# Boosting Just-in-Time Defect Prediction with Specific Features of C/C++ Programming Languages in Code Changes

Chao Ni[†][§], Xiaodan Xu[†][§], Kaiwen Yang[†] and David Lo[‡]
[†]Zhejiang University, China. Email: {chaoni,xiaodanxu,kwyang}@zju.edu.cn
[‡] Singapore Management University, Singapore. Email: davidlo@smu.edu.sg

***Abstract*—Just-in-time (JIT) defect prediction can identify changes as defect-inducing ones or clean ones and many approaches are proposed based on several programming language-independent change-level features. However, different programming languages have different characteristics and consequently may affect the quality of software projects. Meanwhile, the C programming language, one of the most popular ones, is widely used to develop foundation applications (i.e., operating system, database, compiler, etc.) in IT companies and its change-level characteristics on project quality have not been fully investigated. Additionally, whether open-source C projects have similar important features to commercial projects has not been studied much.**

**To address the aforementioned limitations, in this paper, we investigate the impacts of programming language-specific features on the state-of-the-art JIT defect identification approach in an industrial setting. We collect and label the top-10 most starred C projects (i.e., 329,021 commits) on GitHub and 8 C projects in an ICT company (i.e., 12,983 commits). We also propose nine C-specific change-level features and focus our investigations on both open-source C projects on GitHub and C projects at the ICT company considering three aspects: (1) The effectiveness of C-specific change-level features in improving the performance of identification of defect-inducing changes, (2) The importance of features in the identification of defect-inducing changes between open-source C projects and commercial C projects, and (3) The effectiveness of combining language-independent features and C-specific features in a real-life setting at the ICT company.**

***Index Terms*—Just-in-Time, C/C++ programming language, Supervised Methods**

## I. Introduction

Software defect prediction is an issue of general interest of software engineering. Early researchers proposed various module-level defect prediction approaches to identify defect-prone software identities (e.g., classes, packages and files) [1–9]. With the popularity of continuous delivery and the accelerated pace of software development, researchers proposed a series of change-level defect prediction approaches to predict defective entities at a finer-grained level (i.e., code changes) [10–16]. In recent years, change-level defect prediction attracts more and more interest as it is more suitable for an industrial setting.

Change-level defect prediction can identify a defective change at check-in time, and thus is also referred to as just-in-time (JIT) defect prediction [16]. JIT defect prediction is a binary classification problem, which predicts software changes as defective or not. A defective change is a change that introduces one or more defects and causes software anomalies [17]. Compared with module-level defect prediction, JIT defect prediction shows its benefits in the following aspects: ① **Prediction at a fine-grained level.** JIT defect prediction enables developers to locate defects by only inspecting a small amount of codes within the warned changes. Besides, since each change has its corresponding committer, it is convenient to assign the responsible developer to fix the defect. ② **Prediction at check-in time.** JIT defect prediction can identify changes as defect-inducing ones or clean ones immediately after their submissions to the code repository. Thus, developers can inspect defect-prone changes to locate defects while the contexts of these changes are still fresh in their minds.

In practice, the resources allocated for code inspection and quality assurance are often limited. In order to take into consideration the effort to inspect each change, effort-aware JIT defect prediction was introduced [18, 19]. Researchers proposed various effort-aware JIT defect prediction methods [10–13, 18] by fully utilizing programming language-independent features. These approaches have made significant progress of JIT defect prediction with promising results, and consequently are more welcomed by practitioners, since they aim at finding more defective changes within a given budget of inspection effort.

Existing effort-aware JIT defect prediction approaches can be classified into supervised ones [10, 11, 20] and unsupervised ones [13, 14]. These approaches are proposed based on the 14 change-level features, which are introduced by Kamei et al. [10] and are generalizable to projects developed by any programming language. However, software projects from different application domains are often developed using different programming languages, and the varying characteristics of programming languages may have different impacts on software quality according to a recent study on JavaScript projects [21]. Although the widely used 14 change-level features have been verified their effectiveness of identifying defect-inducing changes [10, 12, 21, 22], they are language-agnostic and may not comprehensively capture the characteristics of software quality.

Defect prediction and quality assurance are crucial aspects of software development projects that utilize the C program-

[§] Equal Contribution. Chao Ni is the corresponding author.

ming language, a fundamental programming language distinguished by its unique characteristics of memory management and pointer usage. C programming language finds extensive use in the development of compilers, operating systems, and various software frameworks. For instance, Openharmony [23], a popular open-source operating system project incubated and operated by the OpenAtom Foundation [24], is developed primarily using C and C++. While C programming provides developers with a high degree of flexibility, the use of memory operations and pointers requires extra caution as improper implementation can result in serious defects. Nevertheless, the impact of change-level characteristics of C programming language on project quality has not been comprehensively investigated.

In this paper, considering the importance of C programming language and the limited study of C change-level characteristics, we want to dig into C programming language characteristics. In particular, we collect the top-10 most popular open-source C software projects on GitHub [25] and 8 commercial projects mainly written in C and C++ in an ICT company[1]. Besides, to investigate C programming language's characteristics, we propose nine C-specific change-level features, and conduct a case study on whether these features can help further improve effort-aware JIT defect prediction model's performance on both open-source projects and commercial projects. We also develop a JIT defect prediction tool based on the best-performing approach (i.e., CBS+ [12]), and evaluate its usefulness when it is deployed in the ICT company through a developer study.

Eventually, our paper makes the following contributions:

1) We collect and label the dataset of 10 open-source C projects with 329,021 changes and 8 commercial projects with 12,983 changes using SZZ algorithm [17]. Through manual analysis, we observe that programming language change-level features have impacts on software quality.
2) We firstly introduce nine C-specific change-level features, and prove that C-specific features can further improve JIT defect prediction approach when considering effort-aware performance measures through a comparative experiment.
3) We examine the importance of change-level features to effort-aware JIT defect prediction on both open-source and commercial projects and compare the differences. We find that commercial projects of the ICT company and open-source projects vary in the most important features. In general, feature groups "Experience", "Size", "Diffusion" and "C-Specific" are important for effort-aware JIT defect prediction in both commercial and open-source C projects.
4) We develop a JIT defect prediction tool based on the best-performing supervised approach (i.e., CBS+ [12]). We apply this tool to certain on-going projects at the ICT company, and conduct a developer study to investigate its effectiveness in an actual industrial setting.

[1]The ICT company we studied is Huawei, a provider of information and communication infrastructure and intelligent terminals.

**Paper Structure.** Section II briefly summarizes the related work on JIT defect prediction. Section III describes the experimental settings, including the collected projects, the studied change-level features, the data pre-processing steps, the selected evaluation measures and the statistical analysis methods. Section IV presents the research questions and analyzes the corresponding results. Section V shows some implications and discussion in our study. Section VI describes the threats to validity of our work. Section VII concludes our findings in this paper and discusses future work.

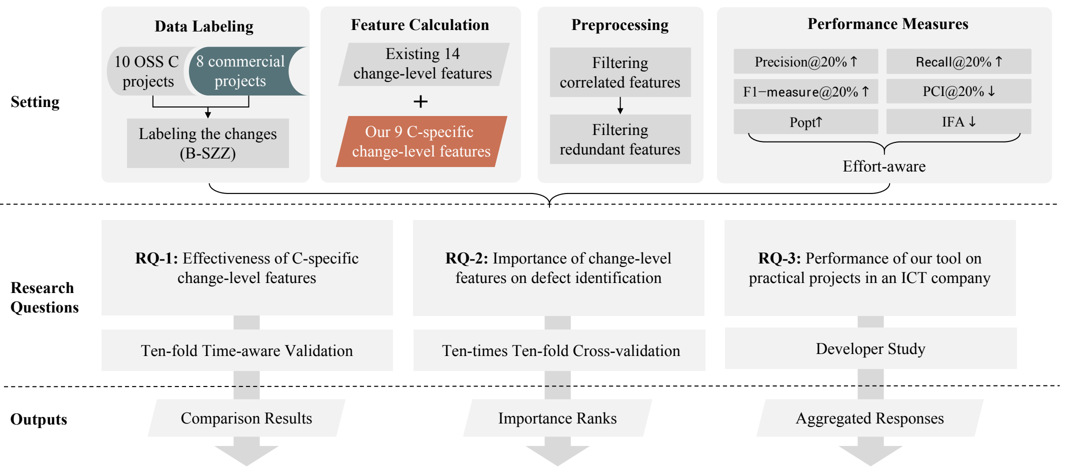
## II. Related Work

**JIT defect prediction.** Mockus and Weiss [26] proposed the first JIT defect prediction method which can predict the risk of Initial Modification Requests (IMRs, i.e., a group of changes) in software projects using change measures. Sliwerski et al. [17] developed an algorithm named SZZ to automatically identify defect-inducing changes, which has greatly improved the efficiency of data labeling and thus has promoted the research on JIT defect prediction. Kim et al. [27] proposed a technique to classify a software change as clean or buggy using complexity metrics and features extracted from change metadata, change log message, source code and file names. Yin et al. [28] investigated the association between defect-fixing changes and defect-inducing changes on certain operating systems. Yang et al. [29, 30] introduced ensemble learning and deep learning techniques to JIT defect prediction. Hoang et al. [31, 32] leveraged advanced deep learning techniques such as deep neural networks (DNNs) to automatically learn feature representations from software changes.

Considering the limitation of resource assigned to code inspection, Kamei et al. [10] proposed the first effort-aware JIT defect prediction method called EALR. They used the number of modified lines to measure the inspection effort required by a change. Huang et al. [18] proposed a supervised model named CBS (i.e., Classify-Before-Sorting) based on the idea that small changes should be inspected first since they are proportionally more defect-prone. Subsequently, Huang et al. [12] proposed an advanced version of CBS named CBS+. According to existing studies [12, 21, 22], CBS+ proposed by Huang et al. [12] statistically performs better than other supervised approaches (i.e., EALR [10] and OneWay [11]) and unsupervised approaches (i.e., LT [13] and Churn [14]).

Therefore, we select CBS+ as the prediction model for our study on commercial and open-source projects. We use effort-aware performance measures to evaluate the effectiveness of CBS+ since inspection resource is limited in practice.

**Change-level features in JIT defect prediction.** Kamei et al. [10] categorized the 14 frequently-used change-level features into five dimensions: diffusion, size, purpose, history, and experience. These features have been widely used in JIT defect prediction studies [10–14, 18]. However, there features are programming language-independent features and do not consider the characteristic of different programming. Recently, Ni et al. [21] conducted an empirical study on 20 JavaScript



Setting
Data Labeling
10 OSS C projects
8 commercial projects
Labeling the changes (B-SZZ)
Feature Calculation
Existing 14 change-level features
+
Our 9 C-specific change-level features
Preprocessing
Filtering correlated features
Filtering redundant features
Performance Measures
Precision@20% ↑
Recall@20% ↑
F1−measure@20% ↑
PCI@20% ↓
Popt↑
IFA ↓
Effort-aware
Research Questions
RQ-1: Effectiveness of C-specific change-level features
RQ-2: Importance of change-level features on defect identification
RQ-3: Performance of our tool on practical projects in an ICT company
Ten-fold Time-aware Validation
Ten-times Ten-fold Cross-validation
Developer Study
Outputs
Comparison Results
Importance Ranks
Aggregated Responses

**Fig. 1:** Overview of this paper

projects on GitHub and proposed 5 JavaScript-specific change-level features. They found that JavaScript-specific features can help improve the performance of prior proposed approach built with the 14 language-independent features. Inspired by Ni et al. [21], we investigate the effectiveness of C-specific change-level features in our work, considering C programming language's unique characteristics in aspect of memory management and pointer operation.

**Empirical Study of JIT defect prediction.** Several existing studies have assessed the effectiveness of JIT defect prediction approaches on real-world software projects. Mockus and Weiss [26] investigated the risk of IMRs of the 5ESS software. Shihab et al. [33] conducted a year-long industrial study and found several important characteristics of defect-prone changes. Kamei et al. [10] performed a large-scale empirical study of effort-aware JIT defect prediction on both open-source and commercial projects. Yan et al. [22] conducted a case study of JIT defect prediction on 14 Alibaba projects. They developed a tool based on CBS+ and assessed its effectiveness through a user study. Ni et al. [21] conduct an empirical study on the top-20 most starred JavaScript projects on Github in their study of JavaScript-specific features.

Our work is inspired but different from the aforementioned studies. Firstly, different from Mockus and Weiss [26] and Shihab et al. [33], we intend to investigate the effectiveness of **effort-aware** JIT defect prediction approaches through an **industrial** study. Second, different from Kamei et al. [10], Yan et al. [22] and Ni et al. [21], we focus on studying whether **C-specific** change-level features can further improve the performance of existing JIT defect prediction approach.

## III. Experimental Setting

This section sequentially introduces the selected appropriate projects, the newly proposed C-specific change-level features, the data preprocessing methods, the evaluation settings, the statistical analysis as well as the studied research questions. The overview of this paper is presented in Fig. 1.

**TABLE I:** Summary of the studied commercial projects in the ICT company.

| Project | # Commit | % C/CPP | # File | # LOC | # Branch | # Developers |
|---|---|---|---|---|---|---|
| P1 | 1,557 | 55.9% | 129 | 17,860 | 9 | 54 |
| P2 | 1,157 | 69.2% | 361 | 65,426 | 10 | 33 |
| P3 | 1,772 | 67.8% | 524 | 222,656 | 29 | 42 |
| P4 | 1,040 | 99.5% | 6,982 | 1,075,598 | 13 | 42 |
| P5 | 2,311 | 99.5% | 7,031 | 1,077,124 | 16 | 54 |
| P6 | 1,709 | 71.7% | 374 | 50,030 | 3 | 36 |
| P7 | 2,245 | 50.9% | 498 | 33,317 | 3 | 32 |
| P8 | 1,192 | 50.9% | 120 | 13,627 | 3 | 25 |

### *A. Data Preparation*

**Project selection.** Our studied projects consist of two parts, namely commercial projects and Open Source Software (OSS) projects.

**Commercial projects**. We study 8 representative C/C++ projects provided by an ICT company, which differ in scales and types. For confidentiality purposes, we hereinafter refer to these projects as P1 to P8. A summary of the these commercial projects is listed in Table I. "% C/CPP" stands for the ratio of C and C++ in all programming languages within the project. We calculate this ratio using a code statistical tool named *cloc* [34].

**OSS projects**. We select the *c* topic on Github Topics and found 42,247 public repositories. We then filter these repositories by *c* programming language, which returns 29,543 public repositories. Finally, we sort the result by *Most stars* to find the most popular projects. We adopt the following inclusion criteria when choosing the most suitable projects: (1) the number of commits in the project should be more than 1,000; (2) the project should conform to the definition of software project. Following these criteria, we choose and clone the top-10 most starred OSS C projects on Github on February 23, 2022. Specifically, the time period of all changes in each selected OSS project starts from its creation time on GitHub and ends by 6:30 pm on February 23, 2022. Statistical

**TABLE II:** Summary of the top-10 most popular C projects on GitHub.

| Project | # Commit | % C | # File | # LOC | # Star | # Fork | # Branch | # Tags | # Developer | # Defective | % Defective |
|---|---|---|---|---|---|---|---|---|---|---|---|
| scrcpy | 2,644 | 73.8% | 240 | 40,520 | 61.6k | 6.6k | 271 | 27 | 95 | 464 | 17.5% |
| git | 68,568 | 49.8% | 4,071 | 1,411,555 | 41.2k | 23k | 6 | 826 | 1,523 | 9,468 | 13.8% |
| obs-studio | 10,620 | 55.0% | 4,495 | 697,120 | 36.4k | 5.6k | 10 | 156 | 477 | 1,815 | 17.1% |
| FFmpeg | 128,364 | 91.7% | 7,653 | 1,977,457 | 28.3k | 9.3k | 32 | 348 | 1,228 | 22,784 | 17.7% |
| curl | 28,336 | 76.7% | 3,430 | 498,038 | 23.8k | 4.9k | 15 | 192 | 803 | 6,488 | 22.9% |
| ss | 2,127 | 81.1% | 96 | 9,969 | 23.3k | 1.3k | 55 | 49 | 216 | 296 | 13.9% |
| mpv | 50,845 | 87.5% | 745 | 244,057 | 17.9k | 2.2k | 74 | 80 | 357 | 14,084 | 27.7% |
| radare2 | 27,490 | 96.8% | 4,139 | 1,430,225 | 15.8k | 2.6k | 47 | 99 | 817 | 15,137 | 55.1% |
| goaccess | 6,164 | 86.5% | 114 | 58,419 | 14.4k | 963 | 2 | 39 | 123 | 555 | 9.0% |
| nnn | 3,863 | 63.1% | 102 | 24,147 | 13.6k | 545 | 1 | 35 | 129 | 889 | 23.0% |

*"ss" refers to project of the_silver_searcher.

**TABLE III:** Summary of the 14 language-agnostic change-level features proposed by Kamei et al. [10]

| Dimension | Feature | Description |
|---|---|---|
| Diffusion | NS | Number of subsystems touched by the current change |
| | ND | Number of directories touched by the current change |
| | NF | Number of files touched by the current change |
| | Entropy | Distribution across the touched files |
| Size | LA | Lines of code added by the current change |
| | LD | Lines of code deleted by the current change |
| | LT | Lines of code in a file before the current change |
| Purpose | FIX | Whether or not the current change is a defect fix |
| History | NDEV | The number of developers that changed the files |
| | AGE | The average time interval (in days) between the last and the change over the files that are touched |
| | NUC | The number of unique last changes to the files |
| Experience | EXP | Developers experience, i.e. the number of changes |
| | REXP | Recent developer experience, i.e. the total experience of the developer in terms of changes, weighted by their age |
| | SEXP | Developer experience on a subsystem, i.e. the number of changes the developer made in the past to the subsystems |

information about these projects can be seen in Table II.

**Data labeling.** Data labeling aims at distinguishing between defect-inducing changes and clean changes. To tackle this problem, Sliwerski et al. [17] developed the initial SZZ algorithm, referred to as B-SZZ, to automatically label historical change of each project as defect-inducing ones or clean ones. Subsequently, variants of the SZZ algorithm (e.g., AG-SZZ [35], MA-SZZ [36], RA-SZZ [37], RA-SZZ*[38], etc) were proposed to resolve the limitations of the original SZZ. However, according to a large-scale experiment which compares the effectiveness of different variants of SZZ algorithms conducted by Rosa et al. [39], B-SZZ performs best in most cases. Therefore, we adopted B-SZZ algorithm in data labeling for efficiency and precision.

## *B. Change-level Features*

Our studied change-level features consist of two sets of features. One group of features are the 14 change-level features proposed by Kamei et al. [10] as used in existing works [10–14, 18, 21, 22]. These features are categorized into five dimensions: diffusion (NS, ND, NF and Entropy), size (LA, LD and LT), purpose (FIX), history (NDEV, AGE and NUC) and experience (EXP, REXP and SEXP). To make the paper self-contained, we summarize the short names and brief descriptions of these features in Table III. More information can be referred to Kamei el al's work [10].

All of the aforementioned 14 change-level features are general and language-independent. However, C programming language possesses unique and non-negligible characteristics in aspect of memory management and pointers. For example, with the help of Dynamic Memory Allocation (DMA), developers can change the size of data structures during runtime by using predefined functions in the C library (i.e., `malloc()`, `calloc()`, `realloc()`, `free()`). Pointer is another useful tool that C programming language provides to help developers interact with the memory directly (i.e., allocating memory, freeing memory, accessing array elements, etc.). DMA and Pointers are so flexible that any improper use of them might bring fatal defects like memory leaks and out-of-memory exceptions. Besides, through our analysis of defective commits in the ICT company, we observe that, in addition to memory management and pointers, the misuses of auto increment or decrement, array indexing and *"goto"* statement are also frequent causes of defects. Therefore, we heuristically propose nine C-specific change-level features (i.e., MemInc, MemDec, MemChg, SinglePointer, MultiplePointer, MaxPointerDepth, Goto, IndexUsed and AutoIncreDecre) in addition to Kamei et al.'s features. Table IV presents a summary of these C-specific change-level features. We illustrate these features in detail as follows.

- **MemInc (MI).** MemInc is short for Memory-used Increment. In C programming, DMA enables developers to request memory according to their needs. We calculate the sum of memory requirement operations included in a change by counting the total number of `malloc()`, `calloc()` and `realloc()` function calls.
- **MemDec (MD).** MemDec is short for Memory-used

**TABLE IV:** Summary of C/C++ specific change-level features

| Feature | Description |
|---|---|
| MemInc | Sum of memory requirement operations by the current change. |
| MemDec | Sum of memory release operations by the current change. |
| MemChg | Sum of "*realloc*" operations by the current change |
| SinglePointer | Sum of pointer definitions or pointer usage operations by the current change (e.g., direct address of a variable, function, array, etc.,) |
| MultiplePointer | Sum of multiple pointer definitions or pointer usage operations by the current change |
| MaxPointerDepth | Max depth of the pointer definition or usage of pointer by the current change |
| Goto | Sum of "*goto*" operations by the current change |
| IndexUsed | Sum of index used in array visiting by the current change |
| AutoIncreDecre | Sum of auto increasing or decreasing operations by the current change |

Decrement. In C programming, it is a good habit to release the requested memory for a dynamic variable by calling `free()` as soon as the variable is no longer used. A memory leak problem might occur if a developer keeps on requesting for memory allocation without releasing any manually. We calculate the sum of memory release operations included in a change by counting the total number of `free()` function calls.

- **MemChg (MC).** MemChg is short for Memory-used Change. In C programming, developers can alter the size of allocated memory space by using `realloc()` whenever their needs for memory space have changed. `realloc()` can either expand or reduce the allocated memory space, and is often used to request a larger space for a variable when the original one is too small. We calculate the sum of memory change operations included in a change by counting the total number of `realloc()` function calls.
- **SinglePointer (SP).** A single pointer stores the value of the direct address of a variable, function, array, etc. We calculate the SinglePointer feature by the number of pointer definition or usage included in a change. For example, `int* ptr1 = &num` defines a single pointer named `ptr1`, and `int var1 = *ptr1` uses the single pointer.
- **MultiplePointer (MP).** Multiple pointer is in effect a pointer indicating to another pointer. Multiple pointer forms a chain of pointers and is used to implement multilevel indirect addressing. In C programming language, operators like `**`, `***` and so on are used to define and dereference multiple pointers. We calculate the MultiplePointer feature by the number of multiple pointer definition and usage included in a change. For example, `int** ptr2 = &ptr1` defines a multiple pointer (i.e., a second rank pointer named `ptr 2`), and `int var2 = **ptr2` uses the multiple pointer.
- **MaxPointerDepth (MPD).** The depth of a pointer can be calculated by simply counting the number of `*` in its definition or usage. For example, `int** ptr` defines a second rank pointer, and its depth is 2, while `int var = ***ptr` indicates that `ptr` is a third rank pointer, and its depth is 3. We calculate the MaxPointerDepth by finding the max depth of the pointer definition or usage of pointer included in a change.
- **Goto (GT).** In C programming, `goto` statement is also called unconditional transfer statement. Developers can use statements like `goto label;` to jump unconditionally to any defined label, which if misused can reduce the readability of programs or even lead to serious defects. We define the Goto feature as the sum of *"goto"* operations included in a change.
- **IndexUsed (IU).** Index is commonly used to access array elements. For example, `arr[0]` is used to visit the first element in array `arr`. According to our manual analyzation of 300 defective commits in the ICT company, the usage of index is one of the frequent causes for out of index errors. Thus, we include the IndexUsed feature in our study and define it as the sum of index used in array visiting by a change.
- **AutoIncreDecre (AID).** AutoIncreDecre stands for Auto Increment or Decrement. For example, `ptr++` implements auto increment, while `--i` implements auto decrement. Auto increment or decrement operations are often used in loop control statements (e.g., `while((index++)<=10))`, which can lead to out of bounds errors or infinite loops if not used properly. We calculate the AutoIncreDecre feature by the number of increasing or decreasing operations included in a change.

### C. Baseline model

According to the large-scale comparison studies conducted by existing works [21, 22], CBS+ proposed by Huang et al. [12] has been identified as the best-performing supervised model among existing supervised [10, 11, 20] and unsupervised [13, 14] effort-aware JIT defect prediction approaches. CBS+ is built upon the 14 language-independent change-level features [10] and operates on the principle that among changes classified as defect-prone, small ones should be inspected first, as they are proportionally more defect-prone.

### D. Data Preprocessing

Considering that correlated features may cause side-effects on models' performance [40], we filter correlated features and redundant features before evaluating the performance of models on selected C projects following [41] when implementing CBS+ to make sure we follow the same processes as its original descriptions [12].

**Filtering correlated features.** First, to deal with data skewness, we apply a standard logarithmic transformation $ln(x+1)$ to values of features except for the binary feature FIX following Kamei et al. [10]. Then, we calculate the correlation coefficient between each pair of features through Spearman rank test [42]. We use *Hmisc*, a R tool-kit [43] for variable clustering analysis, to cluster correlated features. We set the threshold in correlation analysis to 0.8 as suggested by Li et al. [44]. To be specific, if the correlation coefficient of two features is larger than 0.8, they are considered as collinear features and one of the them should be deleted. When removing correlated features for a specific project, we first remove the feature which is correlated with many other features. For each pair of the remaining correlated features, we remove the more complex one as suggested by Kamei et al. [10, 45] and Li et al. [44].

**Filtering redundant features.** A redundant feature is a feature that can be predicted by the combination of other features. Redundant features have no contributions to JIT defect prediction models and might increase the model training cost. Thus, after removing correlated features, we further filter redundant features. We use *redun*, a function in *rms* R tool-kit [46], to filter redundant features.

### E. Evaluation

**Validation setting.** Following previous studies [13, 18, 21], we adopt a *ten-fold time-aware validation setting* which considers the chronological order of changes within the same project. Specifically, for each project, we first sort the changes in chronological order. Then, we divide these changes into approximately 12 equal groups labeled as fold 0 to fold 11, where fold 0 contains the changes that were submitted the earliest. Finally, for each fold $i$ ($1 \leq i \leq 10$), we use all the changes in fold 0 to fold $i-1$ as the training data to build a model, then evaluate its performance on fold $i$. Note that changes in fold 0 and fold 11 will not be used as evaluation set, since fold 0 has no previous folds to build a evaluation model and changes in fold 11 may not be accurately labeled by the SZZ algorithm.

**Performance measures.** Since effort-aware JIT defect prediction is intended to alleviate inspection effort, we mainly adopt effort-aware performance measures in our work. To be specific, we use $Precision@20\%$, $Recall@20\%$, $F1\text{-}measure@20\%$, $PCI@20\%$, $IFA$ and $P_{opt}$ as performance measures. All of these measures, except $P_{opt}$ and $IFA$, are calculated considering code inspection effort. Assuming that we have a dataset with $M$ changes and $N$ defective changes. After inspecting 20% of the total changed lines of code in this dataset, we inspected $m$ changes in total and found $n$ defective ones. Then these measures can be calculated as follows:

$Precision@20\%(P@20\%)$: The proportion of inspected defective changes over the total inspected changes, which calculated as $n/m$.

$Recall@20\%(R@20\%)$: The proportion of inspected defective changes over the total defective changes in the dataset, calculated as $n/N$.

$F1\text{-}measure@20\%(F1@20\%)$: A summary measure of $P@20\%$ and $R@20\%$, calculated as $\frac{2 \times P@20\% \times R@20\%}{P@20\% + R@20\%}$.

$PCI@20\%$: The Proportion of Changed Inspected over the total changes in the dataset, calculated as $m/M$.

$IFA$: The number of Initial False Alarms encountered before we observe the first defective change, calculated as the number of inspected clean changes before observing the first defective change.

$P_{opt}$: Based on the concept of Alberg diagram[47], $P_{opt}$ is used to measure how approximate the model is to the optimal model. For a given prediction model $m$, this measure can be calculated as $P_{opt}(m) = 1 - \frac{Area(Optimal) - Area(m)}{Area(Optimal) - Area(worst)}$, where $Area(m)$ represents the area under the corresponding curve of model $m$ [12, 13].

Note that a larger value is better for $Precision@20\%$, $Recall@20\%$, $F1\text{-}measure@20\%$ and $P_{opt}$, while a smaller value is better for $PCI@20\%$ and $IFA$.

### F. Statistical Analysis

**Improvement.** Improvement stands for the improvement ratio of one method over the other. Specifically, for performance measures where the value is the larger the better (i.e., $Precision@20\%$, $Recall@20\%$, $F1\text{-}measure@20\%$ and $P_{opt}$), the improvement of method A over method B is calculated as $\frac{A-B}{B} \times 100\%$. For performance measures where the value is the smaller the better (i.e., $PCI@20\%$ and $IFA$), the improvement of method A over method B is calculated as $\frac{B-A}{B} \times 100\%$.

$p$**-value.** In order to investigate the statistical significance of the performance difference between two approaches, we conduct the Wilcoxon signed-rank test [48] with a Bonferroni correction[49] on the performance measures. The Wilcoxon signed-rank test is a non-parametric statistical hypothesis test, and the Bonferroni correction is used to offset the problem of multiple comparisons. For each test, the null hypothesis is that there is no difference between the performance of two approaches, and the significance level $\alpha$ is set to 0.05. The null hypothesis is rejected when $p\text{-}value$ is no larger than 0.05; otherwise, the null hypothesis is accepted. That is to say, the performance difference between two approaches is statistically significant if $p\text{-}value$ is no larger than 0.05.

### G. Research Questions

To investigate the effectiveness and characteristic of change-level features, our experiments focus on the following three research questions:

- **RQ-1**: Whether C-specific change-level features can further improve the performance of effort-aware JIT defect approach on selected commercial and OSS projects?
- **RQ-2**: Whether change-level features have varying impact on identification of defect-inducing commits?
- **RQ-3**: How well does model built with C-specific change-level features perform on practical projects?

## IV. Experimental Results

### A. [RQ-1]: Effectiveness of C-specific Features On Identifying Defect-inducing Changes.

**Objective:** Existing supervised [10, 11, 20] and unsupervised [13, 14] effort-aware JIT defect prediction approaches

**TABLE V:** The performance of CBS+ built with/without 9 C-specific features in OSS projects. The best results are highlighted in bold. '↓' stands for 'the smaller the better'; '↑' stands for 'the larger the better'.

| **Project** | **P@20%↑** | | **R@20%↑** | | **F1@20%↑** | | **PCI@20%↓** | | $P_{opt}$ ↑ | | **IFA↓** | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **JIT** | **JIT+C** | **JIT** | **JIT+C** | **JIT** | **JIT+C** | **JIT** | **JIT+C** | **JIT** | **JIT+C** | **JIT** | **JIT+C** |
| scrcpy | **0.33** | **0.33** | 0.38 | **0.41** | 0.34 | **0.36** | **0.24** | 0.26 | 0.55 | **0.58** | 6.10 | **3.90** |
| git | 0.32 | **0.39** | 0.41 | **0.43** | 0.36 | **0.40** | 0.18 | **0.15** | 0.51 | **0.54** | 50.20 | **3.40** |
| obs-studio | 0.28 | **0.31** | 0.59 | **0.60** | 0.38 | **0.40** | 0.31 | **0.28** | 0.61 | **0.61** | 7.80 | **6.40** |
| FFmpeg | 0.24 | **0.31** | 0.42 | **0.43** | 0.31 | **0.36** | 0.29 | **0.23** | 0.51 | **0.55** | 38.90 | **15.70** |
| curl | 0.51 | **0.52** | 0.57 | **0.58** | 0.54 | **0.55** | 0.28 | **0.27** | 0.64 | **0.65** | **2.70** | 3.30 |
| ss | 0.13 | **0.29** | 0.50 | **0.53** | 0.21 | **0.37** | 0.54 | **0.26** | 0.61 | **0.64** | 41.70 | **4.90** |
| mpv | **0.54** | 0.54 | 0.57 | **0.58** | 0.55 | **0.55** | **0.29** | 0.29 | 0.69 | **0.73** | **3.40** | 3.60 |
| radare2 | 0.77 | **0.80** | 0.56 | **0.58** | 0.64 | **0.67** | 0.41 | **0.41** | 0.67 | **0.70** | 2.20 | **1.10** |
| goaccess | 0.12 | **0.14** | 0.55 | **0.60** | 0.20 | **0.22** | **0.33** | 0.34 | 0.59 | **0.62** | 31.90 | **31.60** |
| nnn | 0.34 | **0.40** | 0.45 | **0.46** | 0.37 | **0.41** | 0.33 | **0.30** | 0.53 | **0.57** | 5.40 | **2.50** |
| *Average* | 0.36 | **0.40** | 0.50 | **0.52** | 0.39 | **0.43** | 0.32 | **0.28** | 0.59 | **0.62** | 19.03 | **7.64** |
| *Improvement* | | 13% | | 3% | | 10% | | 13% | | 5% | | 60% |
| *p-value* | | <0.001 | | <0.01 | | <0.001 | | <0.001 | | <0.001 | | <0.001 |
| ***Trend*** | ↗ | | ↗ | | ↗ | | ↗ | | ↗ | | ↗ | |

**TABLE VI:** The performance of CBS+ built with/without 9 C-specific features in commercial projects of the ICT company. The best results are highlighted in bold. '↓' stands for 'the smaller the better'; '↑' stands for 'the larger the better'.

| **Project** | **P@20%↑** | | **R@20%↑** | | **F1@20%↑** | | **PCI@20%↓** | | $P_{opt}$ ↑ | | **IFA↓** | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **JIT** | **JIT+C** | **JIT** | **JIT+C** | **JIT** | **JIT+C** | **JIT** | **JIT+C** | **JIT** | **JIT+C** | **JIT** | **JIT+C** |
| P1 | 0.61 | **0.68** | 0.60 | **0.66** | 0.54 | **0.66** | 0.37 | **0.27** | 0.59 | **0.79** | 13.20 | **1.10** |
| P2 | 0.25 | **0.26** | **0.45** | **0.45** | 0.30 | **0.31** | 0.19 | **0.17** | **0.55** | **0.55** | 4.50 | **4.00** |
| P3 | 0.57 | **0.61** | 0.67 | **0.70** | 0.61 | **0.64** | 0.22 | **0.21** | 0.68 | **0.75** | 4.20 | **2.50** |
| P4 | 0.27 | **0.30** | 0.53 | **0.60** | 0.34 | **0.39** | **0.23** | **0.23** | 0.57 | **0.62** | 7.70 | **6.20** |
| P5 | 0.28 | **0.31** | **0.50** | **0.50** | 0.34 | **0.36** | 0.23 | **0.21** | 0.56 | **0.59** | 6.60 | **6.10** |
| P6 | 0.18 | **0.21** | 0.35 | **0.38** | 0.21 | **0.24** | 0.22 | **0.19** | 0.55 | **0.60** | 10.90 | **8.40** |
| P7 | 0.27 | **0.37** | 0.52 | **0.53** | 0.34 | **0.41** | 0.40 | **0.27** | 0.59 | **0.65** | 17.00 | **3.60** |
| P8 | 0.32 | **0.38** | 0.32 | **0.34** | 0.30 | **0.34** | 0.13 | **0.11** | 0.36 | **0.52** | 3.90 | **3.20** |
| *Average* | 0.34 | **0.39** | 0.49 | **0.52** | 0.37 | **0.42** | 0.25 | **0.21** | 0.56 | **0.63** | 8.50 | **4.39** |
| *Improvement* | | 13% | | 6% | | 12% | | 16% | | 14% | | 48% |
| *p-value* | | <0.001 | | <0.05 | | <0.001 | | <0.01 | | <0.001 | | <0.01 |
| ***Trend*** | ↗ | | ↗ | | ↗ | | ↗ | | ↗ | | ↗ | |

are built with the 14 language-independent change-level features proposed by Kamei et al. [10]. However, each programming language has its own characteristics, which may have varying impacts on software quality. Recently, Ni et al. [21] conducted a study on whether programming language-specific features can help to identify bug-inducing changes and found that JavaScript-specific features can further improve identification performance of existing approaches. Inspired by Ni et al. [21]'s work and considering the importance of C/C++ programming language in the development of fundamental systems, we aim to investigate whether C-specific change-level features can enhance the performance of identifying defect-inducing changes.

**Experiment Design:** We first calculate the 14 change-level features following prior work [10], and use B-SZZ algorithm [17] for data labeling. Then, we propose 9 C-specific change-level features by considering its characteristics, such as memory management and pointer operations, and extract each value for each change. We adopt the best-performing supervised model named CBS+ for building our JIT defect prediction model based on the consideration that the focus of our work is on investigating the effectiveness of C-specific features in identifying defect-inducing changes, rather than proposing new approaches. For the validity of our experiment, we apply the same data pre-processing procedures and use under-sampling on training data to deal with the class-imbalance problem following Huang et al. [12] when implementing CBS+. Following that, we run and evaluate CBS+ in a *ten-fold time-aware validation setting* as described in Section III-E. Specifically, we run the approach on each of the ten folds between the first fold and the last fold, and calculate the average performance for each selected project. Finally, we make a comparison between the performance of CBS+ using only the 14 language-independent features and the performance of the approach using a combination of the 14 language-independent features and our newly proposed 9 C-specific features on both projects from GitHub and projects from the ICT company.

**Results:** The experimental results are shown in Table V and Table VI. The column named "JIT" indicates that CBS+ is built with only 14 language-independent change-level features, while the column named "JIT+C" indicates that CBS+ is built with the combination of 14 language-independent features and 9 C-specific features. The best performance are highlighted in bold. The bottom few lines present the statistical results calculated as described in Section III-F, and the changing trend of each performance measure is shown in the last row.

As for OSS projects shown in Table V, we find that C-specific change-level features can significantly improve the performance of identification of defect-inducing changes in terms of six effort-aware performance measures. More precisely, the model built with "JIT+C" achieves 0.40, 0.52, 0.43, 0.28, 0.62 and 7.74 in terms of P@20%, R@20%, F1@20%, PCI@20%, $P_{opt}$, and IFA, which statistically significantly improve the model built only with "JIT" by 13%, 3%, 10%, 13%, 5% and 60%, respectively. Additionally, we can conclude that with a budget of 20% of the effort to inspect all changes, CBS+ built with the combination of 14 language-independent features and 9 C-specific features can help developers focus on 28% of the changes and identify 52% of the defective changes with 40% precision.

As for commercial projects shown in Table VI, we can also obtain a similar conclusion that C-specific change-level features can significantly improve the performance of identification of defect-inducing changes in terms of six effort-aware performance measures. In particular, the model built with "JIT+C" statistically significantly improves model built only with "JIT" by 13%, 6%, 12%, 16%, 14% and 48%, respectively. That is, considering a budget of 20% of the effort to inspect all changes, CBS+ built with the combination of 14 language-independent features and 9 C-specific features can help developers focus on 21% of the changes and identify 52% of the defective changes with 39% precision.

Comparing the performance between OSS projects and commercial projects, we observe that C-specific change-level can extremely decrease the false alarm in terms of "IFA" (i.e., from 19.03 to 7.64 in OSS projects and from 8.50 to 4.39 in commercial projects), which seems to be friendly when applied to practical application.

✍ **RQ-1** ▶ *C-specific programming language change-level features are helpful to identify defect-inducing changes and can statistically significantly improve the performance of model built only with 14 language-independent change-level features in both OSS projects and commercial projects. The performance of effort-aware JIT defect prediction approach can be further improved by using C-specific change-level features in both commercial and OSS C projects.* ◀

## *B. [RQ-2]: Importance of Change-level Features On Identifying Defect-inducing Changes.*

**Objective:** Kamei et al. [10] introduced 14 language-independent change-level features from five dimensions (as listed in Table III) to identify defect-inducing commits. In this paper, considering the importance of C programming language on building fundamental systems and limited study on its change-level characteristic, we also propose nine C-specific features as shown in Table IV. These features, totally 23 features, describe a commit from different perspectives, and they play varying degrees of importance to the quality of a commit. Thus, we want to investigate whether these features have different importance and which features are the most important ones. Besides, we also want to figure out whether these features have varying importance between OSS projects and commercial projects.

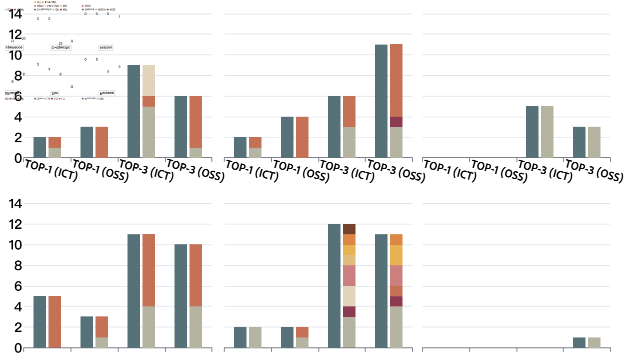


Fig. 2: Number of commercial projects of the ICT company and OSS projects where a feature is ranked as the top-1 or top-3 important feature.

**Experiment Design:** We analyze how change-level features affect CBS+'s performance after two data pre-processing steps: filtering correlated features and filtering redundant features, as described in Section III-D. We rebuild CBS+ with the remaining features according to the suggestion by existing works [21, 50], and investigate features' importance with a $10 \times 10$-fold cross-validation experimental setting.

In general, two phases are involved for investigating importance of the studied features:

**(1) Calculating importance scores.** In each fold, we use the training data to build a CBS+ model. Then, we calculate the generic feature importance score proposed by Tantithamthavorn et al.[51, 52]. The generic feature importance score for a specific feature is calculated by the following two steps. First, we randomly permute the values of this feature in the testing data, and keep all the values of other features as they are. Second, we calculate the performance difference between the $F1\text{-}measure@20\%$ of the CBS+ model applied on the original testing data and the randomly permuted testing data following [22]. A larger difference suggests that this feature has a larger impact on the performance of the model and is more important. Thus, we use the difference values to measure features importance.

**(2) Calculating importance ranks.** After conducting the $10 \times 10$-fold cross-validation, we totally obtain 100 importance scores for each feature. We apply the Scott-Knott ESD (SK-ESD) test on the feature importance scores following existing works [4, 21, 22, 50, 53]. The SK-ESD test is an enhanced version of the Scott-Knott test [54]. On one hand, the SD-ESD test relaxes the assumption of normally distributed data, which is required by the Scott-Knott test, by alleviating the skewness of input data. On the other hand, the SD-ESD test takes the effect size of the input data into account and merges two statistically distinct groups having a negligible effect size into one group.

Finally, based on the ranks of each feature on each project, we count for each feature the number of commercial projects and OSS projects where the feature is ranked as top-1 or top-3 important features.

**Results:** Table VII presents the ranks of the 14 language-independent features and 9 C-specific features in OSS and commercial projects. To better illustrate the ranks of each used feature, we show them in a way of heatmap and also present the rank index for each feature. In this table, each row represents a project and the value in each cell represents the ranks of the corresponding feature in such a project. For the cell with no value, it means that the feature is removed due to correlation or redundancy. Besides, Fig. 2 summarizes for each feature the number of commercial and OSS projects where the feature is ranked as top-1 or top-3 important features. In Fig. 2, the $x$-axis represents the top-1 and top-3 most important features in commercial projects and OSS projects, while the $y$-axis represents the corresponding numbers of project. According to the results, we obtain different conclusions for commercial and OSS projects:

Considering the top-1 important features, "SEXP" (which ranks first in 5 projects), "SinglePointer" (which ranks first in 2 projects), "NS"(which ranks first in 1 project), "NF"(which ranks first in 1 project), "LA"(which ranks first in 5 project) and "LT"(which ranks first in 1 project) are more important than other features in commercial projects, while "LT" (which ranks first in 4 projects), "NF" (which ranks first in 3 projects), "SEXP" (which ranks first in 2 projects), "EXP" (which ranks first in 1 project), "SinglePointer" (which ranks first in 1 project) and "MaxPointerDepth" (which ranks first in 1 project) are more important than other features in OSS projects.

Considering the top-3 important features, "SEXP", "FIX" and "NS" are ranked as top-3 important features for 7, 5 and 5 commercial projects (more than half of the commercial projects), while "LT" and "SEXP" are ranked as top-3 important features for 7 and 6 OSS projects. Thus, "SEXP" is important for both commercial and OSS projects. Apart from "SEXP", both "FIX" and "NS" are important for commercial projects, while "LT" is important for OSS projects.

In general, we conclude that "SEXP", "SinglePointer", "NS", "NF", "LA", "LT" and "FIX" are the most important features for commercial projects, while "LT", "NF", "SEXP", "EXP", "SinglePointer" and "MaxPointerDepth" are the most important features for OSS projects. We find that "SEXP", "LT", "NF" and "SinglePointer" are the most important features for both commercial and OSS projects.

✍ **RQ-2** ► *commercial projects and OSS projects vary in the most important features. In general, "SEXP", "LT", "NF" and "SinglePointer" are the most important features for both commercial and OSS C projects, which indicates that feature groups "Experience", "Size", "Diffusion" and "C-Specific" are important for effort-aware JIT defect prediction in C projects.*◄

### *C. [RQ-3]: Practical Usefulness of C-Specific Features On Identifying Defect-inducing Changes.*

**Objective:** Aforementioned results conducted on CBS+ indicate that our newly proposed 9 C-specific change-level features can further improve CBS+'s performance. Although the CBS+ approach built with the combination of the 14 language-independent change-level features and our 9 C-specific features performs well in terms of the six widely used effort-aware performance measures, its practicability in an actual industrial setting has not been verified. Thus, we developed an effort-aware JIT defect prediction tool based on CBS+ using the combination of the 14 language-independent features and our 9 C-specific features, and conduct a developer study to evaluate its effectiveness when it is deployed in the ICT company. In the developer study, we wish to explore developers' degree of recognition for our effort-aware JIT defect prediction tool.

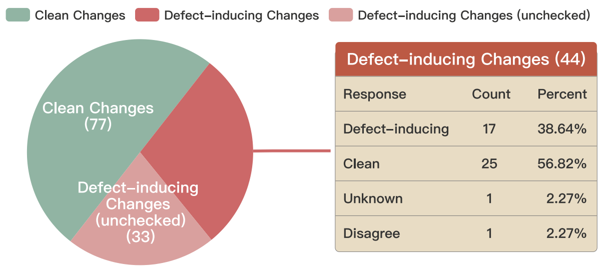
**Experiment Design:** Given the limitation of developers' time and effort, we only conduct the developer study on one of the commercial projects in the ICT company. We use P7 as introduced in Table I, considering that several developers in this project show a willingness to participate in our developer study. In total, we have five participants who are active contributors in P7 with different levels of experience. The developer study consists of the following steps:

Step ①: preparing the review list. In the developer study, we set the reviewing scope (test set) to the 154 recent changes from March 1 to March 31, 2022. The actual labels (i.e., defective or clean) of these changes are not known in advance. These changes are selected because we aim to evaluate our tool in a real industry setting, where most recent changes are not yet fully reviewed or tested, and thus there may be defects that are not identified or fixed. Under such a circumstance, our effort-aware JIT defect prediction tool is used to help developers discover potential defects. After applying our tool on the test set, 77 changes are classified as defective by our tool. Since we set the inspection effort budget to 20% of the total changed LOC in the test set following existing studies [18, 22], the tool finally generates an ordered list of 44 warned changes. In other words, the tool claims that these 44 changes are defect-inducing and should be reviewed at high priority. For each change in the list, we attach the corresponding code review url on the code review system so that developers could go over detailed information of the change conveniently.

Step ②: conducting the survey. We conduct a survey of five participating developers on April 15, 2022. During the survey, developers are asked to review the changed LOC of each change in the list generated in Step ① on the code review system. At the same time, they are required to mark each change as defective, clean or unknown. Before the survey, developers are informed that *Defect-inducing* means that the change may cause bugs in the future, and thus could induce defects in their products and hinder the development process. *Clean* means that the change has a low risk of inducing future

**TABLE VII:** Ranks of the studied features in the selected OSS projects and commercial projects.

| Projects | Diffusion | | | | Size | | | Purpose | History | | | Experience | | | C-Specific | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | NS | ND | NF | Entropy | LA | LD | LT | FIX | NDEV | AGE | NUC | EXP | REXP | SEXP | MI | MD | MC | SP | MP | MPD | GT | IU | AID |
| scrcpy | 3 | 4 | 1 | | 4 | 2 | 3 | 3 | | 5 | | 1 | | | 4 | 3 | 4 | | 5 | 1 | 3 | 4 | 4 |
| git | | 8 | | 6 | 4 | | | 10 | | 13 | | 2 | | 1 | 8 | 7 | 12 | | 3 | | 9 | 5 | 11 |
| obsstudio | | 6 | 1 | 7 | 3 | 6 | 5 | 5 | | 7 | | 2 | | 2 | 5 | 6 | 6 | 4 | 7 | | 5 | 5 | 6 |
| FFmpeg | | 7 | 3 | 10 | 4 | 8 | 5 | 8 | 12 | 13 | | 6 | | 2 | 11 | 11 | 13 | 1 | 12 | 14 | 9 | 7 | 7 |
| curl | | 7 | | 9 | 4 | | 1 | 9 | | 8 | | 9 | | | 7 | 5 | 7 | 2 | 6 | | 8 | 3 | 5 |
| ss | | 7 | | 7 | 4 | 6 | 3 | 5 | | 7 | | 2 | | 1 | 7 | 8 | 8 | 4 | 6 | 7 | 8 | 6 | 8 |
| mpv | | 6 | 1 | 7 | 3 | 4 | 2 | 6 | | 6 | | 5 | | | 6 | 6 | 6 | 3 | 6 | | 6 | 5 | 6 |
| radare2 | 8 | 5 | 2 | | 6 | 5 | 1 | 7 | 3 | 10 | | 9 | | | 11 | 12 | 9 | 4 | 11 | 7 | 11 | 12 | 12 |
| goaccess | | 6 | | 6 | 2 | | 1 | 2 | | 5 | | 5 | | 3 | 4 | 3 | 5 | 6 | 4 | | 6 | 5 | 5 |
| nnn | | 6 | | 6 | 4 | | 1 | 3 | | 6 | | 6 | | 3 | 4 | 6 | 5 | 2 | 5 | | 3 | 4 | 5 |
| P1 | 2 | | 1 | | 3 | | | 4 | | | | 6 | | 3 | 7 | 7 | | 5 | 7 | | 6 | 5 | 6 |
| P2 | 3 | | | 2 | 3 | | | 4 | | | | 2 | | 1 | 4 | 4 | 4 | 4 | 4 | | 4 | 4 | 4 |
| P3 | | 4 | | 3 | 4 | | | 3 | | | | 4 | | 5 | 2 | 3 | 7 | 1 | 5 | | 4 | 6 | 5 |
| P4 | 4 | | | 6 | 4 | | | 5 | | | | 3 | | 1 | | | | 2 | 6 | | | 6 | 5 |
| P5 | | 4 | | 5 | 4 | | 1 | 2 | | | | 5 | | 3 | 6 | 6 | 6 | 6 | 6 | | 6 | 5 | 5 |
| P6 | 1 | | | 6 | | 5 | 2 | 2 | | 5 | | 3 | | 1 | 5 | 5 | 5 | 4 | 5 | | 5 | 5 | 5 |
| P7 | 2 | | | 8 | 5 | | 2 | 3 | | 6 | | 5 | | 1 | 6 | 6 | 6 | 4 | 4 | | 5 | 7 | 6 |
| P8 | 3 | | | 3 | 1 | | | 2 | | | | 3 | | 1 | 3 | 2 | 3 | 1 | 3 | | 3 | 2 | 3 |



Defect–inducing Changes (44)

| Response | Count | Percent |
|---|---|---|
| Defect–inducing | 17 | 38.64% |
| Clean | 25 | 56.82% |
| Unknown | 1 | 2.27% |
| Disagree | 1 | 2.27% |

**Fig. 3:** The aggregated responses of the developer study. Each change is labeled based on developers' responses following the rules as described in Step ③.

bugs or defects. *Unknown* means that the developer cannot decide whether the change is defective or clean for sure. Note that each developer is asked to do the survey separately.

Step ③: aggregating the responses. In order to acquire a comprehensive result, we aggregate the various responses of the participants. Following existing works [22, 55], we aggregate the responses according to the following rules: (1) A change is labeled as *defect-inducing* if at least 2 developers marked the change as defect-inducing, and the number of defect-inducing responses is larger than that of the clean responses; (2) A change is labeled as *clean* on the same condition as (1) but in reverse; (3) A change is labeled as *unknown* if at least 3 developers marked the change as unknown; (4) Otherwise, the change is labeled as *disagreed* if we receive 2 clean responses, 2 defective responses and 1 unknown response.

**Results.** The aggregated responses of the developers are illustrated in Fig. 3. In the review list consisting of 44 defect-inducing changes identified and warned by our tool, 17 are labeled as defect-inducing, 25 are labeled as clean, 1 is labeled as unknown and 1 is labeled as disagree.

The result suggests that with an inspection effort budget of 20% of the changed LOC, our tool helps developers concentrate on 44 of the 154 changes (i.e., 28.6% of all changes in the reviewing scope). Besides, according to the aggregated responses, 17 changes of the 44 warned changes by our tool are truly defect-inducing ones, which suggests that developers agreed with our tool's defect-inducing warning 38.6% of the time. Moreover, in the review list, the first two changes are labeled as clean and the third one is labeled as defective, which suggests that developers need to inspect only two changes before finding the first defective one.

We notice that among the 44 warned changes of our tool the misclassification rate is about 56.82%. Thus, we re-examine the 25 misclassified changes manually for clues as to why our tool might incorrectly classify these changes. We observe that 36% of the 25 changes are in fact refactoring ones (i.e., changing `bool` variable to `enum` variable, renaming functions and files, switching boolean arguments to key strings, etc.). Refactoring changes are the changes which improve the design of a software system but do not change its external behaviour [56]. Refactoring changes may considerably impact SZZ implementations in aspects of producing false positives [37], and mislabeled changes in the training set might negatively impact the performance of our model. Neto et al. [37] propose RA-SZZ to identify refactoring changes in Java projects. However, as far as we know, there are no existing SZZ algorithms which can disregard refactoring changes in C/C++ projects. This suggests a possible direction of improvement for existing data labeling methods.

✍ **RQ-3▶** *In the developer study, our tool helps the developers focus on 28.6% of all changes in the reviewing scope. According to the aggregated responses, the Precision@20% of our tool is 38.6% and the IFA of our tool is 2. Developers think our tool brings them some help in addressing code review tasks.* ◀

## V. Implications and Discussion

Our study reveals the following important practical guidelines for future work of JIT defect prediction.

**Programming Language-Specific Change-level Features.** Different programming languages have varying characteristics which may have different impacts on software quality [21]. In practice, developers often choose different programming languages for various applications. For example, the C programming language is used to develop foundation applications such as operating system since its efficiency, while the Python programming language is used for scientific analysis since its characteristic of being easy to use. Therefore, the widely used 14 programming language-independent change-level features [10] may not fully capture the characteristics of code changes. Meanwhile, the advantages of language-specific features on identifying defect-inducing changes are also illustrated in both existing work [21] and this work. We highly recommend the researchers/developers to investigate more language-specific features when studying the quality of software changes.

**Programming Language-Sensitive SZZ algorithm**. Our results of the developer study indicate that code refactoring operations do have a side-effect on the performance of defect-inducing change identification. The cause may be that we adopt the B-SZZ algorithm to label our dataset since its effectiveness was verified by Rosa et al. [39]. This algorithm cannot filter out the noise from refactoring which may impact the quality of dataset labeling. However, though there exists a newly proposed SZZ (i.e., RA-SZZ [57, 58]) which considers the side-effect of code refactoring operation, RA-SZZ can only be applied to detect refactoring operation on Java projects. That is, we cannot use it in our industry study of C/C++ projects. Therefore, we suggest researchers/developers improve SZZ algorithm by integrating language-sensitive refactoring detection.

**Challenges in JIT Defect Prediction Model**. Our results of the developer study in the ICT company show that there still have two checked changes that cannot obtain an agreement on their label although the developers are active during the process of the project development, which means the JIT defect prediction is still a challenging task. Such observation motivates future researchers to investigate more robust and more performing-well JIT defect prediction approaches in practical usage.

## VI. Threats to Validity

Threats to internal validity are two-folds. One potential threat consists in the potential errors in our implementation of CBS+ using Python programming language. To alleviate the threat, we strictly follow the original descriptions of CBS+ [12], fully utilize reliable third-party libraries and go through multiple inspections and tests on our code. The other potential threat lies in that the SZZ algorithm cannot guarantee 100% precision in change labeling, which means there might be a number of mislabeled changes. In this study, we conduct some extra optimizations when implementing B-SZZ [17] based on the idea of AG-SZZ [35] and MA-SZZ [36] to filter noises such as blank lines, comments and meta-changes.

Threats to external validity primarily consist in the universality of our datasets and conclusions. In this study, we use top-10 most starred open-source C projects in Github, which vary in project sizes and application domains. However, whether our findings in this study can be applied to other open-source projects from other domains is still unknown. To investigate the effectiveness of effort-aware JIT defect prediction in an industrial setting, we study 8 commercial projects and conduct a developer study with five participants at an ICT company. Since we only conduct the case study in one company, whether our findings can be applied to other companies is also unknown. Nevertheless, it is notable that our main contribution in this work is that we propose a series of C-specific features for the first time, and have shown their effectiveness of identifying defect-inducing changes in both OSS and commercial projects. Researchers and companies can refer to our idea, methods and findings in their researches or applications of effort-aware JIT defect prediction approaches.

## VII. Conclusion and Future Work

In this paper, we investigate the effectiveness of effort-aware JIT defect prediction (CBS+) in both 10 open-source projects with 329,021 changes and 8 commercial projects of an ICT company with 12,983 changes by considering six effort-aware performance measures. Moreover, we propose nine C-specific change-level features (i.e., MI, MD, MC, SP, MP, MPD, GT, IU and AID) and find that these features can further improve CBS+'s performance. Then, we explore the importance of features and find that commercial and OSS projects have different conclusions on the most important features, but the features in the dimensions of "Experience", "Size", "Diffusion" and "C-specific" are important for both commercial and OSS projects. Finally, we develop a tool for JIT defect prediction based on CBS+ using both the 14 language-agnostic features and our 9 C-specific features, and evaluate its usefulness through a developer study in the ICT company. Results of the developer study indicate that our tool helps developers focus on 28.6% of all changes in the reviewing scope, and that the $Precision@20\%$ of our tool is 38.6% and the *IFA* is 2. We also release our dataset and model for public verification and further research[2].

Our future work involves extending our evaluation on more product lines in the ICT company.

## Acknowledgment

This research is supported by the National Natural Science Foundation of China (No. 62202419), the Ningbo Natural Science Foundation (No. 2022J184), and Meituan.

## References

[1] H. Hata, O. Mizuno, and T. Kikuno, "Bug prediction based on fine-grained module histories," in *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 200–210.

[2] C. Liu, D. Yang, X. Xia, M. Yan, and X. Zhang, "A two-phase transfer learning model for cross-project defect prediction," *Information and Software Technology*, 2018.

[3] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *IEEE*

[2]https://github.com/jacknichao/C-specific-metrics

*transactions on software engineering*, no. 1, pp. 2–13, 2007.

[4] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "An empirical comparison of model validation techniques for defect prediction models," *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 1–18, 2016.

[5] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, "Cross-project defect prediction: a large scale experiment on data vs. domain vs. process," in *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 2009, pp. 91–100.

[6] T. Menzies, A. Butcher, A. Marcus, and D. Zimmermann, Thomas a nd Cok, "Local vs. global models for effort estimation and defect prediction," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2011, pp. 343–351.

[7] J. Nam, S. J. Pan, and S. Kim, "Transfer defect learning," in *Software Engineering (ICSE), 2013 35th International Conference on*. IEEE, 2013, pp. 382–391.

[8] C. Ni, W.-S. Liu, X. Chen, Q. Gu, D.-X. Chen, and Q.-G. Huang, "A cluster based feature selection method for cross-project software defect prediction," *Journal of Computer Science and Technology*, vol. 32, no. 6, pp. 1090–1107, 2017.

[9] X. Xia, D. Lo, S. J. Pan, N. Nagappan, and X. Wang, "Hydra: Massively compositional model for cross-project defect prediction," *IEEE Transactions on Software Engineering*, vol. 42, no. 10, pp. 977–998, 2016.

[10] 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*, vol. 39, no. 6, pp. 757–773, 2013.

[11] W. Fu and T. Menzies, "Revisiting unsupervised learning for defect prediction," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 72–83.

[12] Q. Huang, X. Xia, and D. Lo, "Revisiting supervised and unsupervised models for effort-aware just-in-time defect prediction," *Empirical Software Engineering*, pp. 1–40, 2018.

[13] Y. Yang, Y. Zhou, J. Liu, Y. Zhao, H. Lu, L. Xu, B. Xu, and H. Leung, "Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 157–168.

[14] J. Liu, Y. Zhou, Y. Yang, H. Lu, and B. Xu, "Code churn: A neglected metric in effort-aware just-in-time defect prediction," in *Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE Press, 2017, pp. 11–19.

[15] M. Yan, X. Xia, Y. Fan, A. E. Hassan, D. Lo, and S. Li, "Just-in-time defect identification and localization: A two-phase framework," *IEEE Transactions on Software Engineering*, vol. 48, no. 1, pp. 82–101, 2022.

[16] C. Ni, W. Wang, K. Yang, X. Xia, K. Liu, and D. Lo, "The best of both worlds: integrating semantic features with expert features for defect prediction and localization," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 672–683.

[17] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *ACM sigsoft software engineering notes*, vol. 30, no. 4. ACM, 2005, pp. 1–5.

[18] Q. Huang, X. Xia, and D. Lo, "Supervised vs unsupervised models: A holistic look at effort-aware just-in-time defect prediction," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2017, pp. 159–170.

[19] T. Mende and R. Koschke, "Effort-aware defect prediction models," in *2010 14th European Conference on Software Maintenance and Reengineering*, 2010, pp. 107–116.

[20] Q. Huang, X. Xia, D. Lo, and G. C. Murphy, "Automating intention mining," *IEEE Transactions on Software Engineering*, vol. 46, no. 10, pp. 1098–1119, 2018.

[21] C. Ni, X. Xia, D. Lo, X. Yang, and A. E. Hassan, "Just-in-time defect prediction on javascript projects: A replication study," *ACM Transactions on Software Engineering and Methodology*, 2021.

[22] M. Yan, X. Xia, Y. Fan, D. Lo, A. E. Hassan, and X. Zhang, "Effort-aware just-in-time defect identification in practice: a case study at alibaba," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1308–1319.

[23] Openharmony, 2022. [Online]. Available: https://gitee.com/openharmony

[24] O. Foundation, 2022. [Online]. Available: https://www.openatom.org

[25] GitHub, 2022. [Online]. Available: https://github.com

[26] A. Mockus and D. M. Weiss, "Predicting risk of software changes," *Bell Labs Technical Journal*, vol. 5, no. 2, pp. 169–180, 2000.

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

[28] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram, "How do fixes become bugs?" in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 2011, pp. 26–36.

[29] X. Yang, D. Lo, X. Xia, and J. Sun, "Tlel: A two-layer ensemble learning approach for just-in-time defect prediction," *Information and Software Technology*, vol. 87,

pp. 206–220, 2017.

[30] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun, "Deep learning for just-in-time defect prediction," in *2015 IEEE International Conference on Software Quality, Reliability and Security*. IEEE, 2015, pp. 17–26.

[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 *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 34–45.

[32] T. Hoang, H. J. Kang, D. Lo, and J. Lawall, "Cc2vec: Distributed representations of code changes," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 518–529.

[33] E. Shihab, A. E. Hassan, B. Adams, and Z. M. Jiang, "An industrial study on the risk of software changes," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 2012, p. 62.

[34] CLOC, 2022. [Online]. Available: https://github.com/AlDanial/cloc

[35] S. Kim, T. Zimmermann, K. Pan, E. James Jr *et al.*, "Automatic identification of bug-introducing changes," in *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*. IEEE, 2006, pp. 81–90.

[36] D. A. Da Costa, S. McIntosh, W. Shang, U. Kulesza, R. Coelho, and A. E. Hassan, "A framework for evaluating the results of the szz approach for identifying bug-introducing changes," *IEEE Transactions on Software Engineering*, vol. 43, no. 7, pp. 641–657, 2016.

[37] E. C. Neto, D. A. da Costa, and U. Kulesza, "The impact of refactoring changes on the szz algorithm: An empirical study," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 380–390.

[38] ——, "Revisiting and improving szz implementations," in *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2019, pp. 1–12.

[39] G. Rosa, L. Pascarella, S. Scalabrino, R. Tufano, G. Bavota, M. Lanza, and R. Oliveto, "Evaluating szz implementations through a developer-informed oracle," in *2021 IEEE/ACM 43rd International Conference on Software Engineering*. IEEE, 2021, pp. 436–447.

[40] C. Tantithamthavorn and A. E. Hassan, "An experience report on defect modelling in practice: Pitfalls and challenges," in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*. ACM, 2018, pp. 286–295.

[41] S. J. Rao, "Regression modeling strategies: with applications to linear models, logistic regression, and survival analysis," 2003.

[42] J. H. Zar, "Spearman rank correlation," *Encyclopedia of Biostatistics*, vol. 7, 2005.

[43] Hmisc, 2022. [Online]. Available: https://cran.r-project.org/web/packages/Hmisc/Hmisc.pdf

[44] H. Li, W. Shang, Y. Zou, and A. E. Hassan, "Towards just-in-time suggestions for log changes," *Empirical Software Engineering*, vol. 22, no. 4, pp. 1831–1865, 2017.

[45] Y. Kamei, T. Fukushima, S. McIntosh, K. Yamashita, N. Ubayashi, and A. E. Hassan, "Studying just-in-time defect prediction using cross-project models," *Empirical Software Engineering*, vol. 21, no. 5, pp. 2072–2106, 2016.

[46] RMS, 2022. [Online]. Available: https://cran.r-project.org/web/packages/rms/rms.pdf

[47] E. Arisholm, L. C. Briand, and E. B. Johannessen, "A systematic and comprehensive investigation of methods to build and evaluate fault prediction models," *Journal of Systems and Software*, vol. 83, no. 1, pp. 2–17, 2010.

[48] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics bulletin*, vol. 1, no. 6, pp. 80–83, 1945.

[49] H. Abdi, "Bonferroni and šidák corrections for multiple comparisons," *Encyclopedia of measurement and statistics*, vol. 3, pp. 103–107, 2007.

[50] Y. Fan, X. Xia, D. Lo, and A. E. Hassan, "Chaff from the wheat: characterizing and determining valid bug reports," *IEEE transactions on software engineering*, 2018.

[51] G. K. Rajbahadur, S. Wang, Y. Kamei, and A. E. Hassan, "The impact of using regression models to build defect classifiers," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 135–145.

[52] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "The impact of automated parameter optimization on defect prediction models," *IEEE Transactions on Software Engineering*, 2018.

[53] X. Xia, L. Bao, D. Lo, P. S. Kochhar, A. E. Hassan, and Z. Xing, "What do developers search for on the web?" *Empirical Software Engineering*, vol. 22, no. 6, pp. 3149–3185, 2017.

[54] A. J. Scott and M. Knott, "A cluster analysis method for grouping means in the analysis of variance," *Biometrics*, pp. 507–512, 1974.

[55] C. Lewis, Z. Lin, C. Sadowski, X. Zhu, R. Ou, and E. J. Whitehead, "Does bug prediction support human developers? findings from a google case study," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 372–381.

[56] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[57] D. Silva and M. T. Valente, "Refdiff: detecting refactorings in version histories," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 269–279.

[58] N. Tsantalis, M. Mansouri, L. Eshkevari, D. Mazinanian, and D. Dig, "Accurate and efficient refactoring detection in commit history," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 483–494.