TraceJIT: Evaluating the Impact of Behavioral Code Change on Just-In-Time Defect Prediction

Anonymous Author(s)

Anonymous

***Abstract*—Just-In-Time (JIT) defect prediction strives to model changes that induce future fixes so that they can be predicted or better understood to inform development practices. Prior work demonstrates that the majority of the predictive/explanatory power of JIT models derives from the size of a change (i.e., larger changes tend to be defect-prone); however, in practice, a misguided change to even a single line of code can lead to defects. While it is clearly the case that larger changes are more likely to alter the product behavior, even small changes are capable of doing this, and when they do, they pose a risk that teams should note. However, to the best of our knowledge, JIT defect prediction models are yet to incorporate features that characterize the change in product behavior when modelling risk.**

This paper is the first to explore the impact of behavioral code change on JIT prediction. Specifically, we propose seven dynamic features that capture the difference in product behavior before and after applying a change. These features are computed using trace logs that are collected during invocations of test suites. Using these logs, we identify which lines of code started/stopped being exercised after a change. We evaluate these features by conducting an empirical study of two large and thriving open-source projects. We observe that, compared to baseline models that use traditional features, adding our proposed set of behavior features leads to improvements of up to 5.9% of ROC-AUC, 44.8% of precision, and 14.1% of PR-AUC. This paper not only demonstrates the importance of behavioral features for JIT defect prediction, but also lays the foundation for future work on behavioral features in other software engineering contexts, such as build outcome prediction and code reviewer recommendation.

***Index Terms*—Just-In-Time defect prediction, dynamic features**

I. Introduction

Since even a single defect may significantly harm an organization reputationally, economically, or otherwise, developers devote considerable time to Software Quality Assurance (SQA). To efficiently allocate SQA resources to the source code that is likely to be defect prone, defect prediction has been studied for decades [1]–[3]. To make recommendations in a change-oriented development style, Just-In-Time (JIT) defect prediction approaches have been proposed [4], [5].

Various sets of features that characterize changes and predictive approaches have been applied to JIT defect prediction; however, recent work [6] demonstrates that a naïve size-based ranking model can outperform even the state-of-the-art JIT models. Even in the original JIT defect prediction studies, the top contributing feature in the model fits was often the size of the change [4], [7]. While useful, it is somewhat superficial that larger changes are more likely to be defect-prone because larger changes modify more lines, which creates more "surface area" for future defect-fixing changes to implicate them.

On the other hand, small changes may also induce bugs. For example, Karampatsis *et al.* [8] reported that even a single line often leads to defects. We conjecture that the riskiness of both large and small changes can be explained by their impact on the behavior of the underlying software product. By their nature, large changes change plenty of code, which creates several opportunities to alter the behavior of the software product; however, even small changes can alter product behavior. It is these (unexpected) changes in behavior that may introduce defects. Indeed, this idea that small changes may introduce defects is at the core of mutation testing [9], which performs small changes to the code (e.g., negating boolean expressions) to simulate defective changes to the codebase. Moreover, the popular notion of mining for "SStuBs" [8], [10], [11], i.e., simple stupid bugs that appear on a single statement and the corresponding fix is within that statement, further suggests that small changes account for a considerable proportion of defect-fixing activity.

These kinds of unexpected behavior changes are typically uncovered during unit testing. However, unit testing tends to focus on inputs and corresponding outputs without paying attention to which lines are exercised. For example, Fraser *et al.* [12] develop EvoSuite to automatically generate test cases. While EvoSuite significantly increases test coverage, it is reported that many real defects are still missed even by the augmented test suites [13].

This paper proposes `TraceJIT`—a JIT defect prediction model that includes features that characterize the behavioral impact of changes. More specifically, the `TraceJIT` approach leverages trace logs that are produced during test execution using dynamic software analysis. These trace logs are then analyzed to compute dynamic behavior features that characterize the change in terms of behavior. The intuition of these behavioral features is that more substantial behavioral changes are likely to be defect-prone. In other words, the prediction model would monitor for abnormal behavior in a similar way that breakpoint debugging does.

To evaluate `TraceJIT`, we conduct an empirical study of two large and thriving open-source systems. We compare `TraceJIT` with the state-of-the-art JIT defect prediction model of Sohn *et al.* [5] in the context of these studied systems. We observe that `TraceJIT` outperforms the previous state of the art, with improvements of 1.3% in terms of ROC-AUC and 23.6% in terms of Precision.

The results of our study demonstrate the importance of features that characterize the behavioral impact of changes in the context of JIT defect prediction. However, we believe that this paper also lays the foundation for future work that applies these behavioral features to other change-based prediction settings in software engineering, such as build outcome prediction [14] and code reviewer recommendation [15].

II. Preliminaries

In this section, we first describe both conventional and state-of-the-art features for defect prediction, as well as their limitations. We then introduce features used to capture the dynamic characteristics of defect-inducing changes. Finally, we provide a motivating example to showcase and differentiate our proposed (dynamic) behavioral change features from existing features.

A. Related Work

Static Features for Defect Prediction. Defect prediction has been a subject of study for several decades, resulting in the proposal of various types of features [4], [5], [16]–[20]. For example, Hassan [16] introduced a novel code complexity metric feature by measuring the entropy of code changes. The empirical evaluation with six software systems demonstrated that the proposed feature outperformed traditional code metrics as a predictor. Rahman and Devanbu [21] explored the usefulness of diverse process metrics in comparison to code metrics, investigating how and why they can be better. Kamei *et al.* [4] categorized 14 change features from previous research on JIT defect prediction into five dimensions of source control repository data. The authors conducted a large-scale empirical study on six open-source software (OSS) projects and five commercial projects, examining the effectiveness of various change features. Given a plethora of features available, previous studies delved into their significance in predicting defects [19], [22]. These investigations highlighted the importance of size features, such as the number of added lines, in the effectiveness of JIT defect prediction: *the larger the change, the greater the defect proneness.*

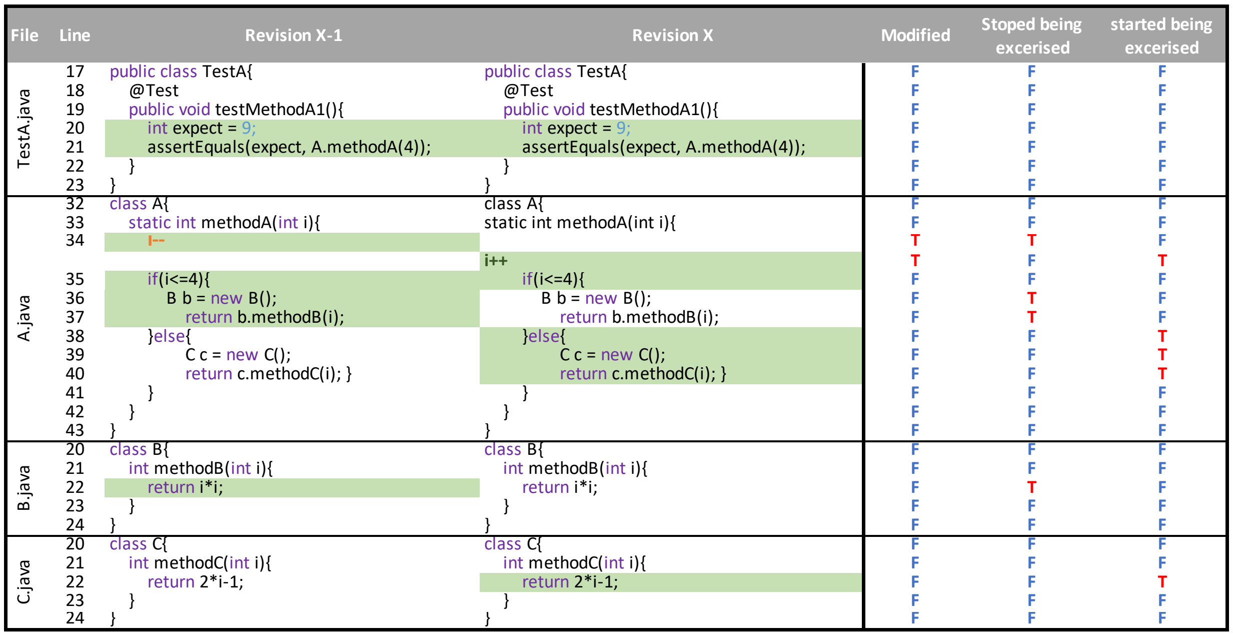
Beyond leveraging static features derived from metadata of code changes, previous studies have explored the use of semantic features extracted from text data, such as source code, through machine learning and text analysis techniques [2], [23]–[31]. For instance, Kim *et al.* [24] employed a bag-of-words model to extract features from change log messages; the performance of the bag-of-words model was evaluated on 12 OSS projects, achieving an accuracy of 78%. Hoang *et al.* [31] utilized a convolutional neural network to derive features from commit messages and code changes, resulting in the creation of a new model called DeepJIT. An empirical study was conducted to evaluate the performance of DeepJIT on two popular open-source software projects, i.e., OpenStack and QT. The results showed that DeepJIT achieved improvements of 9.51-13.69% in terms of ROC AUC on OpenStack and 10.36-11.02% on QT. These semantic features have demonstrated superior performance compared to the traditional features. However, these semantic features are often costly to generate, requiring abundant text data and model training, and more importantly, handling small code changes still remains a challenge. Inspired by prior research on fault localization [32], Sohn *et al.* [5] leveraged the lexical similarity between code and bug reports to enhance JIT defect prediction. While our approach also draws inspiration from fault localization, we rely on the differences in the execution flows of current test suites made by code changes, whereas the work of Sohn *et al.* uses textual similarity between bug reports and code changes.

Dynamic Features for Defect Prediction. Although static data, such as characteristics of the source code and change sets, has been the primary source for deriving features for JIT prediction, several previous studies have leveraged *dynamic information*, such as test execution results, to extract new features for prediction [33]–[36]. For instance, Herzig [34] introduced new dynamic features that associate test execution with defect proneness, such as the number of test failures when the change is integrated. The experimental results on Windows 8 development demonstrated that these metrics outperformed pre-release defect counts in predicting post-release defects. Bowes *et al.* [33], in contrast, defined five features using mutation testing, with four of them leveraging dynamically produced mutant coverage measurements. They evaluated the effectiveness of these new mutation testing-based features with three large real-world systems and showed that the highest prediction performance can be achieved when using both static and dynamic data.

While these studies exploit to what extent the production code still contains defects (*i.e.,* how well-tested), our study also uses tests but examines how different the current behavior of the product is from the previous behavior.

B. Motivating Example

Figure 1 shows the behavior (*i.e.,* trace logs) of two revisions of code, i.e., before and after a change set has been applied; the green-colored lines indicate the code executed during tests. In the figure, line 34 has been changed from `i--` to `i++`. Despite only one line of code being modified, this change causes many lines of the subsequent code to start/stop being executed. For instance, the `if` condition on line 35 evaluates to `false` after the change set has been applied, and `class C` is invoked instead of `class B`. Such a behavioral change should be discovered by a test suite; however, tests often prioritize examining the outputs for each set of inputs, disregarding the coverage of lines. While unexpected behavioral changes can be detected with a robust set of tests in theory, it remains challenging due to the



File	Line	Revision X-1	Revision X	Modified	Stoped being excerised	started being excerised
TestA.java	17	public class TestA{	public class TestA{	F	F	F
	18	@Test	@Test	F	F	F
	19	public void testMethodA1(){	public void testMethodA1(){	F	F	F
	20	int expect = 9;	int expect = 9;	F	F	F
	21	assertEquals(expect, A.methodA(4));	assertEquals(expect, A.methodA(4));	F	F	F
	22	}	}	F	F	F
	23	}	}	F	F	F
A.java	32	class A{	class A{	F	F	F
	33	static int methodA(int i){	static int methodA(int i){	F	F	F
	34	I--		T	T	F
			i++	T	F	T
	35	if(i<=4){	if(i<=4){	F	F	F
	36	B b = new B();	B b = new B();	F	T	F
	37	return b.methodB(i);	return b.methodB(i);	F	T	F
	38	}else{	}else{	F	F	T
	39	C c = new C();	C c = new C();	F	F	T
	40	return c.methodC(i); }	return c.methodC(i); }	F	F	T
	41	}	}	F	F	F
	42	}	}	F	F	F
	43	}	}	F	F	F
B.java	20	class B{	class B{	F	F	F
	21	int methodB(int i){	int methodB(int i){	F	F	F
	22	return i*i;	return i*i;	F	T	F
	23	}	}	F	F	F
	24	}	}	F	F	F
C.java	20	class C{	class C{	F	F	F
	21	int methodC(int i){	int methodC(int i){	F	F	F
	22	return 2*i-1;	return 2*i-1;	F	F	T
	23	}	}	F	F	F
	24	}	}	F	F	F

Fig. 1: Behavior change after a small change.

practical constraints on SQA investments (e.g., testing time, test maintenance costs) [37].

To aid developers in identifying changes that may contain defects, various Just-In-Time (JIT) defect prediction methods have been proposed. These prediction methods typically rely on static features extracted from source code and its changes, often associating commits with many changes as defect-inducing [19], [22]. As a result, these approaches frequently fail to detect unexpected dynamic behavioral changes made by small modifications that lead to defects. To address this issue, we introduce `TraceJIT`—a novel JIT defect prediction approach that characterizes changes to the behavior of a program. To achieve this, `TraceJIT` defines new dynamic features that quantify differences in the execution statuses of lines (e.g., the number of lines that started/stopped being executed after the change). The intuition behind these features is to detect changes that substantially impact the dynamic behavior of code, which we suspect should be riskier than other changes. In this study, we investigate the differences between dynamic features and static features of source code changes (RQ1) and assess whether dynamic features from trace logs improve the traditional JIT models (RQ2).

III. TraceJIT

This section introduces the concept of a spectrum-based defect prediction model (`TraceJIT`).

A. Concept

`TraceJIT` monitors the behavior of the product, exploiting trace logs. Figure 2 outlines the overview of the model-building process. The steps of this process are documented as follows.

1) ***Test exercises.*** `TraceJIT` runs two versions of test suites (*i.e.*, a revision c_i and the previous revision c_{i-1}) to compare the behavior of the products during test exercises before and after a change.
2) ***Trace collection.*** During test exercises, `TraceJIT` executes a dynamic analysis tool to measure dynamic features so that it can identify the behavior change (*trace diff*). Dynamic analysis tools can record various runtime information while programs are running. For example, the tools can measure which lines of code are exercised, what values are returned by each method, and what values are assigned to each variable.
3) ***Build defect prediction model.*** `TraceJIT` detects abnormal behavior changes before and after a change by analyzing the collected trace diff. We extract numerical values from the trace diff and define dynamic features. In addition, we utilize traditional static features to construct `TraceJIT`.

We are the first to explore the dynamic/behavioral characteristics of changes in terms of defect proneness. In line with the previous finding that the size of changes is one of the most effective features in JIT defect prediction, we investigate the impact of the behavioral change size among various dynamic change features. Specifically, we train a JIT defect prediction model using supervised learning with newly proposed behavioral change features, studying their differences with existing features and their effectiveness in defect prediction. Future studies will investigate more diverse (dynamic) features of behavior changes.

B. Features

In this study, we prepare seven simple and new features extracted from trace logs, hereafter referred to as *trace features*.

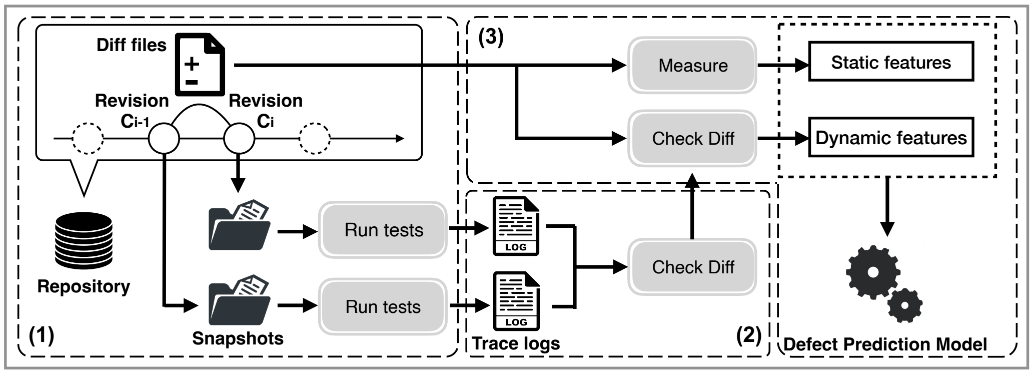


Fig. 2: Overview of `TraceJIT`. It runs tests contained in two revisions (*i.e.,* before and after a change). During testing, `TraceJIT` runs a dynamic analysis tool and collects trace logs. Finally, `TraceJIT` measures the dynamic features by checking diffs between two trace logs, and between diffs of trace logs and changes in source code.

In essence, the trace features in this study analyze which lines of the production code are covered by test suites: *a significant change in the executed lines after applying a change may indicate potential defects in the change.*

Newly exercised or un-exercised lines could be intended and thereby expected by developers. For example, when developers add or drop a substantial number of lines of code, the behavior of the product will likely change dynamically. At the same time, the changes in covered lines may sometimes be unforeseen. For instance, when a developer modifies a class, the change can unexpectedly trigger the execution of another class that underwent no changes, which is reported by several change impact analysis studies [38], [39]. In this study, we aim to differentiate the expected and the unexpected changes when evaluating the behavior changes made by code changes. Hence, we (*i.e.,* `TraceJIT`) further consider which lines of code are modified (*i.e.,* source code changes) when defining the trace features of behavior changes.

In this study, we consider two sets of changes: SC and DC. SC represents static changes in source code, i.e., the modified lines of source code in a change (commit). DC represents (dynamic) changes in the product's behavior, i.e., the lines of source code where a test method started or stopped being exercised after a change. DC is further split into two sets, distinguishing those started (*i.e.,* DC_{start}) and those stopped to be executed (*i.e.,* DC_{stop}).

In the following, we present features measured at two different levels of granularity (*i.e.,* coarse-grained and fine-grained), depending on the extent to which we differentiate dynamic changes (DC), with an example illustrated in Figure 1. Coarse-grained features do not distinguish whether the lines of source code ceased or started to be exercised (*i.e.,* DC), whereas fine-grained features do (*i.e.,* DC_{start} or DC_{stop}).

Coarse-grained dynamic features are computed from the number of traces that appeared or disappeared after a change. These features are designed to determine if the changed traces are directly intended by the source code changes (*i.e.,* whether the changed traces are on the changed lines of the source code or not (*indirect*)), except CT_{all} that involves both direct and indirect behavior changes. The details of the features are described as follows.

1) *# Changed Traces (CT_{all}):* This feature counts the number of lines that are exercised by test suites, quantifying the changes in the product's behavior (*i.e.,* started or stopped being executed) after a change. It can be thereby formulated as "$DC_{start} \cup DC_{stop}$". In Figure 1, all the lines where either column "stopped being exercised" or "started being exercised" is true are marked as CT_{all}.
2) *# Changed Traces on Modified lines (CT_m):* This feature is calculated from the lines of source code that started or stopped being exercised by test suites after a commit and that the commit modified/added/deleted. This feature will be correlated with the changes in source code because a larger size of the production or test code will have more trace logs. This feature can be formulated as "$SC \cap (DC_{start} \cup DC_{stop})$". In Figure 1, all the modified lines where either column "stopped being exercised" or "started being exercised" is true are marked as CT_m (*e.g.,* line 34 and 35 in A.java).
3) *# Changed Traces on Unmodified lines (CT_u):* In contrast to CT_m, this feature represents the lines of source code that started or stopped being exercised by test suites after a commit and that the commit did not modify/add/delete. This feature might be able to detect unexpected behavior changes because the feature is independent of the source code changes. This feature can be formulated as "$\overline{SC} \cap (DC_{start} \cup DC_{stop})$". In Figure 1, all the unmodified lines where either column "stopped being exercised" or "started being exercised" is true are marked as CT_u (*e.g.,* lines 36-40 in A.java).

Fine-grained dynamic features distinguish the lines where test methods started (*i.e.,* Emerged traces) or stopped (*i.e.,* Vanished traces) being exercised by test suites after a change. With the dynamic changes DC_{start}, DC_{stop}, and static changes (*i.e.,* SC), we define the following four features.

4) *# Emerged Traces on Modified lines (ET_m):* This feature represents the number of lines that started being exercised by test suites after a commit and that the commit modified/added/deleted. This feature may detect whether or not the modified lines are well-tested. This feature can be formulated as "$SC \cap DC_{start}$". In Figure 1, all the modified lines where the column "started being exercised" is true are marked as ET_m(*e.g.,* line 34 in A.java in Revision X).
5) *# Vanished Traces on Modified lines (VT_m):* This feature represents the number of lines that stopped being exercised by test suites after a commit and that the commit modified/added/deleted. This feature may find the lines that unexpectedly and wrongly work. This feature can be formulated as $SC \cap DC_{stop}$. In Figure 1, all the modified lines where the column "stopped being exercised" is true are marked as VT_m (*e.g.,* line 34 in A.java in Revision X-1).
6) *# Emerged Traces on Unmodified lines (ET_u):* This feature represents the number of lines that started being exercised by test suites after a commit and that the commit did not modify/add/delete. This feature might capture the lines that unexpectedly started working. This feature can be formulated as "$\overline{SC} \cap DC_{start}$". In Figure 1, all the unmodified lines where the column "started being exercised" is true are marked as ET_u (*e.g.,* lines 38-40 in A.java).
7) *# Vanished Traces on Unmodified lines (VT_u):* This feature represents the number of lines that stopped being exercised by test suites after a commit and that the commit did not modify/add/delete. This feature may find the lines that unexpectedly become unused code, which developers fail to notice. This feature can be formulated as $\overline{SC} \cap DC_{stop}$. In Figure 1, all the unmodified lines where the column "stopped being exercised" is true are marked as VT_u (*e.g.,* line 36 and 37 in A.java).

IV. Experimental Setup

The goal of our study is to investigate the impact of trace logs on defect prediction. In this section, we motivate two research questions and explain why we selected Apache Commons Lang and Math for our study. Then, we describe the details of the evaluation settings.

A. Research Questions

We formulate two research questions to evaluate the utility of trace features when predicting defective commits.

RQ_1**: Do trace features capture distinct and useful features of fix-inducing commits?**

This study introduces the trace features, which monitor the behavior of products during test exercises. In order to make use of the dynamic features in defect prediction, the proposed features need to capture distinct and useful properties of defective and clean commits. As the previous study does [5], RQ1 investigates whether the proposed feature (1) is independent of traditional features and (2) is able to identify clean/defective commits.

RQ_2**: Can the use of trace features improve the performance of defect prediction models?**

To explore the effectiveness of the trace features, RQ2 involves a comparison between `TraceJIT` and a state-of-the-art approach, which shares similarities with `TraceJIT` in being inspired by fault localization but exploits different characteristics [5]. Specifically, we build our model using the base features and the trace features in addition to the state-of-the-art model with the same base features but now with bug report features instead of trace features; we also compare `TraceJIT` with the prediction model trained only with the base features. To further demonstrate the usefulness of the trace features, RQ2 examines how important the trace features are within the features used in `TraceJIT`.

B. Subjects

This study evaluates the performance of JIT defect prediction using the datasets from Apache Commons Lang [40] and Math [41]. Apache Commons Lang and Math projects have a significant number of commits to be analyzed and have been employed by many studies [42]–[44]; the state-of-the-art defect prediction model [5] has been evaluated with these projects. Note that Google Closure Compiler was also used in the previous study [5] to evaluate models, but we could not use it due to the extensive amount of time required by the experiment. For example, `TraceJIT` runs test suites while using a dynamic analysis tool, which requires a vast amount of time to run all the commits. In fact, `TraceJIT` runs tests (including the collection of trace logs) and stored trace logs, which took 278 seconds and 9 hours per commit for the Lang project, respectively. For the Math project, 4,453 seconds to run tests and 51 hours to store traces per commit. The reason why the data store took too much time was that more than 300 containers ran on our cluster and many of them stored the data at the same time.

It is worth noting that the data store is required for only our experiment so that we can debug models and replicate results. In practical use, it is unnecessary to store the data in databases because `TraceJIT` uses trace logs directly. Therefore, compared with the traditional bug prediction models, the additional cost of the time for using `TraceJIT` would be required by testing (*i.e.,* 278 seconds in Lang and 4,453 seconds in Math). More importantly, many modern software development projects adopt continuous integration practices and test their products [45], thus, the actual cost can be considered only the use of the dynamic trace tool.

C. Data Collection

This subsection introduces the procedure for preparing repositories, measuring static features, and dynamic features, as well as building models.

Step 1. Repository Preparation: We clone the repositories of Apache Commons Math and Lang on GitHub, check out each revision on the master/main branch, and extract

TABLE I: Dataset Summary

Project	# commits	# clean commits	# defective commits	Lines of trace logs (Millions)	Studied period	Median test methods	Median passed test methods	Median failed test methods
Lang	1,836	1,782	54	2,182	2007/05 - 2022/06	3,011	2,955	2
Math	3,537	3,337	200	16,148	2008/05 - 2022/06	2,246	2,243	1

TABLE II: 23 Features used in `TraceJIT`

		Feature	Description
Size		Lines Added	# Added lines in a commit
		Lines Deleted	# Deleted lines in a commit
Diffusion		Subsystem	# Modified subsystems in a commit
		Directory	# Modified directory in a commit
		File	# Modified files in as commit
		Entropy	The spread of modified lines across files in a commit
History		Changes	# Changes made to the modified files in the past
		Developers	# Developers who have changed the modified files in a commit in the past
		Age	The time interval to the last changes on the modified files
Experience		Prior changes	# Prior changes to the modified files the authors participated
		Recent changes	# Prior changes to the modified files that the authors participated in weighted by the time interval between changes
		Subsystem changes	# Prior changes to the modified directories the authors participated
		Awareness	The fraction of prior changes to the modified directories that the authors participated
Bug report		$sim2r_{sum}$	The sum of similarities between code changes in a commit and recent bug reports
		$sim2r_{max}$	The maximum of similarities between code changes in a commit and recent bug reports
		$sim2r_{mean}$	The arithmetic mean of similarities between code changes in a commit and recent bug reports
Trace	(Course-grained)	CT_{all}	# Changed trace logs (*i.e.,* CT_m+ CT_u)
		CT_m	# Changed Traces on Modified lines (*i.e.,* ET_m+ VT_m)
		CT_u	# Changed Traces on Unmodified lines (*i.e.,* ET_u+ VT_u)
	(Fine-grained)	ET_m	# Emerged Traces on Modified lines
		VT_m	# Vanished Traces on Modified lines
		ET_u	# Emerged Traces on Unmodified lines
		VT_u	# Vanished Traces on Unmodified lines

features from each commit. When measuring features, merging commits are excluded to prevent generating duplicated results (*i.e.,* the commits that have more than one parent commits). Also, trace features require the difference in trace logs between a target revision and the parent revision. Thus, we use only the target and parent revision that can be tested. In addition, we exclude revisions without additional lines from the dataset because revisions without additional lines are not detected as defects in the SZZ approach described in Section IV. We excluded 3,746 revisions in Lang and 2,940 revisions in Math. Table I shows the statistics of our datasets after filtering.

Step 2. Static feature Extraction: This study collects both base and state-of-the-art static features. We have prepared the 13 *base features* and classified them into four groups: Size, Diffusion, History, and Experience in Table II. These features have been widely employed by previous studies [4], [19], [46].

Three state-of-the-art features from a previous study [5] are measured by calculating the similarity between bug reports and code. These features are made under the assumption that if a commit modifies suspicious parts of the code, the commit is more likely to introduce defects. They found that F1 score and balanced accuracy are improved by 4.2% to 92.2% and by 1.2% to 3.7%. These features are measured with a *time window* parameter. This is the time period during which the bug report is consulted to calculate the features. In this paper, we use the same candidate values as the prior study (30, 60, 90, and 120 days) [5] and use the best value in the experiment. These features are categorized as Bug reports in Table II and hereafter referred to as *report features*.

Step 3. Dynamic feature Extraction: We extract features from trace logs during test exercises, generated by dynamic analysis. Dynamic analysis tools can analyze the order of the product code executed by each test method. To collect trace logs, this study uses the state-of-the-art dynamic analysis tool, SELogger [47]. This tool can save time and computer resources by omitting the records of redundant paths such as loops while dynamic analysis is notorious for being time-consuming and for excessive consumption of memory and disk usage. Also, we use the framework employed in a previous study [48] to run SElogger and test suites parallelly. This tool allows us to deploy numerous containers on our hyper computing clusters and Kubernetes[1] clusters.

When exercising test methods, in the same manner as a previous study [48], we first identify test methods by finding "`@Test`" and run each test method in order to prevent occurring `OutOfMemoryError`. We do not run test methods that are added or modified in a commit in order to eliminate the impact on trace logs. Also, when test methods fail in either of the revisions (before or after a change), we exclude the test methods. The reasons are that (1) we would like to evaluate methods with defects that cannot be identified by tests, and (2) using test failures (*i.e.,* obvious defects) in only `TraceJIT` is not a fair comparison with other methods. In addition, we excluded tests that are likely to have randomness to reduce the chance of generating unstable performance of defect prediction. Specifically, tests with the string "Random" in the test name are excluded (e.g., RandomStringUtilsTest.java).

[1]https://kubernetes.io

Step 4. Label Creation: We need to label each commit, as defective or clean, in order to build prediction models. To identify defective commits, this study uses the SZZ approach [49]. The SZZ approach can identify the commits that induced defects (*i.e.,* fix-inducing commits) by using git-blame on defect-fixing commits. In other words, the approach finds the commits that modified the same lines as those modified by defect-fixing commits. We employ an open-source implementation of the SZZ approach [50]. [2]

D. Data Analysis (RQ1)

We first performed a Spearman correlation analysis on the trace features and the other features to see the degree to which the trace features are distinct. Spearman correlation analysis allows us to evaluate the correlation between two rank variables. A correlation coefficient value below 0.35 indicates a weak correlation, between 0.36 and 0.67 a moderate correlation, and above 0.68 a strong correlation [52]. A lower correlation suggests more distinctness of the trace feature from the other features.

Next, we performed a Mann-Whitney U-test on the trace features to verify whether they can identify defective commits and clean commits ($\alpha = 0.05$). Specifically, we test the following hypothesis:

H_0*: There is no difference in the magnitude of dynamic features between the two groups of defective commits and clean commits.*

H_a*: Defective commits have a higher/lower magnitude of dynamic features than clean commits.*

E. Model Evaluation (RQ2)

Models. RQ2 aims to clarify the impact of using dynamic features on defect prediction performance. We build and compare three models, all of which use base features [4], but two of them use one of the following additional features: report features [5] and trace features. In this paper, these models are denoted by $BaseModel$, $FLModel$ (Fault Localization Model), and $TraceJIT$, respectively.

We employ Random Forest (RF) as the prediction algorithm, which has shown satisfactory performance in previous defect prediction studies [5], [33], [53]–[56]. The number of trees is set to 200, which is the default value of scikit-learn.

Training. This study follows the model training and model evaluation process used in the previous study [5]. Specifically, when creating datasets, we chronologically sort the data and equally divide it into five folds (i.e., F_i) so that models built with past data predict future data. We train a model with the first fold (F_i) and predict a label of each commit in the second fold (F_{i+1}). After that, these data (i.e., F_i and F_{i+1}) are merged as training data and used to predict the labels for the subsequent fold. We iterate this until the last fold is used (i.e., four testing datasets from F_{i+1} to F_{i+4} are used for the testing). This cycle, which is considered as one evaluation, is repeated 30 times, and the mean is used in order to eliminate the randomness of RF.

[2]We used a forked repository [51] due to fatal bugs in the original one.

Note that several features (e.g., Lines Added and ET_m) are expected to be correlated, so there is a possibility of multicollinearity in the model. In spite of this, this study uses all the features because the Random Forest algorithm is one of the models that is less susceptible to multicollinearity than other machine learning models [57].

Performance measures. We employ Precision, ROC-AUC, and PR-AUC as performance measures in the same manner as the previous study [5]. Precision represents the accuracy of the positive predictions made by the model. It is the ratio of true positives to the sum of true positives and false positives. ROC-AUC is simply obtained from the area under the ROC curve. The ROC curve is plotted with the false-positive rate as the x-axis and the true positive rate as the y-axis for each classification threshold in descending order of positive probability. PR-AUC is calculated like the ROC-AUC but utilizes the Precision-Recall curve instead of the ROC curve. The Precision-Recall curve is plotted with the recall on the x-axis and the precision on the y-axis for each classification threshold in descending order of positive probability.

V. Results

This section presents the results of our empirical evaluation with respect to our research questions.

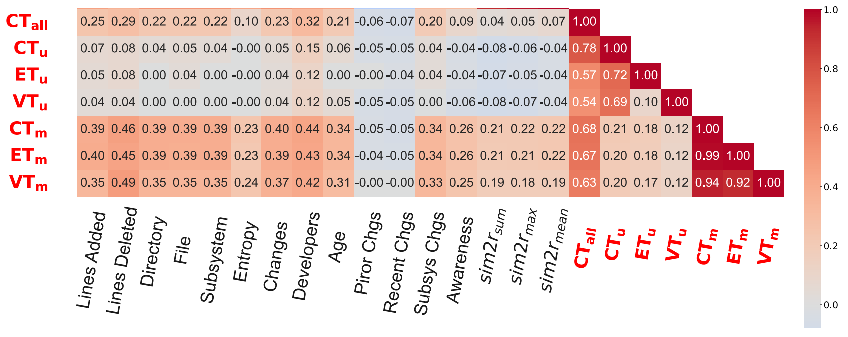
RQ_1*: Do trace features capture distinct and useful features of fix-inducing commits?*

Finding 1: Dynamic features that characterized unchanged lines are independent of static features. Figure 3 shows the Spearman correlation coefficients between the seven trace features and the other 16 features using heatmaps. Cells with lighter shades show weaker correlations and those with darker shades represent stronger correlations.

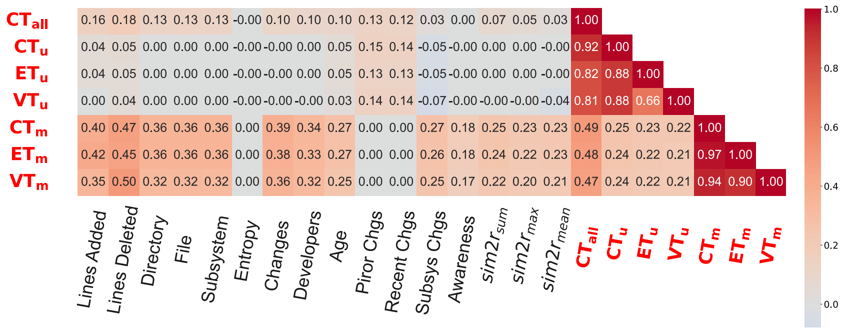
There are correlations between dynamic features regarding modified lines (*i.e.,* CT_m, ET_m, and VT_m) and existing features from source code changes (e.g., Lines Added, Deleted, Directory). In particular, strong correlations (*i.e.,* 0.35 to 0.5) are observed between these dynamic features and $LinesAdded$ as well as $LinesDeleted$. This is not unexpected because CT_m, ET_m, and VT_m are computed by counting dynamically and statically changed lines.

On the other hand, the three dynamic features that characterize unmodified lines (*i.e.,* CT_u, ET_u, and VT_u) are weakly or not correlated with the existing features. This indicates that the signal from these dynamic features is independent from the signal provided by static features. We suspect that this signal will be useful for predicting fix-inducing commits that could not be found using static features.

Finding 2: The values of dynamic features are statistically significantly different when clean commits are compared to defective ones. Figure 4 shows the distribution of trace features in clean commits and defective commits. As for CT_{all}, when comparing the median of each feature, we see different results in Lang and Math. While defective commits have a larger median than clean commits in Lang, they have similar medians in Math. However, after applying

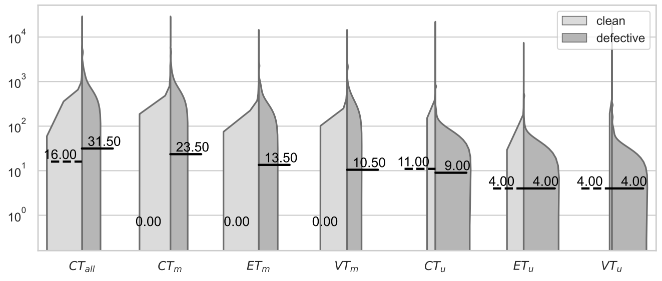


(a) Lang

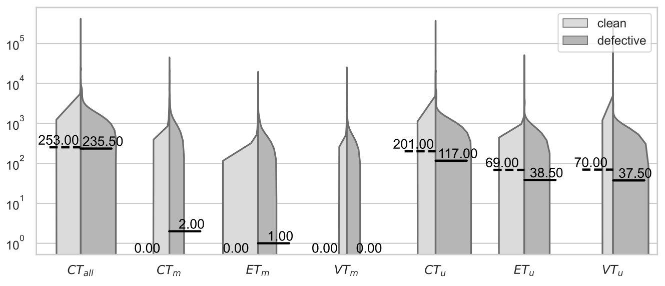


(b) Math

Fig. 3: Heat map of Spearman correlation coefficients between the trace features and the other 18 metrics. Note that the coefficients with more than 0.05 of p-value are replaced with zero.



(a) Lang



(b) Math

Fig. 4: Distribution of dynamic features in defective and clean commits in the Lang and Math projects. The numbers in the figure represent their median.

the statistical test, no statistically significant difference is observed in either project.

As for the trace features on modified lines (*i.e.,* CT_m, ET_m, and VT_m), the medians in defective commits are larger than those in clean commits (except VT_m in Math). After testing four features, the null hypotheses regarding CT_m and ET_m features are rejected, suggesting that behavior changes on modified lines are more likely to be defective. This might be because these features correlate with static change features (i.e., lines added and deleted) that have a strong relationship with fix-inducing commits [4], [7].

Regarding trace features on unmodified lines (*i.e.,* CT_u, ET_u, and VT_u), in Lang, the median of CT_u is larger in the clean commits but no studied project shows a statistically significant difference between these two types of commits. In Math, we observed that clean commits have larger medians than defective ones in terms of CT_u, ET_u, and VT_u. Moreover, hypothesis tests confirm that these differences are statistically significant; however, counter to our expectations, these results suggest that smaller behavior changes are likely defective. The reason for this counter-intuitive result is not yet clear.

RQ_1: Dynamic features on unchanged lines (*i.e.,* CT_u, ET_u, and VT_u) are independent of the existing features but only statistically significant differences between clean and defective commits are observed in the context of the Math project.

TABLE III: The performance of defect prediction models using different features. The percentages in the brackets show the improvements in comparison to the Base Model.

Proj	Model (Metrics)	Base Model (Base)	FL Model (Base+Report)	TraceJIT (Base+Trace)
Lang	ROC-AUC	0.674	0.705 (4.6%)	**0.714 (5.9%)**
	Precision	0.286	0.379 (32.5%)	**0.414 (44.8%)**
	PR-AUC	0.133	**0.134 (0.8%)**	**0.134 (0.8%)**
Math	ROC-AUC	0.762	**0.773 (1.4%)**	0.772 (1.3%)
	Precision	0.213	0.220 (3.3%)	**0.272 (27.7%)**
	PR-AUC	0.149	**0.173 (16.1%)**	0.170 (14.1%)

RQ_2: Can the use of trace features improve the performance of defect prediction models?

Table III shows the performance of three defect prediction models. The values in bold indicate the highest performance among the three models.

Finding 3: `TraceJIT` shows better performance than the FL model in 4 of the 6 cases. Compared with the Base model, the FL and `TraceJIT` show better performance for both Lang and Math across all the performance measures. Specifically, FL and `TraceJIT` have improved 32.5% and 44.8% of Precision for Lang, as well as 16.1% and 14.1% of PR-AUC for Math.

When comparing the FL model and `TraceJIT`, for Lang, `TraceJIT` outperforms the FL model in terms of ROC-AUC (1.3% improvement) and Precision (9.2% improvement). For Math, `TraceJIT` substantially outperforms the FL model in terms of Precision (23.6% improvement). In terms of ROC-AUC and PR-AUC, the FL model technically outperforms `TraceJIT`; however, the improvements are less than one percentage point (0.001 to 0.003). In particular, in Lang, we found three commits that are falsely labeled as defect commits by the FL model, but are correctly labeled by `TraceJIT`. Interestingly, all the trace features have a value of 0 to 3 in each of the commits. `TraceJIT` thus identified these commits as non-fix-inducing commits, leading to higher precision.

Finding 4: Trace features on modified lines and unmodified lines are used as important features in Lang and Math, respectively. Table IV shows the importance of the variables used in the Random Forrest of `TraceJIT`. The most important three features in Lang are “Line added”, “Line deleted”, and “Unique changes”, in particular, “Line added” is also considered one of the important features in previous studies [19], [22]; CT_{all}, CT_m, VT_m, and ET_m are ranked fourth, sixth, seventh, and eighth, respectively. The trace features regarding unmodified lines are ranked at lower positions (VT_u: 17th, CT_u: 18th, ET_u: 19th).

In contrast to the lower rank of the trace features regarding unmodified lines, in Math, CT_u, ET_u, and VT_u rank first, second, and fifth, respectively. CT_{all} ranks in a higher position (fourth) in Math as well as Lang.

TABLE IV: Feature Importance

	Lang		Math	
Rank	Feature	Importance	Feature	Importance
1	Lines Added	0.111	$\boldsymbol{CT_u}$	0.082
2	Lines Deleted	0.106	$\boldsymbol{ET_u}$	0.073
3	Changes	0.061	Changes	0.072
4	$\boldsymbol{CT_{all}}$	0.060	$\boldsymbol{CT_{all}}$	0.062
5	Age	0.056	$\boldsymbol{VT_u}$	0.061
6	$\boldsymbol{CT_m}$	0.054	Recent Chgs.	0.055
7	$\boldsymbol{VT_m}$	0.053	Priror Chgs.	0.054
8	$\boldsymbol{ET_m}$	0.051	Entropy	0.051
9	Awareness	0.046	Lines Added	0.048
10	File	0.046	Awareness	0.045
11	Prior Chgs.	0.044	Age	0.045
12	Recent Chgs.	0.043	Developers	0.044
13	Subsys Chgs.	0.042	Subsys Chgs.	0.043
14	Directory	0.039	Lise deleted	0.041
15	Subsystem	0.038	File	0.041
16	Developers	0.037	Directory	0.040
17	$\boldsymbol{VT_u}$	0.033	Subsystem	0.039
18	$\boldsymbol{CT_u}$	0.032	$\boldsymbol{CT_m}$	0.037
19	$\boldsymbol{ET_u}$	0.026	$\boldsymbol{ET_m}$	0.036
20	Entropy	0.024	$\boldsymbol{VT_m}$	0.032

Overall, while there are different trends in Lang and Math, the dynamic features are considered important features.

RQ_2: `TraceJIT` shows better or equal performance to the FL model in most cases, especially 9.2% and 23.6% improvement of Precision in comparison to the state-of-the-art model (*i.e.,* FL model) in Lang and Math, respectively. In addition, the trace features regarding modified and unmodified lines are selected as important features in Lang and Math, respectively.

VI. Can TraceJIT Detect Regressions?

Regression defects are the defects that previously worked but are now broken due to unexpected behavior changes. One of the main strengths of trace features used in `TraceJIT` is their ability to detect unexpected behavior resulting from changes, which may include defects, by focusing on behavior changes instead of code changes. Hence, `TraceJIT` is inherently more adept at identifying such regression defects caused by unexpected behavioral changes. This study has assessed the performance of `TraceJIT` without differentiating defect types. In this section, we further delve into the potential of `TraceJIT` in predicting regression defects. For this, we identify the commits containing only changes to the existing functions (*i.e.,* methods) to exclude commits introducing new features such as new APIs. Specifically, we separate commits

TABLE V: The effectiveness of trace features

	Test data	MMCs			non-MMCs		
		Base Model	FL Model	`TraceJIT`	Base Model	FL Model	`TraceJIT`
Lang	ROC-AUC	0.665	0.663 (–0.3%)	**0.723** (**8.7%**)	0.610	0.625 (2.5.%)	**0.647** (**6.1%**)
	Precision	0.129	0.157 (21.7%)	**0.414** (**220.9%**)	0.667	**0.833** (**24.9%**)	0.000 (–)
	PR-AUC	0.151	0.151 (0.0%)	**0.169** (**11.9%**)	0.155	**0.156** (**0.6%**)	0.135 (–12.9%)
Math	ROC-AUC	0.686	**0.729** (**6.3%**)	**0.729** (**6.3%**)	0.666	**0.679** (**2.0%**)	**0.679** (**2.0%**)
	Precision	0.191	0.312 (63.4%)	**0.381** (**99.5%**)	0.225	0.208 (–7.6%)	**0.263** (**16.9%**)
	PR-AUC	0.071	**0.079** (**11.3%**)	0.074 (4.2%)	0.179	**0.206** (**15.1%**)	0.199 (11.2%)

depending on whether or not the commits contain only modifications to existing methods. These commits are referred to as *method modification commits (MMCs)*, and the others are as non-MMCs.

To identify MMCs, we employ an Abstract Structure Tree to obtain a list of methods in each commit contained in the testing dataset. We compare the lists before and after a change, and then we find the methods that exist only after the change. By doing so, we separate the testing datasets into two testing datasets including 1,060 MMCs and 408 non-MMCs in Lang, as well as 1,620 MMCs and 1,209 non-MMCs in Math. After labeling each commit in the testing datasets with MMC or non-MMC, we recalculate the results for MMCs and non-MMCs, using the performance measures used in RQ2.

Finding 5: `TraceJIT` shows the best performance for the MMC dataset in most cases while not showing such good performance for the non-MMC dataset. Table V shows the performance for the MMCs and non-MMCs datasets. Comparing the performance of predicting defective MMCs between the three models, we observe that `TraceJIT` outperforms other models across most of the performance measures in both projects. The FL model performs slightly worse than the Base Model in terms of ROC-AUC in Lang.

On the other hand, for non-MMC, `TraceJIT` shows the best performance in terms of ROC-AUC in Lang and Math as well as Precision in Math. However, the FL model shows better performance than `TraceJIT` in some measures (*i.e.,* ROC-AUC in Math, Precision in Lang, PR-AUC in both projects).

Overall, `TraceJIT` performs better for predicting defects in the MMC dataset, implying that `TraceJIT` is likely to be more suited to predict regression defects. Our future work is planning to build and evaluate `TraceJIT` using the dataset containing only MMC or regression defect datasets [58].

`TraceJIT` performed better for predicting defects contained in changes to the existing methods, suggesting that it may be suited for detecting regressions.

VII. Threats to Validity

Internal Validity. Internal validity threats concern factors internal to our study that could influence our results. The number of defective commits in this study is relatively small, which are 54 and 200 in the Lang and Math projects, respectively. The performance difference made by the models might be impacted by a few correct predictions. To mitigate this impact, we iterated evaluations 30 times and applied statistical tests.

Construct Threats. Construct validity threats concern the relationship between theory and observation. Many studies [59]–[61] reported that there are a certain number of flaky tests, which may produce unstable results. This would impact the performance of defect prediction because the lines exercised by test methods vary. Thus, future studies need to eliminate the tests that are likely to be flaky tests.

External Threats. External validity threats concern the generalizability of our findings. We evaluated `TraceJIT` with only two projects that are selected from the previous study [5]. The number of projects is comparable with that of similar previous studies [22], [31] but still needs to be increased to generalize our findings. In particular, the studied projects have numerous test methods, which might lead to clear results. Future work should investigate the impact of the test suite size and examine the effectiveness of automated tests (*e.g.,* EvoSuite [12]) to deal with the impact.

VIII. Conclusions

This paper is the first to explore the impact of behavioral code change on JIT prediction. Specifically, we proposed seven dynamic features that capture the difference in product behavior before and after applying a change. These features are computed using trace logs that are collected during invocations of test suites. Using these logs, we identified which lines of code started/stopped being exercised after a change. We evaluated these features by conducting an empirical study of two large and thriving open-source projects. We observed that, compared to baseline models that use traditional features, adding our proposed set of behavior features leads to improvements of up to 5.9% of ROC-AUC, 44.8% of precision, and 14.1% of PR-AUC. This paper not only demonstrated the importance of behavioral features for JIT defect prediction, but also laid the foundation for future work on behavioral features in other software engineering contexts, such as build outcome prediction and code reviewer recommendation.

Replication package: Our replication package is available on Zenodo [62].

References

[1] N. E. Fenton and M. Neil, "A critique of software defect prediction models," *IEEE Transactions on Software Engineering (TSE)*, vol. 25, no. 5, pp. 675–689, 1999.

[2] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2016, pp. 297–308.

[3] T. Zimmermann, N. Nagappan, H. C. Gall, E. Giger, and B. Murphy, "Cross-project defect prediction: a large scale experiment on data vs. domain vs. process," in *Proceedings of the joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, 2009, pp. 91–100.

[4] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," *IEEE Transactions on Software Engineering (TSE)*, vol. 39, no. 6, pp. 757–773, 2013.

[5] J. Sohn, Y. Kamei, S. McIntosh, and S. Yoo, "Leveraging fault localisation to enhance defect prediction," in *Proceedings of IEEE International Conference on Software Analysis, Evolution, and Reengineering, (SANER)*, 2021, pp. 284–294.

[6] Z. Zeng, Y. Zhang, H. Zhang, and L. Zhang, "Deep just-in-time defect prediction: how far are we?" in *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2021, pp. 427–438.

[7] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2008, pp. 181–190.

[8] R. Karampatsis and C. Sutton, "How often do single-statement bugs occur?: The manysstubs4j dataset," in *Proceedings of the IEEE/ACM International Conference on Mining Software Repositories (MSR)*, 2020, pp. 573–577.

[9] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2005, pp. 402–411.

[10] A. V. Kamienski, L. Palechor, C. Bezemer, and A. Hindle, "Pysstubs: Characterizing single-statement bugs in popular open-source python projects," in *Proceedings of the IEEE/ACM International Conference on Mining Software Repositories (MSR)*, 2021, pp. 520–524.

[11] B. Mosolygó, N. Vándor, G. Antal, and P. Hegedüs, "On the rise and fall of simple stupid bugs: a life-cycle analysis of sstubs," in *Proceedings of the IEEE/ACM International Conference on Mining Software Repositories (MSR)*, 2021, pp. 495–499.

[12] G. Fraser and A. Arcuri, "EvoSuite: automatic test suite generation for object-oriented software," in *Proceedings of the joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, 2011, pp. 416–419.

[13] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri, "Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (T)," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2015, pp. 201–211.

[14] B. Chen, L. Chen, C. Zhang, and X. Peng, "BUILDFAST: history-aware build outcome prediction for fast feedback and reduced cost in continuous integration," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020, pp. 42–53.

[15] M. Chouchen, A. Ouni, M. W. Mkaouer, R. G. Kula, and K. Inoue, "WhoReview: A multi-objective search-based approach for code reviewers recommendation in modern code review," *Applied Soft Computing (ASOC)*, vol. 100, p. 106908, 2021.

[16] A. E. Hassan, "Predicting faults using the complexity of code changes," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2009, pp. 78–88.

[17] V. R. Basili, L. C. Briand, and W. L. Melo, "A validation of object-oriented design metrics as quality indicators," *IEEE Transactions on Software Engineering (TSE)*, vol. 22, no. 10, pp. 751–761, 1996.

[18] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering (TSE)*, vol. 20, no. 6, pp. 476–493, 1994.

[19] M. Kondo, D. M. German, O. Mizuno, and E.-H. Choi, "The impact of context metrics on just-in-time defect prediction," *Empirical Software Engineering (EMSE)*, vol. 25, no. 1, pp. 890–939, 2020.

[20] A. Hindle, M. W. Godfrey, and R. C. Holt, "Reading beside the lines: Indentation as a proxy for complexity metric," in *Proceedings of the International Conference on Program Comprehension (ICPC)*, 2008, pp. 133–142.

[21] F. Rahman and P. T. Devanbu, "How, and why, process metrics are better," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2013, pp. 432–441.

[22] S. McIntosh and Y. Kamei, "Are fix-inducing changes a moving target? A longitudinal case study of just-in-time defect prediction," *IEEE Transactions on Software Engineering (TSE)*, vol. 44, no. 5, pp. 412–428, 2018.

[23] T. Jiang, L. Tan, and S. Kim, "Personalized defect prediction," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2013, pp. 279–289.

[24] S. Kim, E. J. Whitehead Jr, and Y. Zhang, "Classifying software changes: Clean or buggy?" *IEEE Transactions on Software Engineering (TSE)*, vol. 34, no. 2, pp. 181–196, 2008.

[25] O. Mizuno and T. Kikuno, "Training on errors experiment to detect fault-prone software modules by spam filter," in *Proceedings of the joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, 2007, pp. 405–414.

[26] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu, "On the "naturalness" of buggy code," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2016, pp. 428–439.

[27] J. Li, P. He, J. Zhu, and M. R. Lyu, "Software defect prediction via convolutional neural network," in *Proceedings of the Software Quality, Reliability and Security (QRS)*, 2017, pp. 318–328.

[28] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun, "Deep learning for just-in-time defect prediction," in *Proceedings of the Software Quality, Reliability and Security (QRS)*, 2015, pp. 17–26.

[29] Y. Li, S. Wang, T. N. Nguyen, and S. Van Nguyen, "Improving bug detection via context-based code representation learning and attention-based neural networks," *Proceedings of the ACM on Programming Languages (PACMPL)*, vol. 3, no. OOPSLA, pp. 162:1–162:30, 2019.

[30] C. Pornprasit and C. Tantithamthavorn, "Deeplinedp: Towards a deep learning approach for line-level defect prediction," *IEEE Transactions on Software Engineering (TSE)*, vol. 49, no. 1, pp. 84–98, 2023.

[31] T. Hoang, H. K. Dam, Y. Kamei, D. Lo, and N. Ubayashi, "Deepjit: an end-to-end deep learning framework for just-in-time defect prediction," in *Proceedings of the IEEE/ACM International Conference on Mining Software Repositories (MSR)*, 2019, pp. 34–45.

[32] M. J. Harrold, G. Rothermel, K. Sayre, R. Wu, and L. Yi, "An empirical investigation of the relationship between spectra differences and regression faults," *Software Testing, Verification and Reliability (STVR)*, vol. 10, no. 3, pp. 171–194, 2000.

[33] D. Bowes, T. Hall, M. Harman, Y. Jia, F. Sarro, and F. Wu, "Mutation-aware fault prediction," in *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2016, pp. 330–341.

[34] K. Herzig, "Using pre-release test failures to build early post-release defect prediction models," in *Proceedings of the IEEE International Symposium on Software Reliability Engineering (ISSRE)*, 2014, pp. 300–311.

[35] F. Elberzhager, S. Kremer, J. Münch, and D. Assmann, "Guiding testing activities by predicting defect-prone parts using product and inspection metrics," in *Proceedings of the Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2012, pp. 406–413.

[36] A. Amar and P. C. Rigby, "Mining historical test logs to predict bugs and localize faults in the test logs," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2019, pp. 140–151.

[37] V. Garousi, M. Felderer, M. Kuhrmann, K. Herkiloglu, and S. Eldh, "Exploring the industry's challenges in software testing: An empirical study," *Journal of Software: Evolution and Process*, vol. 32, no. 8, 2020.

[38] A. Orso, T. Apiwattanapong, J. Law, G. Rothermel, and M. J. Harrold, "An empirical comparison of dynamic impact analysis algorithms," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2004, pp. 491–500.

[39] T. Apiwattanapong, A. Orso, and M. J. Harrold, "Efficient and precise dynamic impact analysis using execute-after sequences," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2005, pp. 432–441.

[40] Apache, "Commons Lang," https://github.com/apache/commons-lang.

[41] ——, “Commons Math,” https://github.com/apache/commons-math.

[42] F. Palma, T. Abdou, A. Bener, J. Maidens, and S. Liu, “An improvement to test case failure prediction in the context of test case prioritization,” in *Proceedings of the International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE)*, 2018, pp. 80–89.

[43] K. E. Someoliayi, S. Jalali, M. Mahdieh, and S. Mirian-Hosseinabadi, “Program state coverage: A test coverage metric based on executed program states,” in *Proceedings of IEEE International Conference on Software Analysis, Evolution, and Reengineering, (SANER)*, 2019, pp. 584–588.

[44] A. Perera, A. Aleti, M. Böhme, and B. Turhan, “Defect prediction guided search-based software testing,” in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2020, pp. 448–460.

[45] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, “Usage, costs, and benefits of continuous integration in open-source projects,” in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2016, pp. 426–437.

[46] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, “An empirical study of the impact of modern code review practices on software quality,” *Empirical Software Engineering (EMSE)*, vol. 21, no. 5, pp. 2146–2189, 2016.

[47] K. Shimari, T. Ishio, T. Kanda, and K. Inoue, “Near-omniscient debugging for java using size-limited execution trace,” in *Proceeding of the IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2019, pp. 398–401.

[48] Y. Kashiwa, K. Shimizu, B. Lin, G. Bavota, M. Lanza, Y. Kamei, and N. Ubayashi, “Does refactoring break tests and to what extent?” in *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2021, pp. 171–182.

[49] J. Sliwerski, T. Zimmermann, and A. Zeller, “When do changes induce fixes?” in *Proceedings of the IEEE/ACM International Conference on Mining Software Repositories (MSR)*, 2005.

[50] V. Lenarduzzi, F. Palomba, D. Taibi, and D. A. Tamburri, “Openszz: A free, open-source, web-accessible implementation of the SZZ algorithm,” in *Proceedings of the International Conference on Program Comprehension (ICPC)*, 2020, pp. 446–450.

[51] VladyslavBondarenko, “OpenSZZ,” https://github.com/VladyslavBondarenko/OpenSZZ.

[52] J. S. Collofello and S. N. Woodfield, “Evaluating the effectiveness of reliability-assurance techniques,” *The Journal of Systems and Software (JSS)*, vol. 9, no. 3, pp. 191–195, 1989.

[53] J. Nam, W. Fu, S. Kim, T. Menzies, and L. Tan, “Heterogeneous defect prediction,” *IEEE Transactions on Software Engineering (TSE)*, vol. 44, no. 9, pp. 874–896, 2018.

[54] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, “Benchmarking classification models for software defect prediction: A proposed framework and novel findings,” *IEEE Transactions on Software Engineering (TSE)*, vol. 34, no. 4, pp. 485–496, 2008.

[55] T. Mende and R. Koschke, “Effort-aware defect prediction models,” in *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, 2010, pp. 107–116.

[56] S. Tabassum, L. L. Minku, D. Feng, G. G. Cabral, and L. Song, “An investigation of cross-project learning in online just-in-time software defect prediction,” in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2020, pp. 554–565.

[57] X. Zhao, B. Yu, Y. Liu, Z. Chen, Q. Li, C. Wang, and J. Wu, “Estimation of poverty using random forest regression with multi-source data: A case study in bangladesh,” *Remote Sensing*, vol. 11, no. 4, p. 375, 2019.

[58] M. Ohira, Y. Kashiwa, Y. Yamatani, H. Yoshiyuki, Y. Maeda, N. Limsettho, K. Fujino, H. Hata, A. Ihara, and K. Matsumoto, “A dataset of high impact bugs: Manually-classified issue reports,” in *Proceedings of the IEEE/ACM International Conference on Mining Software Repositories (MSR)*, 2015, pp. 518–521.

[59] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, “An empirical analysis of flaky tests,” in *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2014, pp. 643–653.

[60] W. Lam, K. Muslu, H. Sajnani, and S. Thummalapenta, “A study on the lifecycle of flaky tests,” in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2020, pp. 1471–1482.

[61] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn, “A survey of flaky tests,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 1, pp. 17:1–17:74, 2022.

[62] double-blind, “Replication package for TraceJIT: Evaluating the Impact of Behavioral Code Change on Just-In-Time Defect Prediction,” https://doi.org/10.5281/zenodo.10061035.