Long-term Static Analysis Rule Quality Monitoring Using True Negatives

Linghui Luo, Rajdeep Mukherjee, Omer Tripp,
Martin Schäf, Qiang Zhou
Amazon Web Services

Daniel Sanchez
Amazon Alexa

***Abstract*—Static application security testing (SAST) tools have found broad adoption in modern software development workflows. These tools employ a variety of static analysis rules to generate recommendations on how to improve the code of an application.**

Every recommendation consumes the time of the engineer that is investigating it, so it is important to measure how useful these rules are in the long term. But what is a good metric for monitoring rule quality over time? Counting the number of recommendations rewards noisy rules and ignores developers' reactions. Measuring fix rate is not ideal either, because it overemphasizes rules that are easy to fix.

In this paper, we report on an experiment where we use the frequency of true negatives to quantify if developers are able to learn a static analysis rule. We consider a static analysis rule to be ideal if its recommendations are not only *addressed*, but also *internalized* by the developer in a way that prevents the bug from recurring. That is, the rule contributes to code quality not only at present, but also in the future. We measure how often developers produce true negatives, that is, code changes that are relevant to a rule but do not trigger a recommendation, and we compare true-negative rate against other metrics. Our results show that measuring true negatives provides insights that cannot be provided by metrics such as fix rate or developer feedback.

***Index Terms*—software security, human factors, static analysis**

I. Introduction

Static application security testing (SAST) tools are now part of most software development workflows. Several studies indicate that running SAST tools in the code-review stage is most efficient (e.g., [1], [2]). In this scenario, the SAST tool comments on the changed lines of code, just like a human reviewer, and developers can discuss the comments and decide which recommendations to address before the changes are merged.

At Amazon, we use CodeGuru Reviewer[1] during our code review phase. CodeGuru Reviewer provides recommendations on how to improve quality and security of the code under review. Since each recommendation consumes developer time, we want to make sure that these rules actually provide long-term value to our developers.

While in academia metrics such as precision, recall and F-measure are commonly used to assess the quality of static analysis, it is infeasible on industrial code due to the lack of ground truth. In industry, it is more common to assess the value of a SAST rule by counting how often a recommendation leads to a code change within the same code review. This is often referred to as *fix rate*, or *action rate*.

Fix rate is an effective metric to identify rules that perform poorly. If recommendations from a given rule are often ignored, then either the rule produces a high rate of false alarms or the recommendations are not actionable.

However, a disadvantage of using fix rate to assess the value of a rule is that it disproportionately favors rules whose detections are easy and/or important to fix. For example, rules that check for hardcoded credentials traditionally have a high fix rate, but checks that warn about weak cryptographic algorithms tend to have a much lower fix rate. Why is that? Intuitively, removing hardcoded credentials from code is a relatively easy code change, and there is basically no scenario where having hardcoded credentials in code is a good development practice. On the other hand, changing a cryptographic algorithm is not always an easy change. There is usually another piece of code that this code communicates with that expects the same algorithm, so changing the algorithm only in one place may lead to a system failure, and sometimes, this other code can be outside of the developer's control, so it may take weeks or months until such a change can be prioritized.

However, even if a developer does not address a recommendation about a cryptographic algorithm within the same code review, it does not mean that the rule provides no value. If the developer learns what a recommended algorithm is and applies this knowledge in the future, then the rule still provides value. That is, developers may still receive recommendations on their current project because the cryptographic algorithm can't be changed, but they may follow the recommendation when starting their next project, and thus, the rule provided value. Such behavior is not detectable with existing metrics; on the current project, we would observe a low fix rate, and on the next project we would not observe anything, because there are no more recommendations.

In this paper, we examine if we can measure such learning behavior. That is, instead of measuring an immediate reaction where the developer addresses a recommendation with a code change, we want to measure if this reactive behavior turns into proactive behavior where, after a few recommendations, the developer has internalized the rule and writes compliant code without receiving further recommendations.

In an ideal code review system, a developer would see a

[1]https://aws.amazon.com/codeguru

recommendation once, internalize the concept, and then never receive a recommendation from the same rule again. That is, for a given rule, we would see a single recommendation per developer followed by a sequence of true negatives.

In the opposite direction, we assume that a rule does not promote learning if the same developer keeps getting recommendations from this rule at a more or less constant rate. That is, the developer doesn't change their behavior based on the recommendations, and the rule keeps consuming the developer's time.

If a recommendation promotes learning, for a fixed population of developers the frequency of code reviews with recommendations for that rule will decrease over time as developers learn and start writing code that is compliant with the rule. However, it could also be true that we see an increase in fix rate over time if developers learn how to be compliant with the rule, but rely on the tool to remind them. We break this idea down into the following hypotheses that we want to test in this paper:

- H1: For a fixed group of developers and rules, the frequency of code reviews with recommendations will decrease over time, but eventually stabilize.
- H2: For a fixed group of developers and rules, the fix rate is a good metric for monitoring rule quality over time.
- H3: For a fixed group of developers, the true-negative rate is a good metric for monitoring rule quality over time.

Our hypothesis H3 uses the term *True Negative* for code that contains API calls or language constructs that are relevant to a specific rule but does not yield a recommendation. E.g., for our cryptography rule from above, we would report a true negative, on code that calls the `Cipher.getInstance` API, but uses a secure algorithm for which the rule does not produce a recommendation. That is, we distinguish between the case where a recommendation is absent because the analyzed code is irrelevant to the current rule (e.g., because it is written in a different language, or it does not use any API or language construct that is covered by the rule), and the case where the code implements the behavior that is expected by the rule.

To validate or refute the above hypotheses, we conduct an experiment using data collected by CodeGuru Reviewer inside Amazon. We discuss how CodeGuru Reviewer is integrated into the development process and what data it collects in Section III, and explain our experimental setup in Section IV.

Scope of Goals. The goal of our experiment is to understand if true-negative rate is a good metric for monitoring SAST rule quality over time, and how this metric compares to other quality metrics, such as fix rate or developer acceptance of recommendations.

Comparing developers to each other is explicitly out of scope for this experiment. We only collect data where we see an interaction between the developer and the SAST tool to understand changes in their code over time. We do not collect (or have access to) any other data on the developers or their behavior.

II. Related Work

Measuring the usefulness of SAST tools is an important problem. These tools, when deployed company-wide, use a lot of developer time and attention, so it is important that the recommendations they provide are valuable.

Facebook reports on their experience of deploying static analysis at scale and the lessons learned along the way [1], [3]. A key takeaway of these studies is that developers prefer findings at code review time. Their data show that the fix-rate of their system increased from around 20% when they reported findings in an offline fashion (e.g., via an issue tracker) to over 70% when they showed the same findings during code-review time. They also introduce useful terminology, such as *actioned report* rather than *true positive* for recommendations that triggered a code change.

The tool in their study, Infer, predominantly reports issues related to runtime errors and concurrency issues, which can be fixed without knowledge of how the analyzed code will be deployed. That is, in contrast to more context-dependent SAST recommendations, like cross-site scripting, where developers may address an issue reported in backend code by changing code in the frontend, they can assume that findings that go unaddressed are effectively false alarms.

Google conducts similar studies for their ErrorProne tool [4] and their Tricoder static analysis infrastructure [2]. Similar to the study at Facebook, they measure what recommendations trigger a developer response and count all other recommendations as *effective false positives*. An interesting observation in their study, which we confirm in our experiment, is that developers give limited feedback on recommendations that are generated by a tool. For Tricoder, they report that on an average of 93,000 daily recommendations, they receive on average 716 positive reactions and 48 negative reactions [2].

Their study does not explicitly mention what happens to recommendations that are not addressed at code review time, or if developers fix these issues later.

Coverity shares an experience report which suggests that a perceived false positive rate of over 30% leads to developers ignoring the recommendations of a tool [5]. Hence, it is important to develop a better understanding of what a perceived false positive is.

We claim that the fact that a recommendation is not addressed within the same code review does not automatically imply that it is a perceived false positive. During code review, developers try to accomplish a specific task. If addressing the recommendation of a SAST tool does not fit in the scope of this task, there is reason to assume that developers will address the recommendation in a separate code review.

For example, Smith et al. conduct a study where they interview developers while they fix security-related issues reported by a tool called FindSecBugs [6]. In their study, developers see the findings for an entire application at once, rather than only findings for the changed code in a code-review, and they collect the questions that developers ask while fixing the issues. Many of the questions are related to how the code in the recommendation is being used, and if it is reachable from the outside. That is, developers require a lot of context to fix an issue. This motivates our assumption that developers may not be willing to fix a SAST recommendation within a code review because it is outside of their current context, and add it as a backlog item instead.

Another related area of research is measuring the quality of recommendations, and if developers understand the actions that need to be taken. Barik et al. demonstrate how the structure of text in a compiler error message affects the developers ability to fix the issue [7]. They suggest that using Toulmin's model of argument when designing error messages leads to better, and more actionable error messages.

The quality of messages in SAST recommendations is also mentioned as the second most important issue after wrong tool configuration in a study at Microsoft [8]. Weimer shows that providing a patch as part of the recommendation increases the likelihood of a developer reaction significantly [9]. This is also in line with findings reported by Google for their ErrorProne [4].

The quality of messages is not in scope for this paper. We conduct a qualitative analysis of the recommendations that are not addressed in a code review where we can identify if the quality of the message was a significant factor, but we do not investigate if altering the message would lead to a different outcome.

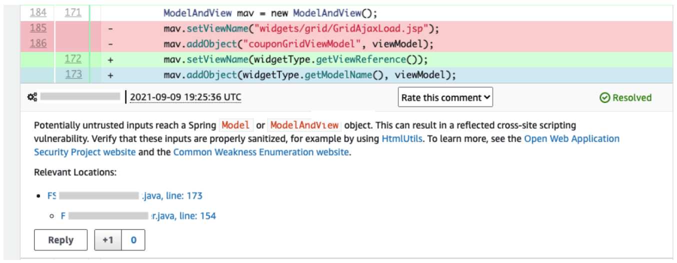
Measuring if a SAST recommendation got fixed is a complex problem in itself. Several different approaches (e.g., [10]–[12]) to track if a code change fixes a recommendation, or just moves or removes it. This problem is out of scope for this paper.

III. Background on CodeGuru Reviewer

To set the context for our experiment, we briefly explain the code review process at Amazon and how our CodeGuru Reviewer is integrated.

Teams at Amazon manage their own development process. Most teams follow an agile or lean development process, such as Scrum or Kanban, where software development is organized into sprints, and developers break down feature development into smaller, well-defined tasks. When developers pick up a task, they clone the Git repositories with the relevant code, make the necessary changes, and create a code review (CR for short) to vet the changes within their team before merging them. Code changes at Amazon have to go through a code review before being merged. The code reviews are created and updated by developers via a command-line tool. Code reviews are based on Git pull requests with some additional tooling to simplify managing the lifecycle of the pull request. Reviewers interact with the code review through a dedicated web interface, where they can view the changes and leave comments as shown in Figure 1.

During the code review, several automated systems can add comments. These systems include tools that run unit tests, search for non-inclusive language, as well as static and dynamic code scanners. CodeGuru Reviewer is part of this workflow, and leaves recommendations on the code review in the same way a



```
184  171       ModelAndView mav = new ModelAndView();
185       -    mav.setViewName("widgets/grid/GridAjaxLoad.jsp");
186       -    mav.addObject("couponGridViewModel", viewModel);
     172  +    mav.setViewName(widgetType.getViewReference());
     173  +    mav.addObject(widgetType.getModelName(), viewModel);
```

[redacted] | 2021-09-09 19:25:36 UTC

Potentially untrusted inputs reach a Spring `Model` or `ModelAndView` object. This can result in a reflected cross-site scripting vulnerability. Verify that these inputs are properly sanitized, for example by using HtmlUtils. To learn more, see the Open Web Application Security Project website and the Common Weakness Enumeration website.

Relevant Locations:

- FS [redacted] .java, line: 173
  - F [redacted] r.java, line: 154

Fig. 1. Example of a comment by our SAST tool on a code review. File names have been redacted.

human reviewer would. CodeGuru Reviewer only comments on changed lines of code. In this workflow, developers merge their changes through the code review web interface after receiving the necessary approvals. Changes are never pushed directly from a developer's machine.

Figure 1 shows a redacted example of such a recommendation by CodeGuru Reviewer. In this example, CodeGuru Reviewer comments that a potential cross-site scripting issue was detected in a Java Spring MVC application. The recommendation states that data from a Spring request parameter is included in the return value (the Spring ModelAndView object) without being sanitized, which could make this code vulnerable to a reflected cross-site scripting attack.

Now it is up to the developer to decide if this recommendation should be addressed as part of this code review. Several factors can affect the decision. For example, is this issue mitigated elsewhere? It could be mitigated in the frontend code. Or, is it mitigated by a Spring Security configuration, or is it clear from the context that this code will never receive inputs from untrusted sources?

Developers and reviewers can provide feedback by replying to the recommendation directly and discuss if it should be addressed in another revision. If developers decide to address recommendations, they modify their code locally, and create another revision of the code review. CodeGuru Reviewer runs every revision of a code review and posts fresh recommendations on each revision.

For each code review and revision thereof, we have access to the changed lines of code, a unique identifier for the author of the review, the state of the review (i.e., if it is still open, if it was merged, or canceled), and the recommendations posted by CodeGuru Reviewer.

We record any feedback on the generated recommendations. Developers can leave a (free-form) text comment to each recommendation. This comment can also be used to discuss the relevance of the recommendation with the human reviewers (e.g., *Can we ignore this?*). For the free-text comments, we finetune a pre-trained BERT model [13] on a dataset of roughly 20000 labeled developer feedbacks to map the sentiment of the comment to either *positive*, or *negative*. Discussing the model is not in scope of the paper, but manual inspection shows that the mapping has an accuracy of over 80%.

For our study, we collect data for three types of developer behavior: 1) the developer reactively changes code in response to a recommendation (accounted through fix-rate), 2) the developer proactively writes the correct code because of what they learned from a previous recommendation (accounted as true negative), or 3) the developer does not react to the recommendation (accounted as code review with recommendations on its last revision).

Fix Rate. We compute fix rate at code review granularity. That is, we consider a rule fixed for a given code review if CodeGuru Reviewer produced recommendations for this rule on one or more revisions but does not produce recommendations for the same rule on the final revision when the changes are merged.

If a rule produces multiple recommendations by the same rule on a code review yet only some are addressed by the time the changes are merged, we still consider the rule not-fixed for this code review. We acknowledge that the choice of granularity might ignore inconsistent behavior inside a code review (e.g., recommendations of a rule are partially fixed), but it was chosen due to technical limitations of our internal API.

True Negatives. CodeGuru Reviewer rules are formulated via a query language [14] over control-flow graph, or as type-state properties [15]. Rules include information about the API calls and language constructs that are relevant. For example, a rule that checks for the use of file streams in Java only triggers if it encounters a call to `Files.list` or `Files.walk`. If these API calls, or language constructs are encountered in the changed lines of a code review, but the rule does not produce a recommendation, we mark it as a true negative.

That is, in the context of this paper, a true negative is code that could trigger a specific rule but does not produce a recommendation. We do not claim that code is correct because a rule did not produce a recommendation; for us, a true negative just means that the developer applied a compliant coding pattern that is accepted by a given rule.

IV. Experiment Setup

For our experiment, we select five internal rules (rules that are only released internally at Amazon) at the time of conducting this experiment. For each rule, we identify the developers that received at least one recommendation. Each rule that goes into CodeGuru Reviewer production is released and evaluated first internally in our code review phase. We emphasize that, because we can not share production data, this experiment is restricted to development data inside Amazon and the rules in this experiment may not be released or modified prior to release to production.

For each rule, we identify all developers from our opt-in list that received a recommendation on at least one of their code reviews. Table I shows the list of rules and the number of developers that received recommendations for each rule, the total number of code reviews with recommendations they received, the total number of recommendations, and the feedback that was provided on these recommendations. Note that a code review can contain multiple recommendations from the same rule.

Rules. We now briefly explain the rules that we use for our experiment. As a threat to validity, we highlight that we could only use rules that were released internally at Amazon at the time of doing the experiment. We did not control the set of internal rules that were released.

Our first rule, R1, checks if developers create a Java Stream of file system objects using API calls, such as `Files.lines` or `Files.walk` without closing it. Not closing these streams can cause the system to run out of file descriptors which will ultimately crash the application. Leaving streams of file system objects open is never correct, but it only becomes a risk if a large number of file system objects are opened, so developers may choose to ignore the recommendation if they know that their code is only running short transactions, like a unit test, or a command line tool.

Rules R2 and R3 check for correct usage of batch operations of two different AWS services: Simple Queue Service (SQS) and DynamoDB. Developers batch multiple operations for sending or deleting messages in SQS, or for writing or deleting entries in DynamoDB, because it can be faster than triggering operations in isolation as it reduces the communication overhead. Not all operations in such a batch have to succeed. The API collects the operations that failed and it is up to the caller of the API to implement proper error handling.

Developers may ignore failures in batch operations if their code has other mechanisms to identify the problem or if it is not expected that all operations succeed. For example, code that batch-deletes entries in a DynamoDB table may ignore failures of the batch operations but raise an alarm if the table is not empty after the operation is completed. Ignoring failures in batch operations can lead to loss of relevant data. Similar to the resource leaks detected by R1, this could allow an attacker to hide their traces, or perform a denial-of-service attack.

Although R2 and R3 are similar in that they check batch operations, we expect variance in results because the objects they are operating on have varying levels of security importance: R2 focuses on batch operation of message queues, while R3 focuses on database operations, and that developers take failed database operations more seriously than failures when interacting with message queues, and thus are more willing to adopt a more defensive programming style.

Rule R4 checks if developers that use the Simple Storage Service, S3, set the expected owner of a storage location when writing data. In S3, the identifiers of resource locations (aka Buckets) are globally unique. Setting the expected owner provides additional protection from accidentally writing to the wrong location. A typical problem that this rule tries to catch is where a developer uses a staging and a production environment with different storage locations. In the staging environment, data might be stored unencrypted for debugging purposes. A code change may accidentally use the storage location of the staging environment in production, which could lead to customer data being stored unencrypted.

Depending on the use cases, developers might not have the expected owner when writing to S3, e.g., if the storage location is from an external source, like another service or the user. In

Rule Name	# unique developers	# code reviews with recommendations	# recommendations	Feedback positive	Feedback negative
R1 Files Resource Leak	246	267	1106	226	40
R2 SQS Batch Operations	252	290	899	115	25
R3 DDB Batch Operations	509	613	1852	198	65
R4 S3 Bucket Owner	1192	1426	5086	401	246
R5 Spring CSRF	317	375	1067	113	12

TABLE I
THE SET OF PRE-PRODUCTION RULES USED IN OUR EXPERIMENT, THE NUMBER OF DEVELOPERS THAT RECEIVED RECOMMENDATIONS FROM EACH RULE, THE NUMBER OF CODE REVIEWS WITH RECOMMENDATIONS FROM EACH RULE, THE NUMBER OF RECOMMENDATIONS, AND THE FEEDBACK FOR THESE RECOMMENDATIONS.

this cases, they will ignore this recommendation.

Rule R5 checks for cross-site request forgery (CSRF) in Spring by checking if a method with a `@RequestMapping` annotation specifies the HTTP Verbs (e.g., `GET` or `POST`) that this method should respond to. Not specifying the HTTP Verb might give an attacker the opportunity to change the state of an application by sending a `POST` or `PUT` request to a method that was only intended to handle read-only `GET` requests. Developers should always set the HTTP Verbs.

Data collection. For each rule, we want to understand if developers accept the recommendation and learn how to write code that is compliant with the rule. To that end, we identify, for each developer and rule, the code review where the developer got a recommendation for the first time.

From this code review on, we go forward in time and collect all future code reviews for the same developer that contain changes in the added/modified lines that may have triggered the rule. That is, in the case of R1, we collect all code reviews for the developer that contain `Files.lines`, `Files.list`, etc. We only include code reviews that are already merged. Reviews that are either in progress or have been canceled are not counted.

The maximum time frame that we could observe in this experiment is 12 months, and we inspect the next relevant code reviews of each developer from the first time they saw a recommendation of a given rule: either it contains recommendations of the rule or it contains compliant code, i.e., true negative. We classify these relevant code reviews into three kinds:

- **Not-fixed code review**: recommendations were not fixed when the code review was merged. We check if there is recommendation in the last revision of the code review.
- **Fixed code review**: recommendations were present on some revisions of the code review but not on the final revision that was merged.
- **True-negative code review**: the code review contains compliant code with a rule that produced recommendation in a previous code review.

In Figure 2, we show the number of not-fixed code reviews (orange), fixed code reviews (blue), and true-negative code reviews (green) over the sequence of code reviews that we identified for the developers and the five internal rules. Because we only start collecting code reviews for a developer after they see the first recommendation for that rule, the first column (orange + blue) of each bar chart in Figure 2 is equal to the number of unique developers in Table I. Thus, the first code review for each developer-rule pair is always either a fixed or not-fixed code review. We expect to see an increasing number of true-negative code reviews over time if developers learn the rule and proactively write compliant code.

The horizontal axis of the bar charts in Figure 2 is the number of code reviews for which we found code changes that are relevant to the specific rule. So, in Figure 2 (a), the first bar indicates that we found a total of 246 developers with a code review for which the rule R1 produced a recommendation (orange + blue). 142 of these developers fixed the issue within the same review (blue), while 104 did not fix (orange). The second bar indicates that, out of these 246 developers, 62 developers had a second code review with code changes relevant to rule R1. Out of these 62 developers, 48 wrote code that is compliant with the rule (green), i.e., true-negative code reviews. 4 developers got recommendations from R1 and fixed it again (blue), and 10 developers got recommendations but did not fix it (orange).

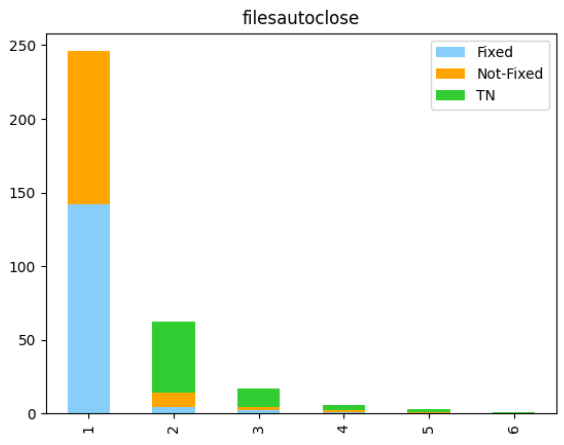
For a rule that can be easily learned, we would expect the bars after the first to be entirely green. That is, in the first bar, the developer gets a recommendation and either fixes it (blue) or not (orange), but from then on, this developer only writes true negatives (green). In the opposite direction, we assume that less green towards the right is a sign that the rule is problematic, in that developers may have difficulty proactively writing code that is compliant with this rule.

V. EVALUATION

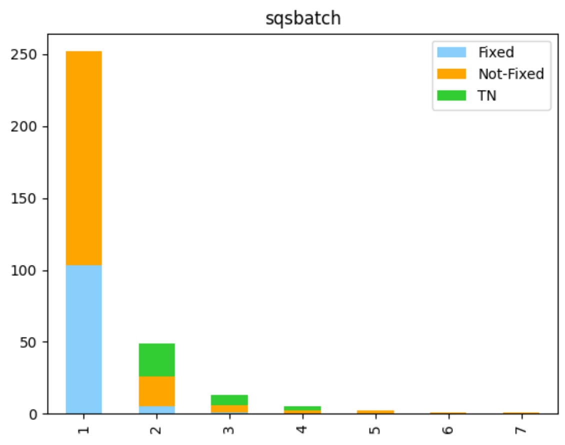
Let us use the collected data to discuss the hypotheses introduced at the beginning of this paper. To understand if recommendations are driving learning, we need a code review that contains the first exposure of a recommendation, and at least one code review after the recommendation to determine if the recommendation acted as a learning event. Thus, we focus on the subset of developers from Figure 2 that had at least two reviews with code that is relevant to a rule so that we can see if they change their behavior after the first review.

a) H1: For a fixed group of developers and rules, the frequency of code reviews with recommendations will decrease over time, but eventually stabilize.

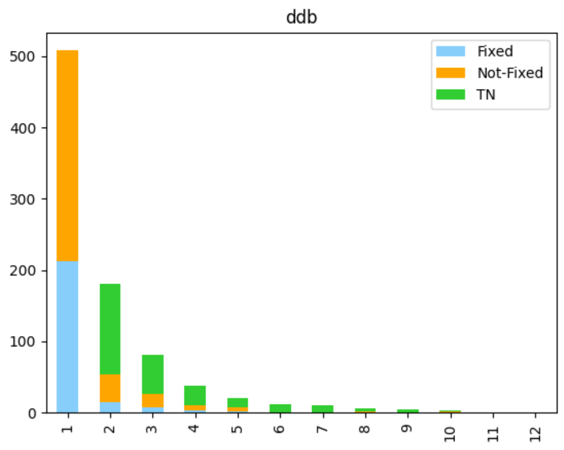
From Figure 2 we can see that, for all rules, the number of developers that write code relevant to the rule multiple times in a row drops almost exponentially, and so does the number of code reviews with recommendations (orange + blue). This



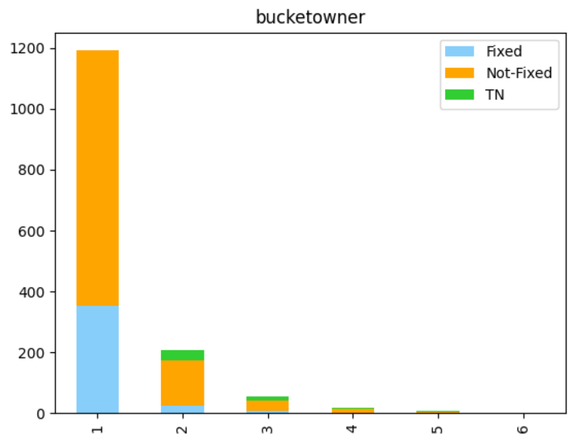
(a) R1 Files Resource Leak



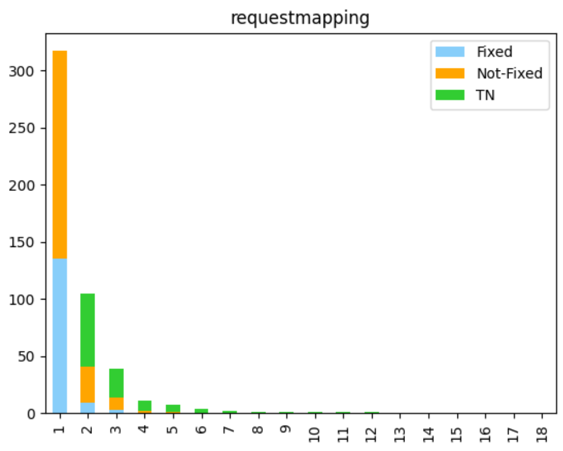
(b) R2 SQS Batch Operations



(c) R3 DDB Batch Operations

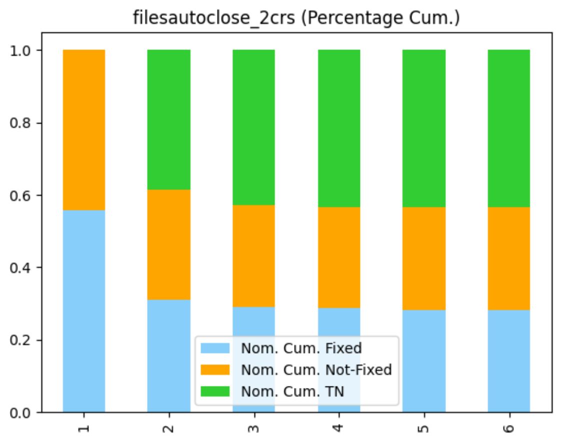


(d) R4 S3 Bucket Owner

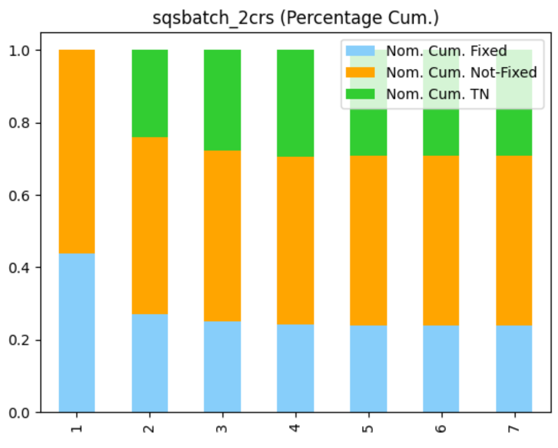


(e) R5 Spring CSRF

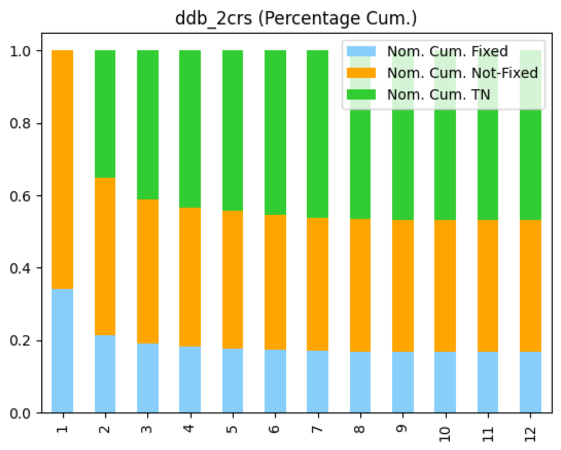
Fig. 2. Number of developers with fixed code reviews (blued), not-fixed code reviews (orange), true-negative code reviews (green) after k relevant code reviews to a rule for our fixed group of developers.



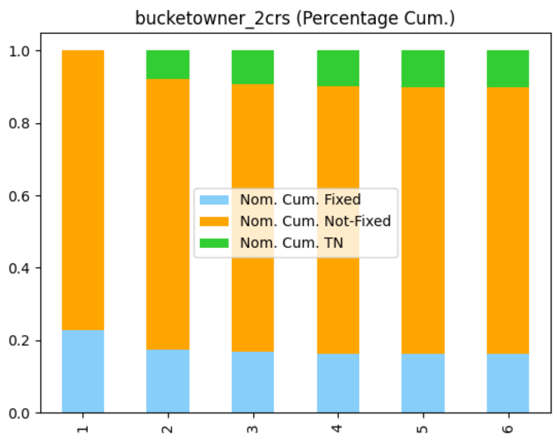
(a) R1 Files Resource Leak



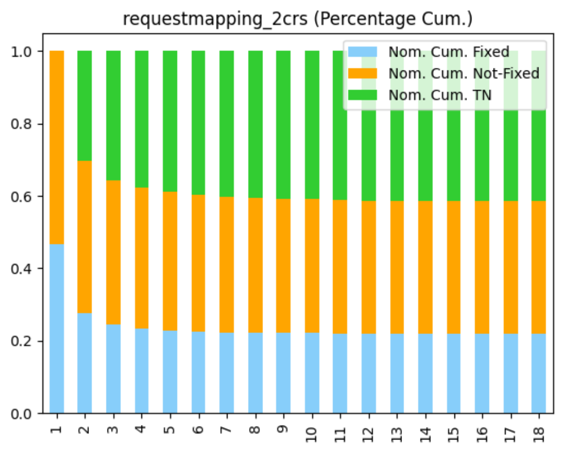
(b) R2 SQS Batch Operations



(c) R3 DDB Batch Operations



(d) R4 S3 Bucket Owner



(e) R5 Spring CSRF

Fig. 3. The percentage of the cumulative number of code reviews with fixed recommendations (blued), not-fixed recommendations (orange), true negatives (green) after k code reviews that include relevant changes to the rule for our fixed group of developers with at least 2 code reviews.

is not unexpected, since developers work on different tasks and rarely need to touch the same APIs multiple times during a project.

Figure 3 shows the percentage of the cumulative number of not fixed code reviews (orange), fixed code reviews (blue), and true-negative code reviews (green) over the time, for all developers that had at least two relevant code reviews.

From this figure, we can see that the ratio of developers that receive recommendations (orange + blue) decreases, the more often the developer writes code that is relevant to this rule. That is, in a fixed population of developers, we will see a steady decrease in the frequency of recommendations for a given rule.

While this is true for all rules, we can see that it is more pronounced for rules like R1 or R3 than for R4. That is, for R4, it is more common that a developer receives a recommendation multiple times.

We can also see that ratio of code reviews with recommendations (orange + blue) shrinks over time. This suggests that the frequency of new recommendations will eventually stabilize. We assume that this is because developers that agree with the recommendation will change their behavior early, and developers that do not agree or ignore the recommendation will not change behavior even if their continuously receive recommendations. Therefore, our data validates H1.

b) H2: For a fixed group of developers and rules, the fix rate is a good metric for monitoring rule quality over time.

Fix-rate is often used as a metric for assessing rule quality: the higher the fix-rate, the better the rule. However, our data show that this is only true early on, after initial exposure to the rule and recommendation. As Figure 3 shows, the fix-rate decreases over time. In fact, we can see that the fix-rate is high initially when developers see a recommendation for the first time but then stabilizes at a lower level. All five rules appear to quickly stabilize to a fix rate around 20% after a few exposures, suggesting that long-term fix rate does not provide useful differentiation about which rule is "better." This can be explained with the increase in true negatives. Some of the developers that fix a recommendation the first time they encounter it will learn this pattern and write code that is compliant with the rule in the future. That is, we cannot see an increase in fix rate over time, but if we combine fix rate and true-negative rate, we see a clear trend. As for hypothesis H1, we see that trends are more pronounced for rules R1 and R3 than for R4 or R2. That is, our data show that hypothesis H2 is not valid.

c) H3: For a fixed group of developers, the true-negative rate is a good metric for monitoring rule quality over time.

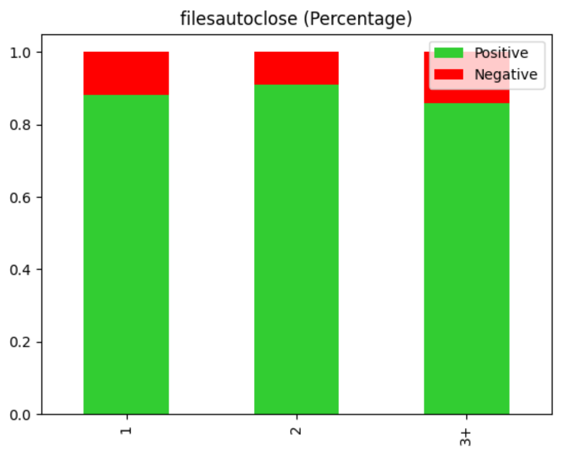
In our data, we can see that the true-negative rate increases quickly in the beginning, but eventually stabilizes. In line with our findings for H1, this suggests that developers either adopt a recommendation from a rule quickly or do not adopt it at all. From the data, we can also see that the true-negative rate significantly differs between rules. Figure 3 shows that about half of our developers produced true negatives for R1, R3, and R5 after their first code reviews, only about 10% did so for R4. If we use the true-negative rate as an indicator to monitor rule quality, then rules R1, R3, and R5 would be ranked highest, followed by R2, and rule R4 ranking lowest.

Is the ranking based on true-negative rate consistent with developer feedback? If we look at the aggregate developer feedback for each rule in Figure 4, we can see that there are consistencies with the true-negative ranking, as rule R4 also ranks lowest, and rules R1 and R5 rank highest. However, the feedback for R2 and R3 is almost identical, even though R2 ranked much lower in terms of true negatives. The feedback similarity for R2 and R3 could be explained by their rule similarity; both rules flag insufficient error handling for AWS batch operations. Manual inspection of the feedback shows similar reaction by developers; they comment that there is error handling in place elsewhere, or that the code is only for experimentation and the data is only short-lived anyway. So, from a feedback perspective, we can see that developers that do not fix recommendations for these rules do it for similar reasons.

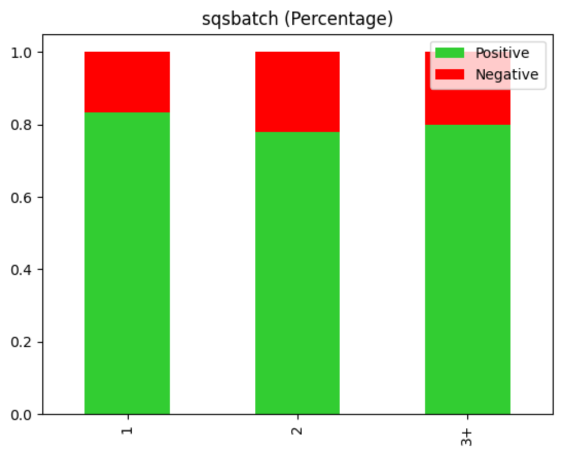
In contrast to developer feedback, rule R3 has a much higher true-negative rate compared to R2. We assume that this is because R2 focuses on batch operation of message queues, while R3 focuses on database operations, and that developers take failed database operations more serious than failures when interacting with message queues, and thus are more willing to adopt a more defensive programming style. If R2 and R3 lead to different true-negative rates because of the different security importance between the rules, we would expect to see that reflected in the developer feedback. However, the developer feedback in Figure 4 does not allow us to distinguish between R2 and R3, suggesting that maybe we're missing a signal in the feedback because developers do not provide feedback for true negatives. So we looked at developers who shipped at least three relevant code reviews for each rule and classify them into to the following types based on their behavior:

- **Proactive User**: after their first code review received recommendations, more than half (K1 = 0.5) of their relevant code reviews afterwards were true-negative code reviews. These developers learned the rule quickly and will code proactively when it comes to the similar problem.
- **Reactive User**: after their first code review received recommendations, more than half (K2 = 0.5) of their relevant code reviews were either fixed or true-negative code reviews. These developers mostly rely on the analysis tool to pinpoint bugs in their code.
- **Random User**: the rest of the developers.

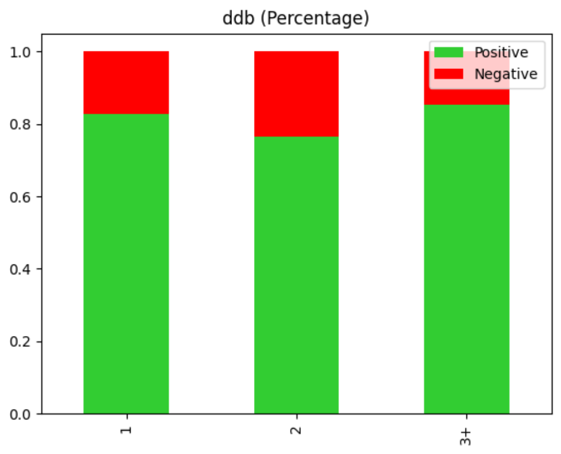
If a rule has a high percentage of Proactive Users, we can assume that the number of feedback and fix rate for Reactive and Random Users will be lower because we will only receive data for those developers that did not learn from the first recommendation.



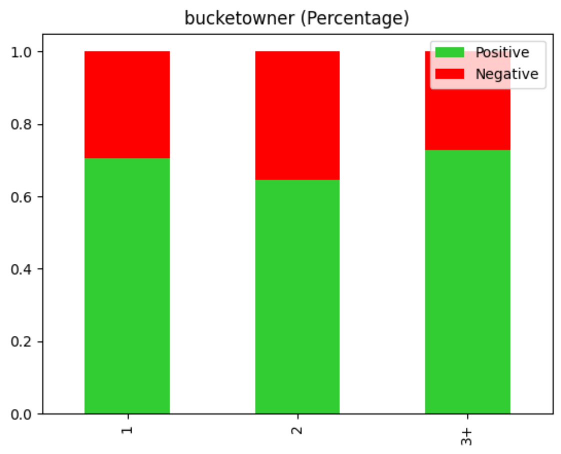
(a) R1 Files Resource Leak



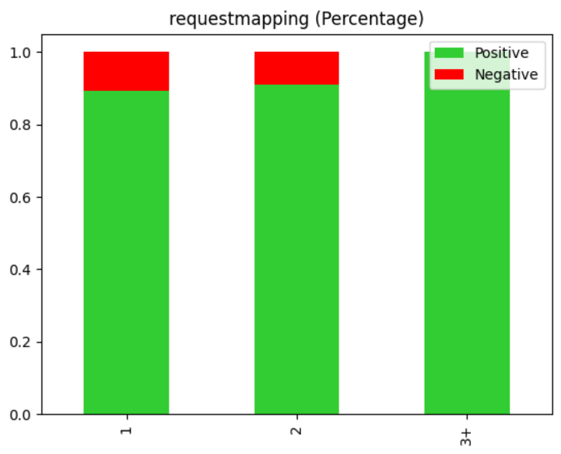
(b) R2 SQS Batch Operations



(c) R3 DDB Batch Operations

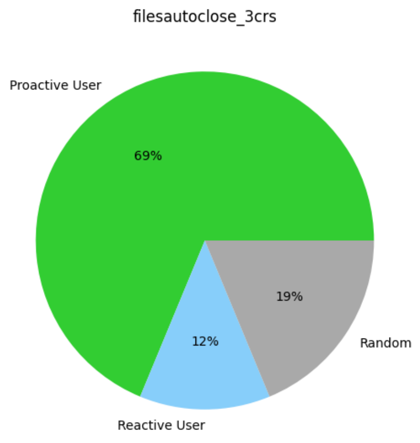


(d) R4 S3 Bucket Owner

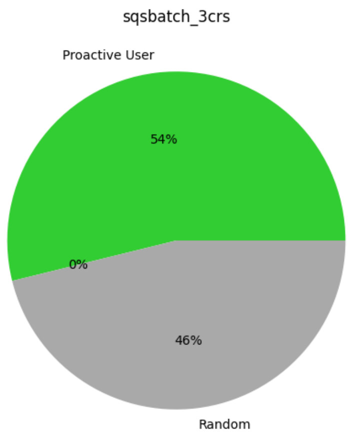


(e) R5 Spring CSRF

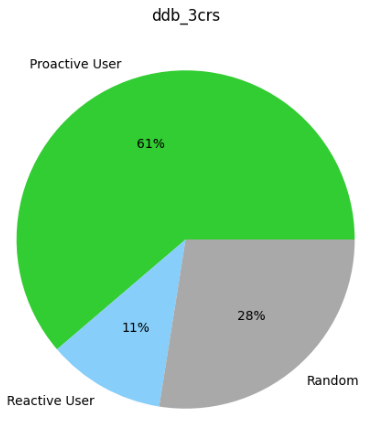
Fig. 4. Percentage of positive vs. negative sentiment regarding developer feedback. X-coordinate: number of code reviews, Y-coordinate: percentage of positive (green) and negative (red) feedback provided by developers with k code reviews received recommendations from each rule.



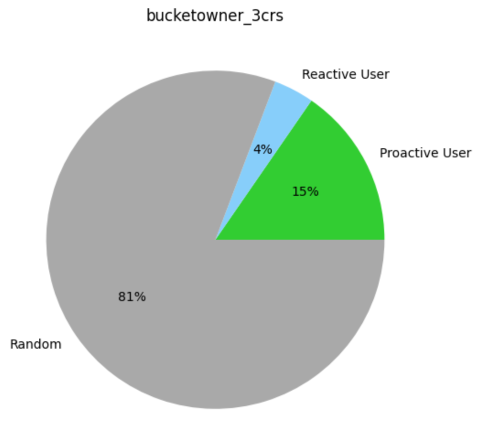
(a) R1 Files Resource Leak



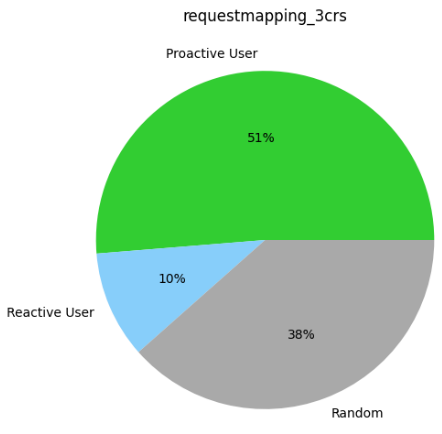
(b) R2 SQS Batch Operations



(c) R3 DDB Batch Operations



(d) R4 S3 Bucket Owner



(e) R5 Spring CSRF

Fig. 5. Different types of users based on at least 3 code reviews per developer. Proactive User (green), Reactive User (blue) and Random User (grey).

Figure 5 shows the percentages of these three types of users for each rule: Proactive Users in green, Reactive Users in blue and Random Users in grey. Here, we can see that that the percentage of developers in the Random group for R2 is significantly higher (46%) than for R3 (28%). Hence, we argue that the reason why we cannot see a difference in developer feedback between R2 and R3 is that the share of developers that agree with the rule stops receiving recommendations and thus does not provide further feedback.

Figure 5 also confirms the remaining ordering of our rules: R1 performs the best, with just 19% of developers in the Random group, followed by R3 (28%), R5 (38%), R2 (46%), and in last position R4 with 81%. Therefore, we believe the ratio of different users can be used as a good indicator for identifying rules that need improvement.

To summarize, our experiment validates hypothesis H1, invalidates H2, and partly validates H3. Taken together, we believe that tracking true negatives is a more reliable metric to monitor rule quality over time. In particular, if the group of developers is relatively constant, we can see that fix rate would favor newer rules, and feedback will give a distorted picture because it will exclude those developers that agree with the rule and proactively write compliant code.

d) Threats to Validity.

We highlight that this is just an experiment and not a user study. We did not control for the type of rules or the group of developers. Further, we used real data, so the number of developers using the same APIs in the same way multiple times is small. None of our results are intended to be statistically significant. We believe that, in order to get sufficient data for a study, one would need a lab setting with a group of developers performing identical task over a period of time, but this is not feasible in a corporate setting.

Lastly, our experiment focused on security rules related to API misuse. It is not clear if this can be generalized to other classes of software defects. E.g., tracking true negatives for Nullpointer exceptions or division-by-zero may be harder. Further, developers are generally aware that one cannot divide by zero and violations of such a rule are most likely slips so it is questionable if we can see a learning effect or increase in true negatives for those rules.

However, for a rule that covers the misuse of an API, we believe our experiment shows that true negatives are an interesting metric to track, and that true negatives provide a different perspective compared to other metrics, such as developer feedback or fix rate.

VI. Conclusion

We presented a study to test if true-negative rate can serve as a useful, long-term indicator of SAST rule quality. Our results show that a share of developers quickly adopts the recommendation of a rule and starts producing true negatives. This confirms our intuition that, a) developers seem to learn from recommendations, and they seem to do so quickly, after just one or two recommendations of the same rule; and b) if the group of developers and rules is fixed, ultimately, only developers that disagree with the rule will receive recommendations. Hence, over time, metrics like fix rate and developer feedback will trend down for rules, because all developers that agreed with the rule already proactively write compliant code.

Our experiment shows that measuring true negatives provides additional insights about rules that cannot be provided by common metrics. More importantly, our experiment suggests that both fix rate and developer feedback become less useful as indicators of quality over time if the population of developers is fixed. Further, true negatives can be used to cluster developers into groups based on their learning behavior, which will allow us to perform more fine-grained analysis of rule performance in the future. Measuring true negatives also opens a door for future research. As we noticed that for some rules, developers have difficulty in proactively writing compliant code even after they've fixed recommendations from the rules. Understanding what makes such rules hard for developers to learn from them could be an interesting research topic for the future.

References

[1] M. Harman and P. O'Hearn, "From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis," in *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2018, pp. 1–23.

[2] C. Sadowski, J. van Gogh, C. Jaspan, E. Söderberg, and C. Winter, "Tricorder: Building a program analysis ecosystem," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. IEEE Press, 2015, p. 598–608.

[3] D. Distefano, M. Fähndrich, F. Logozzo, and P. W. O'Hearn, "Scaling static analyses at facebook," *Commun. ACM*, vol. 62, no. 8, p. 62–70, Jul. 2019. [Online]. Available: https://doi.org/10.1145/3338112

[4] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspan, "Lessons from building static analysis tools at google," *Commun. ACM*, vol. 61, no. 4, p. 58–66, Mar. 2018. [Online]. Available: https://doi.org/10.1145/3188720

[5] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A few billion lines of code later: Using static analysis to find bugs in the real world," *Commun. ACM*, vol. 53, no. 2, p. 66–75, Feb. 2010. [Online]. Available: https://doi.org/10.1145/1646353.1646374

[6] J. Smith, B. Johnson, E. Murphy-Hill, B. Chu, and H. R. Lipford, "How developers diagnose potential security vulnerabilities with a static analysis tool," *IEEE Trans. Softw. Eng.*, vol. 45, no. 9, p. 877–897, Sep. 2019. [Online]. Available: https://doi.org/10.1109/TSE.2018.2810116

[7] T. Barik, D. Ford, E. Murphy-Hill, and C. Parnin, "How should compilers explain problems to developers?" in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 633–643. [Online]. Available: https://doi.org/10.1145/3236024.3236040

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

[9] W. Weimer, "Patches as better bug reports," in *Proceedings of the 5th International Conference on Generative Programming and Component Engineering*, ser. GPCE '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 181–190. [Online]. Available: https://doi.org/10.1145/1173706.1173734

[10] J. Spacco, D. Hovemeyer, and W. Pugh, "Tracking defect warnings across versions," in *Proceedings of the 2006 International Workshop on Mining Software Repositories*, ser. MSR '06. New York, NY, USA:

Association for Computing Machinery, 2006, p. 133–136. [Online]. Available: https://doi.org/10.1145/1137983.1138014

[11] S. Kim and M. D. Ernst, “Which warnings should i fix first?” in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC-FSE ’07. New York, NY, USA: Association for Computing Machinery, 2007, p. 45–54. [Online]. Available: https://doi.org/10.1145/1287624.1287633

[12] P. Avgustinov, A. I. Baars, A. S. Henriksen, G. Lavender, G. Menzel, O. de Moor, M. Schäfer, and J. Tibble, “Tracking static analysis violations over time to capture developer characteristics,” in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE ’15. IEEE Press, 2015, p. 437–447.

[13] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of deep bidirectional transformers for language understanding,” in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 4171–4186. [Online]. Available: https://aclanthology.org/N19-1423

[14] R. Mukherjee, O. Tripp, B. Liblit, and M. Wilson, “Static analysis for AWS best practices in python code,” in *36th European Conference on Object-Oriented Programming, ECOOP 2022, June 6-10, 2022, Berlin, Germany*, ser. LIPIcs, K. Ali and J. Vitek, Eds., vol. 222. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, pp. 14:1–14:28. [Online]. Available: https://doi.org/10.4230/LIPIcs.ECOOP.2022.14

[15] M. Emmi, L. Hadarean, R. Jhala, L. Pike, N. Rosner, M. Schäf, A. Sengupta, and W. Visser, “RAPID: checking API usage for the cloud in the cloud,” in *ESEC/FSE ’21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, D. Spinellis, G. Gousios, M. Chechik, and M. D. Penta, Eds. ACM, 2021, pp. 1416–1426. [Online]. Available: https://doi.org/10.1145/3468264.3473934