues would be caught and 33% think reliability errors would be caught, whereas non-experts have even lower percentages. Therefore, developers who have a better understanding of program analysis also have more trust in its bug-finding capabilities.

More expertise in program analysis could also help in setting expectations with users. As an example, consider that analyzers typically ignore exceptional control flow to improve efficiency and precision (i.e., reduce the number of false positives). However, the responses to our survey did not indicate that developers who would like exceptional control flow to be checked by program analyzers are also willing to tolerate a large number of false positives.

Speed vs. quality. As we previously discussed, there are two camps of developers: those who are willing to wait longer if the quality of the analysis results is higher, and those who are willing to deal with more false positives, more false negatives, or superficial issues if it makes the analysis faster. This indicates that neither kind of program analysis is out of business, there is clear demand for both.

However, there is definitely a correlation between the kind of the analysis and where it fits in the development process. Developers who want to run an analyzer at every state of development are more likely to prefer a fast and superficial analysis. Those who want the results after every compile slightly lean toward slow and deeper analyses. Finally, developers who want the analysis results after a nightly build or right before a code review definitely prefer a slower analysis that detects more intricate code issues. These findings are all statistically significant ($p \ll 0.01$).

Annotations. Our findings indicate that developers may provide additional information to program analyzers, in the form of specifications or annotations, in return for better results (e.g., fewer false positives, suppressed warnings). Still, 21% of the respondents are not willing to write any assertions, preconditions, postconditions, and invariants or do not know what these are. Developers who like to suppress warnings with source code annotations are also more likely to provide specifications to a statistically significant degree (χ^2 test, $p < 0.05$). All this suggests that program analysis should be tunable through annotations but without requiring them.

Trust. To build trust in their analyzers, tool designers should keep in mind that developers care much more about too many false positives than too many false negatives. Moreover, the largest pain point in using program analyzers is found to be that by-default enabled rules or checks do not match what developers want.

3. LIVE SITE INCIDENTS

Our survey allowed us to understand what developers want and are most interested in with regard to program analyzers. We also sought to uncover the distribution of issues that occur in practice. For this we examined *live site incidents* from a set of Microsoft hosted services. We chose services because software development is increasingly focused on services.

A *live site incident* refers to an issue that affects the functionality of a service and requires the on-call engineer to intervene for its resolution. In other words, a live site incident is a high-severity bug that demands immediate attention by a software engineer so that the health of the affected company service is not compromised further.

Here, we categorize live site incidents into the types of code issues that we considered in the survey (see Section 2). We categorized a total of 256 live site incidents, which occurred and were fixed during the first two months of 2016. Note that these incidents affected 17 different Microsoft services. We achieved this categorization by personally interviewing software engineers who were assigned the resolution of these live site incidents, exchanging emails with them when an interview was not possible, attending live site reviews (that is, meetings providing an overview of any live site incidents of the previous week), and by carefully reading the root cause analysis of each live site incident on the company's tracking website for these incidents.

For incidents where our only source of information was the root cause analysis provided by the designated engineers on the tracking website, the categorization comes from one of the authors of this paper rather than the software engineer that handled the incident. This can introduce a threat to validity since the categorization may be subjective. To alleviate this threat, a random sample of 20% of these incidents were coded by another researcher independently and inter-rater reliability [36] was calculated using Fleiss κ on the results [33]. In our case, there was perfect agreement, with $\kappa = 1.0$, indicating that the threat of subjectivity in categorization is quite low.

Figure 7 shows the categorization of all 256 live site incidents using the methodology described above. Note that most live site incidents (65%) are categorized as reliability errors, followed by performance (14%) and dependency (12%) issues. As Figure 7 shows, no live site incidents are categorized as memory and power consumption issues or as style inconsistencies. This makes sense since such code issues are unlikely to cause high-severity bugs.

As part of the survey that we described in the previous section, we asked participants which types of code issues they would like program analyzers to detect. In accordance with Figure 7, the respondents are not very interested in detecting memory and power consumption issues or style inconsistencies, which are ranked fifth, thirteenth, and eleventh, respectively in the survey. Surprisingly, reliability errors are ranked ninth, performance issues fourth, and dependency

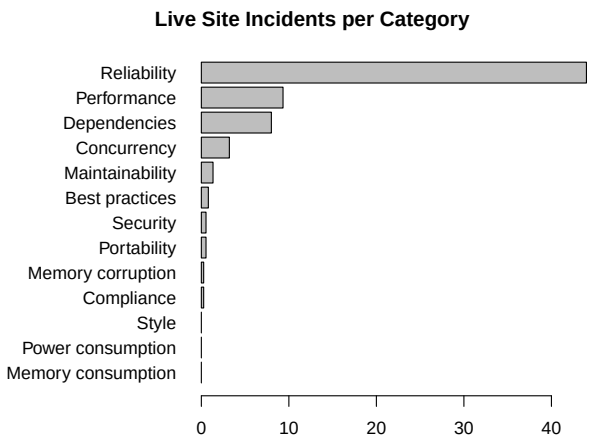

Figure 7: Categorization of live site incidents for 17 different company products.

issues eighth. In other words, high-severity code issues that require the immediate intervention of a software engineer (see Figure 7) are ranked low in the preferences of software engineers.

This mismatch between developers stated desires and the data regarding actual defects is surprising. An explanation for this result is that how much bugs matter depends on your point of view. That is, they matter very much to researchers and designers of program analyzers, but how much they matter to the users of analyzers can be unexpected. During the live site reviews that we attended, we witnessed quotes like *"oh, it'll crash, and we'll get a call"* or *"if developers don't feel pain, they often don't care"*, which were also recorded by Bessey et al. in 2010 [23]. This attitude toward bugs could also explain why the respondents of our survey would not like program analyzers to go after the most painful code issues (presented in Figure 7).

In the survey, we asked participants which types of code issues that they encounter they estimate could have been caught by a program analyzer. The most popular answers were best practices violations (69%) and style inconsistencies (62%), which are both superficial code issues. On the other hand, intricate code issues, like reliability, concurrency, and security errors, were selected by significantly fewer survey respondents (30%, 37%, and 47%, respectively). Figure 7 about live site incidents indicates a strong need for program analyzers to detect reliability errors. Moreover, security and concurrency errors are the first and third most popular answers, respectively, to the previous survey question (about which types of code issues software engineers would like program analyzers to detect).

This suggests that developers do not trust program analyzers to find intricate code issues. We see two possible explanations for this lack of trust, which are also supported by related work. First, due to the various pain points, obstacles, or annoyances that users encounter when using program analyzers, they might not derive the full potential of these tools. In the end, some users might abandon program analysis altogether, incorrectly thinking that it lacks value [21]. Second, it could be the case that users have little understanding of how program analysis works in a particular tool as well as little interest in learning more. As a consequence, they might end up classifying any detected code issue that is even slightly confusing to them as false [23].

> The vast majority of costly bugs in software services are related to reliability.
>
> Developers rank reliability errors low in the types of issues they want program analyzers to detect.
>
> Developers do not seem to trust analyzers to find intricate issues although they want them to detect such issues.

4. THREATS TO VALIDITY

As with any empirical study, there may be limits to our methods and findings [35]. Because one of our primary instruments was a survey, we were concerned that the right questions were included and presented in the right way [34]. To address construct validity [43], we began by examining the landscape of the use of program analysis in Microsoft and interviewing developers. These interviews led to the creation of a beta survey that we deployed and which provided another round of feedback to fine tune the questions in the final survey. Thus, we have high confidence that the questions are asked in a clear and understandable manner and cover the important aspects of program analyzers. We are also confident that the options provided for answers to the questions capture the majority of developer responses.

With regard to external validity [50], our analysis comes wholly from one software organization. This makes it unlikely that our results are completely representative of the views of software developers in general. However, because Microsoft employs tens of thousands of software engineers, works on diverse products in many domains, and uses many tools and processes, we believe that our approach of randomly sampling improves generalizability significantly. In an effort to increase external validity, we have provided a reference to our survey instrument so that others can deploy it in different organizations and contexts.

Not all categorizations of live site incidents came directly from the software engineer that handled the incident. In cases where we could not contact them, we had to categorize them ourselves based on information in the incident tracker, which may introduce subjectivity. As mentioned previously, we mitigated this by having multiple researchers categorize the incidents independently and checking the inter-rater reliability, which resulted in perfect agreement.

5. PROGRAM ANALYZERS IN INDUSTRY

In this section, we give an overview of program analyzers that are currently being used in three large software companies, namely Microsoft, Google, and Facebook. We derived a list of first party program analyzers at Microsoft by interviewing leads and members of the managed and unmanaged static analysis teams in the company. We then asked the participants of our survey (see Section 2) which analyzers they have run the most. We focus on the six most popular first and third party program analyzers for Microsoft. For Google and Facebook, we rely on recent research publications to determine which analyzers are currently being used.

First party program analyzers at Microsoft. The order in which we present the tools is arbitrary and all first

party tools were selected by at least 8% of the respondents.

BinSkim [1] is a binary scanner that validates compiler and linker settings. Specifically, it validates that code has been built using the compiler and linker protections that are required by the Microsoft Security Development Lifecycle (SDL) [17], e.g., that a binary has opted into the Address Space Layout Randomization (ASLR) or the hardware Data Execution Prevention (DEP) features.

FxCop [11] analyzes managed assemblies using a set of pre-defined or custom rules and reports possible design, localization, performance, and security improvements. Many of the detected code issues concern best practices violations.

PoliCheck is an internal tool by Microsoft's Geopolitical Product Strategy team that scans text (code, code comments, and content, including Web pages) for anything that might be politically or geopolitically incorrect. For example, a person's name should not be written in red letters as in some context or culture it may signify that the person is dead.

PREfast [14] performs intraprocedural analysis to identify defects in C and C++ source code. There are multiple PREfast plug-ins for detecting different kinds of code issues, like BadValues for security errors, CppCoreCheck for reliability, security, and compliance errors, DriversDLL for errors in kernel-mode drivers, etc.

PREfix [24] is one of the few analyzers in industry that performs cross-binary dataflow analysis. PREfix detects security, reliability, performance, and memory consumption issues, but without the user providing extensive annotations, it is almost impossible to find any memory or resource leaks.

StyleCop [19] was selected by 50.5% of the respondents and is, by far, the most popular first party tool. It analyzes C# code to enforce a set of style and consistency rules, which is configurable and may be augmented with custom rules.

Third party program analyzers at Microsoft. Here, we present the six most popular third party analyzers that are being or have been used at Microsoft.

ReSharper [16] was selected by 51.2% of the respondents and is the tool that has been run the most out of all first and third party analyzers. ReSharper is a productivity tool with a code analysis component, which finds compiler errors, runtime errors, redundancies, code smells, and possible improvements right while the user is typing. Note that 81.8% of our respondents have heard of ReSharper, out of those, 47.5% are currently using ReSharper, and out of those, 50% rank its code analysis as one of their top three favorite features (29% said it was the most important feature).

Just like ReSharper, *CodeRush* [5] is a productivity tool, which supports easy code investigation, automation of common code creation tasks, easy search and navigation to a required code location, etc. Among its features, CodeRush also provides a code analysis tool that suggests slight code improvements and detects dead or duplicate code, useless code (e.g., containing an unimplemented class member), invalid code (e.g., containing a call to an undeclared method) as well as unreadable code.

Fortify [10] is a tool for identifying security vulnerabilities in source code, with support for 20 programming languages. Its analysis is based on a set of security coding rules, and its detected vulnerabilities are categorized and prioritized based on how much risk they involve and whether they provide an accurate action plan. The Microsoft investigation into Fortify, performed on two very large codebases, revealed that its rules are thorough at the expense of being very noisy.

Checkmarx [2] analyzes source code, also written in a very wide breadth of programming languages, by virtually compiling it and performing queries against it for a set of pre-defined and custom security rules. During an evaluation of the tool at Microsoft, Checkmarx was found to be as accurate as Fortify but easier to configure. Fortify, however, was found to achieve deeper coverage.

Coverity [6, 23] is considered one of the best commercial static analyzers for detecting security and reliability errors in C, C++, C#, and Java. In general, the code issues it reports have a very high fix rate and a very low false positive rate. Coverity performs whole-program analysis and is known to have detected serious bugs involving multiple functions or methods. It is primarily offered as a cloud-based service.

Cppcheck [7] is a rule-based analyzer for C and C++. It mainly detects reliability errors, like null pointer dereferences, use of uninitialized variables or unsafe functions, etc. The goal of Cppcheck is to report no false positives, therefore, it is rarely wrong, but as a consequence, it misses many bugs.

Program analyzers at Google. So far, Google has made several attempts to integrate program analyzers into the development process of software engineers. The most prominent example of such an analyzer is *FindBugs* [9, 21], which cheaply detects defects in Java code, including bad practices, performance, and correctness problems. FindBugs aims at identifying the low-hanging fruit of code issues, instead of finding all possible errors in a particular category. Other analysis tools that have at times been used by Google include *Coverity* [6], *Klocwork* [13], and fault prediction [42]. However, all of these analyzers have gradually been abandoned due to their false positive rates, scalability issues, and workflow integration problems [49].

Recently, Google built Tricorder [20, 49], a program analysis ecosystem for detecting a variety of code issues in a wide breadth of programming languages. Tricorder smoothly integrates into the development workflow, scales, and allows even non-analysis experts to write and deploy custom analyzers. Moreover, any integrated analyzer that is annoying to developers, degrades the performance of Tricorder, or whose reported code issues are never fixed by developers is banned from the ecosystem.

The overview of Tricorder [49] describes some of the tools that have been integrated in the Google analysis ecosystem. These include analysis frameworks, like *ErrorProne* [8] and *ClangTidy* [4], which find bug patterns based on AST matching in Java and C++, respectively, and the Linter analyzer, which detects style issues and contains more than 35 individual linters, such as configured versions of the *Checkstyle* Java linter [3] and the *Pylint* Python linter [15]. Lastly, there are various domain-specific analyzers, like *AndroidLint* for detecting issues in Android applications, and tools that analyze metadata associated with a changelist, like how much code is transitively affected by a particular changelist.

Program analyzers at Facebook. *Infer* [26, 25, 12] is Facebook's most well-known analyzer in the current literature: it is based on academic research in program analysis, and there are many publications on its internals and its large-scale deployment. Facebook uses Infer to find resource leaks and null-pointer exceptions in Android and iOS applications. The tool may report both false positives and false negatives. Infer's analysis is incremental, which means that, when analyzing a changelist, it uses a cache of verification results so that only functions affected by the changelist are analyzed.

Discussion. In this section, we observe that, although there are a few exceptions to this rule, advanced program analysis techniques are generally underdeveloped in industry. Most of the program analyzers that we have presented are productivity tools, linters, or rule-based scanners. We are definitely not claiming that simplistic program analyzers lack value—we are however wondering why many innovative and bright research ideas do not seem to have substantial practical impact. This trend has been observed before [39, 26, 49] in an effort to provide a few reasons for this gap between scientific literature and industry.

Here, we would like to support these suggestions from the literature with data-driven results from our survey. For instance, Calcagno et al. [26, 25, 29] suggest that part of the problem is that research has focused too much on whole-program or specify-first analyses. Indeed, the importance of compositional and incremental analyses is stressed by the fact that 56% of the survey respondents do not currently have the functionality of analyzing only a changelist, instead of an entire codebase, but this functionality would be important to them. Furthermore, 46% find the granularity of functions or methods more suitable for directing an analyzer toward the more critical parts of their code. Concerning program annotations, 21% of the respondents are not willing to write any specifications or do not know what specifications are.

Calcagno et al. also define the "social challenge", which has been described in other related work too [39, 49]. Engineers accumulate trust in an analyzer and start reacting to the bugs it reports when certain features are there: full automation and integration into the development workflow, scalability, precision, and fast reporting. In fact, the top six pain points, obstacles, or annoyances that our survey respondents encountered when using program analyzers are (from most to least annoying): (1) irrelevant checks are turned on by default, (2) bad phrasing of warnings, (3) too many false positives, (4) too slow, (5) no suggested fixes, and (6) difficult to integrate in the workflow (see Figure 1). Moreover, the top six minimum requirements that an analyzer must satisfy for our respondents to start using it are (from most to least minimal): (1) detect issues that are important to me, (2) easy to integrate in the workflow, (3) fast, (4) few false positives, (5) with suppression of warnings, (6) good phrasing of warnings. Lastly, in terms of fast reporting, 21% of the respondents are only willing to wait seconds for a program analyzer to check a changelist, and 53% minutes.

6. OTHER RELATED WORK

Empirical studies. There are relatively few empirical studies that analyze the usage and adoption of program analysis tools in industry, especially from the point of view of software engineers. So far, many studies have analyzed the functionality of program analyzers, mostly from the point of view of tool designers [27, 21, 23].

The work most closely related to ours investigates why software engineers do not use static analysis tools to find bugs [39]. The results are collected from interviews with 20 engineers, and focus on the interviewees' perception of tools, including interacting with their interfaces, and on what could have caused these perceptions. Although their interviewees felt that using static analysis is beneficial, there are certain barriers in their use, like false positives, poorly presented warnings, lack of or weak support from the team, inability to suppress warnings, poor environment integration, long running times of the tools, etc. In Section 2, we discuss that our survey respondents have also identified the same pain points. In terms of support from the team, many of our respondents that have stopped running program analyzers said it was because of a change in the team policy.

Ayewah and Pugh [22] also conducted a survey and a controlled study on how software engineers use the FindBugs tool [9, 21]. Although related, our work is not concerned about a particular program analyzer; we are rather focusing on what the general characteristics of any program analysis should be such that it is industrially relevant.

Lastly, our work is analogous to a recent empirical analysis of programming language adoption [46]: instead of programming languages, it focuses on the adoption of program analyzers in a large software organization.

Unsoundness in program analysis. Around the year 2000, unsoundness in program analysis was controversial in the academic community [23]. By now, researchers have realized that soundness is commonly eschewed in practice [44]. In fact, there have even emerged approaches for measuring the unsoundness in a static analysis and evaluating its practical impact [28]. In industry, as it also becomes evident in Section 5, it is a well-established design decision to build program analyzers to be unsound in order to increase automation, improve performance, achieve modularity, and reduce the number of false positives or the annotation overhead. The full range of program analysis techniques in industry, from heuristics to more advanced methodologies, like in Coverity, PREfix, and Infer, becomes more precise and efficient in detecting software bugs at the cost of ignoring or making unsound assumptions about certain program properties.

In our survey, we presented engineers with common sources of unsoundness in program analyzers, and asked them which of these should not be overlooked by tool designers (see Section 2). We hope that our findings will help designers in finding good trade-offs when building practical analyses.

Other companies. Ebay has suggested techniques for objectively evaluating the cost-effectiveness of different program analyzers and comparing them against each other [38]. IBM has experimented with an online portal that addresses common barriers to the wide usage of static analysis tools and promotes their adoption [47].

Other evaluations. Other evaluations of usage of static analyzers include understanding how to improve their user interfaces [40], and how to use suitable benchmarks for systematically evaluating and comparing these tools [37].

7. CONCLUSION

We have presented a multi-method empirical investigation that we deployed at Microsoft. Our findings shed light on what practitioners want from program analyzers, how program analysis researchers can achieve broad and lasting adoption of their tools, which types of defects are most important to minimize through program analysis, and which tools are currently being used at three large software companies.

We believe that our data-driven answers to these questions are the first step toward narrowing the gap between scientific literature and industry with regard to program analysis.

Acknowledgments. We thank Marc Brockschmidt, Jacek Czerwonka, Laura MacLeod, Wolfram Schulte, Valentin Wüstholz, and Tom Zimmermann for their valuable help. We are also grateful to the developers who participated in our study and the reviewers for their constructive feedback.

8. REFERENCES

[1] BinSkim. https://github.com/Microsoft/binskim.

[2] Checkmarx. https://www.checkmarx.com/.

[3] Checkstyle. http://checkstyle.sourceforge.net/.

[4] ClangTidy. http://clang.llvm.org/extra/clang-tidy/.

[5] CodeRush. https://www.devexpress.com/products/coderush/.

[6] Coverity. http://www.coverity.com/.

[7] Cppcheck. http://cppcheck.sourceforge.net/.

[8] ErrorProne. https://github.com/google/error-prone.

[9] FindBugs. http://findbugs.sourceforge.net/.

[10] Fortify. http://www8.hp.com/us/en/software-solutions/static-code-analysis-sast/.

[11] FxCop. https://msdn.microsoft.com/en-us/library/bb429476(v=vs.80).aspx.

[12] Infer. http://fbinfer.com/.

[13] Klocwork. http://www.klocwork.com/.

[14] PREfast. https://msdn.microsoft.com/en-us/library/ms933794.aspx.

[15] Pylint. https://www.pylint.org/.

[16] ReSharper. https://www.jetbrains.com/resharper/.

[17] Security development lifecycle. https://www.microsoft.com/en-us/sdl/.

[18] SonarQube. http://www.sonarqube.org/.

[19] StyleCop. https://stylecop.codeplex.com/.

[20] Tricorder. https://github.com/google/shipshape.

[21] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh. Using static analysis to find bugs. *IEEE Software*, 25:22–29, 2008.

[22] N. Ayewah and W. Pugh. A report on a survey and study of static analysis users. In *DEFECTS*, pages 1–5. ACM, 2008.

[23] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C.-H. Gros, A. Kamsky, S. McPeak, and D. R. Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. *CACM*, 53:66–75, 2010.

[24] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *SPE*, 30:775–802, 2000.

[25] C. Calcagno and D. Distefano. Infer: An automatic program verifier for memory safety of C programs. In *NFM*, volume 6617 of *LNCS*, pages 459–465. Springer, 2011.

[26] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. W. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez. Moving fast with software verification. In *NFM*, volume 9058 of *LNCS*, pages 3–11. Springer, 2015.

[27] B. Chess and J. West. *Secure Programming with Static Analysis*. Addison-Wesley Professional, 2007.

[28] M. Christakis, P. Müller, and V. Wüstholz. An experimental evaluation of deliberate unsoundness in a static program analyzer. In *VMCAI*, volume 8931 of *LNCS*, pages 336–354. Springer, 2015.

[29] J. Constine. Monoidics. http://techcrunch.com/2013/07/18/facebook-monoidics/.

[30] S. Dowdy, S. Wearden, and D. Chilko. *Statistics for research*, volume 512. John Wiley & Sons, 2011.

[31] H. Esfahani, J. Fietz, Q. Ke, A. Kolomiets, E. Lan, E. Mavrinac, W. Schulte, N. Sanches, and S. Kandula. CloudBuild: Microsoft's distributed and caching build service. In *ICSE Companion*. ACM, 2016. To appear.

[32] M. Fähndrich and F. Logozzo. Static contract checking with abstract interpretation. In *FoVeOOS*, volume 6528 of *LNCS*, pages 10–30. Springer, 2010.

[33] J. L. Fleiss. Measuring nominal scale agreement among many raters. *Psychological bulletin*, 76(5):378, 1971.

[34] F. J. Fowler Jr. *Survey research methods*. Sage publications, 2013.

[35] N. Golafshani. Understanding reliability and validity in qualitative research. *The qualitative report*, 8(4):597–606, 2003.

[36] K. L. Gwet. *Handbook of inter-rater reliability: The definitive guide to measuring the extent of agreement among raters*. Advanced Analytics, LLC, 2014.

[37] S. Heckman and L. Williams. On establishing a benchmark for evaluating static analysis alert prioritization and classification techniques. In *ESEM*, pages 41–50. ACM, 2008.

[38] C. Jaspan, I.-C. Chen, and A. Sharma. Understanding the value of program analysis tools. In *OOPSLA*, pages 963–970. ACM, 2007.

[39] B. Johnson, Y. Song, E. R. Murphy-Hill, and R. W. Bowdidge. Why don't software developers use static analysis tools to find bugs? In *ICSE*, pages 672–681. ACM, 2013.

[40] Y. P. Khoo, J. S. Foster, M. Hicks, and V. Sazawal. Path projection for user-centered static analysis tools. In *PASTE*, pages 57–63. ACM, 2008.

[41] B. A. Kitchenham and S. L. Pfleeger. Personal opinion surveys. In *Guide to Advanced Empirical Software Engineering*, pages 63–92. Springer, 2008.

[42] C. Lewis, Z. Lin, C. Sadowski, X. Zhu, R. Ou, and E. J. Whitehead Jr. Does bug prediction support human developers? Findings from a Google case study. In *ICSE*, pages 372–381. ACM, 2013.

[43] M. S. Litwin. *How to measure survey reliability and validity*, volume 7. Sage Publications, 1995.

[44] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B.-Y. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis. In defense of soundiness: A manifesto. *CACM*, 58:44–46, 2015.

[45] F. Logozzo, S. K. Lahiri, M. Fähndrich, and S. Blackshear. Verification modulo versions: Towards usable verification. In *PLDI*, pages 294–304. ACM, 2014.

[46] L. A. Meyerovich and A. S. Rabkin. Empirical analysis of programming language adoption. In *OOPSLA*, pages 1–18. ACM, 2013.

[47] M. G. Nanda, M. Gupta, S. Sinha, S. Chandra, D. Schmidt, and P. Balachandran. Making defect-finding tools work for you. In *ICSE*, pages 99–108. ACM, 2010.

[48] T. Punter, M. Ciolkowski, B. Freimut, and I. John. Conducting on-line surveys in software engineering. In *Empirical Software Engineering, 2003. ISESE 2003. Proceedings. 2003 International Symposium on*, pages 80–88. IEEE, 2003.

[49] C. Sadowski, J. van Gogh, C. Jaspan, E. Söderberg, and C. Winter. Tricorder: Building a program analysis ecosystem. In *ICSE*, pages 598–608. IEEE Computer

Society, 2015.

[50] J. Siegmund, N. Siegmund, and S. Apel. Views on internal and external validity in empirical software engineering. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, volume 1, pages 9–19. IEEE, 2015.

[51] E. Smith, R. Loftin, E. Murphy-Hill, C. Bird, and T. Zimmermann. Improving Developer Participation Rates in Surveys. In *Proceedings of the International Workshop on Cooperative and Human Aspects of Software Engineering*. IEEE, 2013.

[52] P. K. Tyagi. The effects of appeals, anonymity, and feedback on mail survey response patterns from salespeople. *Journal of the Academy of Marketing Science*, 17(3):235–241, 1989.