Understanding Why and Predicting When Developers Adhere to Code-Quality Standards

Manish Motwani\textsuperscript{3} and Yuriy Brun
University of Massachusetts
Amherst, Massachusetts 01003-9264, USA
\{mmotwani, brun\}@cs.umass.edu

Abstract—Static analysis tools are widely used in software development. While research has focused on improving tool accuracy, evidence at Microsoft suggests that developers often consider some accurately detected warnings not worth fixing: what these tools and developers consider to be true positives differs. Thus, improving tool utilization requires understanding when and why developers fix static-analysis warnings.

We conduct a case study of Microsoft’s Banned API Standard used within the company, which describes 195 APIs that can potentially cause vulnerabilities and 142 recommended replacements. We find that developers often (84\% of the time) consciously deviate from this standard, specifying their rationale, allowing us to study why and when developers deviate from standards. We then identify 23 factors that correlate with developers using the preferred APIs and build a model that predicts whether the developers would use the preferred or discouraged APIs under different circumstances with 92\% accuracy. We also train a model to predict the kind of APIs developers would use in the future based on their past development activity, with 86\% accuracy. We outline a series of concrete suggestions static analysis developers can use to prioritize and customize their output, potentially increasing their tools’ usefulness.

Index Terms—developer-centered static analysis, tools and environments, empirical software engineering

I. INTRODUCTION

Static code analysis is an efficient way to identify software bugs by reasoning about runtime behavior without executing the code. Static analysis tools (SATs) are widely used in both commercial closed-source [9], [51], [66] and open-source [5], [21] development. At Microsoft, teams use SATs to monitor adherence to a variety of code-quality standards, employing engineers to continually improve SATs with state-of-the-art methods. However, studies show that developers tend to fix only a small fraction of the bugs reported by SATs [27], [43]. The common view is that the frequent false-positive warnings these tools report is one of the main reasons developers underutilize the tools [22], [35], [43], [55], [71]. Accordingly, a large research effort [14], [23], [24], [31]–[33], [64], [83], [85] has focused on methods for increasing SATs’ accuracy. However, anecdotal evidence at Microsoft shows that developers often find true-positive warnings not worth fixing. At Microsoft, teams typically designate developers to analyze a SAT’s warnings to select the ones worth fixing. Other developers then either fix the warnings, or explicitly label them as “won’t fix”, stating their rationale. When we presented a product team with a 95\%-accurate SAT for detecting deviations from an important code-quality standard, the team considered only ~45\% of the tool’s true-positive warnings as worth fixing. Analyzing why the team made these choices, we found that teams often consider fixing warnings in non-production code (e.g., tests) and code not targeting a current ship cycle, as not worth their effort. Further, in some cases, fixing a warning may degrade performance, and so teams elect to violate the standard, but mitigate the associated risks by code reviews and extra safety checks. These observations suggest that making SATs more accurate may not be sufficient to achieve a desired fix rate. In essence, what the teams consider to be true positives differs from what SAT designers traditionally have.

The central goal of this paper is understanding when and why development teams fix SAT warnings, and developing models that accurately predict whether warnings are worth bringing to the developers’ attention. These contributions can then lead to concrete suggestions for SAT designers to improve SAT utilization, complementing existing, accuracy-based methods.

Representing the first tangible step toward that goal, this paper performs a case study on a single code quality standard and a single SAT (both used at Microsoft), and develops a machine-learning-based approach for predicting which SAT warnings the developers fix. At Microsoft, teams follow Microsoft’s Security Development Lifecycle (SDL) process [48], which consists of a set of practices to develop secure and compliant software. Our study uses Microsoft’s Banned API Standard [25], [63], which is part of SDL, and sarif-patternmatcher [50], which is one of several SAT solutions [49] developed at Microsoft to detect potential violations. (Note that the standard’s name is a misnomer. In practice, today, the APIs it describes are merely discouraged rather than banned completely from use. Developers are expected to make reasoned decisions on API use, and, as this paper describes, there exist contexts in which the use of these discouraged APIs is, in fact, acceptable.) The Banned API Standard describes 195 discouraged APIs, which are known to sometimes be unreliable and can cause software vulnerabilities, and 142 preferred C/C++ APIs, which are recommended replacements for the discouraged ones. For example, the standard recommends

\textsuperscript{3}Work completed during an internship with Microsoft’s IES Security Tools Group.
using `strcpy_s`, instead of `strcpy` as the latter can result in buffer overflows [12].

Banned API Standard and sarif-pattern-matcher are well-suited for our study for three reasons:

First, sarif-pattern-matcher is highly accurate for this standard. It is developed by Microsoft software quality assurance experts and has very few false positives, and properly hides warnings in natural language and in commented-out source. Improving the accuracy of SATs is orthogonal (but complementary) to our study’s goals, making this standard with an existing accurate SAT a good case study candidate.

Second, a large number of teams at Microsoft are expected to follow this standard. Our analysis of 119,869 repositories finds that 9,021 repositories use the Banned API Standard (the main limiting factor is the standard’s applicability only to C/C++). As violating the standard is known to cause critical security vulnerabilities [72], teams consider the standard very seriously. Our analysis is limited to 162 of the 9,021 repositories that are actively maintained, have at least 100 source files and 1,000 commits, and have more contributions from internal Microsoft developers than external open-source developers. These 162 repositories have code contributions from around 15,000 developers, and belong to 49 projects from 15 diverse teams at Microsoft.

Third, developers surprisingly often (84% of the time) chose to deviate from this standard and use discouraged APIs, with justification, providing ample opportunities to understand developer reasoning. In the 162 repositories, we detected 479,987 discouraged and 93,663 preferred API usages (573,650 total) associated with 3,137 developers.

This observation suggests that the vast majority of the warnings sarif-pattern-matcher reports, despite seemingly being true positives, in fact, do not result in code improvements. These uses of discouraged APIs enable us to understand developer reasoning toward improving the SAT. The developers make their decisions based on assessed risk, the availability of a safe API alternative, and the complexity of correctly using that API. For example, we find that teams tend to adhere to the standard more in code that is under active development and less in code borrowed from other projects. When reusing legacy code with discouraged APIs, developers elect to implement additional safety checks rather than reimplementing that legacy functionality using preferred APIs. Further, teams use preferred APIs more in non-test code and source files that involve more editing and have multiple contributors. With the goal of understanding such intricacies of API usages toward improving SAT design, we identify and analyze 25 factors that encode the circumstances in which teams implement code-quality standards. We use these factors to model and predict warning usefulness. This can allow SATs to prioritize and customize their warnings by considering the development context, improving their utility in the development process.

While our analysis focuses on this one standard, the methodology we describe should generalize to other code-quality standards and SATs, as both the methodology and the 25 factors are agnostic of the standard.

To make our study possible, we extend sarif-pattern-matcher to also detect preferred API call sites, to trace relevant developer information, and to perform temporal analysis. In addition to analyzing the 162 repositories, we do a deeper, temporal case study on one repository over a 10-year span.

We answer three research questions:

**RQ1:** What factors correlate with development teams’ adherence to the Banned API Standard?

**Answer:** Complexity of the project, work environment, whether the code was under active development or non-shipping, developer’s coding experience, and whether a developer is an internal or an external employee.

**RQ2:** Can the correlated factors predict adherence to the Banned API Standard?

**Answer:** An artificial-neural-network (ANN)-based model can predict whether development teams will use preferred or discouraged APIs with 92% accuracy.

**RQ3:** Can development team’s past activity, encoded in terms of correlated factors, predict future standard adherence?

**Answer:** An ANN-based model trained on past behavior can predict whether team will use preferred or discouraged APIs in the future with 86% accuracy.

SATs designed to detect usage of specific APIs can use these prediction models to prioritize and customize their warnings, improving utilization. Our findings drive our concrete recommendations for improving SATs:

1. SATs can prioritize the warnings the developers are more likely to address. For example, our analysis showed that developers are more likely to address violations in production code, and for some operations (e.g., stream buffering) than for others (e.g., memory copy), often for performance reasons. By prioritizing violations developers are likely to fix, SATs can ensure developers see more of the warnings they will act on, earlier.

2. SATs can customize warnings to include suggestions on how to address them, to improve developer compliance. For example, our analysis showed that more experienced developers use preferred APIs in reused legacy code, developers often opted to add safety checks instead of replacing those APIs.

3. SATs can customize warning content based on who is using them. Our analysis showed that more experienced developers used more preferred APIs and may need only a simple prompt to fix discouraged APIs. However, novice developers may need to see more context about the associated risks and potential solutions. A SAT can also suggest code reviews when it believes other developers would fix a warning left unfixed.

4. Team culture plays an important role in API use, which can help customize SATs. For example, SATs could raise an alarm if a team fails to follow a standard when other teams likely would have, or allow a team to consciously limit the warnings to those of the highest priority.

**II. BACKGROUND**

SATs reason about program behavior without running the program. They typically combine different kinds of analy-
ses that consist of independent rules. Each rule describes a requirement that correct, high-quality code should meet. A set of rules constitutes a code-quality standard. If the tool encounters code that violates a rule, it reports a warning. It is up to the developer to decide whether the warning is real and worth addressing. In practice, SATs can detect software bugs and vulnerabilities that range from purely syntactic ones [26] to complex semantic ones that can be used to perform program optimization [2] and fix security vulnerabilities [65]. However, the warnings produced by SATs are often coarse-grained because they over-approximate program behavior to encompass all possible inputs [44]. To address this, researchers have explored the kinds of bugs that SATs can accurately detect [22], [75] and the kinds of bugs developers tend to fix, along with the time they take to fix them [27], [43], [45], and methods to suggest automated fixes [40], [45].

Even though SATs provide several benefits, there exists ample evidence that these tools are underutilized by developers [27], [40], [43]. A lack of accuracy (i.e., a high percentage of false-positive warnings) and the manual effort required for comprehending warnings to identify bugs that are worth fixing are the two main issues that complicate the adoption of SATs [43], [71]. Accordingly, a large research effort has focused on methods for increasing SATs’ accuracy [14], [23], [24], [31]–[33], [64], [83], [85]. These approaches consider different aspects of a SAT’s warning, including factors related to source code [23], [39], [85], repository history [80], and historical data about fixing warnings [34] to detect true-positive warnings. Further, researchers have also proposed ways to automatically generate the fixes for the true-positive warnings [40], [44]. A recent study, which is closest to our work, investigates how developers use SATs in different development contexts defined in terms of the coding activity (e.g., code review, local programming, continuous integration), types of SAT warnings (e.g., naming convention, logic, concurrency), and whether developers update the configuration of SATs [77]. Their findings aim to produce novel automated strategies to help developers pay attention to the right warnings depending on the context they are in. Our proposed approach to help SATs prioritize their true-positive warnings considering factors that influence developers’ decision to act on them complements these existing studies.

III. METHODOLOGY

Figure 1 shows our study’s five-step methodology.

Step-1: Enumerate factors that may influence development teams’ adherence to code-quality standards. To identify the factors that can be objectively measured and could influence development teams’ adherence to code-quality standards in a general software development setting, we analyzed (a) what SATs Microsoft teams use in their workflow, (b) the information available about developers in Microsoft’s developer database, and (c) the meta-data of 119K+ repositories stored in Microsoft’s repository database. We also considered the 23 “Golden Features,” the most important features for detecting actionable warnings identified by Wang et al. [78] by performing a systematic evaluation of the features (or factors) that have been proposed in the literature to improve SAT’s accuracy. Figure 2 shows the 25 factors our analysis identified along with their computation method. To help the reader, we sort these factors into three categories — developer, code base, and quality standard — based on the information sources used to derive the factor. In general, these factors measure developers’ expertise, the complexity of development tasks, work environment, and development teams’ motivation for adhering to a code-quality standard. Note that the process of identifying these factors and the factors themselves are agnostic of the Banned API Standard, and thus can generalize to other standards and SATs. Further, note that all of the 25 factors may/may not affect developers’ adherence to a selected standard. In Step-4, we describe how to identify the factors that are associated with the standard of interest.

Step-2: Define code-quality standard, extend SAT to detect bugs and fixes along with their authors, and compute factors. To perform the analysis, we first had to define a code-quality standard that can be verified using a SAT. This involves carefully crafting rules (code patterns) to accurately detect code that adheres to or deviates from the code-quality standard. Currently, SATs only use rules to detect the violation
### Factors Derived from Developer Information

<table>
<thead>
<tr>
<th>Factor</th>
<th>Description</th>
<th>Computation Method</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CareerStageName</td>
<td>designation of the developer</td>
<td>computed from developer database</td>
<td>categorical</td>
</tr>
<tr>
<td>YearsOfMSEmployee</td>
<td>number of years developer has worked at Microsoft (excluding the duration between the time when a developer left and rejoined the company)</td>
<td>computed using developer database joining or re-joining dates</td>
<td>numerical</td>
</tr>
<tr>
<td>EmployeeLevel</td>
<td>position of developer in organization hierarchy (CEO is 1 and other junior employees have higher values)</td>
<td>computed from developer database</td>
<td>numerical</td>
</tr>
<tr>
<td>JobTitleName</td>
<td>job category that classifies developer’s designation (e.g., Service Engineering, Software Engineering, Program Management, Technical Delivery)</td>
<td>computed from developer database</td>
<td>categorical</td>
</tr>
<tr>
<td>StandardTitle</td>
<td>type of role within a job category that a developer is occupying (e.g., Senior Service Engineer, Software Development Engineer, Consultant)</td>
<td>computed from developer database</td>
<td>categorical</td>
</tr>
<tr>
<td>AuthorChurn</td>
<td>a developer’s code churn in a repository. This varies from [-1, 1] and negative value indicates that developer rewrites her code more often.</td>
<td>computed by parsing and matching patterns in repository</td>
<td>numerical</td>
</tr>
<tr>
<td>WorkLocation</td>
<td>country where the development team is located (e.g., Convex Ski Eng) to which developer belongs</td>
<td>computed by parsing and matching patterns in repository</td>
<td>categorical</td>
</tr>
<tr>
<td>Department</td>
<td>department (e.g., Convex Ski Eng) to which developer belongs</td>
<td>computed by parsing and matching patterns in repository</td>
<td>categorical</td>
</tr>
<tr>
<td>ProjectType</td>
<td>type of the project (e.g., OCAS Word) assigned to developer</td>
<td>computed from developer database</td>
<td>categorical</td>
</tr>
<tr>
<td>ProjectName</td>
<td>project (e.g., OCAS-Word iOS/Mac) assigned to developer</td>
<td>computed from developer database</td>
<td>categorical</td>
</tr>
<tr>
<td>IsInternalMSEmployee</td>
<td>whether developer is an internal (true) or external (false) Microsoft employee</td>
<td>computed from developer database meta-data</td>
<td>dichotomous</td>
</tr>
<tr>
<td>EmployeeType</td>
<td>whether a developer is full-time employee, contract-based vendor, or other (e.g., intern)</td>
<td>computed from developer meta-data</td>
<td>categorical</td>
</tr>
</tbody>
</table>

### Factors Derived from Code Base Information

<table>
<thead>
<tr>
<th>Factor</th>
<th>Description</th>
<th>Computation Method</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>SourceLOC</td>
<td>number of source code lines in a file excluding comments</td>
<td>implemented sarif-pattern-matcher module to parse source files and count lines excluding comments</td>
<td>numerical</td>
</tr>
<tr>
<td>RepoLOC</td>
<td>number of source code lines in a repository excluding comments</td>
<td>implemented sarif-pattern-matcher module to count the lines of code in all source files</td>
<td>numerical</td>
</tr>
<tr>
<td>FileChurn</td>
<td>how often the source file is modified by developers. This varies from [0, 1] and lower value indicates that file is modified more often.</td>
<td>computed by parsing the commit history of source file to identify distinct developers</td>
<td>numerical</td>
</tr>
<tr>
<td>SourceIntermittentChanges</td>
<td>Number of developers who have contributed to a source file prior to a given commit</td>
<td>computed using developer database</td>
<td>numerical</td>
</tr>
<tr>
<td>TeamSize</td>
<td>number of peers of a developer who report to same manager</td>
<td>obtained from repository database</td>
<td>numerical</td>
</tr>
<tr>
<td>RepositoryOrganization</td>
<td>the project umbrella to which a repository belongs (e.g., Office)</td>
<td>obtained from repository database</td>
<td>categorical</td>
</tr>
<tr>
<td>RepositoryProject</td>
<td>the project (within a product umbrella) to which the repository belongs (e.g., Office, OC, OE under office)</td>
<td>obtained from repository database</td>
<td>categorical</td>
</tr>
<tr>
<td>IsTestCode</td>
<td>whether the source file is a test code (true) or not (false)</td>
<td>computed by parsing and matching patterns in the source file enlistment path</td>
<td>dichotomous</td>
</tr>
<tr>
<td>IsActiveCode</td>
<td>whether the source file is under active development (true) or an external dependency (false)</td>
<td>computed by parsing and matching patterns in the source file enlistment path</td>
<td>dichotomous</td>
</tr>
</tbody>
</table>

### Factors Derived from Quality Standard Information

<table>
<thead>
<tr>
<th>Factor</th>
<th>Description</th>
<th>Computation Method</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>RuleType</td>
<td>conceptual topic to which a code-quality standard rule belongs (e.g., APIs in Banned API Standard related to string copying)</td>
<td>manually analyzed quality standard rules to categorize them into conceptual topics</td>
<td>categorical</td>
</tr>
<tr>
<td>QualityStandardApplicability</td>
<td>prevalence of distinct code-quality standard rules in a repository</td>
<td>fraction of #distinct rules detected and total #rules defined in the quality standard</td>
<td>numerical</td>
</tr>
<tr>
<td>QualityStandardDensity</td>
<td>prevalence of code-quality standard in a repository</td>
<td>fraction of #rules detected and total LOC</td>
<td>numerical</td>
</tr>
</tbody>
</table>

Fig. 2: Factors that can influence development teams’ adherence to code-quality standards.
for the discouraged APIs. The discouraged APIs are known to, in some situations, cause severe vulnerabilities, such as remote code execution, local elevation of privilege, arbitrary code execution, and system compromise. Therefore, all development teams are recommended to use the preferred APIs, wherever applicable. For example, the standard recommends using `strcpy_s`, `wcscpy_s`, `_mbscpy_s`, instead of `strcpy`, `wcscpy`, `_mbscpy`, respectively, which can cause buffer overruns.

We extended sarif-pattern-matcher [50], Microsoft’s open-source SAT solution for detecting deviations from the Banned API Standard in three ways. First, we augmented the existing set of rules that detect discouraged API use with new rules to detect preferred APIs. This required creating regular-expression-based patterns with criteria to minimize false positives (e.g., excluding API detection in commented out source code and natural language code comments). Second, using Git blame [13], we identified the developer who used the detected preferred or discouraged API. Third, we performed historical, temporal scanning of a code base for a given time duration (specified using a start and end dates) at a given fixed time interval. Finally, with help of an Microsoft data mining expert, we mined developer and source code metadata from the developer and repository databases using the Kusto query language [11] to compute the 25 factors (recall Figure 2) associated with each detected API use and annotated the API use with the computed factors. As developer behavior can change over time, we computed these factors at the instant the developers used the detected API, approximated with the time-stamp of the containing commit.

**Step-3: Execute sarif-pattern-matcher to identify repositories of interest.** We executed sarif-pattern-matcher on 119,869 repositories, which include large product repositories, such as Microsoft Windows, Microsoft Office, Skype, and Bing, as well as small prototype repositories, sarif-pattern-matcher found 9,021 repositories that use at least one of the 337 APIs defined by the Banned API Standard. From the 9,021 repositories, we selected the 162 repositories that have: (1) at least 100 #source files (to ensure that it is not a toy repository), (2) at least 1,000 #commits (to ensure that repository has sufficient development history), (3) more contributions from internal Microsoft developers than the external open-source developers (to ensure that majority of developers know about the Banned API Standard), and (4) an associated product or service that is under active development or maintenance (to ensure that repository is not deprecated). These 162 repositories have contributions made by more than 15,000 developers and belong to 15 diverse product teams at Microsoft. sarif-pattern-matcher detected a total of 573,651 (93,675 preferred and 479,976 discouraged) API usages in these 162 repositories from 3,302 developers. We next annotated each API usage with all the factors (recall Figure 2). While annotating, we found that values for many developer-specific factors, such as CareerStageName, EmployeeLevel, and Department, were either marked as “unknown” or stored in private repositories to which we did not have access. Figure 3 (“all cases”) shows the diversity in all of the annotated APIs while “complete cases” shows the diversity in 27,136 subset of APIs for which all of the factors are annotated successfully.

**Step-4: Empirically measure which factors correlate with development teams’ adherence to the code-quality standards.** We represented development teams’ choice to use a preferred or discouraged API as a binary dependent variable. Numerically, we used “1” for “preferred” and “0” for “discouraged”. Depending on each factor’s data type (see Figure 2), we measured its correlation with development teams’ choice of preferred or discouraged APIs using the appropriate non-

---

**Table 1: Diversity in terms of distinct values for factors in the annotated dataset of 162 repositories that contain 573,651 APIs defined by the Banned API Standard.**

<table>
<thead>
<tr>
<th>Factor</th>
<th># of Distinct Values</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>all cases</td>
</tr>
<tr>
<td>CareerStageName</td>
<td>15</td>
</tr>
<tr>
<td>YearsOfMSExperience</td>
<td>36</td>
</tr>
<tr>
<td>EmployeeLevel</td>
<td>9</td>
</tr>
<tr>
<td>JobTitleName</td>
<td>19</td>
</tr>
<tr>
<td>StandardTitle</td>
<td>62</td>
</tr>
<tr>
<td>AuthornChurn</td>
<td>1,278</td>
</tr>
<tr>
<td>WorkLocation</td>
<td>20</td>
</tr>
<tr>
<td>Department</td>
<td>241</td>
</tr>
<tr>
<td>ProjectType</td>
<td>531</td>
</tr>
<tr>
<td>PositionName</td>
<td>260</td>
</tr>
<tr>
<td>IsInternalMSEmployee</td>
<td>2</td>
</tr>
<tr>
<td>EmployeeType</td>
<td>3</td>
</tr>
<tr>
<td>SourceLOC</td>
<td>3,299</td>
</tr>
<tr>
<td>RepoLOC</td>
<td>161</td>
</tr>
<tr>
<td>FileChurn</td>
<td>1,105</td>
</tr>
<tr>
<td>SourceIntermittentChanges</td>
<td>79</td>
</tr>
<tr>
<td>TeamSize</td>
<td>49</td>
</tr>
<tr>
<td>RepositoryOrganization</td>
<td>15</td>
</tr>
<tr>
<td>RepositoryProject</td>
<td>49</td>
</tr>
<tr>
<td>Repository</td>
<td>162</td>
</tr>
<tr>
<td>IsTestCode</td>
<td>2</td>
</tr>
<tr>
<td>IsActiveCode</td>
<td>2</td>
</tr>
<tr>
<td>RuleType</td>
<td>19</td>
</tr>
<tr>
<td>QualityStandardApplicability</td>
<td>82</td>
</tr>
<tr>
<td>QualityStandardDensity</td>
<td>36</td>
</tr>
</tbody>
</table>

Fig. 3: Diversity in terms of distinct values for factors in the annotated dataset of 162 repositories that contain 573,651 APIs defined by the Banned API Standard. “All Cases” shows the diversity in all of the annotated APIs while “complete cases” shows the diversity in 27,136 subset of APIs for which all of the factors are annotated successfully.
parametric statistical tests, which do not make any assumptions on the underlying distribution of the data.

For each categorical or dichotomous factor, we created a $n \times 2$ contingency table where $n$ represents the number of distinct values the factor can take. For each value, the table contains the distribution of preferred and discouraged API usage by developers. We then ran a Chi-Square test of independence [46] on the contingency table to see if the development teams’ choice to use preferred or discouraged APIs is independent of that factor. The Chi-Square test computes a p-value ($p$) that indicates whether our null hypothesis (a developers choice to use a preferred or discouraged APIs is independent of the factor) should be rejected. For each numerical factor, we split the distribution of that factor’s values into two distribution samples: (1) the distribution of the factor’s values for which developers used preferred APIs, and (2) the distribution of the factor’s values for which developers used discouraged APIs. We used Mann-Whitney U test [47], which measures rank biserial correlation [19] to determine if the difference between the two distribution samples was statistically significant. That is, the test computes a p-value ($p$) that indicates whether our null hypothesis (the two distribution samples are statistically indistinguishable) should be rejected. If the two distribution samples are statistically distinguishable, it indicates that the development teams’ adherence correlates with that factor.

We used the standard criteria for statistical significance, $p \leq 0.05$, to mean the factor correlates with development teams’ adherence, and $p > 0.1$ to mean the factor does not correlate. We also computed the effect size ($es$) i.e., the strength of the correlations using Cramers’ V [42] for categorical and dichotomous factors, and Somers’ D [57] for numerical factors. While Cramers’ V $\in \{0,1\}$, Somers’ D $\in \{-1,1\}$. As per Gignac et al. [18], we mapped effect size values to adjectives: very weak ($|es| < 0.1$), weak ($0.1 \leq |es| < 0.2$), moderate ($0.2 \leq |es| < 0.3$), and strong ($0.3 \leq |es|$). To verify that correlations observed were not arbitrary, we computed the 95% confidence interval (CI) that tells us the range of the effect size with 95% confidence. We therefore considered correlations to be statistically significant when $p \leq 0.05$ and 95% CI does not span 0.

Step-5: Predict adherence to the code-quality standard from correlated factors. We formulated the problem of predicting development teams’ adherence to the code-quality standard from correlated factors as follows: Let $F_1,F_2,...,F_n$ represent the $n$ independent factors that influence development teams’ adherence to the quality standard, and let $y$ represent the binary dependent variable indicating whether the team adheres to $(y=1)$ or deviated from $(y=0)$ the quality standard. Our goal was to compute a function $f(F_1,F_2,...,F_n)$ that evaluates to $P(y=1)$, which is the probability that a team will adhere to the quality standard. In the case of the Banned API Standard, $P(y=1)$ represents team decides to use a preferred API.

To solve this problem, we experimented using the logistic regression and artificial neural network (ANN) [73] models, and found that ANN performed better than logistic regression. We trained and evaluated a five-layer ANN model consisting of four hidden layers (h1 and h3 as dense while h2 and h4 as dropout layers) and one output layer that used non-linear factor combinations of $sigmoid$ and $relu$ functions to learn and predict adherence to the Banned API Standard. We used $sigmoid$ for the outermost hidden layer because its output is a value between 0 and 1, making it useful for binary classification; we used $relu$ for inner hidden layers because the fast computation made the model more efficient. Figure 4 shows the model’s architecture, along with its parameters and the non-linear functions it uses. The model’s input is a matrix whose columns represent multiple factors and each row represents a development context (encoded in terms of the factors) in which teams used preferred or discouraged APIs. The model combines multiple factors using non-linear functions to compute the output matrix that contains the probability that a team would use a preferred API for each row of the input matrix. The model used supervised learning with a training and a test phase. During the training phase, the model took as input both the input matrix and the expected team choice to use a preferred or discouraged API (ground truth), and learned the parameters (values of hidden layers) to predict the ground truth from the given factors. To ensure that the model did not over-fit on the training set, it was validated on a fraction of the training set (i.e., the validation set) that was not used for learning. If a model’s accuracy on the validation set started to decrease, it starts to over-fit on the training set. The model was trained until the max number of epochs were reached or the model started to over-fit. During the test phase, the trained model took as input only the matrix and output the probability of teams’ choice to use preferred $(P(y=1))$ or discouraged $(P(y=0))$ APIs for each row in the input matrix. We considered $P(y=1) > 0.5$ to imply that a team will use a preferred API.
We evaluated the ANN model under the following two scenarios: (1) **Analysis over code bases.** This scenario considered the dataset of annotated API usages from all 162 repositories and used uniform random sampling to create training and test sets that were representative of all repositories. (2) **Analysis across time.** This scenario considered the dataset of annotated API usages detected from the case study repository at fixed time intervals (15 days) starting from July 15, 2011 and ending on Sept 30, 2021. (Our analysis uses fixed time intervals instead of every commit for performance reasons. Analyzing every commit over 10 years would require executing the SAT on ∼25,000 revisions; our analysis handled ∼2,400 revisions.) The training set consisted of annotated API usages detected before April 30, 2020, and the test set consisted of API usages detected on or after April 30, 2020. The model learned from the development team’s past API usage to predict their behavior in terms of using preferred or discouraged APIs in the future. The hyper-parameters used for training the models were: batch size = 64; validation split = 0.3; decaying learning rate (lr) starting from 0.001 and decaying by a factor of 0.1; dropout rate (h2) = 0.4; dropout rate (h4) = 0.3; loss function = binary cross entropy; optimizer = adam.

**IV. Evaluation**

This section answers three research questions about the developers’ adherence to the Banned API Standard, first describing the dataset and metrics used to answer these questions.

### A. Dataset

**Correlation Analysis.** To identify correlations between each factor and development teams’ adherence to the Banned API Standard, we used the dataset of 573,651 API usages in 162 repositories annotated with 25 factors listed in Figure 2. As not all APIs could be annotated with all the factors (recall Step-3 in Section III), to analyze the correlation between a factor and development teams’ adherence to the Banned API Standard, we used a subset of the API usages that could be annotated with that factor. While presenting correlation results, we present the distribution of preferred and discouraged API usages annotated with that factor.

**Prediction over code bases.** To make predictions about development teams’ adherence to Banned API Standard, we used the subset of 27,309 API usages that could be annotated with all 23 of the correlated factors (with > 1 distinct values in Figure 3 (“complete cases”) and with p < 0.05 in Figure 5). We split this dataset into a training set consisting of 19,117 API usages (∼70%) and a test set, which consisting of 8,192 API usages (∼30%). To do this, we randomly sampled the API usages from each class (preferred and discouraged) to ensure that the overall class distribution (∼20% preferred and ∼80% discouraged) was preserved in the training and test sets. To prevent classification bias, we used the Synthetic Minority Oversampling Technique (SMOTE) [10], a technique to balance training sets, to balance the distribution of preferred and discouraged APIs in the training set. This increased the training set size from 19,117 to 27,545 with ∼43% preferred and ∼57% discouraged API usages. Note that it is challenging for techniques like SMOTE to exactly balance the distribution of the two kinds of API usages. To address the remaining slight imbalance, we used additional evaluation metrics, as described in Section IV-B, to assess the performance of trained models.

**Prediction across time.** To make predictions about development team’s use of preferred or discouraged APIs in the future using the factors that encode their past API usage, we use the case study repository that has been under active development since July 5, 2011. We used sarif-pattern-matcher to scan this repository at an interval of 15 days starting from July 15, 2011 to Sept 30, 2021. For each time-stamp, the API usages detected by sarif-pattern-matcher were annotated with the factor values computed for that time-stamp, as these values can change over time. sarif-pattern-matcher detected a total of 199,872 API usages (143,287 discouraged and 56,585 preferred) across 170 timestamps. We split the dataset into before/after a single timestamp of April 30, 2020, such that all annotated API usages before this date comprise the training set and all usages on or after the selected date comprise the test set. This resulted in 140,188 API usages (∼70%) in the training set and 59,684 (∼30%) in the test set. As many of the factors remain constant for a single repository, for this analysis we used the 7 factors (“AuthorChurn”, “SourceLOC”, “IsActiveCode”, “IsTestCode”, “RuleType”, “FileChurn”, and “SourceIntermittentChanges”) that change more frequently over time along with the path of source files. Finally, we used SMOTE [10] to balance the distribution of preferred and discouraged API usages that resulted in a training set of 283,150 API usages with ∼43% preferred and ∼57% discouraged APIs.

### B. Evaluation Metrics

To measure correlation between individual factors and development teams’ adherence to the Banned API Standard, we use the p-value, effect size, and 95% CI computed for that factor using appropriate statistical tests (see Step-4 in Section III). We consider a correlation to be statistically significant if p < 0.05 and 95% CI does not span 0. We evaluate prediction models using the following metrics.

1) **Accuracy:** The ratio of correct predictions to all predictions (note that predicting adherence is equally important as predicting deviation for improving SAT’s utilization).

2) **95% CI:** The range of Accuracy with 95% confidence.

3) **AUC-ROC:** Area under the ROC curve, which gives the average accuracy of correctly predicting preferred API usages and discouraged API usages. This varies from 0 to 1, where 1 corresponds to 100% accuracy. A value between 0.7 to 0.8 is considered acceptable, 0.8 to 0.9 is excellent, and more than 0.9 is outstanding [41].

4) **Confusion Matrix:** A matrix to visualize the model predictions and analyze the errors.

### C. Results

**RQ1:** What factors correlate with development teams’ adherence to the Banned API Standard?
with specific geographical locations (WorkLocation), departments (Department), projects (ProjectType and ProjectName) and products (RepositoryOrganization, RepositoryProject, and Repository). This indicates that adherence can be consistent within teams but can differ across teams within the company and these results corroborate the findings of Lavallée et al. [36]. Due to non-disclosure agreement, we can not share the specific factor values associated with developer or product related details that were found to be associated with use of discouraged and preferred APIs. Finally, teams tend to use more preferred APIs in their non-test code (IsTestCode) and code that is under active development or maintenance (IsActiveCode) than in the external code borrowed from other projects. Our qualitative analysis of these findings suggests teams behave this way because test code is rarely shipped and therefore considered less relevant, especially when chasing a release deadline. External code was left as-is to avoid conflicts with updates outside of their control.

Statistical tests for numeric factors revealed how adherence to the Banned API Standard can be influenced by changing factor values. For example, a very weak positive correlation was observed for YearsOfMSExperience, suggesting that developers who have worked at Microsoft longer are slightly more likely to use preferred APIs. On the other hand, a weak negative correlation observed for EmployeeLevel indicates that junior developers, whose EmployeeLevel is higher, are likely to use more discouraged APIs. Similarly, we found that developers who edit less code (AuthorChurn) or work on large code bases (SourceLOC and RepoLOC) tend to use more discouraged APIs, while developers who work on source files that involve a lot of changes (FileChurn) and have multiple developers working on them (SourceIntermittentChanges) tend to use more preferred APIs. Some of these correlations observed (e.g., for SourceLOC and AuthorChurn) are consistent with the prior studies [61], [62].

**RQ2: Can the correlated factors predict adherence to the Banned API Standard?**

The graphs at the top of Figure 6 show the ANN model’s performance for 100 epochs on the training set. As shown, loss decreases and accuracy increases on the training and validation sets with increasing epochs and the convergence is achieved after ~85 epochs. After 100 epochs, the accuracy obtained on the training set is 93.40% and 77.41% for the validation set.
Fig. 6: Performance of our ANN model for predicting development teams’ choice to use preferred or discouraged APIs over different code bases. The graphs show the model’s performance during the training phase. “Confusion Matrix” and “Evaluation Metrics” show the model’s performance on the test set. Our model predicts choices with 92% accuracy.

The bottom of Figure 6 shows the trained ANN model’s performance on the test set in terms of the evaluation metrics. As shown, our model can predict whether teams will use preferred or discouraged APIs with 92% accuracy, and the 0.8459 AUC-ROC shows this is excellent performance for this prediction task.

An artificial-neural-network (ANN)-based model can predict whether developers will use preferred or discouraged APIs with 92% accuracy.

RQ3: Can development team’s past activity, encoded in terms of the correlated factors, predict future standard adherence? The graphs at the top of Figure 7 show the training of the ANN model for 100 epochs. As shown, the loss decreases and the accuracy increases on the training and validation sets with increasing epochs and the convergence is achieved after ~88 epochs. After 100 epochs, the accuracy on the training set was 93.60% and the validation set was 70.80%.

The bottom of Figure 7 shows the trained ANN model’s performance on the test set in terms of the evaluation metrics. As shown, the model predicts if developers will use preferred or discouraged APIs with 86% accuracy, and the 0.7879 AUC-ROC indicates the model’s performance is acceptable.

For the large code base with a decade of development activity, an ANN-based model can predict if a team will use preferred or discouraged APIs in the future with 86% accuracy based on 7 factors computed from past development activity.

D. Discussion and Threats to Validity

The factors presented in this paper are not an exhaustive list and other factors could influence development teams’ compliance. For example, studies [78], [81]–[83] show that factors related to SAT warnings are useful to improve SAT accuracy. However, we do not consider such factors because recent research showed that these factors cause data leakage, leading to misleading results [28]. Further, a factor can encode organization-specific context, and it is important to understand the impact of such factors on the study. For example, instead of using years of experience, we use YearsOfMSExperience

440
that measures the developers’ work experience at Microsoft and disregards their previous experiences. We made this choice because other companies may not use the Banned API Standard, and so even experienced developers who are new to Microsoft may be a novice when it comes to Banned API Standard. Therefore, we consider such developers to be novices. Although our methodology and factors generalize to other software development scenarios, the results presented for the Banned API Standard detected using sarif-pattern-matcher are observational and rely on the dataset used for our analysis.

The ANN model to predict development team’s future adherence to the Banned API Standard from their past behavior performed reasonably well by using only 7 of the 23 correlated factors in a single repository. However, when we used the same set of factors to perform analysis over code bases, the accuracy of the model dropped significantly. This indicates that factors that vary across different teams contribute significantly when predicting adherence over multiple code bases. These results are consistent with a prior study [36] and our observation that different teams at Microsoft had different adherence behaviors.

We address the threat to external validity by ensuring that our evaluation datasets consist of the large and diverse repositories that are representative of real-world software development. We address the threat to internal validity by computing necessary evaluation metrics (“effect size”, “95% CI” and “AUC-ROC”), which give us statistical assurance that our results are not influenced merely by the large size of our dataset. Our collected dataset is incomplete, missing some values, particularly about the developers. This is not uncommon in data collected from databases from multiple, real-world, complex systems. We address threats posed by incomplete data by only using the data points for which we have complete data for our prediction modeling, and only using non-missing data points for correlation computations. Our methods of detecting API usages and computing IsActiveCode and IsTestCode features use a heuristics-based approach; their accuracy may influence our findings. While we use SMOTE, a popular technique to balance the imbalanced training sets to prevent influence our findings. While we use SMOTE, a popular technique to balance the imbalanced training sets to prevent classification bias, our results for predictions show that our technique to balance the imbalanced training sets to prevent classification bias by computing AUC-ROC and confusion matrix to analyze predictions for discouraged and suggested preferred API usages separately.

V. ACTIONABLE ADVICE FOR SAT DESIGNERS

Our analysis focused on the Banned API Standard and sarif-pattern-matcher, but our methodology (Figure 1) and the influencing factors (Figure 2) are standard and tool agnostic. We next advise SAT designers how to improve SATs.

Prioritizing warnings based on context. SATs can predict which warnings the teams are more likely to address — whether by fixing the warnings or by adding safety checks or taking other risk-mitigation steps — and rank those warnings higher. For example, our analysis shows that prioritizing warnings in production code, actively-under-development code, code being worked on by more developers, and code with higher churn is likely to get the warnings teams address to be seen sooner.

Prioritizing warnings based on prior behavior and improve fix suggestions. SATs can observe teams’ behavior on code and extrapolate that behavior to prioritize warnings. For example, our analysis observed that when reusing legacy code, teams were less likely to replace discouraged APIs. If the SAT identifies a warning in legacy code that has been reused elsewhere by other teams, and those teams elected not to replace the APIs, making them as “won’t fix,” the current team is likely to do the same. The SAT can further make suggestions for fixes similar to prior teams, such as adding additional safety checks, rather than replacing the APIs. A SAT that checks adherence to multiple standards can prioritize the warnings of the standard more often complied to by the team.

Improving warnings content based on developer needs. Our analysis identified that more experienced developers are more likely to address warnings than novice developers. Potentially, SATs can improve by customizing the information displayed with the warning. Novice developers may need to see more context and deeper explanations of why a discouraged API is unsafe, as well as potential solutions. More experienced developers may only need to see the discouraged API use and suggested preferred API.

Modern IDEs can run static analysis with every build (and some proposals even suggest running static and dynamic analyses in the background, continuously [29], [67], [68]). SATs running in this way can use our models in a form of speculative analysis [7], [8], [56] to predict teams’ actions and proactively suggest the best coding practices suitable for the given development context. Displaying all warnings in such a continuous manner is likely to overwhelm the developer, but using our predictive models can help focus the developer on the most pertinent warnings that they are likely to address.

VI. RELATED WORK

Several studies have been conducted to identify developer factors that affect code quality. Salamea et al. [69] analyzed two open-source projects to investigate how code quality, measured in terms of technical debt (TD) (number of code smells (e.g., duplicated code) produced per developer), is affected by developer participation, experience, and communication skills. They found that the level of participation (lines of code contributed) of the developers and their experience (number of months active) in the project have a positive correlation with the amount of TD while their communication skills have barely any impact. Alfayez et al. [3] analyzed 38 Apache projects to investigate how developers’ frequency of commits, their seniority, and the size of time lapses between their commits relate to code quality. They found that developers’ frequency of commits and seniority are negatively correlated with introducing defects and the size of the time lapses.
between developers’ commits is positively correlated with introducing defects in code.

Tufano et al. [76] analyzed five open-source Java projects to identify how commit coherence (i.e., how much it is focused on a specific topic), developers’ experience (measured using the metrics that compute developers’ familiarity with the files modified by them based on their commits), and the interfering changes performed by other developers on the files involved influence the likelihood that a commit induces a fix. They found that fix-inducing commits are less coherent, are produced by more experienced developers, and have a higher number of past interfering changes than commits that did not induce a fix. Li et al. [38] analyzed data of 76 developers in four open-source software projects to understand how developers’ bug-introducing commit rates change over time, and its possible dependency on lines of code (LOC). They showed that developers’ bug-introducing commit rates tend to include two phases: a first increasing phase and a decreasing phase. Further, larger (in terms of LOC) commits introduce a higher number of software defects. Qui et al. [61] analyzed six open-source projects to investigate how developer quality (rate of non-bug-introducing commits) relates to software evolution. They found that developer quality tends to increase with software evolution, developers with more code contribution are more likely to have higher developer quality, and source code ownership does not have a consistent and significant correlation with developer quality. Rahman et al. [62] analyzed four open-source projects to understand how developers’ file ownership, file experience, and overall experience impact code quality. They reported that multiple developers contributing to the same file reduces defective code, developers introduce fewer defects in their own files, and developer general experience (measure in terms of the deltas committed to a repository up to a particular time) has a weak relationship with code quality. Lavallée et al. [36] investigated how organizational factors such as structure and culture impact the working conditions of developers. They found that some software teams tend to prefer quick-and-dirty solutions over time-consuming ones, incurring technical debt due to fear of exceeding their budget. All of these studies consider factors that are related to either developer or code base properties.

Our work extends the state-of-the-art by using factors covering both, developer and code base aspects, along with the properties of code-quality standards.

Due to the variable nature of code quality standards and the onerous manual efforts required for compliance, the industry is becoming more efficient by using automated means of recognizing, verifying, delegating, and monitoring compliance-related tasks [16], [17]. Code quality is affected not only by the developers but also by the process the developers follow. For example, enforcing static analysis checks at commit time can increase compliance [66], and adhering to testing practices can improve code quality [4]. While certain types of tests can be automatically generated, e.g., regression tests via Ran-doop [58] or EvoSuite [15], most testing for new functionality requires human-written oracles. Recent work has aimed to generate such oracles automatically from comments or natural-language specifications. Swami [52] generates oracles from semi-structured specification and is able to discover defects in code and ambiguities in specifications in mature projects, such as the Rhino and Node.js JavaScript implementations. Toradacu [20] and Jdoctor [6] and @tComment [74] extract oracles from Javadoc comments, with Toradacu and Jdoctor focusing on extracting oracles for exceptional behavior and @tComment on extracting preconditions related to nullness of parameters. C2S [86] captures developers’ intent by automatically translating natural language comments in the source code into formal program specifications.

In addition to static analyses that make suggestions for developers for improving code quality, it is possible to improve code quality automatically. Automated program repair aims to automatically repair failing tests by modifying the source code via a variety of methods [37]. For example, genetic algorithms and random search through the space of modifications can often find patches that pass the supplied tests [59], [79]. Unfortunately, these repair mechanisms can often break untested or undertested behavior [54], [60], [70], and while some recent advances have improved the frequency with which automated program repair techniques produce correct patches [1], [30], [53], [84], [87], improving the quality of automatically generated patches remains an open problem.

VII. FUTURE RESEARCH DIRECTIONS

This paper proposes a novel approach to predict circumstances in which development teams are likely to adhere to a quality standard and act on the true positive warnings raised by SATs. To accomplish this, our approach detects instances of adherence to and noncompliance with a quality standard, and developer and programming-context factors that correlate with these instances. Our next steps are to use our findings to improve SATs for the Banned API Standard and conduct user studies to evaluate the effects of our improvements in practice. However, this paper is merely a first step towards understanding the relationships between these factors and teams’ adherence. While our work potentially allows SATs to detect context-aware situations in which to interpret rule applicability, future research can consider methods for detecting context-aware situations that violate or adhere to code-quality standards undetectable by rules alone. This can extend SATs to detecting violations and compliances of more complex code-quality standards than is possible today.

VIII. CONTRIBUTIONS

We have studied the problem of identifying when and why development teams adhere to or deviate from code-quality standards with the aim of improving SATs’ utilization. Our large-scale case study at Microsoft has identified the factors that affect the teams’ adherence and allowed the creation of models for predicting situations in which teams will choose to deviate from or adhere to the standards. SATs can use these models to prioritize and customize their warnings, improving their utility to the development process.
ACKNOWLEDGMENTS

We thank Michael C. Fanning, Larry Golding, Harleen K. Kohli, Suvam Mukherjee, and Danielle Gonzalez for their contributions to this work. This work is supported by the National Science Foundation under grant no. CCF-1763423.

REFERENCES


