Are Slice-Based Cohesion Metrics Actually Useful in Effort-Aware Post-Release Fault-Proneness Prediction? An Empirical Study

Yibiao Yang, Yuming Zhou, Hongmin Lu, Lin Chen, Zhenyu Chen, *Member, IEEE*, Baowen Xu, Hareton Leung, *Member, IEEE*, and Zhenyu Zhang, *Member, IEEE*

Abstract—*Background*. Slice-based cohesion metrics leverage program slices with respect to the output variables of a module to quantify the strength of functional relatedness of the elements within the module. Although slice-based cohesion metrics have been proposed for many years, few empirical studies have been conducted to examine their actual usefulness in predicting fault-proneness. *Objective*. We aim to provide an in-depth understanding of the ability of slice-based cohesion metrics in effort-aware post-release fault-proneness prediction, i.e. their effectiveness in helping practitioners find post-release faults when taking into account the effort needed to test or inspect the code. *Method*. We use the most commonly used code and process metrics, including size, structural complexity, Halstead's software science, and code churn metrics, as the baseline metrics. First, we employ principal component analysis to analyze the relationships between slice-based cohesion metrics and the baseline metrics. Then, we use univariate prediction models to investigate the correlations between slice-based cohesion metrics and post-release fault-proneness. Finally, we build multivariate prediction models to examine the effectiveness of slice-based cohesion metrics in effort-aware post-release fault-proneness prediction when used alone or used together with the baseline code and process metrics. *Results*. Based on open-source software systems, our results show that: 1) slice-based cohesion metrics are not redundant with respect to the baseline code and process metrics; 2) most slice-based cohesion metrics are significantly negatively related to post-release fault-proneness; 3) slice-based cohesion metrics in general do not outperform the baseline metrics when predicting post-release fault-proneness; and 4) when used with the baseline metrics together, however, slice-based cohesion metrics can produce a statistically significant and practically important improvement of the effectiveness in effort-aware post-release fault-proneness prediction. *Conclusion*. Slice-based cohesion metrics are complementary to the most commonly used code and process metrics and are of practical value in the context of effort-aware post-release fault-proneness prediction.

Index Terms—Cohesion, metrics, slice-based, fault-proneness, prediction, effort-aware

1 Introduction

Cohesion refers to the relatedness of the elements within a module [1], [2]. A highly cohesive module is one in which all elements work together towards a single function. Highly cohesive modules are desirable in a system as they are easier to develop, maintain, and reuse, and hence are less fault-prone [1], [2]. For software developers, it is expected to automatically identify low cohesive modules targeted for software quality enhancement. However, cohesion is a subjective concept and hence is difficult to use in practice [14]. In order to attack this problem, program slicing is applied to develop quantitative cohesion metrics, as it provides a means of accurately quantifying the interactions between the elements within a module [12]. In the last three decades, many slice-based cohesion metrics have been developed to quantify the degree of cohesion in a module at the function level of granularity [3], [4], [5], [6], [7], [8], [9], [10]. For a given function, the computation of a slice-based cohesion metric consists of the following two steps. At the first step, a program reduction technology called program slicing is employed to obtain the set of program statements (i.e. program slice) that may affect each output variable of the function [9], [11]. The output variables include the function return value, modified global variables, modified reference parameters, and variables printed or other outputs by the function [12]. At the second step, cohesion is computed by leveraging the commonality among the slices with respect to different output variables. Previous studies showed that slice-based cohesion metrics provided an excellent quantitative measure of cohesion [3], [13], [14]. Hence, there is a reason to believe that they should be useful predictors for fault-proneness. However, few empirical studies have so far been conducted to examine the actual usefulness of slice-based cohesion metrics for predicting fault-proneness, especially compared with

- *Y. Yang, Y. Zhou, H. Lu, L. Chen, and B. Xu are with the State Key Laboratory for Novel Software Technology, Department of Computer Science and Technology, Nanjing University, Nanjing 210023, China. E-mail: yangyibiao@smail.nju.edu.cn, {zhouyuming, hmlu, lchen, bwxu}@nju.edu.cn.*
- *Z. Chen is with the State Key Laboratory for Novel Software Technology, School of Software, Nanjing University, Nanjing 210023, China. E-mail: zychen@software.nju.edu.cn.*
- *H. Leung is with the Department of Computing, Hong Kong Polytechnic University, Hung Hom, Hong Kong, China. E-mail: cshleung@inet.polyu.edu.hk.*
- *Z. Zhang is with the State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China. E-mail: zhangzy@ios.ac.cn.*

Manuscript received 16 Feb. 2013; revised 24 Oct. 2014; accepted 29 Oct. 2014. Date of publication 11 Nov. 2014; date of current version 17 Apr. 2015.
Recommended for acceptance by T. Menzies.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2014.2370048

the most commonly used code and process metrics [5], [15], [16], [17], [18].

In this paper, we perform a thorough empirical investigation into the ability of slice-based cohesion metrics in the context of effort-aware post-release fault-proneness prediction, i.e. their effectiveness in helping practitioners find post-release faults when taking into account the effort needed to test or inspect the code [35]. In our study, we use the most commonly used code and process metrics, including size, structural complexity, Halstead's software science, and code churn metrics, as the baseline metrics. We first employ principal component analysis (PCA) to analyze the relationships between slice-based cohesion metrics and the baseline code and process metrics. Then, we build univariate prediction models to investigate the correlations between slice-based cohesion metrics and post-release fault-proneness. Finally, we build multivariate prediction models to examine the effectiveness of slice-based cohesion metrics in effort-aware post-release fault-proneness prediction when used alone or used together with the baseline code and process metrics. In order to obtain comprehensive performance evaluations, we evaluate the effectiveness of effort-aware post-release fault-proneness prediction under the following three prediction settings: cross-validation, across-version prediction, and across-project prediction. More specifically, cross-validation is performed within the same version of a project, i.e. predicting faults in one subset using a model trained on the other complementary subsets. Across-version prediction uses a model trained on earlier versions to predict faults in later versions within the same project, while across-project prediction uses a model trained on one project to predict faults in another project. The subject projects in our study consist of five well-known open-source C projects: Bash, Gcc-core, Gimp, Subversion, and Vim. We use a mature commercial tool called Understand[1] to collect the baseline code and process metrics and use a powerful source code analysis tool called Frama-C to collect slice-based cohesion metrics [57]. Based on the data collected from these five projects, we attempt to answer the following four research questions:

- *RQ1*. Are slice-based cohesion metrics redundant with respect to the most commonly used code and process metrics?
- *RQ2*. Are slice-based cohesion metrics statistically significantly correlated to post-release fault-proneness?
- *RQ3*. Are slice-based cohesion metrics more effective than the most commonly used code and process metrics in effort-aware post-release fault-proneness prediction?
- *RQ4*. When used together with the most commonly used code and process metrics, can slice-based cohesion metrics significantly improve the effectiveness of effort-aware post-release fault-proneness prediction?

The purpose of RQ1 and RQ2 investigates whether slice-based cohesion metrics are potentially useful post-release fault-proneness predictors. The purpose of RQ3 and RQ4 investigates whether slice-based cohesion metrics can lead to significant improvements in effort-aware post-release fault-proneness prediction. These research questions are critically important to both software researchers and practitioners, as they help to answer whether slice-based cohesion metrics are of practical value in view of the extra cost involved in data collection. However, little is currently known on this subject. Our study attempts to fill this gap by a comprehensive investigation into the actual usefulness of slice-based cohesion metrics in the context of effort-aware post-release fault-proneness prediction.

The contributions of this paper are listed as follows. First, we compare slice-based cohesion metrics with the most commonly used code and process metrics including size, structural complexity, Halstead's software science metrics, and code churn metrics. The results show that slice-based cohesion metrics measure essentially different quality information than the baseline code and process metrics measure. This indicates that slice-based cohesion metrics are not redundant with respect to the most commonly used code and process metrics. Second, we validate the correlations between slice-based cohesion metrics and fault-proneness. The results show that most slice-based cohesion metrics are statistically related to fault-proneness in an expected direction. Third, we analyze the effectiveness of slice-based cohesion metrics in effort-aware post-release fault-proneness prediction compared with the most commonly used code and process metrics. The results, somewhat surprisingly, show that slice-based cohesion metrics in general do not outperform the most commonly used code and process metrics. Fourth, we investigate whether the combination of slice-based cohesion metrics with the most commonly used code and process metrics provide better results in predicting fault-proneness. The results show that the inclusion of slice-based cohesion metrics can produce a statistically significant improvement of the effectiveness in effort-aware post-release fault-proneness prediction under any of the three prediction settings. In particular, in the ranking scenario, when testing or inspecting 20 percent of the code of a system, slice-based cohesion metrics lead to a moderate to large improvement (Cliff's δ: 0.33-1.00), regardless of which prediction setting is considered. In the classification scenario, they lead to a moderate to large improvement (Cliff's δ: 0.31-0.77) in most systems under cross-validation and lead to a large improvement (Cliff's δ: 0.55-0.72) under across-version prediction. In summary, these results reveal that the improvement is practically important for practitioners, which is worth the relatively high time cost for collecting slice-based cohesion metrics. In other words, for practitioners, slice-based cohesion metrics are of practical value in the context of effort-aware post-release fault-proneness prediction. Our study provides valuable data in an important area for which otherwise there is limited experimental data available.

The rest of this paper is organized as follows. Section 2 introduces slice-based cohesion metrics and the most commonly used code and process metrics that we will investigate. Section 3 gives the research hypotheses on slice-based cohesion metrics, introduces the investigated dependent and independent variables, presents the employed modeling technique, and describes the data analysis methods. Section 4 describes the experimental setup in our study, including the data sources and the method we used to

[1] www.scitools.com

collect the experimental data sets. Section 5 reports in detail the experimental results. Section 6 examines the threats to validity of our study. Section 7 discusses the related work. Section 8 concludes the paper and outlines directions for future work.

2 The Metrics

In this section, we first describe slice-based cohesion metrics investigated in this study. Then, we describe the most commonly used code and process metrics that will be compared against when analyzing the actual usefulness of slice-based cohesion metrics in effort-aware post-release fault-proneness prediction.

2.1 Slice-Based Cohesion Metrics

The origin of slice-based cohesion metrics can be traced back to Weiser, who used backward slicing to describe the concepts of *coverage, overlap, and tightness* [9], [19]. A *backward slice* of a module at statement n with respect to variable v is the sequence of all statements and predicates that might affect the value of v at n [9], [19]. For a given module, Weiser first sliced on every variable where it occurred in the module. Then, Weiser computed *Coverage* as the ratio of average slice size to program size, *Overlap* as the average ratio of non-unique to unique statements in each slice, and *Tightness* as the percentage of statements common in all slices. As stated by Ott and Bieman [10], however, Weiser "did not identify actual software attributes these metrics might meaningfully measure", although such metrics were helpful for observing the structuring of a module.

Longworth [7] demonstrated that *Coverage*, a modified definition of *Overlap* (i.e. the average ratio of the size of non-unique statements to slice size), and *Tightness* could be used as cohesion metrics of a module. In particular, Longworth sliced on every variable once at the end point of the module to obtain *end slices* (i.e. *backward slices* computed from the end of a module) and then used them to compute these metrics. Later, Ott and Thuss [3] improved the behavior of slice-based cohesion metrics through the use of *metric slices* on output variables. A metric slice takes into account both the *uses* and *used by* data relationships [3]. More specifically, a metric slice with respect to variable v is the union of the backward slice with respect to v at the end point of the module and the *forward slice* computed from the definitions of v in the backward slice. A *forward slice* of a module at statement n with respect to variable v is the sequence of all statements and predicates that might be affected by the value of v at n. Ott and Thuss argued that the purpose of executing a module was indicated by its output variables, including function return values, modified global variables, printed variables, and modified reference parameters. Furthermore, the slices on the output variables of a module capture the specific computations for the tasks that the module performs. Therefore, we could use the relationships among the slices on output variables to investigate whether the module's tasks are related, i.e. whether the module is cohesive. They redefined *Overlap* as the average ratio of the slice interaction size to slice size and added *MinCoverage* and *MaxCoverage* to the metrics suite. *MinCoverage* and *MaxCoverage* are respectively the ratio of the size of the smallest slice to the module size and the ratio of the size of the largest slice to the module size. Consequently, the slice-based cohesion metrics suite proposed by Ott and Thuss consists of five metrics: *Coverage*, *Overlap*, *Tightness*, *MinCoverage*, and *MaxCoverage*. Note that these metrics are computed at the statement level, i.e. statements are the basic unit of metric slices. Ott and Bieman [20] refined the concept of metric slices to use data tokens (i.e. the definitions of and references to variables and constants) rather than statements as the basic unit of which slices are composed of. They called such slices *data slices*. More specifically, a data slice for a variable v is the sequence of all data tokens in the statements that comprise the metric slice of v. This leads to five slice-based data-token-level cohesion metrics.

Bieman and Ott [4] used data slices to develop three cohesion metrics: *SFC* (strong functional cohesion), *WFC* (weak functional cohesion), and *A* (Adhesiveness). They defined the slice abstraction of a module as the set of data slices with respect to its output variables. In particular, a data token is called a "glue token" if it lies on more than one data slices, and is called a "super-glue token" if it lies on all data slices in the slice abstraction. As such, *SFC* is defined as the ratio of the number of super-glue tokens to the total number of data tokens in the module. *WFC* is defined as the ratio of the number of glue tokens to the total number of data tokens in the module. *A* is defined as the average adhesiveness for all the data tokens in the module. The adhesiveness of a data token is the relative number of slices that it glues together. If a data token is a glue token, its adhesiveness is the ratio of the number of slices that it appears in to the total number of slices. Otherwise, its adhesiveness is zero. Indeed, *SFC* is equivalent to the data-token-level *Tightness* metric and *A* is equivalent to the data-token-level *Coverage* metric proposed by Ott and Bieman [20].

Counsell et al. [5] proposed a cohesion metric called normalized Hamming distance (*NHD*) based on the concept of slice occurrence matrix. For a given module, the slice occurrence matrix has columns indexed by its output variables and rows indexed by its statements. The (i, j)th entry of the matrix has a value of 1 if the ith statement is in the end slice with respect to the jth output variable and otherwise 0. In this matrix, each row is called a slice occurrence vector. *NHD* is defined as the ratio of the total actual slice agreement between rows to the total possible agreement between rows in the matrix. The slice agreement between two rows is the number of places in which the slice occurrence vectors of the two rows are equal.

Dallal [8] used a data-token-level slice occurrence matrix to develop a cohesion metric called similarity-based functional cohesion metric (*SBFC*). For a given module, the data-token-level slice occurrence matrix has columns indexed by its output variables and rows indexed by its data tokens. The (i, j)th entry of the matrix has a value of 1 if the ith data token is in the end slice with respect to the jth output variable and otherwise 0. *SBFC* is defined as the average degree of the normalized similarity between columns. The normalized similarity between a pair of columns is the ratio of the number of entries where both columns have a value of 1 to the total number of rows in the matrix.

Table 1 summarizes the formal definitions, descriptions, and sources of the slice-based cohesion metrics that will be investigated in this study. In this table, for a given module

TABLE 1
Definitions of Slice-Based Cohesion Metrics

Metric	Definition	Description	Source
Coverage	$Coverage = \frac{1}{\lvert V_o\rvert}\sum_{i=1}^{\lvert V_0\rvert}\frac{\lvert SL_i\rvert}{length(M)}$	The extent to which the slices cover the module (measured as the ratio of the mean slice size to the module size)	[3], [20]
MaxCoverage	$MaxCoverage = \frac{1}{length(\mathrm{M})}\max_i\lvert SL_i\rvert$	The extent to which the largest slice covers the module (measured as the ratio of the size of the largest slice to the module size)	
MinCoverage	$MinCoverage = \frac{1}{length(\mathrm{M})}\min_i\lvert SL_i\rvert$	The extent to which the smallest slice covers the module (measured as the ratio of the size of the smallest slice to the module size)	
Overlap	$Overlap = \frac{1}{\lvert V_o\rvert}\sum_{i=1}^{\lvert V_0\rvert}\frac{\lvert SL_{int}\rvert}{\lvert SL_i\rvert}$	The extent to which slices are interdependent (measured as the average ratio of the size of the "cohesive section" to the size of each slice)	
Tightness	$Tightness = \frac{\lvert SL_{int}\rvert}{length(M)}$	The extent to which all the slices in the module belong together (measured as the ratio of the size of the "cohesive section" to the module size)	
SFC	$SFC = \frac{\lvert SG(SA(M))\rvert}{\lvert tokens(M)\rvert}$	The extent to which all the slices in the module belong together (measured as the ratio of the number of super-glue tokens to the total number of data tokens of the module)	[4]
WFC	$WFC = \frac{\lvert G(SA(M))\rvert}{\lvert tokens(M)\rvert}$	The extent to which the slices in the module belong together (measured as the ratio of the number of glue tokens to the total number of data tokens of the module)	
A	$A = \frac{\sum_{t\in G(SA(M))} slices\ containing\ t}{\lvert tokens(M)\rvert\times\lvert SA(M)\rvert}$	The extent to which the glue tokens in the module are adhesive (measured as the ratio of the amount of the adhesiveness to the total possible adhesiveness)	
NHD	$NHD = 1 - \frac{2}{lk(k-1)}\sum_{j=1}^{l} c_j\big(k - c_j\big)$	The extent to which the statements in the slices are the same (measured as the ratio of the total slice agreement between rows to the total possible agreement between rows in the statement-level slice occurrence matrix of the module)	[5]
SBFC	$SBFC = \begin{cases} 1 & if\lvert V_o\rvert = 1 \\ \frac{\sum_{t=1}^{\lvert tokens(M)\rvert} x_i(x_i-1)}{\lvert tokens(M)\rvert\times\lvert V_o\rvert\times(\lvert V_o\rvert-1)} & otherwise \end{cases}$	The extent to which the slices are similar (measured as the average degree of the normalized similarity between columns in the data-token-level slice occurrence matrix of the module)	[8]

M, V_o denotes the set of its output variables, *length(M)* denotes its size, *SA(M)* denotes its slice abstraction, and *tokens(M)* denotes the set of its data tokens. SL_i is the slice obtained for $v_i \in V_o$ and SL_{int} (called "cohesive section" by Harman et al. [6]) is the intersection of SL_i over all $v_i \in V_o$. In particular, *G(SA(M))* and *SG(SA(M))* are respectively the set of glue tokens and the set of super-glue tokens. In the definition of *NHD*, *k* is the number of statements, *l* is the number of output variables, and c_i is the number of 1s in the *j*th column of the statement-level slice occurrence matrix. In the definition of *SBFC*, x_i is the number of 1s in the *i*-th row of the data-token-level slice occurrence matrix.

Note that all the slice-based cohesion metrics can be computed at the statement or data-token level, although some of them are originally defined at either the statement level or the data-token level. The data-token level is at a finer granularity than the statement level since a statement might contain a number of data tokens. We next use an example function *fun* shown in Table 2 to illustrate the computations of the slice-based cohesion metrics at the data-token level. In Table 2: (1) the first column lists the statement number (excluding non-executable statements such as blank statements, "{", and "}"); (2) the second column lists the code of the example function; (3) the third to fifth columns respectively list the end slices for the largest, smallest, and range variables; (4) the sixth to eighth columns respectively list the forward slices from the definitions of the largest, smallest, and range variables in the backward slices; and (5) the ninth to eleventh columns list the metric slices for the largest, smallest, and range variables. Here, a vertical bar "|" in the last nine columns denotes that the indicated statement is part of the corresponding slice for the named output variable. This example function determines the smallest, the largest, and the range of an array, which is a modified version of the example module used by Longworth [7]. For this example, V_o consists of largest, smallest, and range. The former two variables are the modified reference parameters and the latter is the function return value. Table 3 shows the data-token level slice occurrence matrix of the *fun* function under end slices and metric slices, where T_i indicates the *i*th data token for *T* in the function.

Table 4 shows the computations of twenty data-token-level slice-based cohesion metrics. In this table, the second to eleventh rows show the computations for end-slice-based cohesion metrics and the 12th to 21st rows show the computations for metric-slice-based cohesion metrics. As can be seen, end-slice-based metrics indicate with typical values around 0.5 or 0.6, while metric-slice-based metrics indicate with typical values around 0.7 or 0.8. In particular, for each cohesion metric (except *MaxCoverage*), the metric-slice-based version has a considerably larger value than the corresponding end-slice-based version. When looking at the

TABLE 2
End Slice Profile and Metric Slice Profile for Function *Fun*

Line	Code	End slice			Forward slice			Metric slice		
		largest	smallest	range	largest	smallest	range	largest	smallest	range
	int fun(									
1	int A[]	|	|	|				|	|	|
2	int size,	|	|	|				|	|	|
3	int *largest,	|		|				|		|
4	int *smallest)	|	|	|				|	|	|
	{									
5	int i;	|	|	|				|	|	|
6	int range;			|						|
7	i = 1;	|	|	|				|	|	|
8	range = 0;									
9	*smallest = A[0];	|	|	|		|		|	|	|
10	*largest = *smallest;	|		|	|	|		|	|	|
11	while(i < size) {	|	|	|				|	|	|
12	if(*smallest > A[i])		|	|		|			|	|
13	*smallest = A[i];		|	|		|			|	|
14	if(*largest < A[i])	|		|	|	|		|	|	|
15	*largest = A[i];	|		|	|	|		|	|	|
16	i++;	|	|	|				|	|	|
	}									
17	range = *largest - *smallest;			|	|	|	|	|	|	|
18	return range;			|	|	|	|	|	|	|
	}									

Data tokens included in the end slice for the variable smallest are indicated by the underline.

example function *fun* shown in Table 2, we find that, except an unnecessary initialization statement (statement 8 in Table 2: range = 0;), all the rest statements are all related to the computation of the final outputs. In other words, intuitively, this function has a high cohesion. In this sense, when measuring its cohesion, it appears that metric-slice-based cohesion metrics are more accurate than end-slice-based cohesion metrics.

As mentioned above, *Coverage*, *MaxCoverage*, *MinCoverage*, *Overlap*, *Tightness*, *SFC*, *WFC*, and *A* are originally based on metric slices [4], [6], [13], [15], [16], [17], [18], [21]. However, *NHD* and *SBFC* are originally based on end slices. In this study, we will use metric slices to compute all the cohesion metrics. In particular, we will use metric-slice-based cohesion metrics at the data-token level to investigate the actual usefulness of slice-based cohesion metrics in effort-aware post-release fault-proneness prediction. The reason for choosing the data-token level rather than the statement level is that the former is at a finer granularity. Previous studies suggested that software metrics at a finer granularity would accordingly have a higher discriminative power and hence may be more useful for fault-proneness prediction [62], [63]. Note that, at the data-token level, *SFC* is equivalent to *Tightness* and *A* is equivalent to *Coverage*. Therefore, in the subsequent analysis, only the following eight metric-slice-based cohesion metrics will be examined: *Coverage*, *MaxCoverage*, *MinCoverage*, *Overlap*, *Tightness*, *WFC*, *NHD*, and *SBFC*. During our analysis, a function is regarded as a module and the output variables of a function consist of the function return value, modified global variables, modified reference parameters, and standard outputs by the function.

2.2 The Most Commonly Used Code and Process Metrics

In this study, we employ the most commonly used code and process metrics as the baseline metrics to analyze the actual usefulness of slice-based cohesion metrics in effort-aware post-release fault-proneness prediction. As shown in Table 5, the baseline code and process metrics cover 16 product metrics and three process metrics. These 16 product metrics consist of 1 size metric, 11 structural complexity metrics, and 4 software science metrics. The size metric *SLOC* simply counts the non-blank non-commentary source lines of code (SLOC) in a function. There is a common belief that a function with a larger size tends to be more fault-prone [22], [23], [24], [25]. The structural complexity metrics, including the well-known McCabe's Cyclomatic complexity metrics, assume that a function with complex control flow structure is likely to be fault-prone [26], [27], [28], [29]. The Halstead's software science metrics estimate reading complexity based on the counts of operators and operands, in which a function hard to read is assumed to be fault-prone [30]. Note that we

TABLE 3
Data-Token Level Slice Occurrence Matrix with Respect to End Slice Profile and Metric Slice Profile

		End slice			Metric slice		
Line	Token	largest	smallest	range	largest	smallest	range
1	A$_1$	1	1	1	1	1	1
2	size$_1$	1	1	1	1	1	1
3	largest$_1$	1	0	1	1	0	1
4	smallest$_1$	1	1	1	1	1	1
5	i$_1$	1	1	1	1	1	1
6	range$_1$	0	0	1	0	0	1
7	i$_2$	1	1	1	1	1	1
7	1$_1$	1	1	1	1	1	1
8	range$_2$	0	0	0	0	0	0
8	0$_1$	0	0	0	0	0	0
9	smallest$_2$	1	1	1	1	1	1
9	A$_2$	1	1	1	1	1	1
9	0$_2$	1	1	1	1	1	1
10	largest$_2$	1	0	1	1	1	1
10	smallest$_3$	1	0	1	1	1	1
11	i$_3$	1	1	1	1	1	1
11	size$_2$	1	1	1	1	1	1
12	smallest$_4$	0	1	1	0	1	1
12	A$_3$	0	1	1	0	1	1
12	i$_4$	0	1	1	0	1	1
13	smallest$_5$	0	1	1	0	1	1
13	A$_4$	0	1	1	0	1	1
13	i$_5$	0	1	1	0	1	1
14	largest$_3$	1	0	1	1	1	1
14	A$_5$	1	0	1	1	1	1
14	i$_6$	1	0	1	1	1	1
15	largest$_4$	1	0	1	1	1	1
15	A$_6$	1	0	1	1	1	1
15	i$_7$	1	0	1	1	1	1
16	i$_8$	1	1	1	1	1	1
17	range$_3$	0	0	1	1	1	1
17	largest$_5$	0	0	1	1	1	1
17	smallest$_6$	0	0	1	1	1	1
18	range$_4$	0	0	1	1	1	1

do not include the other Halstead's software science metrics such as N, n, V, D, and E [30]. The reason is that these metrics are fully based on $n1$, $n2$, $N1$, and $N2$ (for example $n = n1 + n2$). Consequently, they are highly correlated with $n1$, $n2$, $N1$, and $N2$. When building multivariate prediction models, highly correlated predictors will lead to a high multicollinearity and hence might lead to inaccurate coefficient estimates [61]. Therefore, our study only takes into account $n1$, $n2$, $N1$, and $N2$. The process metrics consist of three relative code churn metrics, i.e. the normalized number of added, deleted, and modified source lines of code. These code churn metrics assume that a function with more added, deleted, or modified code would have a higher possibility of being fault-prone.

The reasons for choosing these baseline code and process metrics in this study are three-fold. First, they are widely used product and process metrics in both industry and academic research [22], [23], [24], [25], [26], [27], [28], [29], [50], [52], [54], [55], [64]. Second, they can be automatically and cheaply collected from source code even for very large software systems. Third, many studies show that they are useful indicators for fault-proneness prediction [26], [27], [28], [50], [52], [54], [55], [64]. In the context of effort-aware post-release fault-proneness prediction, we believe that slice-based cohesion metrics are of practical value only if: (1) they have a significantly better fault-proneness prediction ability than the baseline code and process metrics; or (2) they can significantly improve the performance of fault-proneness prediction when used together with the baseline code and process metrics. This is especially true when considering the expenses for collecting slice-based cohesion metrics. As Meyers and Binkley stated [13], slicing techniques and tools are now mature enough to allow an intensive empirical investigation. In our study, we use Frama-C, a well-known open-source static analysis tool for C programs [57], to collect slice-based cohesion metrics. Frama-C provides scalable and sound software analyses for C programs, thus allowing accurate collection of slice-based cohesion metrics on industrial-size systems [57].

3 Research Methodology

In this section, we first give the research hypotheses relating slice-based cohesion metrics to the most commonly used code and process metrics and to fault-proneness. Then, we describe the investigated dependent and independent variables, the employed modeling technique, and the data analysis methods.

3.1 Research Hypotheses

The first research question (RQ1) of this study investigates whether slice-based cohesion metrics are redundant when compared with the most commonly used code and process metrics. It is widely believed that software quality cannot be measured using only a single dimension [29]. As stated in Section 2.2, the most commonly used code and process metrics measure software quality from size, control flow structure, and cognitive psychology perspectives. However, slice-based cohesion metrics measure software quality from the perspective of cohesion, which are based on control-/data-flow dependence information among statements. Our conjecture is that, given the nature of the information and counting mechanism employed by slice-based cohesion metrics, they should capture different underlying dimensions of software quality than the most commonly used code and process metrics capture. From this reasoning, we set up the following null hypothesis $\boldsymbol{H1_0}$ and alternative hypothesis $\boldsymbol{H1_A}$ for RQ1:

$\boldsymbol{H1_0}$. *Slice-based cohesion metrics do not capture additional dimensions of software quality compared with the most commonly used code and process metrics.*

$\boldsymbol{H1_A}$. *Slice-based cohesion metrics capture additional dimensions of software quality compared with the most commonly used code and process metrics.*

The second research question (RQ2) of this study investigates whether slice-based cohesion metrics are statistically related to post-release fault-proneness. In the software engineering literature, there is a common belief that low cohesion indicates an inappropriate design [1], [2]. Consequently, a function with low cohesion is more likely to be fault-prone than a function with high cohesion [1], [2]. From Section 2.1, we can see that slice-based cohesion metrics leverage the commonality among the slices with respect to different output variables of a function to quantify its cohesion. Existing studies showed that they provided an

TABLE 4
Example Metrics Computations at the Data-Token Level

Type	Metric	Computation	Value
End slice	Coverage	$= 1/3 \times (21/34 + 18/34 + 32/34)$	$= 0.696$
	MaxCoverage	$= 32/34$	$= 0.941$
	MinCoverage	$= 18/34$	$= 0.529$
	Overlap	$= 1/3 \times (12/21 + 12/18 + 12/32)$	$= 0.538$
	Tightness	$= 12/34$	$= 0.353$
	SFC	$= 12/34$	$= 0.353$
	WFC	$= 27/34$	$= 0.794$
	A	$= (21 + 18 + 32)/(3 \times 34)$	$= 0.696$
	SBFC	$= (12 \times 3 \times 2 + 15 \times 2 \times 1)/(34 \times 3 \times 2)$	$= 0.500$
	NHD	$= 1-2/(3 \times 34 \times 33) \times (21 \times 13 + 18 \times 16 + 32 \times 2)$	$= 0.629$
Metric slice	Coverage	$= 1/3 \times (25/34 + 30/34 + 32/34)$	$= 0.853$
	MaxCoverage	$= 32/34$	$= 0.941$
	MinCoverage	$= 25/34$	$= 0.735$
	Overlap	$= 1/3 \times (24/25 + 24/30 + 24/32)$	$= 0.837$
	Tightness	$= 24/34$	$= 0.706$
	SFC	$= 24/34$	$= 0.706$
	WFC	$= 31/34$	$= 0.912$
	A	$= (25 + 30 + 32)/(3 \times 34)$	$= 0.853$
	SBFC	$= (24 \times 3 \times 2 + 7 \times 2 \times 1)/(34 \times 3 \times 2)$	$= 0.775$
	NHD	$= 1-2/(3 \times 34 \times 33) \times (25 \times 9 + 30 \times 4 + 32 \times 2)$	$= 0.757$

excellent quantitative measure of function cohesion [5], [13]. In particular, for each of the investigated slice-based cohesion metrics, a large value indicates a high cohesion. From this reasoning, we set up the following null hypothesis $\boldsymbol{H2_0}$ and alternative hypothesis $\boldsymbol{H2_A}$ for RQ2:

$\boldsymbol{H2_0}$. *There is no significant correlation between slice-based cohesion metrics and post-release fault-proneness.*

$\boldsymbol{H2_A}$. *There is a significant correlation between slice-based cohesion metrics and post-release fault-proneness.*

The third research question (RQ3) of this study investigates whether slice-based cohesion metrics predict post-release fault-prone functions more accurately than the most commonly used code and process metrics do. From Table 5, we can see that the most commonly used code and process metrics are based on either simple syntactic information or control flow structure information among statements in a function. In contrast, slice-based cohesion metrics make use of the semantic dependence information among the statements in a function. In other words, they are based on program behaviors as captured by program slices. In this sense, slice-based cohesion metrics provide a higher level quantification of software quality than the most commonly used code and process metrics. Consequently, it is reasonable to expect

TABLE 5
The Most Commonly Used Code and Process Metrics (i.e. the Baseline Metrics in This Study)

Category	Characteristic	Metric	Description
Product	Size	SLOC	Source lines of code in a function (excluding blank lines and comment lines)
	Structural complexity	FANIN	Number of calling functions plus global variables read
		FANOUT	Number of calling functions plus global variables set
		NPATH	Number of possible paths, not counting abnormal exits or gotos
		Cyclomatic	Cyclomatic complexity
		CyclomaticModified	Modified cyclomatic complexity
		CyclomaticStrict	Strict cyclomatic complexity
		Essential	Essential complexity
		Knots	Measure of overlapping jumps
		Nesting	Maximum nesting level of control constructs
		MaxEssentialKnots	Maximum Knots after structured programming constructs have been removed
		MinEssentialKnots	Minimum Knots after structured programming constructs have been removed
	Software science	n1	Total number of distinct operators of a function
		n2	Total number of distinct operands of a function
		N1	Total number of operators of a function
		N2	Total number of operands of a function
Process	Code churn	Added	Added source lines of code, normalized by function size
		Deleted	Deleted source lines of code, normalized by function size
		Modified	Modified source lines of code, normalized by function size

that slice-based cohesion metrics are more closely related to fault-proneness than the most commonly used code and process metrics. From this expectation, we set up the following null hypothesis ***H3*$_0$** and alternative hypothesis ***H3*$_A$** for RQ3:

***H3*$_0$**. *Slice-based cohesion metrics are not more effective in effort-aware post-release fault-proneness prediction than the most commonly used code and process metrics.*

***H3*$_A$**. *Slice-based cohesion metrics are more effective in effort-aware post-release fault-proneness prediction than the most commonly used code and process metrics.*

The fourth research question (RQ4) of this study investigates whether the model built with slice-based cohesion metrics and the most commonly used code and process metrics together has a better ability to predict post-release fault-proneness than the model built with the most commonly used code and process metrics alone. This issue is indeed raised by the null hypothesis ***H1*$_0$**. If the null hypothesis ***H1*$_0$** is rejected, it means that slice-based cohesion metrics capture different underlying dimensions of software quality that are not captured by the most commonly used code and process metrics. In this case, we will naturally conjecture that combining slice-based cohesion metrics with the most commonly used code and process metrics should give a more complete indication of software quality. Consequently, the combination of slice-based cohesion metrics with the most commonly used code and process metrics will form a better indicator of post-release fault-proneness than the combination of the most commonly used code and process metrics alone. From this reasoning, we set up the following null hypothesis ***H4*$_0$** and alternative hypothesis ***H4*$_A$** for RQ4:

***H4*$_0$**. *The combination of slice-based cohesion metrics with the most commonly used code and process metrics are not more effective in effort-aware post-release fault-proneness prediction than the combination of the most commonly used code and process metrics.*

***H4*$_A$**. *The combination of slice-based cohesion metrics with the most commonly used code and process metrics are more effective in effort-aware post-release fault-proneness prediction than the combination of the most commonly used code and process metrics.*

3.2 Variable Description

The dependent variable in this study is a binary variable Y that can take on only one of two different values. In the following, let the values be 0 and 1. Here, $Y = 1$ represents that the corresponding function has at least one post-release faults and $Y = 0$ represents that the corresponding function has no post-release fault. In this paper, we use a modeling technique called logistic regression (described in Section 3.3) to predict the probability of $Y = 1$. The probability of $Y = 1$ indeed indicates post-release fault-proneness, i.e. the extent of a function being post-release faulty. As stated by Nagappan et al. [66], for the users, only post-release failures matter. It is hence essential to predict post-release fault-proneness of functions in a system in practice, as it enables developers to take focused preventive actions to improve quality in a cost-effective way. Indeed, much effort has been devoted to post-release fault-proneness prediction [27], [34], [36], [42], [54], [60], [64], [65], [66].

The independent variables in this study consist of two categories of metrics: (i) the most commonly used 19 code and process metrics, and (ii) eight slice-based cohesion metrics. All these metrics are collected at the function level. The objective of this study is to empirically investigate the actual usefulness of slice-based cohesion metrics in the context of effort-aware post-release fault-proneness prediction, especially when compared with the most commonly used code and process metrics. With these independent variables, we are able to test the four null hypotheses described in Section 3.1.

3.3 Modeling Technique

Logistic regression is a standard statistical modeling technique in which the dependent variable can take two different values [28]. It is suitable for building fault-proneness prediction models because the functions under consideration are divided into two categories: faulty and not-faulty. Let $Pr(Y = 1|X_1, X_2, \ldots, X_n)$ represent the probability that the dependent variable $Y = 1$ given the independent variables X_1, X_2,..., and X_n (i.e. the metrics in this study). Then, a multivariate logistic regression model assumes that $Pr(Y = 1|X_1, X_2 \ldots, X_n)$ is related to $X_1, X_2, \ldots, X_n$ by the following equation:

$$Pr(Y = 1|X_1, X_2, \ldots X_n) = \frac{e^{\alpha+\beta_1 X_1+\beta_2 X_2+\ldots\beta_n X_n}}{1 + e^{\alpha+\beta_1 X_1+\beta_2 X_2+\ldots\beta_n X_n}},$$

where α and β_is are the regression coefficients and can be estimated through the maximization of a log-likelihood. Odds ratio is the most commonly used measure to quantify the magnitude of the correlation between the independent and dependent variables in a logistic regression model. For a given independent variable X_i, the odds that $Y = 1$ at $X_i = x$ denotes the ratio of the probability that $Y = 1$ to the probability that $Y = 1$ at $X_i = x$, i.e.

$$Odds(Y = 1|X_i = x) = \frac{\Pr(Y = 1|\ldots, X_i = x, \ldots)}{1 - \Pr(Y = 1|\ldots, X_i = x, \ldots)}.$$

In this study, similar to [33], we use ΔOR, the odds ratio associated with one standard deviation increase, to provide an intuitive insight into the impact of the independent variable X_i:

$$\Delta OR(X_i) = \frac{Odds(Y = 1|X_i = x + \sigma_i)}{Odds(Y = 1|X_i = x)} = e^{\beta_i \sigma_i},$$

where β_i and σ_i are respectively the regression coefficient and the standard deviation of the variable X_i. $\Delta OR(X_i)$ can be used to compare the relative magnitude of the effects of different independent variables, as the same unit is used [42]. $\Delta OR(X_i) > 1$ indicates that the independent variable is positively associated with dependent variable. $\Delta OR(X_i) = 1$ indicates that there is no such correlation. $\Delta OR(X_i) < 1$ indicates that there is a negative correlation. The univariate logistic regression model is a special case of the multivariate logistic regression model, where there is only one independent variable.

3.4 Data Analysis Method

In the following, we describe the data analysis method for testing the four null research hypotheses.

3.4.1 Principal Component Analysis for RQ1

In order to answer RQ1, we use principal component analysis to determine whether slice-based cohesion metrics capture different underlying dimensions of software quality than the most commonly used code and process metrics. PCA is a powerful statistical technique used to identify the underlying, orthogonal dimensions that explain the relations among the independent variables in a data set. These dimensions are called principal components (PCs), which are linear combinations of the standardized independent variables. In our study, for each data set, we use the following method to determine the corresponding number of PCs. First, the stopping criterion for PCA is that all the eigenvalues for each new component are greater than zero. Second, we apply the varimax rotation to PCs to make the mapping of the independent variables to components clearer where the variables have either a very low or a very high loading. This helps identify the variables that are strongly correlated and indeed measure the same property, though they may purport to capture different properties. Third, after obtaining the rotated component matrix, we map each independent variable to the component having the maximum loading. Fourth, we only retain the components to which at least one independent variable is mapped. In our context, the null hypothesis $\boldsymbol{H1_0}$ corresponding to RQ1 will be rejected when the result of PCA shows that slice-based cohesion metrics define new PCs of their own compared with the most commonly used code and process metrics.

3.4.2 Univariate Logistic Regression Analysis for RQ2

In order to answer RQ2, we use univariate logistic regression to examine whether each slice-based cohesion metric is negatively related to post-release fault-proneness at the significant level α of 0.10. From a scientific perspective, it is often suggested to work at the α level 0.05 or 0.01. However, the choice of a particular level of significance is ultimately a subjective decision and other levels such as $\alpha = 0.10$ are also common [51]. In this paper, the minimum significance level for rejecting a null hypothesis is set at $\alpha = 0.10$, as we are aggressively interested in revealing unclosed correlations between metrics and fault-proneness. When performing univariate analysis, we employ the Cook's distance to identify influential observations. For an observation, its Cook's distance is a measure of how far apart the regression coefficients are with and without this observation included. If an observation has a Cook's distance equal to or larger than 1, it is regarded as an influential observation and is hence excluded for the analysis [32]. Furthermore, for each metric, we use ΔOR, the odds ratio associated with one standard deviation increase in the metric, to quantify its effect on fault-proneness [33]. This allows us to compare the relative magnitude of the effects of individual metrics on post-release fault-proneness. Note that previous studies reported that module size (i.e. function size in this study) might have a potential confounding effect on the relationships between software metrics and fault-proneness [43], [53]. In other words, module size may falsely obscure or accentuate the true correlations between software metrics and fault-proneness. Therefore, there is a need to remove the potentially confounding effect of module size in order to understand the essence that a metric measures [53]. In this study, we first apply the linear regression method proposed by Zhou et al. [53] to remove the potentially confounding effect of function size. After that, we use univariate logistic regression to examine the correlations between the cleaned metrics and fault-proneness. For each metric, the null hypothesis $\boldsymbol{H2_0}$ corresponding to RQ2 will be rejected if the result of univariate logistic regression is statistically significant at the significant level of 0.10.

3.4.3 Multivariate Logistic Regression Analysis for RQ3 and RQ4

In order to answer RQ3 and RQ4, we perform a stepwise variable selection procedure to build three types of multivariate logistic regression models: (1) the "B" model (using only the most commonly used code and process metrics); (2) the "S" model (using only slice-based cohesion metrics); and (3) the "B+S" model (using all the metrics). As suggested by Zhou et al. [53], before building the multivariate logistic regression models, we remove the confounding effect of function size (measured by *SLOC*). In addition, many metrics used in this study are defined similarly with each other. For example, *CyclomaticModified* and *CyclomaticStrict* are the revised *Cyclomatic* complexity versions. These highly correlated predictors may lead to a high multicollinearity and hence inaccurate coefficient estimates in a logistic regression model [61]. Variance inflation factor (VIF) is a widely used indicator of multicollinearity. In this study, we use the recommended cut-off value 10 to deal with multicollinearity in a regression model [59]. If an independent variable has a VIF value larger than 10, it will be removed from the multivariate regression model. More specifically, we use the following algorithm *BUILD-MODEL* to build the multivariate logistic regression models. As can be seen, when building a multivariate model, our algorithm takes into account: (1) the confounding effect of function size; (2) the multicollinearity among the independent variables; and (3) the influential observations.

Algorithm 1. BUILD-MODEL

```
Input dataset $D$($X$: set of independent variables, $Y$: dependent variable)
Step
1: Remove the confounding effect of function size from each
   independent variable in $X$ for $D$. [53]
2: Use the backward stepwise variable selection method to
   build the logistic regression model $M$ on $D$.
3: Calculate the variance inflation factors for all independent
   variables in the model $M$.
4: If all the $VIF$s are less than or equal to 10, goto step 6; other-
   wise, goto step 5.
5: Remove the variable $x_i$ with the largest $VIF$ from $X$, and goto
   step 2.
6: Calculate the Cook's distance for all the observations in $D$. If
   the maximum Cook's distance is less than or equal to 1, then
   goto step 8; otherwise, goto step 7.
7: Update D by removing the observations whose Cook's dis-
   tances are equal to or larger than 1. Goto step 2.
8: Return the model $M$.
```

After building the above models, we compare the prediction effectiveness of the following two pairs of models: “S” vs. “B” and “B+S” versus “B”. To obtain an adequate and realistic comparison, we use the prediction effectiveness data generated from the following three methods:

- *Cross-validation*. Cross-validation is performed within the same version of a project, i.e. predicting faults in one subset using a model trained on the other complementary subsets. In our study, for a given project, we use 30 times three-fold cross-validation to evaluate the effectiveness of the prediction models. More specifically, at each three-fold cross-validation, we randomly divide the data set into three parts of approximately equal size. Each part is used to compute the effectiveness for the prediction models built on the remainder of the data set. The entire process is then repeated 30 times to alleviate possible sampling bias in random splits. Consequently, each model has $30 \times 3 = 90$ prediction effectiveness values. Note that we choose to perform three-fold cross validation rather than 10-fold cross-validation due to the small percentage of post-release faulty functions in the data sets.
- *Across-version prediction*. Across-version prediction uses a model trained on earlier versions to predict faults in later versions within the same project. There are two kinds of approaches for the across-version prediction [50]. The first approach is next-version prediction, i.e. building a prediction model on a version i and then only applying the model to predict faults in the next version $i+1$ of the same project. The second approach is follow-up-version prediction, i.e. building a prediction model on a version i and then applying the model to predict faults in any follow-up version j (i.e. $j > i$) of the same project. In our study, we adopt both approaches. If a project has m versions, the first approach will produce $m-1$ prediction effectiveness values for each model, while the second approach will produce $m \times (m-1)/2$ prediction effectiveness values for each model.
- *Across-project prediction*. Across-project prediction uses a model trained on one project to predict faults in another project [50]. Given n projects, this prediction method will produce $n \times (n$ - 1) prediction effectiveness values for each model.

In each of the above-mentioned three prediction settings, all models use the same training data and the same testing data. Based on these setups, we employ the Wilcoxon signed-rank test to examine whether two models have a significant difference on the prediction effectiveness. In particular, we use the Benjamini-Hochberg (BH) corrected p-values to examine whether a difference is significant at the significance level of 0.10. The null hypothesis $\boldsymbol{H3_0}$ corresponding to RQ3 will be rejected when the comparison shows that the “S” model outperforms the “B” model and the difference is significant. The null hypothesis $\boldsymbol{H4_0}$ corresponding to RQ4 will be rejected when the comparison shows that the “B+S” model outperforms the “B” model and the difference is significant. Furthermore, we use the Cliff’s δ, which is used for median comparison, to examine whether the magnitude of the difference between the prediction performances of two models are important from the viewpoint of practical application [34]. By convention, the magnitude of the difference is considered either trivial ($|\delta| < 0.147$), small (0.147-0.33), moderate (0.33-0.474), or large (> 0.474) [58].

We test the null Hypotheses $\boldsymbol{H3_0}$ and $\boldsymbol{H4_0}$ in the following two typical application scenarios: ranking and classification. In the ranking scenario, functions are ranked in order from the most to the least predicted relative risk. With this ranking list in hand, software practitioners can simply select as many high-risk functions targeted for software quality enhancement as available resources will allow. In the classification scenario, functions are first classified into two categories in terms of their predicted relative risk: high-risk and low-risk. After that, those functions classified as high-risk are targeted for software quality enhancement. In both scenarios, we take into account the effort to test or inspect those functions predicted as high-risk when evaluating the prediction effectiveness of a model. Following previous work [34], we use the source lines of code in a function f as a proxy to estimate the effort required to test or inspect the function. In particular, we define the relative risk of the function f as $R(f) = Pr/SLOC(f)$, where Pr is the probability of the function f being faulty predicted using the logistic regression model. In other words, *R*(*f*) can be regarded as the predicted fault-proneness per *SLOC*. In the context of effort-aware fault-proneness prediction, prior studies used defect density [35, [36], [37], i.e. *#Error*(*f*) / *SLOC*(*f*), as the dependent variable to build the prediction model. In this study, we first use the binary dependent variable to build the logistic regression model and then use *R*(*f*) to estimate the relative risk of a given function *f*. Next, we describe the effort-aware prediction performance indicators used in this study for ranking and classification.

(1) Effort-aware ranking performance evaluation. We use the cost-effectiveness measure *CE* proposed by Arisholm et al. [34] to evaluate the effort-aware ranking effectiveness of a fault-proneness prediction model. The *CE* measure is based on the concept of the “*SLOC*-based” Alberg diagram. In this diagram, the *x*-axis is the cumulative percentage of *SLOC* of the functions selected from the function ranking and the *y*-axis is the cumulative percentage of post-release faults found in the selected functions. Consequently, each fault-proneness prediction model corresponds to a curve in the diagram. Fig. 1 is an example “*SLOC*-based” Alberg diagram showing the ranking performance of a prediction model *m* (in our context, the prediction model *m* could be the “B” model, the “S” model, and the “B+S” model). To compute *CE*, we also consider two additional curves, which respectively correspond to “random” model and “optimal” model. In the “random” model, functions are randomly selected to test or inspect. In the “optimal” model, functions are sorted in decreasing order according to their actual post-release fault densities. Based on this diagram, the effort-aware ranking effectiveness of the prediction model m is defined as follows [34]:

$$CE_{\pi}(m) = \frac{Area_{\pi}(m) - Area_{\pi}(random\ model)}{Area_{\pi}(optimal\ model) - Area_{\pi}(random\ model)}.$$

Here, Area$_{\pi}$(*m*) is the area under the curve corresponding to model *m* for a given top $\pi \times 100\%$ percentage of *SLOC*.

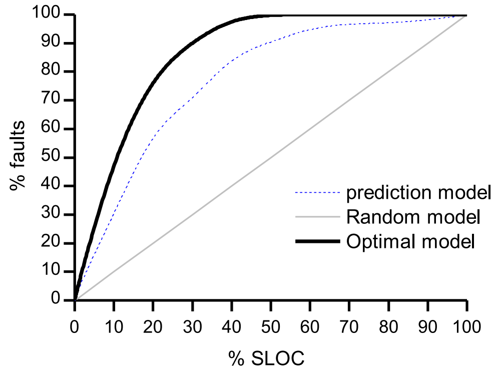


Fig. 1. SLOC-based Alberg diagram.

The cut-off value π varies between 0 and 1, depending on the amount of available resource for testing or inspecting functions. Indeed, $CE_\pi(m)$ is the normalized "average recall" of faults in relation to the optimal model and the random model over the interval from 0 percent *SLOC* to $\pi \times 100\% SLOC$. The range of $CE_\pi(m)$ is [−1, 1] and a larger value means a better ranking effectiveness. In the literature, $CE_\pi(m)$ has been considered as an effective way to measure the ranking performance of fault-proneness prediction models [38].

(2) Effort-aware classification performance evaluation. We use the *SLOC* testing or inspection reduction measure *ER* (effort reduction, originally called "LIR") proposed by Shin et al. [39] to evaluate the effort-aware classification effectiveness of a post-release fault-proneness prediction model. The *ER* measure calculates the proportion of the reduced source lines of code to test or inspect by using a classification model compared with random selection to achieve the same recall of faults. To simplify the presentation, we assume that the system under analysis consists of *n* functions. Let s_i be the *SLOC* of function *i* and f_i be the number of post-release faults in function *i*, $1 \le i \le n$. For a given prediction model *m*, let p_i be 1 if function *i* is predicted as high-risk by the model and 0 otherwise, $1 \le i \le n$. In the classification scenario, only those functions predicted to be high-risk will be tested or inspected for software quality enhancement. In this context, the effort-aware classification effectiveness of the prediction model *m* can be formally defined as follows:

$$ER(m) = \frac{Effort(random) - Effort(m)}{Effort(random)}.$$

Here, *Effort(m)* is the ratio of the total *SLOC* in those predicted high-risk functions to the total *SLOC* in the system, i.e.

$$Effort(m) = \sum_{i-1}^{n} s_i \times p_i \Big/ \sum_{i-1}^{n} s_i.$$

Effort(random) is the proportion of *SLOC* to test or inspect to the total *SLOC* in the system that a random selection model needs to achieve the same recall of post-release faults as the prediction model *m*, i.e.

$$Effort(random) = \sum_{i=1}^{n} f_i \times p_i \Big/ \sum_{i=1}^{n} f_i.$$

Before computing ER, we need to know the classification threshold for the prediction model *m*. In the literature, there are two popular methods to determine the classification threshold on a training set. The first method is called balanced-pf-pd (BPP) method. This method employs the ROC curve corresponding to *m* to determine the classification threshold. In the ROC curve, probability of detection (*pd*) is plotted against probability of false alarm (*pf*) [40]. Intuitively, the closer a point in the ROC curve is to the perfect classification point (0, 1), the better the model predicts. In this context, the "balance" metric

$$balance = 1 - \sqrt{(0 - pf)^2 + (1 - pd)^2} \Big/ \sqrt{2}$$

can be used to evaluate the degree of balance between *pf* and *pd* [40]. For a given training data set, BPP chooses the threshold having the maximum "balance". The second method is balanced-classification-error (BCE) method. Unlike BPP, BCE chooses the threshold to roughly equalize two classification error rates: false positive rate and false negative rate. As stated by Schein et al. [41], such an approach has the effect of giving more weight to errors from false positives in imbalanced data sets. This is especially important for fault data, as they are typically imbalanced (i.e. most functions have no faults). For the simplicity of presentation, the effort reduction metrics under the BPP and BCE thresholds are respectively called "*ER-BPP*" and "*ER-BCE*". Our study will use "*ER-BPP*" and "*ER-BCE*" to evaluate the effort-aware classification effectiveness of a prediction model. However, it may be not adequate to use these two thresholds for model evaluation, as there are many other possible thresholds. In practice, it is possible that a model is good under the BPP and BCE thresholds but is poor under the other thresholds. Consequently, for software practitioners, it may be misleading to use the "*ER-BPP*" and "*ER-BCE*" metrics to select the best classification model from a number of alternatives. In this paper, we use an additional metric "*ER-AVG*" proposed by Zhou et al. [53] to alleviate this problem. For a given model, the "*ER-AVG*" metric is the average effort reduction of the model over all possible thresholds on the test data set. Therefore, the "*ER-AVG*" metric is indeed independent from specific thresholds. Consequently, it can provide a complete picture of the classification performance.

4 Experimental Setup

In this section, we first introduce the projects used in our study and the method for collecting the data. Then, we give a brief description of the distribution of the metrics in the data sets.

4.1 Studied Projects

We use five well-known open-source projects to investigate the actual usefulness of slice-based cohesion metrics in effort-aware post-release fault-proneness prediction: Bash, Gcc-core, Gimp, Subversion, and Vim. Of these projects, Bash, Gcc-core, and Gimp are GNU projects, while Subversion is an Apache project. More specifically, Bash is a command language interpreter, Gcc-core is a famous GNU compiler collection, Gimp is a GNU image manipulation program, Subversion is a well-known open-source version control system, and Vim is an advanced text editor distributed with

TABLE 6
Studied Projects and Version Information

	Subject release							Previous release	
				Common functions before preprocessing		Common functions after preprocessing			
System	Version	Release date	KSLOC	#function	#faulty function	#function	#faulty function	Version	Release date
Bash	3.0	03–08–2004	55	1476	43	1403	40	2.05b	17–07–2002
Gcc-core	3.4.0	18–04–2004	411	6139	219	6066	210	3.3	14–05–2003
Gimp	2.0.0	23–03–2004	434	12110	469	11521	447	1.3.0	13–11–2001
Subversion	1.2.0	23–05–2005	181	2350	30	2003	29	1.1.0	29–09–2004
Vim	6.2	01–07–2003	123	2400	407	2342	398	6.1	24–03–2002

most UNIX systems. We choose these five projects as the subjects of our study for the following reasons: (1) their patch files or bug-fixing release versions are publicly available, thus allowing us to collect post-release fault data at the function level; (2) they can be successfully analyzed by the value analysis plug-in in the code analysis tool Frama-C [57], thus enabling us to collect slice-based cohesion metrics; (3) they have moderate percentage post-release faulty functions which is suitable for our experiments; and (4) they are nontrivial software belonging to different problem domains.

In our study, we collect the data for Bash 3.0, Gcc-core 3.4.0, Gimp 2.0.0, Subversion 1.2.0, and Vim 6.2. We use these five systems to evaluate the prediction effectiveness of post-release fault-proneness prediction models under the cross-validation and across-project prediction methods. It is easy to know that, under the across-project prediction method, each prediction model will produce $5 \times (5 - 1) = 20$ prediction effectiveness values. Furthermore, we collect the data for Bash 3.1, 3.2, 4.0, 4.1, 4.2, and 4.3. Note that, Bash 4.3, released on 26 February 2014, is the latest version of the Bash system till now. We use Bash 3.0, Bash 3.1, Bash 3.2, Bash 4.0, Bash 4.1, Bash 4.2, and Bash 4.3 to evaluate the prediction effectiveness of post-release fault-proneness prediction models under the across-version prediction method. Under the first across-version prediction approach (i.e. next-version prediction), each prediction model will produce $7 - 1 = 6$ prediction effectiveness values. However, under the second across-version prediction approach (i.e. follow-up-version prediction), each prediction model will produce $7 \times (7 - 1)/2 = 21$ prediction effectiveness values.

4.2 Data Collection

We collected the data from the above-mentioned five projects. Each data point of a data set corresponds to one C function and consists of: 1) 16 product metrics (1 size metric + 11 structural complexity metrics + 4 Halstead's software science metrics); 2) three process metrics (i.e. code churn metrics); 3) eight slice-based cohesion metrics; and 4) the faulty/not-faulty labels of the functions after release. We obtained the data by the following steps:

- *Step 1: Collect the baseline product metrics for each function using the tool "Understand"*. For each system, we first generated an Understand database using the program-understanding tool "Understand".[2] This database stored information about entities (such as functions and variables) and references (such as function call and variable references). Then, we collected the most commonly used 16 product metrics for each function of a system.
- *Step 2: Collect the baseline process metrics for each function*. For each project, we used the tool "Understand" to generate two Understand databases: one for the investigated version and another for the previously released version. After that, we collected the three code churn metrics by using the commonly used diff algorithm [56] to compare the functions appearing in these two databases. In this study, the blank line and comments in those functions are not counted when computing the code churn metrics. The last two columns in Table 6 show for each project the previously released version used for computing the code churn metrics.
- *Step 3: Determine the faulty or not-faulty labels for each function after release*. On the one hand, the project websites for Bash[3] and Vim[4] publish a number of patch files for fixing bugs reported after release. Each patch file not only describes the problem reported but also gives the patch to fix the corresponding problem. By analyzing these patches, we were able to determine which functions needed to be changed for fixing the problem. If a function had code changes by these patches, it will be marked as a faulty function and otherwise not-faulty. On the other hand, Gcc-3.4.6,[5] Gimp 2.0.6,[6] and Subversion 1.2.3[7] are the latest bug-fixing releases to Gcc-core 3.4.0, Gimp 2.0.0, and Subversion 1.2.0, respectively. These bug-fixing releases did not add any new features to the corresponding systems, thus enabling us to determine which functions had code changes for fixing bugs. If a function had code changes in the latest bug-fixing releases, it will be marked as a faulty function. Otherwise, the function is a not-faulty function. This is one of the most commonly used ways to determine *faulty* functions [31].
- *Step 4: Compute slice-based cohesion metrics for each function using the tool "Frama-C"*. We use intra-

[2] http://www.scitools.com

[3] http://ftp.gnu.org/gnu/bash

[4] http://ftp.vim.org/pub/vim/patches

[5] http://gcc.gnu.org/releases.html

[6] http://www.gimpusers.com/forums/gimp-user/1786-announce-gimp-2-0-6

[7] http://subversion.apache.org/docs/release-notes/1.2.html

procedural slicing to compute slice-based cohesion metrics for each function. In other words, metric slices are computed within a single function. In our study, calls to other functions are handled conservatively. More specifically, we developed two Frama-C plug-ins named INFER CONtract (INFERCON) and SLIce-Based COhesion Metrics (SLIBCOM) to compute slice-based cohesion metrics for each function. The INFERCON plugin is used to infer the contract for a called function conservatively. For a function, if it has a return value, INFERCON assumes that the returned value is data-dependent on all the arguments passed to the function. For any pointer argument p in the function, INFERCON assumes that the value pointed by p will be changed at the end of the function and hence p is data-dependent on all the arguments passed to the function. In addition, INFERCON assumes that any function is a terminating function. The SLIBCOM plugin is used to collect slice-based cohesion metrics for each function in each system. The SLIBCOM plug-in was based on the INFERCON plug-in and the other four plug-ins provided by Frama-C, namely "Value analysis", "Outputs", "Slicing", and "Impact analysis". For each function m in the system, INFERCON first inferred contracts for the functions called by m. Next, SLIBCOM employed the "Value analysis" plug-in to perform the value analysis in a context-insensitive way by using the inferred contracts for the called functions. Then, based on the results from the value analysis, SLIBCOM used the "Outputs" plug-in to obtain the output variables of m. After that, SLIBCOM leveraged the "Slicing" plug-in to obtain the end slices for each output variable. Based on those end slices, SLIBCOM used the "Impact analysis" plug-in to obtain the corresponding "forward slices" and then combined them to obtain the metric slices for each output variable. Finally, SLIBCOM used the metric-slice information to calculate slice-based cohesion metrics for the function m. Note that, the cohesion metric value of a function was set to undefined if either of the following two conditions was satisfied: (1) the execution time for the value analysis was very long; (2) the "Outputs" plug-in did not find any output variable. The reason for the former case is unknown. In our study, we terminated the value analysis when the execution time was longer than 30 min. The latter case occurred when the function under analysis did not return anything and had no side effect as well. In this case, the "Outputs" plug-in was unable to identify any output variable for the function under analysis.

Table 6 summarizes the projects studied in this study (the time cost for collecting the slice-based cohesion metrics is shown in Table 12 in Appendix A, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TSE.2014.2370048). The second to the fourth columns are the version number, the release date, and the total source lines of code of the subject release, respectively. The fifth and the sixth columns are respectively the total number of functions and the total number of faulty functions that can be identified by both Understand and Frama-C. The seventh and the eighth columns are respectively the total number of functions and the total number of faulty functions after removing the functions that have an "undefined" metric value. As can be seen, for each system, faulty functions detected during the post-release phase concentrated in a very small number of functions (only around 1.234~16.994 percent of all functions). In all the subsequent analyses (in Section 5), we use only the pre-processed data sets. The last two columns respectively provide the version number and the release date for the previous releases. The previous releases are used for computing the code churn metrics for each system (described in *Step 2* in Section 4.2). We choose these previous five versions as the baseline versions to compute code churn metrics, as they all are the first previous minor versions for each system. We can find that, on average, the previous release is released 17 months before the subject version is released.

4.3 Data Distribution

Table 7 presents the descriptive statistics for each data set. Columns "25", "50", and "75" percent state for each metric the first quartile, the median value, and the third quartile, respectively. From Table 7, we have the following observations. First, for the code metrics, we can see that Gcc-core 3.4.0 has the largest function size, the highest Cyclomatic complexity, and the maximum depth of nesting. This indicates that this compiler collection has a more complex control flow than the other systems. Second, for the process metrics, we can see that the functions in Gcc-core 3.4.0 undergo more code changes. This is probably because the Gcc 3.4.0 has many improvements in the C++ frontend.[8] Third, for the slice-based cohesion metrics, we can see that Vim 6.2 in general has a smaller cohesion value than the other systems. In other words, its functions are less cohesive than the functions in the other four systems. From Table 6, we observe that, of the five systems, Vim 6.2 has the largest percentage of faulty functions. One possible explanation is that lower cohesive functions are more likely to be faulty functions. This is consistent with our intuition. Fourth, for most metrics, there are large differences between the lower 25th percentile, the median, and the 75th percentile, thus showing strong variations across functions.

All of the metrics have more than five observations that are nonzero, and hence, are considered for further analysis [33].

5 Experimental Results

In this section, we elaborate on the experimental results for slice-based cohesion metrics. In Section 5.1, we present the results from examining their redundancy with the most commonly used code and process metrics (RQ1). In Section 5.2, we give the results from examining their correlations with post-release fault-proneness (RQ2). In Section 5.3, we show the results from examining their ability for predicting post-release fault-proneness compared with the most commonly used code and process metrics (RQ3). In Section 5.4, we report the results from examining the usefulness of their combination with the most commonly used code and

[8] http://www.gnu.org/software/gcc/gcc-3.4/changes.html

TABLE 7
Description Statistics for Each Data Set

	Bash 3.0			Gcc-core 3.4.0			Gimp 2.0.0			Subversion 1.2.0			Vim 6.2		
Metric	25%	50%	75%	25%	50%	75%	25%	50%	75%	25%	50%	75%	25%	50%	75%
SLOC	8	14	26	10	18	39	9	17	31	10	16	31	9.25	17	38
FANIN	3	4	7	5	9	15	3	5	9	5	7	11.5	4	7	12
FANOUT	2	3	6	2	4	9	3	5	10	3	5	9	2	4	9
NPATH	1	3	7	2	4	18	1	2	5	1	3	8	2	4	16
Cyclomatic	1	3	6	2	4	9	1	2	4	1	4	9	2	3	8
CyclomaticModified	1	3	6	2	4	8	1	2	4	1	4	9	2	3	8
CyclomaticStrict	1	3	8	2	5	12	1	2	5	1	4	10	2	4	10
Essential	1	1	3	1	1	4	1	1	1	1	3	7	1	1	4
Knots	0	0	3	0	1	5	0	0	1	0	2	8	0	1	4
Nesting	0	1	2	1	2	3	0	1	2	0	2	3	1	1	3
MaxEssentialKnots	0	0	1.5	0	0	3	0	0	0	0	2	7	0	0	3
MinEssentialKnots	0	0	1	0	0	3	0	0	0	0	2	7	0	0	3
n1	8	12	17	9	14	20	8	10	14	10	13	17	9	14	21
n2	7	12	21	11	18	32	11	18	30	12	19	32	10	17	32
N1	18	37	75	25	51.5	114	19	39	81	25	47	93.5	21	46	114
N2	13	25	53	19	40	88	17	33	70	19	36	71.5	16	36	85
Added	0	0	0	0	0	0.067	0	0.176	0.882	0	0	0.039	0	0	0
Deleted	0	0	0	0.667	0.867	1	0	0	0.819	0	0	0.833	0	0	0
Modified	0	0	0	0	0.037	0.111	0	0	0	0	0	0	0	0	0
Coverage	0.500	0.773	1	0.583	0.840	1	0.644	0.838	1	0.667	0.812	1	0.496	0.737	1
MaxCoverage	0.667	0.928	1	0.778	0.961	1	0.850	0.965	1	0.875	0.977	1	0.727	0.936	1
MinCoverage	0.286	0.679	1	0.350	0.765	1	0.380	0.721	1	0.378	0.667	1	0.222	0.605	1
Overlap	0.494	0.963	1	0.623	0.971	1	0.672	0.946	1	0.639	0.873	1	0.454	0.905	1
Tightness	0.182	0.653	1	0.281	0.750	1	0.350	0.710	1	0.353	0.652	1	0.143	0.571	1
WFC	0.481	0.847	1	0.590	0.895	1	0.692	0.917	1	0.729	0.889	1	0.454	0.813	1
SBFC	0.356	0.903	1	0.469	0.892	1	0.520	0.817	1	0.539	0.739	1	0.290	0.754	1
NHD	0.602	0.758	1	0.644	0.798	1	0.644	0.778	1	0.640	0.756	1	0.619	0.757	1

process metrics in effort-aware post-release fault-proneness prediction (RQ4).

5.1 RQ1: Are Slice-Based Cohesion Metrics Redundant with Respect to the Most Commonly Used Code and Process Metrics?

We use the results from principal component analysis to answer RQ1. Table 8 summarizes the rotated components from PCA for each data set (the detailed information are shown in Tables 13, 14, 15, 16, and 17 in Appendix B, available in the online supplemental material). In Table 8, the first and second columns are respectively the system name and the PC name. The third to fifth columns report the eigenvalue, the percentage of variance explained by each rotated component, and the cumulative percentage of variance explained, respectively. The sixth column shows which metrics are clustered into each PC. The last column marks the PCs consisting of only slice-based cohesion metrics. In particular, slice-based cohesion metrics are shown in bold face.

From Table 8, we can see that the metrics (the most commonly used code and process metrics and the slice-based cohesion metrics) are clustered into ten to thirteen distinct orthogonal components, which describe around 91~95 percent of the variance in the data. Furthermore, we have the following observations:

- The most commonly used code and process metrics are distributed in seven to 10 components, which describe around 72 percent of the variance in the data. Of the code metrics, *SLOC*, most cyclomatic complexity metrics and Halstead's software science metrics fall in the same component (PC1). This indicates that cyclomatic complexity metrics and Halstead's software science metrics essentially measure the size dimension. *Knots*, *MaxEssentialKnots*, and *MinEssentialKnots* fall in a component (PC3) different from most of the other code metrics (that mainly go in PC1), and the same happens for *FANIN* and for *NPATH*. In particular, code churn metrics defines new PCs of their own compared with code metrics, indicating that process metrics and code metrics measure different information.
- Slice-based cohesion metrics are distributed in three distinct orthogonal components, which describe around 28 percent of the variance in the data. As can be seen, most slice-based cohesion metrics fall in the same component (PC2). It is interesting that *NHD* always defines a PC of its own, regardless of which data set is considered. This indicates that *NHD* is different from the other slice-based cohesion metrics. This is also true for *MaxCoverage* to a limited extent.

The core observation from Table 8 is that there is no overlap between all the PCs by the most commonly used code and process metrics and all the PCs by slice-based cohesion metrics. In other words, slice-based cohesion metrics indeed define the PCs of their own compared with the most commonly used code and process metrics. Therefore, our PCA analysis results, from five different data sets, consistently

TABLE 8
Results of Principal Component Analysis

Sys.	PC	Eigen value	%Var.	%Cum Var.	Clustered metrics	
Bash 3.0	PC1	8.160	0.302	0.302	SLOC + FANOUT + Cyclomatic + CyclomaticModified + CyclomaticStrict + n1 + n2 + N1+ N2	
	PC2	6.201	0.230	0.532	**Coverage** + **MinCoverage** + **Overlap**+ **Tightness** + **WFC** + **SBFC**	√
	PC3	3.397	0.126	0.658	Essential + Knots + MaxEssentialKnots + MinEssentialKnots	
	PC4	1.078	0.040	0.698	Modified	
	PC5	1.008	0.037	0.735	FANIN	
	PC6	1.007	0.037	0.772	Deleted	
	PC7	1.005	0.037	0.809	Added	
	PC8	0.998	0.037	0.846	NPATH	
	PC9	0.767	0.028	0.875	**MaxCoverage**	√
	PC10	0.669	0.025	0.900	Nesting	
	PC11	0.590	0.022	0.921	**NHD**	√
Gcc-core 3.4.0	PC1	7.990	0.296	0.296	SLOC + Cyclomatic + CyclomaticModified + CyclomaticStrict + Essential + n2 + N1 + N2	
	PC2	5.790	0.214	0.510	**Coverage** + **MinCoverage** + **Overlap** + **Tightness** + **WFC** + **SBFC**	√
	PC3	3.418	0.127	0.637	Knots + MaxEssentialKnots + MinEssentialKnots	
	PC4	1.012	0.037	0.674	Modified	
	PC5	1.007	0.037	0.712	Added	
	PC6	1.002	0.037	0.749	Deleted	
	PC7	0.993	0.037	0.786	**MaxCoverage**	√
	PC8	0.989	0.037	0.822	FANIN	
	PC9	0.898	0.033	0.856	NPATH	
	PC10	0.869	0.032	0.888	Nesting	
	PC11	0.677	0.025	0.913	**NHD**	√
	PC12	0.557	0.021	0.933	n1	
	PC13	0.547	0.020	0.954	FANOUT	
Gimp 2.0.0	PC1	6.239	0.231	0.231	SLOC + Cyclomatic + CyclomaticModified + CyclomaticStrict + Nesting + n1 + N1 + N2	
	PC2	5.004	0.185	0.416	**Coverage** + **MinCoverage** + **Overlap** + **Tightness** + **SBFC**	√
	PC3	3.068	0.114	0.530	Knots + MaxEssentialKnots + MinEssentialKnots	
	PC4	1.879	0.070	0.600	**MaxCoverage** + **WFC**	√
	PC5	1.704	0.063	0.663	Modified	
	PC6	1.604	0.059	0.722	FANOUT + n2	
	PC7	1.014	0.038	0.760	NPATH	
	PC8	1.008	0.037	0.797	FANIN	
	PC9	1.003	0.037	0.834	Added	
	PC10	1.001	0.037	0.871	Deleted	
	PC11	0.547	0.020	0.892	**NHD**	√
	PC12	0.486	0.018	0.910	Essential	
Subversion 1.2.0	PC1	8.179	0.303	0.303	SLOC + FANOUT + Cyclomatic + CyclomaticModified + CyclomaticStrict + Essential + n2 + N1 + N2	
	PC2	4.674	0.173	0.476	**MinCoverage** + **Overlap** + **Tightness** + **SBFC**	√
	PC3	3.064	0.113	0.590	Knots + MaxEssentialKnots + MinEssentialKnots	
	PC4	2.570	0.095	0.685	**Coverage** + **MaxCoverage** + **WFC**	√
	PC5	1.007	0.037	0.722	Deleted	
	PC6	1.006	0.037	0.759	NPATH	
	PC7	1.005	0.037	0.797	Added	
	PC8	1.005	0.037	0.834	Modified	
	PC9	1.001	0.037	0.871	FANIN	
	PC10	0.616	0.023	0.894	n1	
	PC11	0.521	0.019	0.913	Nesting	
	PC12	0.376	0.014	0.927	**NHD**	√
Vim 6.2	PC1	9.253	0.343	0.343	SLOC + FANOUT + NPATH + Cyclomatic + CyclomaticModified + CyclomaticStrict + Essential + n1 + n2 + N1 + N2	
	PC2	6.037	0.224	0.566	**Coverage** + **MinCoverage** + **Overlap** + **Tightness** + **WFC** + **SBFC**	√
	PC3	3.295	0.122	0.688	Knots + MaxEssentialKnots + MinEssentialKnots	
	PC4	1.031	0.038	0.727	Modified	
	PC5	1.016	0.038	0.764	Nesting	
	PC6	1.013	0.038	0.802	Added	
	PC7	1.000	0.037	0.839	Deleted	
	PC8	0.978	0.036	0.875	FANIN	
	PC9	0.906	0.034	0.909	**MaxCoverage**	√
	PC10	0.587	0.022	0.930	**NHD**	√

TABLE 9
Results of Univariate Logistic Regression Analysis Before Removing the Potentially Confounding Effect of Function Size

Metric	Bash 3.0			Gcc-core 3.4.0			Gimp 2.0.0			Subversion 1.2.0			Vim 6.2		
	Coeff.	p-value	ΔOR	Coeff.	p-value	ΔOR	Coeff.	p-value	ΔOR	Coeff.	p-value	ΔOR	Coeff.	p-value	ΔOR
SLOC	0.008	0.035	1.262	0.010	<0.001	1.716	0.011	<0.001	1.416	0.013	<0.001	1.461	0.020	<0.001	4.399
FANIN	0.003	0.843	1.039	0.011	<0.001	1.215	0.004	0.135	1.053	0.001	0.939	1.012	0.044	<0.001	1.710
FANOUT	0.038	0.055	1.244	0.071	<0.001	1.798	0.064	<0.001	1.641	0.083	<0.001	1.723	0.118	<0.001	3.452
NPATH	<0.001	0.797	<0.001	<0.001	<0.001	1.250	<0.001	0.993	1.000	<0.001	0.939	<0.001	<0.001	0.030	3.683
Cyclomatic	0.028	0.035	1.261	0.028	<0.001	1.538	0.043	<0.001	1.277	0.036	0.001	1.421	0.092	<0.001	5.285
CyclomaticModified	0.033	0.034	1.279	0.045	<0.001	1.675	0.053	<0.001	1.294	0.038	0.001	1.433	0.101	<0.001	4.415
CyclomaticStrict	0.018	0.039	1.241	0.021	<0.001	1.646	0.038	<0.001	1.289	0.033	0.001	1.430	0.065	<0.001	5.030
Essential	−0.001	0.992	0.998	0.048	<0.001	1.405	0.033	0.028	1.079	0.039	0.001	1.401	0.131	<0.001	2.904
Knots	−0.003	0.869	0.959	0.007	<0.001	1.377	0.001	0.767	1.018	0.013	0.015	1.250	0.057	<0.001	6132.536
Nesting	0.313	0.034	1.517	0.313	<0.001	1.669	0.288	<0.001	1.420	0.497	<0.001	2.068	0.515	<0.001	2.456
MaxEssentialKnots	−0.013	0.740	0.840	0.008	<0.001	1.374	<0.001	0.993	0.998	0.013	0.016	1.244	0.051	<0.001	2482.538
MinEssentialKnots	−0.014	0.740	0.822	0.007	<0.001	1.368	<0.001	0.992	0.995	0.013	0.016	1.243	0.052	<0.001	3015.089
n1	0.049	0.035	1.404	0.099	<0.001	2.065	0.074	<0.001	1.551	0.119	<0.001	1.847	0.137	<0.001	3.123
n2	0.014	0.038	1.279	0.021	<0.001	1.767	0.020	<0.001	1.553	0.019	<0.001	1.474	0.038	<0.001	4.270
N1	0.002	0.035	1.268	0.003	<0.001	1.726	0.002	<0.001	1.309	0.005	<0.001	1.524	0.006	<0.001	4.711
N2	0.003	0.035	1.257	0.004	<0.001	1.733	0.003	<0.001	1.326	0.004	0.001	1.373	0.009	<0.001	4.426
Added	0.596	0.108	1.175	0.393	<0.001	1.183	0.471	<0.001	1.268	1.845	<0.001	1.842	0.546	0.001	1.172
Deleted	0.272	0.045	1.191	0.155	0.017	1.105	0.005	0.831	1.013	0.483	0.141	1.258	1.049	<0.001	2.011
Modified	−1.590	0.845	0.793	−6.409	<0.001	0.004	−0.035	0.853	0.984	−0.228	0.767	0.876	−0.064	0.633	0.950
Coverage	−1.183	0.055	0.726	−1.395	<0.001	0.702	−0.910	<0.001	0.820	−1.854	0.043	0.703	−1.944	<0.001	0.589
MaxCoverage	−0.598	0.455	0.868	−0.570	0.058	0.885	1.049	0.001	1.209	−0.658	0.636	0.907	−0.966	<0.001	0.806
MinCoverage	−1.363	0.034	0.619	−1.536	<0.001	0.590	−1.932	<0.001	0.532	−1.781	0.005	0.570	−1.952	<0.001	0.486
Overlap	−1.067	0.034	0.682	−1.421	<0.001	0.622	−1.927	<0.001	0.573	−1.753	0.004	0.627	−1.458	<0.001	0.596
Tightness	−1.255	0.034	0.613	−1.384	<0.001	0.599	−1.745	<0.001	0.551	−1.793	0.004	0.559	−1.682	<0.001	0.513
WFC	−0.760	0.134	0.781	−0.096	0.677	0.972	0.896	<0.001	1.250	0.346	0.767	1.075	−0.436	0.009	0.868
SBFC	−1.112	0.034	0.666	−1.142	<0.001	0.682	−1.113	<0.001	0.732	−1.703	0.023	0.658	−1.435	<0.001	0.590
NHD	−2.495	0.034	0.622	−1.829	<0.001	0.728	−1.028	<0.001	0.839	−4.757	0.001	0.453	−2.737	<0.001	0.610

All p-values have been adjusted using the Benjamini-Hochberg method.

reject the null hypothesis $\boldsymbol{H1_0}$ corresponding to RQ1: “Slice-based cohesion metrics do not capture additional dimensions of software quality compared with the most commonly used code and process metrics”. In other words, our results are in favor of the alternative hypothesis $H1_A$, i.e. there is a strong evidence that slice-based cohesion metrics are not redundant with respect to the most commonly used code and process metrics.

5.2 RQ2: Are Slice-Based Cohesion Metrics Statistically Significantly Correlated to Post-Release Fault-Proneness?

We use the results from univariate logistic regression analysis to answer RQ2. Tables 9 and 10 respectively report the results of univariate logistic regression analysis for individual slice-based cohesion metrics before and after removing the confounding effect of function size measured by *SLOC*. The column “Metric” shows the independent variable used for the corresponding univariate logistic regression model. The columns “Coeff.”, “p-value”, and “ΔOR” state for each model the estimated regression coefficient, the statistical significance of the coefficient from Z test, and the odds ratio associated with one standard deviation increase, respectively. Since there are 27 metrics, the total number of statistical Z tests on each data set is 27. Because multiple tests on the same data set may result in spurious statistically significant results, we use the Benjamini-Hochberg correction of p-values to control for false discovery [47]. In other words, all p-values reported in Tables 9 and 10 have been adjusted using the Benjamini-Hochberg method. In particular, those p-values greater than the $\alpha = 0.10$ significance level are marked in gray.

From Table 9, we can see that most of the investigated baseline code and process metrics are significantly positively related to post-release fault-proneness. This indicates that functions with a high code and process complexity tend to be post-release fault-prone. Furthermore, we observe that almost all the slice-based cohesion metrics have a significant correlation with post-release fault-proneness. This is true especially for Gcc-core 3.4.0, Gimp 2.0.0, and Vim 6.2. Overall, most slice-based cohesion metrics are significantly related to post-release fault-proneness negatively, thus supporting the intuition that functions with a low cohesion tend to be fault-prone. According to ΔOR, *Tightness* and *MinCoverage* have the strongest impact on post-release fault-proneness. However, the results shown in Table 9 do not reflect the true correlations of the investigated metrics with post-release fault-proneness, as the potentially confounding effect of function size is not taken into account [53].

Table 10 summarizes the univariate analysis results after removing the confounding effect of function size. As can be seen, for all the data sets except Bash 3.0, more than half of the baseline code and process metrics are significantly related to post-release fault-proneness. Even for Bash 3.0, there are still five significant code metrics. These results suggest that the characteristics captured by these baseline code and process metrics are indeed different from function size and can be used as post-release fault-proneness indicators. More importantly, we can see that, after removing the

TABLE 10
Results of Univariate Logistic Regression Analysis After Removing the Potentially Confounding Effect of Function Size

Metric	Bash 3.0			Gcc-core 3.4.0			Gimp 2.0.0			Subversion 1.2.0			Vim 6.2		
	Coeff.	p-value	ΔOR	Coeff.	p-value	ΔOR	Coeff.	p-value	ΔOR	Coeff.	p-value	ΔOR	Coeff.	p-value	ΔOR
SLOC	0.008	0.064	1.262	0.010	<0.001	1.716	0.011	<0.001	1.416	0.013	<0.001	1.461	0.020	<0.001	4.399
FANIN	−0.006	0.842	0.940	0.002	0.518	1.039	−0.006	0.351	0.914	−0.002	0.892	0.962	0.009	0.047	1.100
FANOUT	0.015	0.771	1.059	0.041	<0.001	1.270	0.071	<0.001	1.489	0.112	<0.001	1.777	0.063	<0.001	1.418
NPATH	<0.001	0.428	0.866	<0.001	0.047	1.101	<0.001	<0.001	0.319	<0.001	<0.001	0.027	<0.001	0.012	0.851
Cyclomatic	0.033	0.672	1.097	−0.059	<0.001	0.705	−0.043	0.012	0.869	0.043	0.037	1.343	−0.016	0.255	0.926
CyclomaticModified	0.087	0.100	1.255	0.021	0.135	1.103	−0.008	0.715	0.977	0.045	0.035	1.353	0.020	0.240	1.066
CyclomaticStrict	0.005	0.873	1.025	0.004	0.572	1.037	−0.016	0.331	0.941	0.041	0.035	1.371	<0.001	0.984	0.999
Essential	−0.155	0.059	0.708	−0.056	<0.001	0.760	−0.239	<0.001	0.622	0.040	0.054	1.300	−0.072	<0.001	0.760
Knots	−0.060	0.012	0.509	−0.007	<0.001	0.128	−0.068	<0.001	0.235	−0.002	0.868	0.947	−0.020	<0.001	0.213
Nesting	0.286	0.103	1.325	−0.110	0.054	0.862	−0.003	0.955	0.997	0.491	0.005	1.848	0.265	<0.001	1.411
MaxEssentialKnots	−0.060	0.012	0.504	−0.007	<0.001	0.129	−0.092	<0.001	0.206	−0.002	0.864	0.935	−0.021	<0.001	0.200
MinEssentialKnots	−0.062	0.012	0.494	−0.007	<0.001	0.129	−0.092	<0.001	0.203	−0.003	0.864	0.932	−0.021	<0.001	0.202
n1	0.026	0.652	1.128	−0.018	0.172	0.906	0.027	0.028	1.118	0.114	0.016	1.639	0.099	<0.001	1.805
n2	0.006	0.771	1.057	0.013	0.012	1.171	0.049	<0.001	1.637	0.055	0.001	1.803	0.022	<0.001	1.450
N1	0.003	0.561	1.094	0.003	0.087	1.103	<0.001	0.702	1.018	0.011	<0.001	1.820	0.002	0.027	1.114
N2	0.003	0.672	1.080	0.004	0.006	1.166	0.001	0.144	1.043	0.016	0.002	1.543	0.002	0.025	1.119
Added	0.603	0.158	1.178	0.393	0.001	1.183	0.486	<0.001	1.277	1.809	<0.001	1.818	0.494	0.004	1.154
Deleted	0.229	0.173	1.151	0.064	0.497	1.042	−0.001	0.955	0.997	0.216	0.722	1.104	0.550	<0.001	1.415
Modified	−0.935	0.815	0.872	0.105	0.318	1.046	0.010	0.955	1.005	−0.120	0.864	0.932	−0.020	0.840	0.984
Coverage	−0.972	0.173	0.771	−0.881	0.002	0.801	−0.435	0.067	0.911	−1.396	0.171	0.770	−1.215	<0.001	0.722
MaxCoverage	−0.597	0.561	0.868	−0.711	0.025	0.859	1.413	<0.001	1.290	−0.397	0.864	0.943	−0.901	<0.001	0.817
MinCoverage	−1.124	0.064	0.683	−0.773	<0.001	0.771	−1.458	<0.001	0.636	−1.375	0.037	0.655	−1.031	<0.001	0.696
Overlap	−0.902	0.080	0.730	−0.920	<0.001	0.738	−1.587	<0.001	0.650	−1.460	0.035	0.684	−0.743	<0.001	0.777
Tightness	−1.042	0.064	0.674	−0.727	<0.001	0.767	−1.282	<0.001	0.659	−1.391	0.035	0.645	−0.895	<0.001	0.712
WFC	−0.770	0.173	0.779	−0.252	0.300	0.927	0.967	<0.001	1.272	0.470	0.769	1.104	−0.506	0.003	0.849
SBFC	−0.924	0.080	0.719	−0.643	0.002	0.808	−0.714	<0.001	0.822	−1.324	0.100	0.726	−0.859	<0.001	0.734
NHD	−2.060	0.064	0.680	−0.745	0.087	0.880	−0.512	0.103	0.917	−3.581	0.015	0.562	−1.585	<0.001	0.754

All p-values have been adjusted using the Benjamini-Hochberg method.

confounding effect of function size, most slice-based cohesion metrics are still significantly related to fault-proneness. Overall, our univariate logistic regression analysis results, from five different data sets, reject the null hypothesis $\boldsymbol{H2_0}$ corresponding to RQ2: "Slice-based cohesion metrics do not have a significant correlation with post-release fault-proneness." In other words, our results are in favor of the alternative hypothesis $\boldsymbol{H2_A}$, i.e. slice-based cohesion metrics are in general significantly related to the occurrence of post-release faults in functions.

5.3 RQ3: Are Slice-Based Cohesion Metrics More Effective than the Most Commonly Used Code and Process Metrics in Effort-Aware Post-Release Fault-Proneness Prediction?

In order to answer RQ3, we first use the procedure described in Section 3.4.3 to build the "S" model and the "B" model on each data set. Table 11 summarizes the "B" and "S" models for each data set. The first column indicates the data set and the second column is the type of the model. The third to fourth columns show the number of (excluded) influential observations and the maximum VIF among the independent variables in each model, respectively. The remaining columns, starting from the fifth column, show the selected independent variables. Below each variable name, we present the corresponding coefficient and p-value. Then, we compare the prediction effectiveness of the "S" and "B" models with respect to both ranking and classification under cross-validation, across-version prediction, and across-project prediction.

5.3.1 *Cross-Validation*

Fig. 2 employs the box-plot to describe the distributions of the *CE*s at different cut-off values and the *ER-BPP*s/*ER-BCE*s/*ER-AVG*s obtained from 30 times three-fold cross-validation for the "B" model and the "S" model. For each model, the box-plot shows the median (the horizontal line within the box), the 25th and 75th percentiles (the lower and upper sides of the box) as well as the mean performance value (the small rectangle inside the box). In practice, practitioners are more interested in the ranking performance of a prediction model at the top fraction. Therefore, we report the *CE* performances at $\pi = 0.1$ and 0.2 for each model. In addition, we report the *CE* performance at $\pi = 1.0$ in order to provide a more complete picture of the ranking performance for each model.

From Fig. 2, we have the following observations:

- *Ranking performance*. For Bash 3.0, the "S" model has a slightly larger median *CE* than the "B" model. The BH corrected p-value in the Wilcoxon signed-rank test is not significant for both $CE_{0.1}$ and $CE_{1.0}$ but is significant for $CE_{0.2}$. The effect sizes (i.e. the magnitudes of the difference between the "S" model and the "B" model) are trivial or small in terms of the Cliff's δ $(0.046 \leq |\delta| \leq 0.115)$. For Gcc-core 3.4.0 and Vim 6.2, the "S" model has a significantly lower median *CE* than the "B" model (the BH corrected p-values $<$ 0.001). The effect sizes are moderate or large in terms of the Cliff's δ

TABLE 11
The “B”, “S”, and “B+S” Models for Each Data Set

Sys.	Model	k	VIF														
Bash 3.0	B	0	1.094	Constant	Nesting	MaxEssentialKnots	Added										
				−4.281	0.362	−0.074	0.748										
				<0.001	0.007	<0.001	0.029										
	S	0	5.698	Constant	Coverage	MinCoverage	NHD										
				−2.198	3.047	−2.421	−2.589										
				0.002	0.060	0.016	0.056										
	B+S	0	2.548	Constant	Nesting	MaxEssentialKnots	n2	Added	Modified	NHD							
				−2.483	0.322	−0.083	−0.035	0.868	0.536	−2.236							
				0.001	0.026	<0.001	0.087	0.016	0.030	0.021							
Gcc-core 3.4.0	B	0	2.418	Constant	SLOC	FANOUT	Cyclomatic	CyclomaticStrict	Essential	MaxEssentialKnots	n2	N1	Added	Modified			
				−4.623	0.011	0.044	−0.029	0.013	0.032	−0.001	0.011	−0.002	0.597	0.200			
				<0.001	<0.001	<0.001	0.019	0.073	0.019	0.002	0.022	0.077	<0.001	0.011			
	S	0	5.586	Constant	Coverage	Overlap	WFC										
				−2.580	−1.635	−0.717	1.384										
				<0.001	0.009	0.006	0.002										
	B+S	0	3.031	Constant	SLOC	FANOUT	Cyclomatic	CyclomaticStrict	Essential	MaxEssentialKnots	n2	N1	Added	Modified	MaxCoverage	Overlap	WFC
				−3.568	0.011	0.028	−0.031	0.015	0.035	−0.001	0.011	−0.002	0.602	0.200	−1.049	−0.929	0.752
				<0.001	<0.001	0.005	0.014	0.044	0.01	0.005	0.034	0.072	<0.001	0.013	0.056	<0.001	0.084
Gimp 2.0.0	B	1	3.612	Constant	SLOC	FANIN	FANOUT	NPATH	CyclomaticModified	MinEssentialKnots	n1	n2	N1	Added			
				−4.706	0.009	−0.018	0.032	−4.949E-09	0.052	−0.052	0.044	0.028	−0.005	0.362			
				<0.001	<0.001	0.010	<0.001	0.058	<0.001	0.001	<0.001	<0.001	<0.001	<0.001			
	S	0	7.738	Constant	MaxCoverage	MinCoverage	Overlap	WFC									
				−5.026	1.570	−3.201	0.865	2.236									
				<0.001	0.002	<0.001	0.043	<0.001									
	B+S	1	7.906	Constant	SLOC	FANIN	Cyclomatic	MinEssentialKnots	n1	n2	N1	Added	MaxCoverage	MinCoverage	Overlap	WFC	
				−6.606	0.011	−0.013	0.040	−0.039	0.035	0.028	−0.004	0.353	1.755	−3.099	1.230	1.934	
				<0.001	<0.001	0.077	0.001	0.005	0.003	<0.001	<0.001	<0.001	0.001	<0.001	0.004	<0.001	
Subversion 1.2.0	B	0	1.495	Constant	FANOUT	NPATH	Nesting	Added									
				−5.854	0.066	−9.370E-08	0.339	1.282									
				<0.001	0.037	0.041	0.037	<0.001									
	S	0	2.107	Constant	WFC	NHD											
				−2.616	7.631	−10.496											
				0.038	<0.001	<0.001											
	B+S	0	2.682	Constant	SLOC	Added	WFC	NHD									
				−2.523	0.013	1.183	7.430	−11.497									
				0.061	<0.001	<0.001	<0.001	<0.001									
Vim 6.2	B	0	1.238	Constant	SLOC	FANOUT	CyclomaticStrict	n1	Modified								
				−3.684	0.016	0.061	0.042	0.083	0.186								
				<0.001	<0.001	<0.001	<0.001	<0.001	0.009								
	S	0		Constant	Coverage	Overlap	WFC	SBFC									
				−0.815	−2.402	0.662	1.905	−1.296									
				<0.001	<0.001	0.05	<0.001	0.002									
	B+S	0	1.548	Constant	SLOC	FANOUT	CyclomaticStrict	Essential	n1	Modified	MaxCoverage	NHD					
				−2.402	0.016	0.047	0.055	−0.038	0.082	0.177	−0.583	−0.991					
				<0.001	<0.001	<0.001	<0.001	0.024	<0.001	0.013	0.086	0.045					

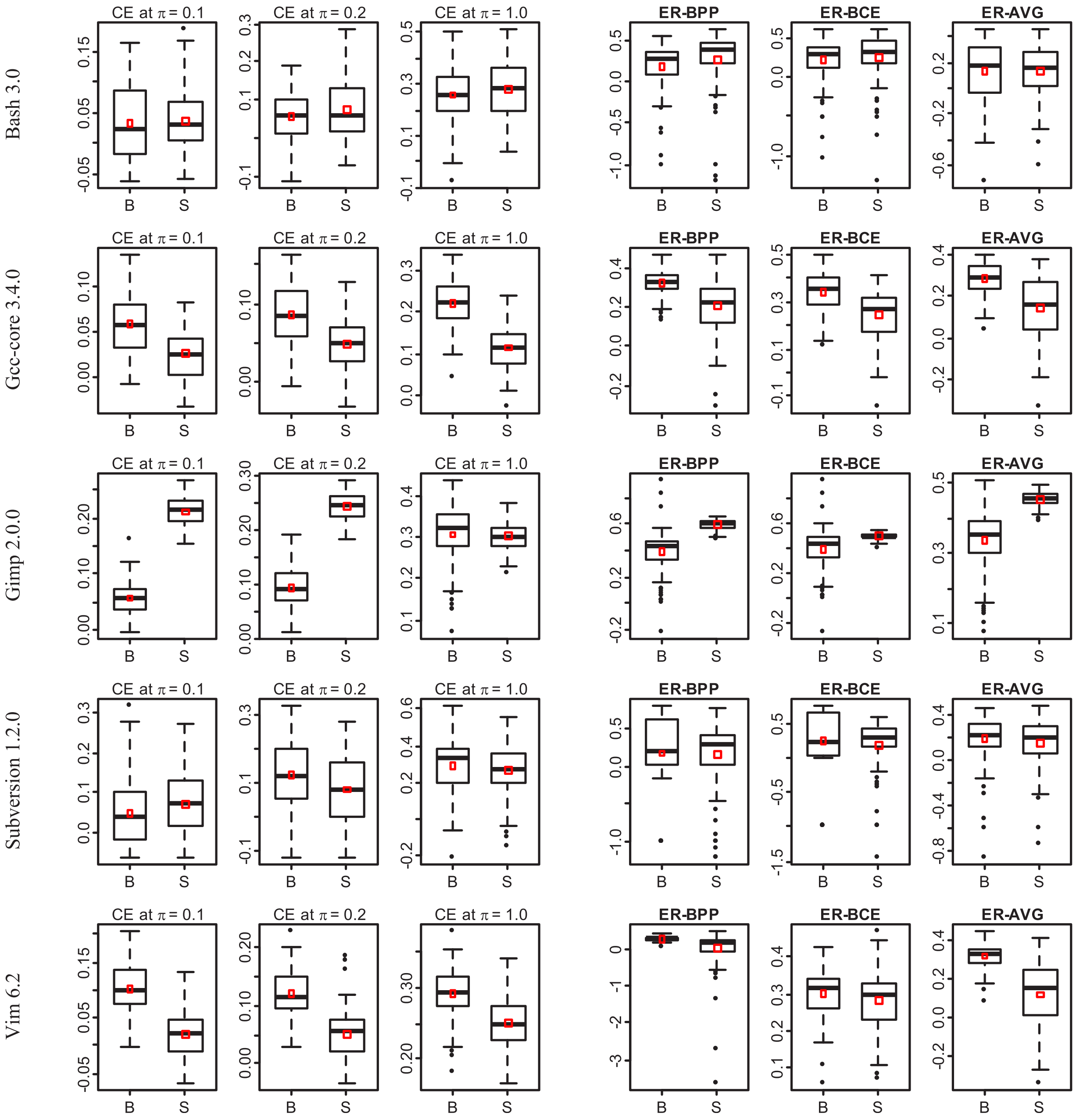


Fig. 2. Ranking/Classification performance comparison under cross-validation: the “B” model versus the “S” model.

$(0.516 \leq |\delta| \leq 0.842)$. In particular, for Vim 6.2, we can see that the median $CE_{0.1}$ is around zero, indicating a performance similar to the random model. For Subversion 1.2.0, the “S” model is similar to the “B” model from the viewpoint of practical application $(0.126 \leq |\delta| \leq 0.236)$. For Gimp 2.0.0, the “S” model is considerably better than the “B” model for the CE at $\pi = 0.1$ and $\pi = 0.2$. However, they are similar at $\pi = 1$ (the BH corrected p-value is 0.688, $|\delta| = 0.213$). The core observation is that, from the viewpoint of practical application, the “S” model has a similar or worse ranking performance than the “B” model in most systems.

- *Classification performance*. For Bash 3.0, the “S” model has a higher median *ER-BPP*, a similar median *ER-BCE*, and a lower median *ER-AVG* compared with the “B” model. The BH corrected p-value in the Wilcoxon signed-rank test is significant for *ER-BPP* but is not significant for both *ER-BCE* and *ER-AVG*. The effect sizes are trivial or small in terms of the Cliff’s δ $(0.046 \leq |\delta| \leq 0.281)$. For Gcc-core 3.4.0, the “S” model has a significantly lower median *ER-BPP*/*ER-BCE*/*ER-AVG* (the BH corrected p-values < 0.001). The effect sizes are large in terms of the Cliff’s δ $(0.573 \leq |\delta| \leq 0.604)$. The results from Vim 6.2 are similar to those from Gcc-core 3.4.0. For subversion 1.2.0, the “S” model is not significantly different from the “B” model, regardless of whether *ER-BPP*, *ER-BCE*, or *ER-AVG* is considered. Again, for Gimp 2.0.0, the “S” model is considerably better than the “B” model. The core observation from *ER-AVG* is that, from the viewpoint of practical application, the “S” model has a similar or worse classification performance than the “B” model in most systems.

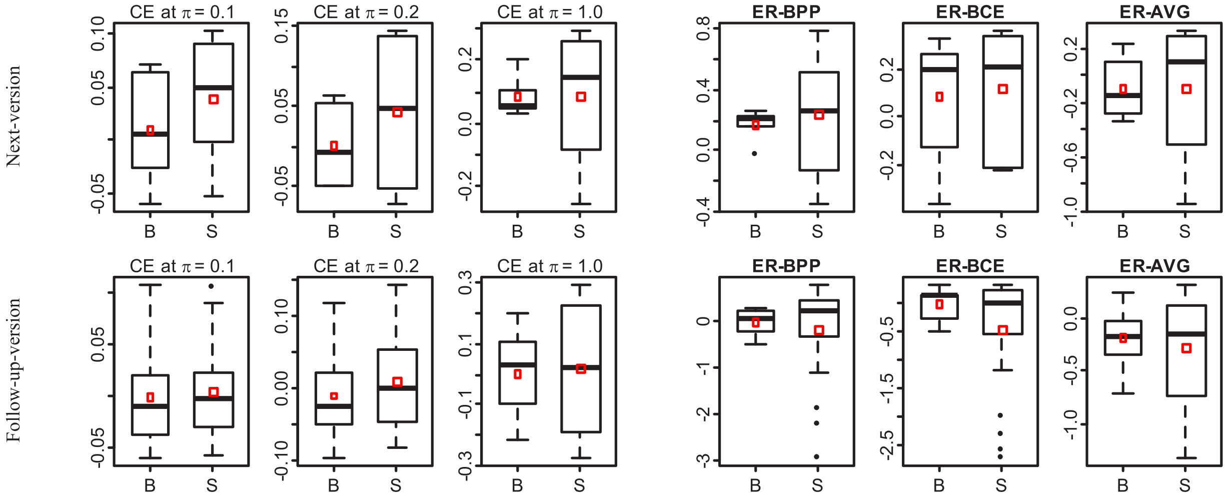


Fig. 3. Ranking/Classification performance comparison under across-version prediction: the “B” model versus the “S” model on the continuous versions of the Bash system.

Overall, the above observations suggest that the “S” model does not outperform the “B” model in effort-aware post-release fault-proneness prediction under the cross-validation evaluation in most systems.

5.3.2 Across-Version Prediction

Fig. 3 employs the box-plot to describe the distributions of the *CE*s at different cut-off values and the *ER-BPP*s/*ER-BCE*s/*ER-AVG*s obtained from across-version prediction for the “B” model and the “S” model for Bash. More specifically, the upper and lower parts respectively report the performance of post-release fault-proneness prediction models under next-version prediction and the performance under follow-up-version prediction.

From Fig. 3, we have the following observations:

- *Ranking performance*. Under the next-version prediction, the “S” model has a larger median *CE* than the “B” model. However, their difference is not statistically significant. The BH corrected p-value in the Wilcoxon signed-rank test is larger than 0.620. Under the follow-up-version prediction, the difference between the “S” model and the “B” model is smaller (compared with next-version prediction). Again, their difference is not statistically significant (the BH corrected p-values > 0.740). The core observation is that the “S” model does not have a significantly better ranking performance than the “B” model.
- *Classification performance*. Under the next-version prediction, the “S” model has a similar median *ER-BPP*/*ER-BCE* and a larger median *ER-AVG* compared with the “B” model. The BH corrected p-values ($>$0.98) in the Wilcoxon signed-rank test show that there is no statistically significant difference between the “S” model and the “B” model. Under the follow-up-version prediction, the “S” model has a similar median *ER-BPP*/*ER-BCE*/*ER-AVG*. Again, their difference is not statistically significant according to the BH corrected p-values ($>$0.720). The core observation is that the “S” model does not have a significantly better classification performance than the “B” model.

Overall, the above observations suggest that the “S” model does not outperform the “B” model in effort-aware post-release fault-proneness prediction under the across-version evaluation.

5.3.3 Across-Project Prediction

Fig. 4 employs the box-plot to describe the distributions of the *CE*s at different cut-off values and the *ER-BPP*s/*ER-BCE*s/*ER-AVG*s obtained from across-project prediction for the “B” model and the “S” model. The five subject systems are Bash 3.0, Gcc-core 3.4.0, Gimp 2.0.0, Subversion 1.2.0, and Vim 6.2.

From Fig. 4, we have the following observations:

- *Ranking performance*. At $\pi = 0.1$, the “S” model has a median *CE* close to zero, thus indicating a ranking performance similar to that of the random model. The “B” model has a higher median *CE* compared with the “S” model. The BH corrected p-value (0.475) in the Wilcoxon signed-rank test shows that

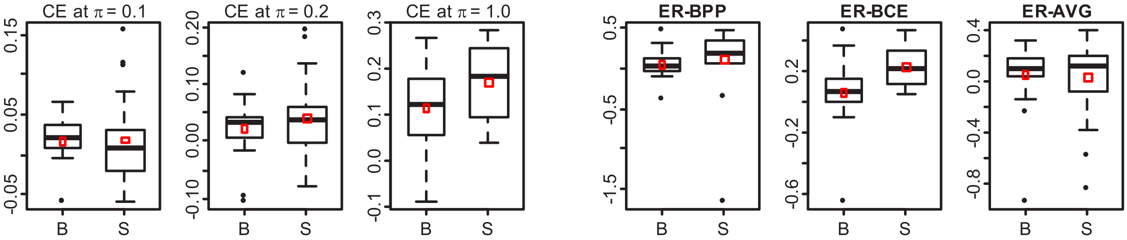


Fig. 4. Ranking/Classification performance comparison under across-project prediction: the “B” model vs. the “S” model on the five data sets.

there is no statistically significant difference between the "S" model and the "B" model. This conclusion also holds for $CE_{0.2}$. At $\pi = 1.0$, the "S" model is better than the "B" model (the BH corrected p-value < 0.05, $|\delta| = 0.320$). The core observation is that, from the viewpoint of practical application, the "S" model is not superior to the "B" model when ranking post-release fault-prone functions.

- *Classification performance*. The "S" model has a higher median *ER-BPP*/*ER-BCE* than the "B" model. For both *ER-BPP* and *ER-BCE*, the BH corrected p-values are significant in the Wilcoxon signed-rank test. However, for *ER-AVG*, the BH corrected p-value (0.622) is not significant and the effect size is trivial ($|\delta| = 0.045$). Therefore, the core observation is that, from the viewpoint of practical application, the "S" model is not superior to the "B" model when classifying post-release fault-prone functions.

Overall, the above observations suggest that the "S" model does not outperform the "B" model in effort-aware post-release fault-proneness prediction under the across-project evaluation.

Combining the results from Sections 5.3.1, 5.3.2, and 5.3.3, we do not have enough evidence to support that the "S" models are superior to the "B" models in the ranking and classification scenarios. Therefore, multivariate logistic regression analyses, from five different data sets, fail to reject the null hypothesis ***H3₀*** corresponding to RQ3: "Slice-based cohesion metrics are not more effective in effort-aware post-release fault-proneness prediction than the most commonly used code and process metrics". This conclusion contrasts with the general expectation that slice-based cohesion metrics should have a stronger predictive ability as they make use of semantic dependence information. One possible explanation for this is that the most commonly used code and process metrics explain significantly more variations in the data than slice-based cohesion metrics (72 versus 28 percent, as shown in the PCA results in Section 5.1), thus having a better predictive ability.

5.4 RQ4: When Used Together with the Most Commonly Used Code and Process Metrics, Can Slice-Based Cohesion Metrics Significantly Improve the Effectiveness of Effort-Aware Post-Release Fault-Proneness Prediction?

In order to answer RQ4, we first use the procedure described in Section 3.4.3 to build the "B+S" model and the "B" model on each data set (Table 11 shows the parameters for each model, including the selected metrics, the regression coefficients, and the corresponding p-values). Then, we compare the prediction effectiveness of the "B+S" and "B" models with respect to both ranking and classification under cross-validation, across-version prediction, and across-project prediction.

5.4.1 Cross-Validation

Fig. 5 employs the box-plot to describe the distributions of the *CE*s at different cut-off values and the *ER-BPP*s/*ER-BCE*s/*ER-AVG*s obtained from 30 times three-fold cross-validation for the "B" model and the "B+S" model. Similar to Section 5.3.1, we report the *CE* performances at $\pi = 0.1$, 0.2, and 1.0 for each model.

From Fig. 5, we have the following observations:

- *Ranking performance*. For all systems, the "B+S" model has a larger median *CE* than the "B" model. For $CE_{0.1}$ in Vim 6.2, the BH corrected p-value is significant (0.081) in the Wilcoxon signed-rank test. For all the other cases, the BH corrected p-values are very significant (<0.001). For Bash 3.0, Gimp 2.0.0, and Subversion 1.2.0, the effect sizes are moderate to large in terms of the Cliff's δ ($0.433 \leq |\delta| \leq 1$). For Gcc-core 3.4.0, the effect size is large ($|\delta| = 0.482$) for $CE_{0.1}$ and is moderate ($|\delta| = 0.366$) for $CE_{0.2}$. For $CE_{0.2}$ and $CE_{1.0}$ in Vim 6.2, the effect sizes are large in terms of the Cliff's δ ($0.426 \leq |\delta| \leq 0.456$). The core observation is that, from the viewpoint of practical application, the "B+S" model has a substantially better ranking performance than the "B" model.
- *Classification performance*. For all systems except Gcc-core 3.4.0, the "B+S" model has a higher median *ER-BPP*/*ER-BCE*/*ER-AVG* than the "B" model. The BH corrected p-values in the Wilcoxon signed-rank test are very significant (< 0.006). For *ER-BPP*, the effect sizes ($0.581 \leq |\delta| \leq 0.781$) are large on Bash 3.0, Gimp 2.0.0, and Vim 6.2 and the effect size is moderate on Subversion 1.2.0 ($|\delta| = 0.311$). For *ER-BCE*, the effect sizes ($0.455 \leq |\delta| \leq 0.490$) are moderate to large on Bash 3.0 and Vim 6.2. For *ER-AVG*, the effect sizes are moderate to large in terms of the Cliff's δ ($0.421 \leq |\delta| \leq 0.772$) on Bash 3.0, Gimp 2.0.0, and Subversion 1.2.0. For Gcc-core 3.4.0, the effect size of *ER-AVG* is around moderate ($|\delta| = 0.308$). The core observation is that the "B+S" model has a better classification performance than the "B" model.

Overall, the above observations suggest that the "B+S" model outperforms the "B" model in effort-aware post-release fault-proneness prediction under the cross-validation evaluation.

5.4.2 Across-Version Prediction

Fig. 6 employs the box-plot to describe the distributions of the *CE*s at different cut-off values and the *ER-BPP*s/*ER-BCE*s/*ER-AVG*s obtained from across-version prediction for the "B+S" model and the "B" model for Bash. The upper and lower parts respectively report the performance under next-version prediction and under follow-up-version prediction.

From Fig. 6, we have the following observations:

- *Ranking performance*. Under the next-version prediction, the "B+S" model has a larger median *CE* than the "B" model. The BH corrected p-values in the Wilcoxon signed-rank test are smaller than 0.10, thus indicating significant differences. In particular, the effect sizes are large ($0.611 \leq |\delta| \leq 0.778$), regardless of whether $CE_{0.1}$, $CE_{0.2}$, or $CE_{1.0}$ is considered. Under the follow-up-version prediction, the "B+S" model also has a larger median *CE* than the "B" model. According to the BH corrected p-values (<0.001), there is a very significant difference. The effect sizes are large for $CE_{0.1}$ and $CE_{0.2}$ ($0.587 \leq |\delta| \leq 0.692$). The core

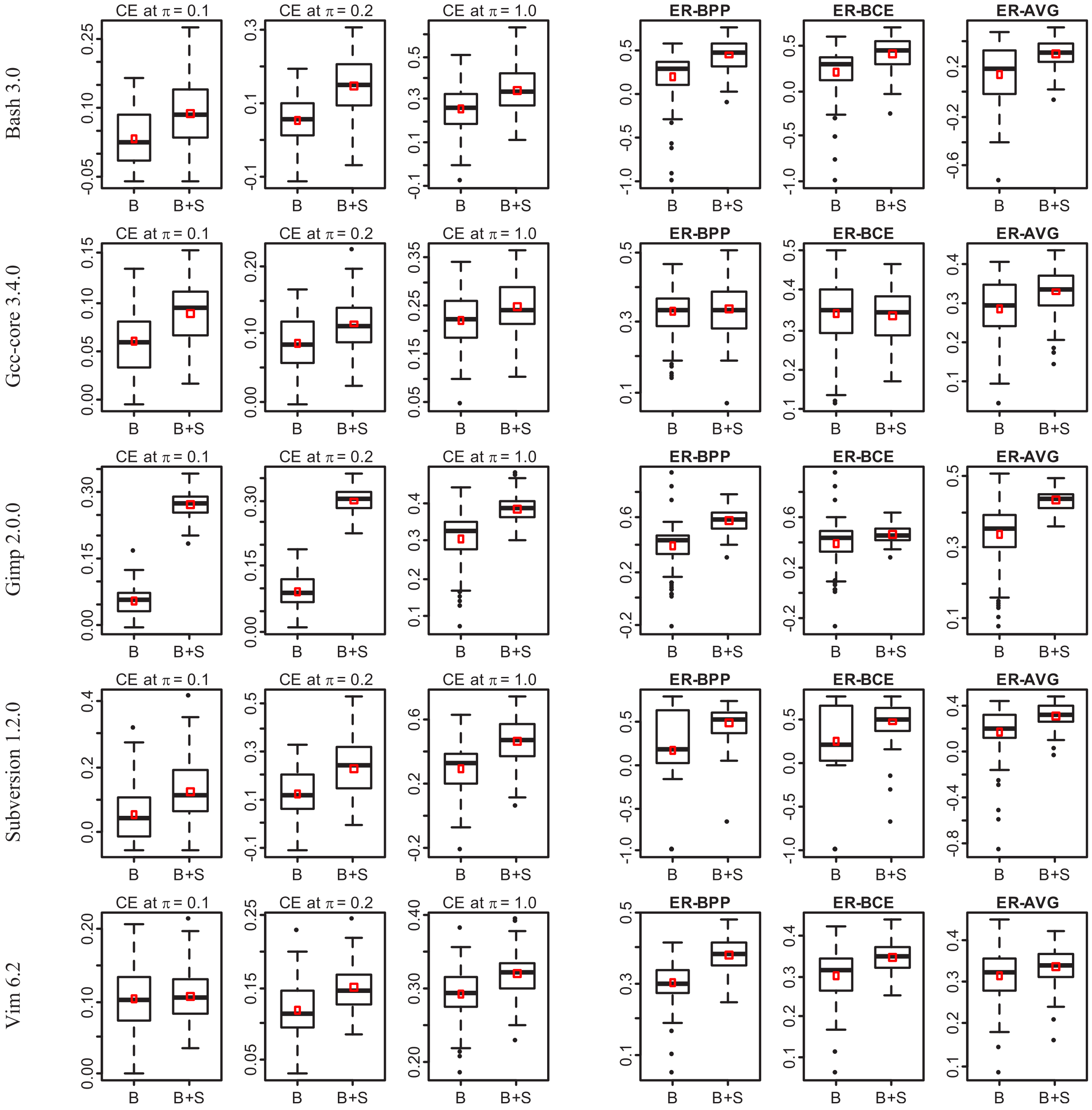


Fig. 5. Ranking/Classification performance comparison under cross-validation: the “B” model versus the “B+S” model.

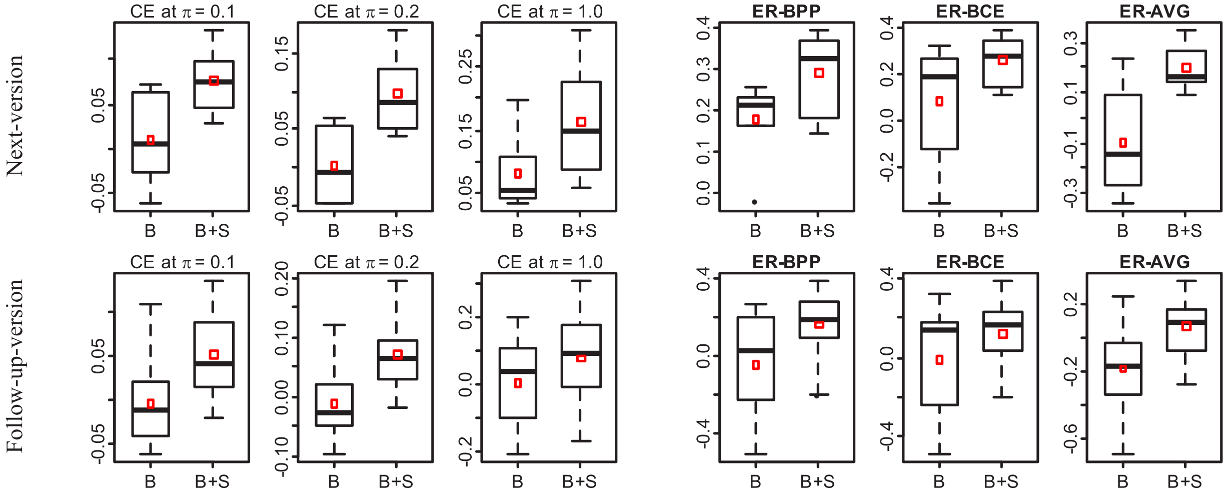


Fig. 6. Ranking/Classification performance comparison under across-version prediction: the “B” model versus the “B+S” model on the continuous versions of the Bash system.

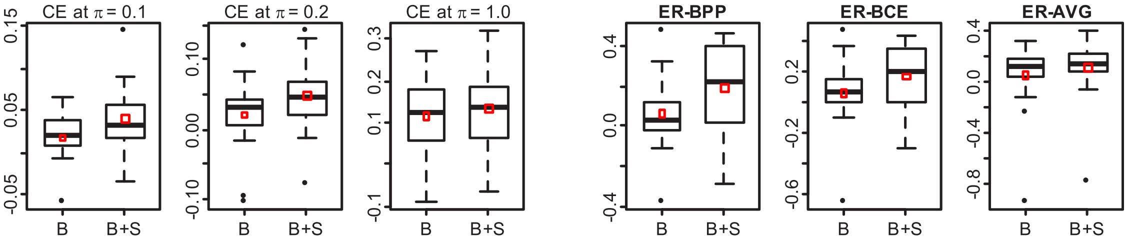


Fig. 7. Ranking/Classification performance comparison under across-project prediction: the “B” model versus the “B+S” model for the five data sets.

observation is that the “B+S” model has a substantially better ranking performance than the “B” model.

- *Classification performance*. Under the next-version prediction, the “B+S” model has a higher median *ER-BPP*/*ER-BCE*/*ER-AVG* than the “B” model. The BH corrected p-values (<0.07) show that there is a statistically significant difference. The effect sizes are large ($0.444 \leq |\delta| \leq 0.722$), regardless of whether *ER-BPP*, *ER-BCE*, or *ER-AVG* is considered. Under the follow-up-version prediction, the “B+S” model also has a higher classification performance value. The BH corrected p-values (<0.002) show that there is a very significant difference. The effect sizes are moderate to large in terms of the Cliff's δ ($0.438 \leq |\delta| \leq 0.551$) for *ER-BPP*/*ER-AVG*. The core observation is that the “B+S” model has a substantially better classification performance than the “B” model.

Overall, the above observations suggest that the “B+S” model outperforms the “B” model in effort-aware post-release fault-proneness prediction under the across-version evaluation. In addition, from Fig. 6, we also have the following two interesting observations. First, the “B” model has a median $CE_{0.1}/CE_{0.2}$ close to zero and a negative median *ER-AVG*. This indicates that the “B” model is not better than the random model under across-version prediction. However, the “B+S” model is much better than the random model. Second, for a given model, the next-version prediction in general produces a better performance than the follow-up-version prediction. This suggests that, in practice, we should use the data from the latest versions (rather than the out-of-date versions) of a system to build a post-release fault-proneness prediction model.

5.4.3 Across-Project Prediction

Fig. 7 employs the box-plot to describe the distributions of the *CE*s at different cut-off values and the *ER-BPP*s/*ER-BCE*s/*ER-AVG*s obtained from across-project prediction for the “B” and “B+S” models.

From Fig. 7, we have the following observations:

- *Ranking performance*. The “B+S” model has a higher median *CE* than the “B” model. For $CE_{0.1}$ and $CE_{0.2}$, the BH corrected p-values show that there is a statistically significant difference. In particular, the effect sizes are almost moderate in terms of the Cliff's δ ($0.320 \leq |\delta| \leq 0.325$). The core observation is that, from the viewpoint of practical application, the “B+S” model is superior to the “B” model when ranking post-release fault-prone functions.
- *Classification performance*. The “B+S” model has a higher median *ER-BPP*/*ER-BCE*/*ER-AVG* than the “B” model. The BH corrected p-values (< 0.07) show that there is a statistically significant difference. For *ER-BPP*, the effect size is moderate ($|\delta| = 0.362$). For *ER-BCE*/*ER-AVG*, the effect sizes are small ($0.235 \leq |\delta| \leq 0.248$). The core observation is that the “B+S” model is superior to the “B” model when classifying post-release fault-prone functions.

Overall, the above observations suggest that the “B+S” model outperforms the “B” model in effort-aware post-release fault-proneness prediction under the across-project evaluation.

Combining the results from Sections 5.4.1, 5.4.2, and 5.4.3, we have a strong evidence to support that the “B+S” models are superior to the “B” models in the ranking and classification scenarios. Therefore, multivariate logistic regression analyses, from five different data sets, reject the null hypothesis ***H4₀*** corresponding to RQ4: “The combination of slice-based cohesion metrics with the most commonly used code and process metrics are not more effective in effort-aware post-release fault-proneness prediction than the combination of the most commonly used code and process metrics”. In other words, our results are in favor of the alternative hypothesis ***H4_A***, i.e. the combination of slice-based cohesion metrics with the most commonly used code and process metrics lead to a more effective effort-aware post-release fault-proneness prediction. This conclusion is consistent with intuition as the experimental results in Section 5.1 show that slice-based cohesion metrics are complementary to the most commonly used code and process metrics. As such, using slice-based cohesion metrics and the most commonly used code and process metrics together should provide a better ability to predict post-release fault-proneness.

6 Threats to Validity

In this section, we analyze the most important threats to the construct, internal, and external validity of our study. Construct validity is the degree to which the variables used in a study accurately measure the concept they purport to measure. Internal validity is the degree to which conclusions can be drawn about the causal effect of independent variables on the dependent variables. External validity is the degree to which the findings of a study can be generalized to data not used in that particular study.

6.1 Construct Validity

The dependent variable used in this study is a binary variable that represents the detection or non-detection of a post-release fault in a function. For each investigated system, we collected this information by contrasting the latest patch version or the latest bug-fixing version. Each patch version or bug-fixing version added no new features to the corresponding system. In this sense, the fault data were reliable. Thus, the construct validity of the dependent variable in our study is considered acceptable.

The independent variables used in this study are the most commonly used code and process metrics and slice-based cohesion metrics. On the one hand, we leveraged Understand, a mature commercial tool, to collect the most commonly used code and process metrics. This tool has been used to collect the metric data in many previous studies [42], [44], [48], [49]. On the other hand, we developed two plug-ins called INFERCON and SLIBCOM based on Frama-C, a powerful and extensible framework for the analysis of industrial-size C programs, to compute slice-based cohesion metrics. We took two measures to ensure that slice-based cohesion metrics were correctly calculated. First, we used INFERCON and SLIBCOM to collect the metric data for a number of functions (developed by ourselves) in which different control flows and data flows were implemented. We found that INFERCON and SLIBCOM always produced correct results in our test. Second, we randomly examined a small number of (about 50 functions in each system) C functions in Bash 3.0, Gcc-core 3.4.0, Gimp 2.0.0, Subversion 1.2.0, and Vim 6.2 and found that the metric data produced by INFERCON and SLIBCOM were reliable. Therefore, the construct validity of the independent variables in our study is also satisfactory.

6.2 Internal Validity

There are four threats to the internal validity of our study. The first threat is the unknown effect of the exclusion of the functions that have an “undefined” value. In our study, a number of functions were assigned an “undefined” metric value when they cannot be analyzed within 30 min or have no output variables reported by Frama-C. These functions were excluded before we conducted the subsequent analyses. However, this exclusion should not have a large influence on our conclusions, as only a small percentage of functions were excluded for the five subject systems.

The second threat to the internal validity of our study is the unknown effect of the deviation of the independent variables from the normal distribution. In our study, we used the raw data to build the logistic regression models when investigating RQ3 and RQ4. In other words, we did not take into account whether the independent variables follow a normal distribution. The reason is that, in logistic regression, there is no assumption related to normal distribution. However, previous studies suggested applying the log transformation to the independent variables to make them close to a normal distribution, as it might lead to a better model [40]. To eliminate this threat, we applied the log transformation and rerun the analyses. We found that the conclusions for RQ3 and RQ4 did not change before and after the log transformation.

The third threat to the internal validity of our study is the unknown effect of the method to define the relative risk of a function. In our study, for each function, we use the ratio of the predicted value by a naïve logistic regression model to its *SLOC* as the relative risk of the function. However, in the literature, most studies use the predicted value by the naïve logistic regression model as the relative risk of a function. To eliminate this threat, we used the predicted value by the naïve logistic regression model as the relative risk and rerun the analyses for RQ3 and RQ4. We found that the results were largely identical.

The fourth threat to the internal validity of our study is the unknown effect of the actual development process of the code. In our study, we use post-release faults to examine the usefulness of slice-based cohesion metrics in fault-proneness prediction when investigating RQ2, RQ3 and RQ4. It is possible that some functions have no post-release faults simply because they undergo more rigorous quality-assurance procedures (e.g., inspections, walkthroughs, and testing). In other words, the actual development process may have a confounding effect on our findings. However, we believe that this confounding effect should not have a substantial influence, as similar findings are obtained from different systems in our study. Nonetheless, this threat needs to be eliminated by controlled experiments in the future work.

6.3 External Validity

The most important threat to the external validity of this study is that our findings may not be generalized to other systems. In our experiments, we use five long-lived and widely used software systems, including a command language interpreter, a compiler collection, an image manipulation program, an open-source version control system, and an advanced text editor, as the subject systems. The experimental results drawn from these subject systems, which belong to different application domains, are quite consistent. Furthermore, the data sets collected from these systems are large enough to draw statistically meaningful conclusions. We believe that our study makes a significant contribution to the software engineering body of empirical knowledge about the usefulness of slice-based cohesion metrics. Nonetheless, we do not claim that our findings can be generalized to all systems, as the subject systems under study might not be representative of systems in general. To mitigate this threat, we hope that other researchers will replicate our study across a wide variety of systems in the future.

7 Related Work

Although slice-based cohesion metrics have been proposed for many years, to date little work has been performed to empirically relate them to code quality. Meyers and Binkley [13], [18] conducted a large-scale empirical study of five statement-level slice-based cohesion metrics, including *Coverage*, *MaxCoverage*, *MinCoverage*, *Tightness* and *Overlap*. They analyzed the relations between slice-based cohesion metrics and code size metrics and found that slice-based cohesion metrics provided a unique view of a program. This finding is consistent with one of our findings that slice-based cohesion metrics capture additional dimensions of software quality compared with the most commonly used code and process metrics. From two longitudinal studies, Meyers and Binkley found that slice-based cohesion metrics were able to quantify the deterioration of a program as it evolved. They also provided the baseline values for slice-based cohesion metrics, which can be used to focus the attention of reengineering effort. However, they did not relate slice-based cohesion metrics to fault-proneness. Another major difference between their study and our study is the definition of the output variables of a function. In their study, the output variables of a function consist of function return value and modified global variables. In our study, in addition to function return value and modified global variables, the output variables of a function also include modified reference parameters and standard outputs.

Black et al. [15] empirically investigated the ability of two statement-level slice-based cohesion metrics, *Tightness* and *Overlap*, to distinguish between faulty and not-faulty functions. In their study, they combined the nineteen versions of a small program called Barcode to obtain a single data set. In particular, Black et al. extracted the fault data manually using the online report logs of Barcode but did not explicitly mention whether they were pre-release or post-release faults. Their results showed that not-faulty functions tended to have higher *Tightness* and *Overlap* values compared with faulty functions. The difference between faulty and not-faulty functions for *Tightness* was significant at the significance level of 0.05, while the difference for *Overlap* was significant at the significance level of 0.10. This indicated that *Tightness* had a stronger correlation with fault-proneness. This finding is somewhat consistent with our univariate logistic regression analysis result in Section 5.2. In our study, we found that *Tightness* and *Overlap* were both significantly related to fault-proneness and furthermore *Tightness* had a stronger correlation with fault-proneness (as indicated by ΔOR). Black et al. [17] had planned to test the hypotheses relating three statement-level slice-based cohesion metrics (*Tightness*, *Overlap*, and *Coverage*) and fault-proneness. However, they failed to do this due to lack of data. Compared with their work, we performed an in-depth and comprehensive study on the relationships between slice-based cohesion metrics and post-release fault-proneness.

Dallal [8] proposed a data-token-level slice-based cohesion metric called *SBFC*. He used six very small programs developed by students to examine the correlations between *SBFC* and the other three data-token-level metrics (*SFC*, *WFC*, and *A*). He found that there was a strong correlation between *SBFC* and these metrics and hence concluded that *SBFC* provided a useful alternative to them. However, the actual usefulness of *SBFC* for predicting fault-proneness was not examined. Counsell et al. [5] proposed a novel statement-level slice-based cohesion metric called *NHD* and compared it with the statement-level *Tightness*/*Overlap*. Based on Barcode, they found that *NHD* had a lower correlation to module size (measured by *SLOC*) compared with *Tightness* and *Overlap*. Again, the actual usefulness of *NHD* for predicting fault-proneness was not examined. In our study, we used univariate logistic regression to relate *SBFC*/*NHD* to post-release fault-proneness. Our results showed that data-token-level *SBFC* and *NHD* were significantly related to post-release fault-proneness.

8 Conclusion and Future Work

In this paper, we empirically examine the actual usefulness of slice-based cohesion metrics in the context of effort-aware post-release fault-proneness prediction. Our findings from industrial-size systems show that they are not redundant to a set of the most commonly used code and process metrics. Consistent with the general expectation, we find that most of them are significantly negatively related to post-release fault-proneness. Contrary to the general expectation, we find that they do not outperform the most commonly used code and process metrics when predicting post-release fault-proneness for both ranking and classification in most systems. However, we find that the combination of slice-based cohesion metrics and the most commonly used code and process metrics produces more effective models for the prediction of post-release fault-proneness than the combination of the most commonly used code and process metrics alone. These results provide valuable data for better understanding slice-based cohesion metrics and for guiding the development of better post-release fault-proneness prediction models in practice.

Our study only investigates the actual usefulness of slice-based cohesion metrics for procedural systems. In the last decade, object-orientation has been the dominant programming paradigm. In particular, many slice-based cohesion metrics for classes have been proposed [10], [46]. However, little is currently known on whether they are of practical value for predicting post-release fault-prone classes. In the future, an interesting work is hence to extend our current study to object-oriented systems.

Acknowledgments

The authors would like to thank the editor and anonymous reviewers for their very insightful comments and constructive suggestions in greatly improving the quality of this paper. They also wish to thank Professor Mark Harman for his very helpful suggestions when revising their paper. The work in this paper is supported by the National Key Basic Research and Development Program of China (2014CB340702), the National Natural Science Foundation of China (61432001, 91318301, 61321491, 61379045, 61272082, 61300051, 61472175), the National Natural Science Foundation of Jiangsu Province (BK20130014), the National Science and Technology Major Project of China (2012ZX01039-004), the Hong Kong Competitive Earmarked Research Grant (PolyU5219/06E), and PolyU Grant (4-6934). Yuming Zhou is the corresponding author of this paper.

References

[1] E. Yourdon and L. L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, vol. 5. Englewood Cliffs, NJ, USA: Prentice-Hall, 1979.

[2] M. Page-Jones, *The Practical Guide to Structured Systems Design*, vol. 2. Englewood Cliffs, NJ, USA: Prentice-Hall, 1988.

[3] L. M. Ott and J. J. Thuss, “Slice based metrics for estimating cohesion,” in *Proc. 1st Int. Softw. Metrics Symp.*, 1993, pp. 71–81.

[4] J. M. Bieman and L. M. Ott, “Measuring functional cohesion,” *IEEE Trans. Softw. Eng.*, vol. 20, no. 8, pp. 644–657, Aug. 1994.

[5] S. Counsell, T. Hall, and D. Bowes, “A theoretical and empicial analysis of three slice-based metrics for cohesion,” *Int. J. Softw. Eng. Knowl. Eng.*, vol. 20, no. 05, p. 609, 2010.

[6] M. Harman, S. Danicic, B. Sivagurunathan, B. Jones, Y. Sivagurunathan, and E. Grove, “Cohesion Metrics,” in *Proc. 8th Int. Qual. WEEK*, San Francisco, CA, USA, May 29, 1995, pp. 1–14.

[7] H. D. Longworth, “Slice based program metrics,” Master thesis, Michigan Technol. Univ., Houghton, MI, USA, 1985.

[8] J. Al Dallal, “Software similarity-based functional cohesion metric,” *IET Softw.*, vol. 3, no. 1, pp. 46–57, Feb. 2009.

[9] M. Weiser, “Program Slicing,” *IEEE Trans. Softw. Eng.*, vol. SE-10, no. 4, pp. 352–357, Jul. 1984.

[10] L. M. Ott and J. M. Bieman, "Program slices as an abstraction for cohesion measurement," *Inf. Softw. Technol.*, vol. 40, no. 11/12, pp. 691–699, Dec. 1998.

[11] K. J. Ottenstein and L. M. Ottenstein, "The program dependence graph in a software development environment," *ACM Sigplan Notices*, vol. 19, no. 5, pp. 177–184, 1984.

[12] P. D. Green, P. C. R. Lane, A. Rainer, and S. Scholz, "An introduction to slice-based cohesion and coupling metrics," Tech. Rep. No. 488, Univ. Hertfordshire, Hertfordshire, AL, USA, 2009.

[13] T. M. Meyers and D. Binkley, "An empirical study of slice-based cohesion and coupling metrics," *ACM Trans. Softw. Eng. Methodol.*, vol. 17, no. 1, pp. 1–27, Dec. 2007.

[14] L. M. Ott and J. J. Thuss, "The relationship between slices and module cohesion," in *Proc. 11th Int. Conf. Softw. Eng.*, 1989, pp. 198–204.

[15] S. Black, S. Counsell, T. Hall, and D. Bowes, "Fault analysis in OSS based on program slicing metrics," in *Proc. 35th Euromicro Conf. Softw. Eng. Adv. Appl.*, Washington, DC, USA, 2009, pp. 3–10.

[16] S. Counsell, T. Hall, E. Nasseri, and D. Bowes, "An analysis of the 'inconclusive' change report category in OSS assisted by a program slicing metric," in *Proc. 36th Euromicro Conf. Softw. Eng. Adv. Appl.*, 2010, pp. 283–286.

[17] S. Black, S. Counsell, T. Hall, and P. Wernick, "Using program slicing to identify faults in software," in *Proc. Dagsthul Seminar Beyond Program Slicing*, Nov. 06–11, 2005.

[18] T. M. Meyers and D. Binkley, "Slice-based cohesion metrics and software intervention," in *Proc. 11th Working Conf. Reverse Eng.*, 2004, pp. 256–265.

[19] M. Weiser, "Program slicing," in *Proc. 5th Int. Conf. Softw. Eng.*, 1981, pp. 439–449.

[20] L. M. Ott and J. M. Bieman, "Effects of software changes on module cohesion," in *Proc. Conf. Softw. Maintenance*, 1992, pp. 345–353.

[21] D. Bowes, T. Hall, and A. Kerr, "Program slicing-based cohesion measurement: The challenges of replicating studies using metrics," in *Proc. 2nd Int. Workshop Emerging Trends Softw. Metrics*, 2011, pp. 75–80.

[22] V. R. Basili and B. T. Perricone, "Software errors and complexity: An empirical investigation," *Commun. ACM*, vol. CACM-27, no. 1, pp. 42–52, Jan. 1984.

[23] N. E. Fenton and N. Ohlsson, "Quantitative analysis of faults and failures in a complex software system," *IEEE Trans. Softw. Eng.*, vol. 26, no. 8, pp. 797–814, Aug. 2000.

[24] L. Hatton, "Reexamining the fault density component size connection," *IEEE Softw.*, vol. 14, no. 2, pp. 89–97, Mar. 1997.

[25] K.-H. Moller and D. J. Paulish, "An empirical investigation of software fault distribution," in *Proc. 1st Int. Softw. Metrics Symp.*, 1993, pp. 82–90.

[26] J. C. Munson and T. M. Khoshgoftaar, "The detection of fault-prone programs," *IEEE Trans. Softw. Eng.*, vol. 18, no. 5, pp. 423–433, May 1992.

[27] N. Ohlsson and H. Alberg, "Predicting fault-prone software modules in telephone switches," *IEEE Trans. Softw. Eng.*, vol. 22, no. 12, pp. 886–894, Dec. 1996.

[28] V. R. Basili, L. C. Briand, and W. L. Melo, "A validation of object-oriented design metrics as quality indicators," *IEEE Trans. Softw. Eng.*, vol. 22, no. 10, pp. 751–761, Oct. 1996.

[29] D. P. Darcy, C. F. Kemerer, S. A. Slaughter, and J. E. Tomayko, "The structural complexity of software an experimental test," *IEEE Trans. Softw. Eng.*, vol. 31, no. 11, pp. 982–995, Nov. 2005.

[30] M. H. Halstead, *Elements of Software Science (Operating and Programming Systems Series)*. New York, NY, USA: Elsevier, 1977.

[31] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for Eclipse," in *Proc. Int. Workshop Predictor Models Softw. Eng.*, 2007, pp. 9–9.

[32] D. A. Belsley, E. Kuh, and R. E. Welsch, *Regression Diagnostics: Identifying Influential Data and Sources of Collinearity*, vol. 571. New York, NY, USA: Wiley, 2005.

[33] L. C. Briand, J. Wüst, J. W. Daly, and D. Victor Porter, "Exploring the relationships between design measures and software quality in object-oriented systems," *J. Syst. Softw.*, vol. 51, no. 3, pp. 245–273, 2000.

[34] E. Arisholm, L. C. Briand, and E. B. Johannessen, "A systematic and comprehensive investigation of methods to build and evaluate fault prediction models," *J. Syst. Softw.*, vol. 83, no. 1, pp. 2–17, Jan. 2010.

[35] T. Mende and R. Koschke, "Effort-aware defect prediction models," in *Proc. 14th Eur. Conf. Softw. Maintenance Reeng.*, Washington, DC, USA, 2010, pp. 107–116.

[36] Y. Kamei, S. Matsumoto, A. Monden, K. Matsumoto, B. Adams, and A. E. Hassan, "Revisiting common bug prediction findings using effort-aware models," in *Proc. IEEE Int. Conf. Softw. Maintenance*, Washington, DC, USA, 2010, pp. 1–10.

[37] 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 Trans. Softw. Eng.*, vol. 39, no. 6, pp. 757–773, Jun. 2013.

[38] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, "A systematic literature review on fault prediction performance in software engineering," *IEEE Trans. Softw. Eng.*, vol. 38, no. 6, pp. 1276–1304, Nov. 2012.

[39] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," *IEEE Trans. Softw. Eng.*, vol. 37, no. 6, pp. 772–787, Nov. 2011.

[40] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *IEEE Trans. Softw. Eng.*, vol. 33, no. 1, pp. 2–13, Jan. 2007.

[41] A. I. Schein, L. K. Saul, and L. H. Ungar, "A generalized linear model for principal component analysis of binary data," in *Proc. 9th Int. Workshop Artif. Intell. Statist.*, 2003, vol. 38, p. 46.

[42] Y. Zhou, B. Xu, and H. Leung, "On the ability of complexity metrics to predict fault-prone classes in object-oriented systems," *J. Syst. Softw.*, vol. 83, no. 4, pp. 660–674, 2010.

[43] K. El Emam, S. Benlarbi, N. Goel, and S. N. Rai, "The confounding effect of class size on the validity of object-oriented metrics," *IEEE Trans. Softw. Eng.*, vol. 27, no. 7, pp. 630–650, Jul. 2001.

[44] Y. Zhou, H. Leung, and B. Xu, "Examining the potentially confounding effect of class size on the associations between object-oriented metrics and change-proneness," *IEEE Trans. Softw. Eng.*, vol. 35, no. 5, pp. 607–623, Sep./Oct. 2009.

[45] J. Cohen, *Statistical Power Analysis for the Behavioral Sciences*. Evanston, IL, USA: Routledge, 2013.

[46] Y. Zhou, J. Wang, J. Zhao, H. Lu, and B. Xu, "A novel class cohesion measure based on slices," later breaking paper, in *Proc. 10th Int. Softw. Metrics Symp.*, 2004.

[47] Y. Benjamini and Y. Hochberg, "Controlling the false discovery rate: A practical and powerful approach to multiple testing," *J. Roy. Stat. Soc. Series B Methodol.*, vol. 57, no. 1, pp. 289–300, 1995.

[48] K. Pan, S. Kim, and J. E. Whitehead, "Bug classification using program slicing metrics," in *Proc. 6th IEEE Int. Workshop Source Code Anal. Manipulation*, Sep. 2006, pp. 31–42.

[49] A. G. Koru and J. Tian, "Comparing high-change modules and modules with the highest measurement values in two large-scale open-source products," *IEEE Trans. Softw. Eng.*, vol. 31, no. 8, pp. 625–642, Aug. 2005.

[50] F. Rahman, D. Posnett, and P. Devanbu, "Recalling the 'imprecision' of cross-project defect prediction," in *Proc. ACM SIGSOFT 20th Int. Symp. Found. Softw. Eng.*, 2012, pp. 61:1–61:11.

[51] L. C. Briand, C. Bunse, and J. W. Daly, "A controlled experiment for evaluating quality guidelines on the maintainability of object-oriented designs," *IEEE Trans. Softw. Eng.*, vol. 27, no. 6, pp. 513–530, Jun. 2001.

[52] P. L. Li, J. Herbsleb, M. Shaw, and B. Robinson, "Experiences and results from initiating field defect prediction and product test prioritization efforts at ABB Inc.," in *Proc. 28th Int. Conf. Softw. Eng.*, 2006, pp. 413–422.

[53] Y. Zhou, B. Xu, H. Leung, and L. Chen, "An in-depth study of the potentially confounding effect of class size in fault prediction," *ACM Trans. Softw. Eng. Methodol.*, vol. 23, no. 1, pp. 10:1–10:51, Feb. 2014.

[54] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Proc. 27th Int. Conf. Softw. Eng.*, New York, NY, USA, 2005, pp. 284–292.

[55] 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 *Proc. 7th Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng.*, 2009, pp. 91–100.

[56] J. W. Hunt and M. D. McIlroy, *An Algorithm for Differential File Comparison*. Murray Hill, NJ, USA: Bell Laboratories, 1976.

[57] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-C: A software analysis perspective," in *Proc. 10th Int. Conf. Softw. Eng. Formal Methods*, 2012, pp. 233–247.

[58] J. Romano, J. D. Kromrey, J. Coraggio, and J. Skowronek, "Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen's d for evaluating group differences on the NSSE and other surveys," in *Proc. Annu. Meeting Florida Assoc. Institutional Res.*, 2006, pp. 1–33.

[59] M. H. Kutner, C. Nachtsheim, and J. Neter, *Applied Linear Regression Models*. New York, NY, USA: McGraw-Hill/Irwin, 2004.

[60] F. Rahman, S. Khatri, E. T. Barr, and P. Devanbu, "Comparing static bug finders and statistical prediction," in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 424–434.

[61] D. E. Farrar and R. R. Glauber, "Multicollinearity in regression analysis: The problem revisited," *Rev. Econ. Stat.*, vol. 49, no. 1, pp. 92–107, 1967.

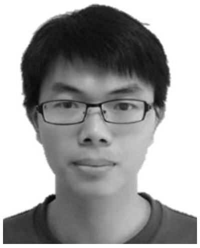
[62] J. Al Dallal, "Measuring the discriminative power of object-oriented class cohesion metrics," *IEEE Trans. Softw. Eng.*, vol. 37, no. 6, pp. 788–804, Nov. 2011.

[63] J. Al Dallal, "Fault prediction and the discriminative powers of connectivity-based object-oriented class cohesion metrics," *Inf. Softw. Technol.*, vol. 54, no. 4, pp. 396–416, 2012.

[64] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *Proc. 30th Int. Conf. Softw. Eng.*, New York, NY, USA, 2008, pp. 181–190.

[65] R. Shatnawi and W. Li, "The effectiveness of software metrics in identifying error-prone classes in post-release software evolution process," *J. Syst. Softw.*, vol. 81, no. 11, pp. 1868–1882, 2008.

[66] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *Proc. 28th Int. Conf. Softw. Eng.*, 2006, pp. 452–461.



Yibiao Yang received the BS degree from Southwest Jiaotong University in 2010. He is currently working toward the PhD degree in the Department of Computer Science and Technology at Nanjing University. His main research interests include empirical software engineering and software analysis.

Yuming Zhou received the PhD degree in computer science from Southeast University in 2003. From January 2003 to December 2004, he was a researcher at Tsinghua University. From February 2005 to Februray 2008, he was a researcher at Hong Kong Polytechnic University. He is currently a professor in the Department of Computer Science and Techology at Nanjing University. His main research interests are empirical software engineering.

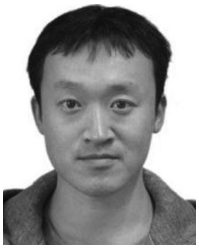
Hongmin Lu received the PhD degree in computer science from Southeast University in 2013. She is a lecturer in the Department of Computer Science and Technology at Nanjing University. Her research interests include software metrics and software maintenance.

Lin Chen received the PhD degree in computer science from Southeast University in 2009. He is currently a lecturer in the Department of Computer Science and Technology at Nanjing University. His research interests include software analysis and software maintenance.

Zhenyu Chen received the BS and PhD degrees in mathematics from Nanjing University. He was a postdoctoral researcher at the School of Computer Science and Engineering, Southeast University, China. He is currently an associate professor in the School of Software at Nanjing University. His research interests include software analysis and testing. He has about 70 publications at major venues including *TOSEM*, *JSS*, *SQJ*, *IJSEKE*, ISSTA, ICST, QSIC, etc. He has served as a PC co-chair of QSIC 2013, AST2013, IWPD2012 and the program committee member of many international conferences. He has received research funding from several competitive sources such as NSFC. He is a member of the IEEE.

Baowen Xu received the BS, MS, and PhD degrees in computer science from Wuhan University, Huazhong University of Science and Technology, and Beihang University, respectively. He is currently a professor in the Department of Computer Science and Technology at Nanjing University. His main research interests include programming languages, software testing, software maintenance, and software metrics. He is a member of the IEEE and the IEEE Computer Society.

Hareton Leung received the PhD degree in computer science from University of Alberta. He is an associate professor and the director at the Laboratory for Software Development and Management in the Department of Computing, the Hong Kong Polytechnic University. He currently serves on the editorial board of *Software Quality Journal* and *Journal of the Association for Software Testing*. His research interests include software testing, software maintenance, quality and process improvement, and software metrics.



Zhenyu Zhang received the PhD degree in computer science from University of Hong Kong. He is an associate professor at the State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences. His current research interests are program debugging and testing for software and systems, and the reliability issues of web-based services and cloud-based systems. He has published research results in venues such as *Computer, TSC*, ICSE, FSE, ASE, and WWW.