# Slice-Based Cognitive Complexity Metrics for Defect Prediction

Basma S. Alqadi
*Department of Computer Science*
*Imam Muhammad Ibn Saud Islamic University*
Riyadh, Saudi Arabia
*Kent State University*
balqadi@kent.edu

Jonathan I. Maletic
*Department of Computer Science*
*Kent State University*
Kent, Ohio 44242
jmaletic@kent.edu

***Abstract*—Researchers have identified several quality metrics to predict defects, relying on different information however, these approaches lack metrics to estimate the effort of program understandability of system artifacts. In this paper, novel metrics to compute the cognitive complexity based on program slicing are introduced. These metrics help identify code that is more likely to have defects due to being challenging to comprehension. The metrics include such measures as the total number of slices in a file, the size, the average number of identifiers, and the average spatial distance of a slice. A scalable lightweight slicing tool is used to compute the necessary slicing data. A thorough empirical investigation into how cognitive complexity correlates with and predicts defects in the version histories of 10 datasets of 7 open source systems is performed. The results show that the increase of cognitive complexity significantly increases the number of defects in 94% of the cases. In a comparison study to metrics that have been shown to correlate with understandability and with defects, the addition of cognitive complexity metrics shows better prediction by up to 14% in F1, 16% in AUC, and 35% in R$^2$.**

***Keywords— Cognitive Complexity, Software Metrics, Defect Prediction, Mining Software Repositories, Regression Model***

## I. Introduction

Defect prediction has generated widespread interest for a considerable period of time. The driving scenario is the limitation of resources for software quality assurance (QA), which may include manual code inspections, technical review meetings, and intensive testing. Such resources are always limited by time and by cost. Defect prediction techniques result in a list of defect-prone software modules and thus QA teams can effectively allocate limited resources by spending more effort on the modules that are likely to be defective. As the size of software projects becomes larger, defect prediction techniques play an important role to support developers and speed up time to market with more reliable software.

As such, researchers have identified several quality metrics and developed models to predict the quality of software. These approaches rely on different information, such as code metrics [1, 2], process metrics [2–4] or finer grained details of the source code such as anti-patterns [5–7] and code smells [8, 9]. However, these approaches lack metrics to estimate program understandability effort of the source code. It has been shown that software defects are often the result of the incomplete or incorrect comprehension of a program segment [10]. Therefore, the location of code that presents a comprehension challenge to the developer can be the basis for identifying code that has a greater risk of defects. Existing metrics for understandability are typically tied to readability or syntactic features of source code such as structural complexity. However, understandability has a cognitive and semantic aspect; a developer can find a piece of code readable, but still difficult to understand [11]. Campbell [12] attempts to measure the cognitive complexity by extending the cyclomatic complexity [13] to address modern language structures and to produce values that are meaningful at the class and application levels.

Much of the research on cognitive models explains how programmers comprehend complex code using a bottom-up approach [14, 15]. The programmer analyzes the source code statement by statement and gradually develops control-flow and data-flow abstractions through the process of chunking [16]. Program chunks are grouped together to form larger chunks, until the entire program is understood. In this way a hierarchical semantic representation of the program is built from the bottom-up. Thus, assessing the cognitive complexity of program semantic chunks can be a criterion for characterizing defects for defect prediction. Specifically, in order to make accurate predictions, the metrics need to be discriminative: capable of distinguishing one instance of code region from another of different cognitive complexity.

In this paper, we introduce a novel set of cognitive complexity metrics by utilizing *program slicing* to predict defects. Short introductory of these metrics is recently reported in [17]. Program slicing is a reduction technique that traces the data and control dependencies for determining only those parts of the original program that are relevant to the computation of a given feature of interest [18]. Program slicing has been successfully employed for program comprehension during different maintenance tasks such as testing and debugging [19–21]. Unlike straightforward code metrics based on line counts and statement counts, slice-based cognitive complexity metrics have the potential to consider more insightful code properties based on program behaviors, as captured by program slices and obtained from program analysis and points-to analysis. The slice-based metrics give different weights to each statement based on their significance in the control dependence and flow dependence in the program. For example, a *while* predicate that encloses multiple statements will contribute more than one control dependence in slice-based cognitive complexity metrics, while in code metrics it typically contributes only one source code line.

We perform an empirical investigation into how cognitive complexity correlates with and predict defects on parts of the version history of 10 datasets extracted from 7 open-source systems. Like other work on defect prediction, machine learning techniques are used to build regression models from the metric data applied to older versions of a system and evaluate the prediction models on more recent versions of the system. Our results show that an increase in cognitive complexity in source code significantly increases the number of defects in 94% of the cases. In comparison to metrics that have been shown to correlate with understandability and defects [3, 11, 22, 23], the addition of slice-based cognitive complexity metrics give better prediction of defects by up to 14% in F1, 16% in AUC, and 35% in $R^2$.

The contributions of our work are as follow. 1) A new set of slice-based cognitive complexity metrics are presented. While others have proposed slicing as a means for defect prediction [24–26], there is no other work that provides empirical results of the use on realistically sized software systems and indicates evidence of cognitive complexity. 2) Our study is one of the first that empirically studies the relationship between program understandability and the probability of defects. 3) Practitioners can easily adopt the proposed metrics for defect prediction as computing them is scalable to large systems. Generally, program slicing is time consuming to compute, however here we take advantage of a lightweight high scalable and publicly available program slicing approach to compute the necessary information.

The remainder of this paper begins with descriptions of the proposed cognitive complexity metrics (II), followed by an overview of the approach used to create a corpus (III). The methodology applied is discussed in Section IV. Results and discussion from applying different models are presented in Section V. Section VI discusses related work. VII and VIII conclude the paper with threats to validity and future work.

## II. Related Work

There exists a large body of research aiming to model software defects using product (e.g., LOC) and process metrics (e.g., changes metrics). D'Ambros et al. [1] presented a comparison of well-known defect prediction approaches (i.e., models with product and process metrics), together with novel approaches. Nevertheless, code metrics are lightweight alternatives with overall good performance. In a comparison, Di Nucci et al. [4] measured the scattering of changes performed by developers working on a component. Rahman and Devanbu examined the effects of ownership and experience on quality, using a fine-grained level of analysis based on fix-inducing code-fragment [27]. Some defect prediction studies have focused on finer grained details of the source code. Khomh et al. [5] and Taba et al. [6] considered the use of anti-patterns because they are more actionable than other metrics. Padua and Shang [7] investigated the exception handling anti-patterns and found them to have significant relationship with defects. Palomba et al. [9] proposed a smell-aware defect prediction model and found the accuracy of the model increases by adding code smell intensity as predictor.

Various researches proposed metrics quantifying other aspects of software engineering in order to model software quality. For example, Shihab et al. [28] consider branching activities; Shang et al. [29] investigate logging characteristics, Zhang et al. [30] examine editing patterns, and McIntosh et al. [31] study code reviews. To the best of our knowledge, this paper is the first attempt to study the relationship between cognitive complexity based on program slicing, and software quality. The metrics from the aforementioned research only focus on single elements without taking the interactions between elements into account. However, the nature of defects has changed and today most defects in bug databases are of semantic nature [32].

In term of slicing literature, Weiser presented in [33] the concept of program slicing and several slicing based metrics, such as parallelism, and tightness. Ott and Thuss introduced two new metrics supplementing Weiser metrics [34]. Later, Bieman and Ott used slicing in the context of tokens rather than statements [35]. Although Weiser introduced program slicing as a comprehension method used by programmers while debugging [33], many slice-based metrics have been developed to quantify the degree of cohesion in a module. Such metrics calculated in which the numbers of statements that are shared by multiple slices represent cohesion [17, 18, 23, 32– 34].

In term of software quality, to date little work has been performed to empirically relate slice metrics to code quality. Meyers and Binkley undertook an empirical study of five slice metrics and analyzed the relations between these metrics and code size metrics [17, 33]. They found that slice-based metrics provided a unique view of a program. Black et al. [24] empirically investigated the functions cohesion using two of Weiser metrics, Tightness and Overlap, to distinguish between faulty and not-faulty functions. They combined the nineteen versions of a small program to obtain a single data set. In [26], Black et al. had planned to test the hypotheses relating three slice-based metrics and defect-proneness. However, they failed to do this due to lack of data. Work by Pan and Kim used C language slicing metrics to compare the classification of defects with code metrics for C++ [25]. The calculation of their metrics is based on the notion of a Program Dependence Graph (PDG) [38] such as edge count and vertices count [39]. Yang et al. [40] studied the usefulness of Weiser [33] and Ott and Thuss [34] cohesion metrics in effort aware defect prediction. The 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. Yet we still know very little about software semantic and their cognitive complexity. To the best of our knowledge, this is the first work that applies program slicing to measure characteristics of cognitive complexity and investigate their relationship to defect propensity from an evolutionary viewpoint.

## III. Slice-Based Cognitive Complexity Metrics

For years researchers have devoted their efforts trying to understand how programmers comprehend code and several cognitive model have been proposed [14, 15]. A study by Siegmund et al. looked at the process of bottom-up program comprehension with (fMRI), a technique used by to understand brain regions activated by cognitive tasks, and found a network of brain areas activated that are related to natural-language comprehension, problem solving, and working memory [41]. Klemola argues that measuring complexity should reflect attributes of human comprehension since complexity is relative to human cognitive characteristics. They focus on aspects of cognition which involves both short-term and long-term memory. Overloading over a short period of time affects short-term memory (STM) while long term memory (LTM) is affected by the frequency of exposure to a concept over time [42].

Researchers theorize that all information processed for comprehension must at some time occupy (STM). For the purposes of natural-language comprehension, the capacity of STM has been measured at 4 concepts [43]. This suggests that any code segment that is using more than 4 concepts to make a point unfamiliar to the reader might not be immediately understood.

In coping with these demands and limitations, the programmer must have mental capacity for dealing with large workloads for short periods of time and cognitive mechanisms for locating the code relevant to a particular feature. Program slicing was introduced by Weiser [18] after noticing programmers try to identify program bugs by using slices of the program composed of statements, which affect the computation of interest [44]. Thus, slicing process removes from consideration parts of the program that are determined to have no effect upon the semantics of interest in a similar way as it would be perceived by developer during the process of comprehension [44]. A slice is a cognitive chunk of the program that preserves control flow and data flow dependences relevant to a specific point of interest. It is possible to determine the parts with different behaviors by comparing the slices of two artifacts. With slice granularity, a hierarchical internal semantic representation of the whole program can be measured in addition to a detailed analysis of the comprehension effort required to retrace and inspect particular function. In the following we define four categories of slice-based cognitive complexity measures.

***sliceSize.*** This measure provides an indicator of the cognitive effort required to comprehend a particular slice. As stated earlier, program slice consists of all the statements that may influence the values of a variable at a program point [18]. It includes program artifacts that are data and control dependent to the function or variable of interest. *sliceSize* is *the mean count of individual statements of a slice in a module*. A study by Alomari et al. shows that the growth of the slice size over time in Linux kernel is related to the maintenance activity being made [19]. An increase in the slice size requires increase in the cognitive effort in analyzing the code related to the slice. Failing in uncovering the causal interactions between components force programmer to make unverified assumptions and eventually introducing defects [10, 41]. sliceSize for a module *x* can be defined as: $sliceSize(x) = \sum_{i=1}^{k} S_i/k$, where *S* is the number of statements in slice *i,* and *k* is the number of slices in module *x*.

***sliceIdentifier.*** Identifiers play a crucial role in program comprehension, since developers express domain knowledge through the names that they assign to the code entities at different levels (i.e., packages, classes, methods, variables) [42– 44]. Thus, source code lexicon impacts the psychological complexity of a program [45, 46]. For the purpose of cognitive complexity metrics, a high identifier density may overload STM and lead to errors. The risk of comprehension error has been observed to rise with the increase of general identifier density metric in program code [41, 47] and the increase of concept density in text as well [51]. However, the problem with calculating a general identifier density metric is that it only represents a general view on the system under investigation [52]. A small block with highly dense identifier may be buried in a large block of simple code resulting in a low value for the large block. Therefore, it is essential to refine identifier density metric to reflect a more realistic assessment based on the development task on hand. Program slicing allows for such refinement, by focusing the metric only on these parts that are relevant with respect to a particular feature or variable. *sliceIdentifier* can be defined as the *mean distinct occurrences of programmer defined labels within a slice in a module*. For a module $x$ it can be formally defined as: $sliceIdentifier(x) = \sum_{i=1}^{k} SI_i/k$, where *SI* is the number of identifiers in slice *i,* and *k* is the number of slices in module *x*.

***sliceCount.*** The count of slices focuses on the overall cognitive complexity of source code parts. Program segment that has a high number of slices will have high number of features leading to a higher concentration of identifiers, method invocations and relevant control and data dependencies. When there are many possible paths to be taken within a module the time spent tracing references increases and at the same time the use of identifiers must be carefully observed and retained in human memory to arrive at a correct understanding. When time is limited, a program segment with a high value of slice count can be difficult to interpret. *sliceCount* is defined *as number of slices within a module* and more formally as follows: *sliceCount(x) = K*, where *k* is the number of slices in *x*.

***sliceSpatial.*** Spatial complexity measures account for the difficulty of reading the source code of a program for understanding, in terms of the lexical distance (measured in lines of code) that the maintainer is required to traverse to follow control and/or data dependencies as they build a mental model [53, 54]. This type of complexity was based on the spatial distance between the definition and direct use of various program elements. However, understanding of the use of a program element also requires knowledge of control and data flow in which the program element has been used [53]. More details about the elements are understood through its use in a particular sequence and the use of other artifacts that influence the behavior of the element of interest. Without program slicing, it would be impossible to find all relevant uses that might effect or affected by the element value. The greater the distance in lines of code, the more is the cognitive effort required to understand the purpose and data flow of that slice. Thus, we define *sliceDistance* as *the spatial distance in LOC between the definition and the last use of the slice divided by the module size.* $SliceDistance\ (i) = Sm_i - Sn_i/q$, where *Sm* is the line number of the first statement in slice *i, Sn* is the line number of the last statement in slice *i,* and *q* is the module size in LOC. Accordingly, for module *x, sliceSpatial* measured as *the mean of the individual slice distance in x.* $sliceSpatial\ (x) = \sum_{i=1}^{k} sliceDistance(i)/k,$ where $k$ is the number of slices in *x*.

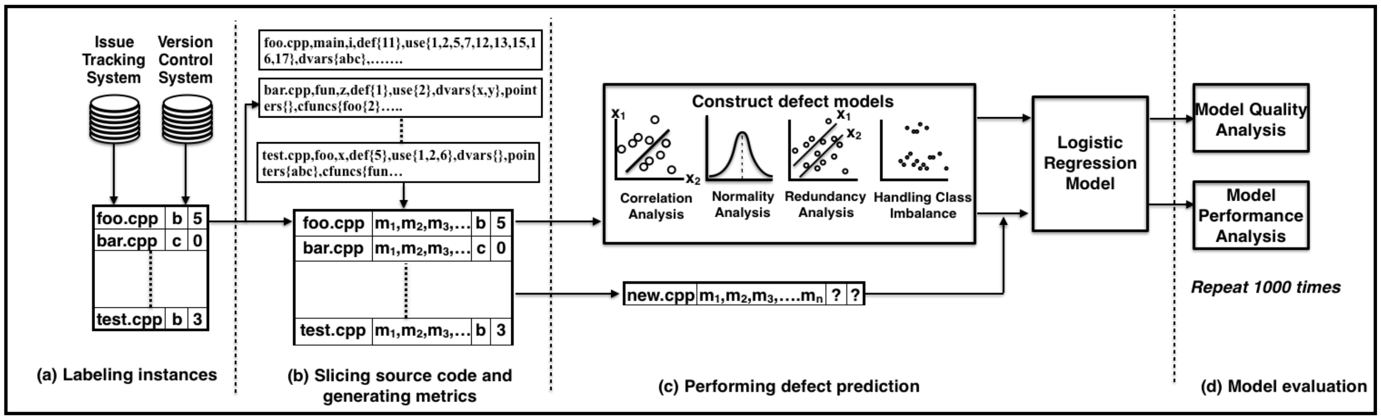
Issue Tracking System
Version Control System
foo.cpp b 5
bar.cpp c 0
test.cpp b 3
(a) Labeling instances
foo.cpp,main,i,def{11},use{1,2,5,7,12,13,15,16,17},dvars{abc},.......
bar.cpp,fun,z,def{1},use{2},dvars{x,y},pointers{},cfuncs{foo{2}.....
test.cpp,foo,x,def{5},use{1,2,6},dvars{},pointers{abc},cfuncs{fun...
foo.cpp m1,m2,m3,... b 5
bar.cpp m1,m2,m3,... c 0
test.cpp m1,m2,m3,... b 3
(b) Slicing source code and generating metrics
Construct defect models
Correlation Analysis
Normality Analysis
Redundancy Analysis
Handling Class Imbalance
new.cpp m1,m2,m3,....mn ? ?
Logistic Regression Model
(c) Performing defect prediction
Model Quality Analysis
Model Performance Analysis
Repeat 1000 times
(d) Model evaluation

**Figure 1. Overview of the study design.**

## IV. APROACH

Figure. 1 shows the file-level defect prediction process used in this paper. This is a typical prediction process commonly used in the literature [1, 4, 55]. The key insight behind these models is learning from software evolution history such as Git.

Recorded data include change history, change log messages, and bug fixes that cover years of data which can be a useful resource for learning from previous defects and predicting the new ones. The first step in building the model is to collect instances and historic information from software archives. Instances can represent different granularity, however, due to the file-based nature of Git, a file-level granularity is used for prediction in this paper. Processing the raw data falls into two folds: (1) Labeling instances as defective/non-defective or defects count, and (2) Extracting metrics to determine useful patterns in bug-fix or occurrence that can be applied for prediction. After generating the corpus, i.e., instances with metrics and labels, the final step is training a prediction model, so it can predict whether a new instance has a defect or not. Following subsections provide details of our approach steps.

### *A. Creating a Labeled Dataset*

Since we evaluate the performance of defect prediction using program slicing metrics, it needs to determine whether a file examined in the experiment actually contains a defect and how many times a file is included in a bug fixing task. Typically, defects are discovered and reported to an issue tracking system and later fixed by the developers. In order to link files with defects, for each system we clone the repositories from Git, and for each repository *r*, we create a series of patch files $\{P_i\}_{i=1}^{n}$, where *n* is the latest revision number for repository *r* at specific release. Each patch file $P_i$ is responsible for transforming repository *r* from revision $r_{i-1}$ to revision $r_i$ , where $r_1$ is the initial revision for specific release. By initially setting repository *r* to revision 1 and then applying all patches $\{P_i\}_{i=1}^{n}$, in a sequential manner, the revision history for that repository *r* is essentially replayed. Conceptually, this is equivalent to the case of all developers performing their commits sequentially one by one according to their chronological order. To perform defect labeling, we assume that a file has a defect if it is involved in a bug fixing transaction along the whole history for the period considered (i.e., release-level labeling).

The labeling begins with links between bugs reported in the issue tracking system and the specific revision that fixes the bug; we call this a bug-fixing revision. Additionally, various keywords such as “bug”, “fixed”, numerical bug ids, etc. mentioned in the commit log, are extracted and linked back to the issue tracking system’s identifiers. Finally, manual inspection is performed to remove spurious linking as much as possible. These are introduced and used in a number of defect prediction studies [1, 27].

In cases where the lack of supporting information (e.g., undescriptive ticket and /or commit message) prevents us from classifying a certain commit with satisfactory confidence, that commit is dropped from the dataset. Overall, we dropped 11% (3670), of the defect commits. We repeat this routine for all commits involved in the considered period. Furthermore, we excluded any commit for fixing a broken unit test or build since these are not ones that matter for the users of a program.

**Extracting Metrics**. This step is to enrich the model with the metrics. For each retrieved file we use the *srcSlice* tool [56, 57] to compute the slicing metrics at the file level in every repository. We check out each file of the preceding revision that fixed a specific defect (i.e., revision hash~1). In this manner, we compute slicing metrics prior to fixing the defect to evaluate the nature of defect-prone files. We calculate metrics just before fixing the defect instead of the common way that calculates the metrics at the end or beginning of the release. For files that appear in a non-fixing commit (i.e., non-defective files), we check out each file of the specific revision that changed the file. In case a file appears in more than one transaction throughout the considered period (i.e., release-level), we calculate the average of each extracted metric. The reason for choosing this method is that files can undergo massive changes through the same release, so metrics at the end or beginning of the release are often not representative and thus less accurate. We applied this method to extract all metrics through the revision history. An overview description of the theory and implementation of *srcSlice* tool, calculation of slice-based metrics, and baseline metrics is now presented.

***srcSlice* Tool.** The *srcSlice* tool [57] is a fast and scalable, slicing approach that is freely available. It has practical means to estimate the source code semantics for very large systems within efficient time frames which makes it suitable for this work. Program slicing is typically based on the notion of a

Program Dependence Graph (PDG) [38] or one of its variants. Unfortunately, building the PDG is quite costly in terms of computational time and space. As such, slicing approaches generally do not scale well (time wise). *srcSlice* addresses this limitation by eliminating the time and effort needed to build the entire PDG. It combines a text-based approach with a lightweight static analysis infrastructure that only computes dependence information as needed while computing the slice for each variable in the program. The approach was first introduced in [58], and then evaluated to a total of 18 open source systems through a comparison study [56].

The *srcSlice* tool implements a forward, static slicing technique. Forward static program slicing refers to the computation of program points that are affected by other program points [59]. The forward slice from program point *p* includes all the program points in the forward control flow affected by the computation at *p*. *srcSlice* uses the initial variable declaration as the starting point. The tool is enabled by the *srcML* [60, 61] infrastructure (see *srcML.org*). Source code is first converted to *srcML* and then a stream-oriented approach to compute the slice is performed. *srcML* (SouRce-Code Markup Language) augments source code with abstract syntactic information from the AST to add explicit structure to program source code. This syntactic information is used to identify program dependencies as needed when computing the slice. It supports C/C++, C#, and Java. The approach is very scalable and can compute the slice on all variables in the Linux kernel in approximately 7 minutes [57].

Given a system (in the *srcML* format), *srcSlice* gathers data about every file, function, and variable throughout the system, storing it all in a three-tier dictionary. Each entry of the system dictionary is a single slice profile for an identifier, which contains all data gathered about that identifier during the slicing process. The following is a list of that information:

- *File, function, and variable names*
- *Def* – list of line numbers a variable is defined or redefined on. *Def* is used to differentiate between variables with the same name but in differing scopes.
- *Use* – list of line numbers a variable is used. This refers to a variable being used in a computation with no modification to its value. Can be used to construct def-use chains.
- *Dvars* - list of variables data dependent on the slice variable.
- *Pointers* - list of aliases of the slicing variable. The elements of this list are variables to which the slicing variable is a pointer.
- *Cfuncs* - list of functions called using the slicing variable.

The tool produces a system dictionary for all the slice profiles of all variables in a system. It is 3-tiered and consists of three maps. On the first level is a map from files to functions, on the second level is a map from functions to variable names, and on the third level is a map from variable names to slice profiles.

**Slice-Based Metrics**. Using the output generated from *srcSlice*, we parse it into a data structure and then calculate the metrics. The first metric is *sliceCount*, it measures the file slices, which is equivalent to the number of paths in the code representation model (e.g., program dependence graph). This simply counts the number of entries in the system dictionary produced by *srcSlice* for each file. Using the information stored in each slice profile, we can easily retrieve the size of the slice for each variable in the system, identifier density, and slice spatial. *sliceSize* represents the mean size of a variable slice in a file, measured in number of lines of code. It indicates how much the slice profiles depend on each other by intra-procedural or inter-procedural control or data dependencies. *sliceSize* counts all lines that a variable is defined or used. This is the union between *def* and *use* in the slice profile. Therefore, the *sliceSize* is the ratio of all slice sizes to the *sliceCount*.

*Dvars, pointers* and *cfunc* fields in the slice profile capture the distinct identifiers appeared in the slice including variable names and methods invocation. For individual slice, number of identifiers is the count of items in *dvars, pointers*, and *cfunc* fields. Accordingly, file *sliceIdentifier* is the mean of all slice identifiers to the *sliceCount*. *sliceSpatial* is the extent to which the slice scatter within a file. For an individual slice, we calculate *sliceDistance* as the distance in LOC between the definition and the last use of the slicing variable divided by the file size. As stated earlier, *Def* and *use* list all the line numbers in ascending order where the slicing variable is defined or used. Thus, *sliceDistance* is the subtraction of the first use from the last use divided by file size. *sliceSpatial* of the file is then measured as the mean of the individual *sliceDistance*.

We also include *sliceCoverage* metric similar to the one proposed by Weiser [33]. However, *srcSlice* is a static slicing technique, which consider subsets of the program with respect to all possible executions/behaviors, while Weiser uses dynamic slicing that is suitable to identify code fragment with respect to one execution. We include *sliceCoverage* because of its relation to comprehension as it represents the active portion of the file that the programmer needs to traverse and comprehend [33]. By comparing the slice size to the file size, we can measure the *sliceCoverage*, which is the mean slice size relative to file size. Generally, high file-level values of these metrics indicate more logically complex code and potentially more complex behaviors that are difficult to understand, inspect and trace in maintenance activities and hence exhibit more system defects. Note that slice metrics can be calculated at different level of granularity (i.e., system, file, method, and variable), however, due to the file-based nature of Git, a file-level granularity is used in this paper.

| Category | Metric | Description |
|---|---|---|
| Slice-based cognitive complexity | sliceCount | Number of slices within a file |
| | sliceSize | Average slice size measured in LOC |
| | sliceIdentifier | Average of distinct occurrences of programmer defined labels within a slice |
| | sliceSpatial | Average of spatial distance in LOC between the definition and the last use of the slice divided by the file size |
| | sliceCoverage | Average slice size relative to LOC |
| Size | NLOC | Average non-commentary source lines of code in a function |
| Structural complexity | CCN | Average Cyclomatic complexity of a function |
| Software science | Program Length | Average number of operators and operands of a function |
| Process | lineChange | Average number of lines added and deleted of a function |
| | funcChange | Average number of functions changed within a file |

**Table 1. Definitions of file level metrics.**

| Subject | Application type | Prog. lang. | Period | # Revisions | Defect revision rate % | # Instances | Defective instances rate % | EPV |
|---|---|---|---|---|---|---|---|---|
| **Linux 3.13** | Operating system | C | 01-19-2014~03-29-2014 | 13844 | 30% | 35,397 | 10% | 718 |
| **Eclipse 3.1** | IDE | Java | 06-27-2005~06-28-2006 | 2283 | 29% | 1,045 | 50% | 104 |
| **Eclipse 3.2** | | | 06-29-2006~06-24-2007 | 1643 | 35% | 1,122 | 41% | 92 |
| **Koffice 2.0** | Office suite | C++ | 05-20-2009~11-20-2009 | 1632 | 32% | 4,424 | 6% | 51 |
| **Apache HTTP 2.0** | Web server | C | 04-06-2002~02-07-2005 | 5919 | 19% | 266 | 38% | 20 |
| **Apache HTTP 2.2** | | | 09-11-2012~11-16-2013 | 465 | 15% | 402 | 17% | 14 |
| **Dolphin 14.11~18.8** | File manager | C++ | 11-08-2014~09-06-2018 | 709 | 21% | 327 | 23% | 15 |
| **Lucene 3.0** | Information retrieval | Java | 11-25-2009~03-29-2011 | 2696 | 21% | 4599 | 14% | 129 |
| **KDE Krita 3.0~3.1.3** | Graphics editor | C++ | 05-30-2016~04-28-2017 | 2396 | 31% | 5,518 | 10% | 111 |
| **KDE Krita 3.1.4~4.0** | | | 29-04-2017~08-01-2018 | 1689 | 28% | 5,166 | 7% | 68 |
| **Total** | - | - | - | 3328 | 26% | 5827 | 22% | - |

**Table 2. Revisions, file instances and % of defective files.**

**Baseline Metrics**. In order to quantify the contribution of slice-based metrics, we include traditional code and process metrics as a control set for providing a comparison. We select three code metrics: a size metric (i.e., *NLOC*), a structural complexity metric (i.e., *McCabe's* [13]), and software science metric (i.e., *Halstead's Program Length* [62]). These are known to be powerful indicators for defect prediction [23] and at the same time are commonly used in understandability studies [11, 22]. We exclude other class-level code metrics such as CK and OO metrics since the analysis of this work is a file-level granularity. We also do not include the other Halstead's metrics because these metrics are fully based on the counts of operators and operands. Consequently they are highly correlated with each other [63]. Additionally, process metrics are found to be powerful indicators in defect modeling and show improvement when combined with code metrics [3]. Therefore, we include two widely used change metrics in defect prediction, namely *lineChange* the number of lines changed (i.e., added and removed) and *funcChange* the number of functions changed within a file. Table 1 describes these baseline metrics. By choosing these five baseline metrics we are able to perform a balanced (5 vs. 5) analysis which ensures that any overfitting or multidimensionality problem that commonly happened with large number of variables is avoided.

## V. EXPEREMENTAL DESIGN

In this section, we introduce the projects used in our study and the research questions relating slice-based metrics to defect-proneness. Then, we describe the modeling techniques and the methods for data analysis.

### A. *Systems*

We use 10 datasets of 7 open-source projects to investigate the usefulness of cognitive complexity metrics in defect prediction. In selecting the systems, we consider three important criteria:

**Criterion 1—**Different corpora: To extend the generality of our conclusions, we choose systems from different corpora and domains. The included systems are non-trivial software that belong to different problem domains and different programming languages.

**Criterion 2—**Sufficient EPV: Prior studies show that the Events Per Variable (EPV) (i.e., the ratio of the frequency of the least occurring class in the outcome variable to the number of features that are involved in training of a classifier) has a significant influence on the performance of defect classifiers [64]. In particular, defect classifiers trained with datasets with a low EPV value yield unstable results [64, 65]. To ensure the stability of our results, we ensure that included datasets have an EPV value that is larger than 10 [65]. Particularly, the systems we select have EPV ranging from 14 to 718 (see Table 2).

**Criterion 3—** Defect rate: Since it is unlikely that more software modules have defects than are free of defects, we choose to study datasets that have defective rate ranging from (6%) to (50%). Table 2 summarizes the details of the projects examined in this study.

### B. *Research Questions*

The general goal of this paper is to understand whether the proposed cognitive complexity metrics have a relationship with the probability of defects. To achieve the goal, we attempt to answer the following two research questions:

***RQ1.*** Do slice-based cognitive complexity metrics significantly correlate to defects?

***RQ2.*** Do slice-based cognitive complexity metrics contribute to the prediction of the probability of defects?

The purpose of these questions is to investigate whether cognitive complexity metrics can effectively lead to significant relationship to defect prediction. These questions are critically important to both software researchers and practitioners, as they help to answer whether slice-based cognitive complexity metrics are of practical value. We choose to use defects as one widely used indicator of software quality and known as a result of comprehension difficulty.

### C. *Correlation Analysis*

In order to investigate (RQ1), we determine the Spearman rank correlation between the number of defects and each slice-based cognitive complexity metric since all datasets suffer from high skewed distribution. The Spearman rank correlation is a robust technique that can be applied when the association between values is non-linear. The closer the value of correlation is to −1 or +1, the higher two metrics are correlated-positively for +1 and negatively for −1. A value of 0 indicates that two metrics are independent. The 95% confidence interval (CI) of a Spearman's rank correlation coefficient is computed by bootstrapping with 1,000 replicates [66]. The significance (p-values) of the correlation are computed using algorithm AS 89 for n<1290 when exact compute was allowed [67] otherwise Edgeworth series approximation with cutoff modification from the original [68].

### *D. Model Construction*

To answer (RQ2), we build multiple regression models where the number of defects forms the dependent variable in binary classification, representing whether an instance is defective or non-defective. We build separate models for two sets of independent variables:

- **BMM** (baseline metrics model): This set consists of all code and process metrics listed in Table 1.
- **SBCCM** (slice-based cognitive complexity model): This set of variables includes the addition of slice-based metrics that were introduced in section (III.B.2) and Table 1 to the baseline metrics. In the rest of the subsection, we present the detail of our model construction process.

**Normality analysis**. Regression models expect normality in the outcome and in the predictors. Defect prediction datasets suffer from high skewed data typically do not follow a normal distribution [31, 28] (e.g., defects exist only in a small portion of the files). Therefore, we apply a log transformation log2 (x+1) to reduce the skew and adequate the data to the regression assumption.

**Correlation analysis.** Software metrics can be highly correlated to each other [55]. Highly correlated metrics (i.e., $|\rho| > 0.7$) can lead to an inflated variance in the estimation of the outcome [69]. Prior to modeling, we evaluate the correlations among our extracted metrics. We use Spearman pair-wise rank correlation to better account for collinearity between predictors in the data. Afterword, we use Principal Component Analysis (PCA) [70] to build the regression models using sets of principal components (PC), which are independent instead of the actual independent variable (i.e., metrics). Therefore, these components do not suffer from multicollinearity, while at the same time they account for as much sample variance as possible (i.e., feature selection). We use *prcomp* function from *stats* R package. We include PCs that account for at least 95% of the variance. Across systems, SBCCMs need an average of 80% of the components to account 95% of the data variance, while the BMMs need an average of 67% of the components.

**Redundancy analysis.** To ensure principle components of the PCA do not include redundant predictors, the redundancy analysis is performed in an iterative manner in which components are dropped until no components can be predicted with an $R^2$ or adjusted $R^2$ higher than 0.9. Hence, we use the *redun* function from *Hmisc* R package and find no redundant PCs in all datasets [71].

**Handling Category Imbalance.** Table 1 shows that our dependent variables are imbalanced, e.g., there are more non-defective instances than defective ones. If left untreated, the models will favor the majority category, since it offers more predictive power. To combat this bias, we use the SMOTE technique [72] (provided by the *DMwR* R package [73]) which creates artificial data based on the feature space similarities from the minority modules. The SMOTE technique has been shown to improve AUC and been used in previous defect prediction studies [74].

**Binary logistic regression.** We conduct our experiments using binary logistic regression model. This technique is a standard statistical modeling technique in which the dependent variable can take two different values. It is suitable for building defect prediction models because the files under consideration are divided into two categories: defective and non-defective. We choose logistic regression over other modeling techniques since it is found to yields the best performance for models that combines both process and code metrics [3]. We use the method *lrm* from the *RMS* R package [75].

**Out-of-sample bootstrap.** In order to ensure that the conclusions that we draw about our models are robust, we use the out-of-sample bootstrap validation technique, which has been shown to yield the best balance between the bias and variance [64]. Unlike the ordinary bootstrap, the out-of-sample bootstrap technique fits models using the bootstrap samples, but rather than testing the model on the original sample, the model is instead tested using the rows that do not appear in the bootstrap sample [64]. Thus, the training and testing corpora do not share overlapping observations. The entire bootstrap process is repeated 1000 times with the function *validate* from the *RMS* R package [75], and the average out-of-sample performance is reported as the performance estimate.

**Model analysis.**

*1) Logistic regression model explanatory power*

**Area under the ROC curve (AUC):** The ROC curve is plotted by false positive rate (FP) and true positive rate (TP). FP and TP vary based on threshold for prediction probability of each classified instance. Thus, the AUC value characterizes the accuracy of the model across all possible cutoff values. Larger AUC values indicate better performance [76].

**Nagelkerke R2:** is a specialized $R^2$ typically used for logistic regression models [77]. Nagelkerke $R^2$ with larger values indicating more variability explained by the model and less unexplained variation—a high Nagelkerke $R^2$ (provided by the *lrm* method) value indicates good explanative power, but not predictive power.

*2) Logistic regression model prediction abilities*

To assess the models prediction abilities using 0.5 cutoff, we calculate **precision** as the proportion of files that are correctly labeled as defective, i.e., $P = TP/(TP + FP)$, **recall** as the proportion of files that are correctly labeled, i.e., $R = TP/(TP + FN)$, and **F-Measure** which is a harmonic mean of precision and recall, i.e., $F = (2 \times P \times R)/(P + R)$.

## VI. Result and Disscussion

***RQ1. Do slice-based cognitive complexity metrics correlate significantly to defects?***

Table 3 shows the Spearman correlations computed by bootstrapping with 1,000 replicates. All correlations are significant at 0.95-confidence level except the correlations that

| Subject | sliceCount | | | sliceSize | | | sliceCoverage | | | sliceIdentifier | | | sliceSpatial | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $r_s$ | 95%CI | | $r_s$ | 95%CI | | $r_s$ | 95%CI | | $r_s$ | 95%CI | | $r_s$ | 95%CI | |
| | | Lower | Upper | | Lower | Upper | | Lower | Upper | | Lower | Upper | | Lower | Upper |
| **Linux 3.13** | **0.31** | 0.29 | 0.33 | **0.19** | 0.17 | 0.21 | **-0.28** | -0.30 | -0.26 | **0.17** | 0.15 | 0.19 | **-0.16** | -0.18 | -0.14 |
| **Eclipse 3.1** | **0.33** | 0.26 | 0.4 | **0.31** | 0.24 | 0.37 | **-0.36** | -0.43 | -0.3 | **0.23** | 0.16 | 0.30 | -0.03 | -0.1 | 0.04 |
| **Eclipse 3.2** | **0.45** | 0.38 | 0.52 | **0.36** | 0.29 | 0.43 | **-0.39** | -0.45 | 0.32 | **0.26** | 0.19 | 0.33 | -0.02 | -0.11 | 0.06 |
| **Koffice 2.0** | **0.24** | 0.17 | 0.31 | **0.1** | 0.02 | 0.16 | **-0.26** | -0.32 | -0.19 | **0.1** | 0.03 | 0.16 | **-0.15** | -0.22 | -0.07 |
| **Apache HTTP 2.0** | **0.48** | 0.37 | 0.58 | **0.20** | 0.1 | 0.32 | **-0.50** | -0.6 | -0.4 | **0.24** | 0.12 | 0.36 | **-0.33** | -0.44 | -0.22 |
| **Apache HTTP 2.2** | **0.32** | 0.18 | 0.46 | **0.2** | 0.05 | 0.34 | **-0.27** | -0.4 | -0.13 | **0.2** | 0.05 | 0.34 | **-0.16** | -0.31 | -0.0005 |
| **Dolphin 14.11** | **0.45** | 0.27 | 0.60 | **0.2** | 0.06 | 0.35 | **-0.47** | -0.61 | -0.29 | **0.24** | 0.09 | 0.38 | **-0.25** | -0.41 | -0.08 |
| **Lucene 3.0** | **0.28** | 0.24 | 0.33 | **0.16** | 0.11 | 0.21 | **-0.25** | -0.30 | -0.20 | **0.1** | 0.03 | 0.14 | 0.02 | -0.03 | 0.07 |
| **KDE Krita 3.0** | **0.31** | 0.26 | 0.36 | **0.13** | 0.07 | 0.18 | **-0.32** | -0.37 | -0.26 | **0.14** | 0.1 | 0.19 | **-0.15** | -0.20 | -0.1 |
| **KDE Krita 3.1.4** | **0.29** | 0.23 | 0.34 | **0.16** | 0.1 | 0.21 | **-0.26** | -0.3 | -0.21 | **0.13** | 0.07 | 0.19 | **-0.11** | -0.17 | -0.04 |
| **Average** | 0.35 | 0.27 | 0.42 | 0.20 | 0.12 | 0.28 | -0.34 | -0.40 | -0.19 | 0.18 | 0.10 | 0.26 | -0.13 | -0.22 | -0.05 |

**Table 3. Spearman correlation coefficient $r_s$, *p*-value and confidence interval (CI) between defect counts and cognitive complexity metrics. All correlation coefficient values are statistically significant except values not bolded.**

are not bolded. Most of the investigated slice-based metrics are significantly correlated with defects. In 94% (i.e., 47 out of 50) of the cases, slice-based metrics have p-value $\leq 0.05$ and 95% CI that does not include zero. The *SliceCount*, *sliceSize*, and *sliceIdentifier*, show a consistent positive relationship with defect counts across all systems, which means that an increase in the aforementioned metrics leads to an increase in number of defects. This finding suggests that code that is divided into many parts (i.e., higher *sliceCount*), have a higher concentration of method invocations with parameters (i.e., higher *sliceIdentifier*), and have more tracing activity (i.e., higher *sliceSize*) during comprehension process, have a higher probability of defects. The increase of the aforementioned metrics indeed increases the cognitive complexity implying that developers should carefully handle files with high percentage of slice size, slice count and slice identifier.

The characteristics of cognitive complexity captured by slice-based metrics express a statistically significant relationship with the probability of defects, suggesting that slice based cognitive complexity metrics can be used as defects indicators.

Conversely, *sliceCoverage* shows an expected consistent negative relationship across systems, which means an increase in *sliceCoverage* comes with a decrease in number of defects. *sliceCoverage* measures the average sliceSize relative to file LOC (i.e., sliceSize/sliceCount*file LOC). Thus, an increase in the file LOC and *sliceCount* decreases the *sliceCoverage*, leading to more cognitive complexity and eventually defects.

We find that *sliceSpatial* has a negative relationship with the probability of defects. This suggests that being in large and scattered slices do not necessarily result in a high number of defects. While, this correlation is very weak (i.e., -0.12) and all of the insignificant correlation cases (i.e., 3 out of 3) occurred in relation to *sliceSpatial*, one would expect a high slice spatial to increase the cognitive complexity leading to more defects. This might be due to the file level granularity of the analysis by taking the average of *sliceSpatial* within a file. A highly scattered slice might be hidden in a large block of unite code resulting in a low value for the large block. Therefore, *sliceSpatial* might provide a higher and positive correlation with defects if a finer granularity of analysis is used (e.g., at slice-level). However, without further investigation to support this analysis, we cannot claim such argument.

Overall, the individual correlations of each cognitive complexity metrics indicate a weak/moderate monotonic relationship with number of defects in most of the cases. The metrics with the highest observed correlations are *sliceCount* and *sliceCoverage.* These metrics are more related to defects than finer grained metrics (*sliceIdentifier, sliceSize*, and *sliceSpatial*). This finding suggests that handling files with high slice coverage and slice count is more challenging and requires better understanding of the source code by developers. The results in addition suggest that these metrics might have more association if used together in relation to defects which is the analysis to be done in RQ2.

slice count, slice size and slice identifier metrics have a consistent and significant positive relationship with the probability of defects across systems, while slice coverage and slice spatial have a significant negative relationship, suggesting that handling files with higher cognitive complexity captured by slice-based metrics is more challenging and requires a better understanding of the source code by developers.

***RQ2. Do slice-based cognitive complexity metrics contribute to the prediction of the probability of defects?***

To address RQ2, we follow the process described in section (IV.B) to train multiple regression models for two different sets of predictors- SBCCM (slice based cognitive complexity model) and BMM (baseline metrics model). We then measure the explanatory power of models from the bootstrap training samples and measure the prediction performance of the models on the bootstrap testing samples. This process is validated by repeating the bootstrapping for 1,000 times.

| Subject | BMM | | | | | SBCCM | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | AUC | Nag. $R^2$ | Recall | Precision | F-1 | AUC (*) | Nag. $R^2$ (*) | Recall (*) | Precision (*) | F-1 (*) |
| **Linux 3.13** | 0.62 | 0.04 | 48 | 60 | 54 | 0.72 **(+10%)** | 0.18 **(+14%)** | 65 **(+17%)** | 65 **(+5%)** | 65 **(+11%)** |
| **Eclipse 3.1** | 0.67 | 0.15 | 54 | 67 | 60 | 0.78 **(+11%)** | 0.30 **(+15%)** | 65 **(+11%)** | 69 **(+2%)** | 67 **(+7%)** |
| **Eclipse 3.2** | 0.73 | 0.20 | 61 | 67 | 64 | 0.79 **(+6%)** | 0.32 **(+12%)** | 67 **(+6%)** | 73 **(+6%)** | 70 **(+6%)** |
| **Koffice 2.0** | 0.75 | 0.23 | 61 | 68 | 64 | 0.80 **(+5%)** | 0.34 **(+11%)** | 69 **(+8%)** | 72 **(+4%)** | 70 **(+6%)** |
| **Apache HTTP 2.0** | 0.73 | 0.20 | 68 | 65 | 66 | 0.87 **(+14%)** | 0.49 **(+29%)** | 81 **(+13%)** | 78 **(+13%)** | 79 **(+13%)** |
| **Apache HTTP 2.2** | 0.74 | 0.23 | 61 | 67 | 63 | 0.82 **(+8%)** | 0.41 **(+18%)** | 71 **(+10%)** | 70 **(+3%)** | 70 **(+7%)** |
| **Dolphin 14.11~18.8** | 0.71 | 0.16 | 61 | 64 | 62 | 0.87 **(+16%)** | 0.51 **(+35%)** | 75 **(+14%)** | 77 **(+13%)** | 76 **(+14%)** |
| **Lucene 3.0** | 0.69 | 0.14 | 58 | 62 | 60 | 0.78 **(+9%)** | 0.30 **(+16%)** | 71 **(+13%)** | 70 **(+8%)** | 71 **(+11%)** |
| **KDE Krita 3.0~3.1.3** | 0.70 | 0.15 | 58 | 67 | 62 | 0.76 **(+6%)** | 0.26 **(+11%)** | 66 **(+8%)** | 70 **(+3%)** | 68 **(+6%)** |
| **KDE Krita 3.1.4~4.0** | 0.74 | 0.21 | 55 | 68 | 61 | 0.80 **(+6%)** | 0.35 **(+14%)** | 70 **(+15%)** | 73 **(+5%)** | 72 **(+11%)** |
| **Average** | 0.71 | 0.17 | 58 | 66 | 62 | 0.80 **(+9%)** | 0.35 **(+18)** | 70 **(+12%)** | 72 **(+6%)** | 71 **(+9%)** |

* The values between ( ) represent the improvement between BMM and SBCCM.

**Table 4. Logistic regression models average AUC, Nagelkerke R2, recall, precision and F1 values across systems using 1000 bootstrap validation (Bold font highlights the best performance).**

The generated classifiers are evaluated using the AUC obtained from the training bootstrap samples. Table 4 reports the average AUC for each studied datasets. Overall, we find that SBCCM has an average AUC of 0.8 while BMM has an average of 0.71. This means SBCCM increases the AUC by 9% on average compared to BMM. In particular, the SBCCM reveals an increase in the AUC values across all systems by up to 16% and not less than 5%. This indicates that the addition of slice-based metrics certainly increases the discriminating power of the models. To measure the significance of the AUC differences between the two models, we use a Wilcoxon signed-rank test [78] since it does not need the data to follow a normal distribution and it tests paired results. The test reveals that the differences between SBCCM and BMM are significant in all datasets.

Figure 2 highlights the AUC values distribution of the 1000 iterations of bootstrap of SBCCM and BMM. Both models have a high level of agreement on the computed AUCs, as the boxplots are comparatively short and whiskers do not stretch over a wider range of values indicating reliable results. We observe a consistent trend of SBCCM outperforming BMM among all systems. SBCCM have a larger median and mean than the BMM. Across all systems, the lower whiskers of SBCCM are larger than BMM median. No sharable Inter-quartile ranges (IQR, i.e., middle box that represents 50% of AUC distribution) across systems. Moreover, all differences are significant using Wilcoxon signed-rank test. Additionally, Nag. $R^2$ values of SBCCM in Table 3 show improvement in all datasets that reaches up to 35% with an average increase of 18% over BMM which indicates an improvement in the fitted model. Thus, SBCCM model has a substantially better classification performance than the BMM.

The addition of slice based cognitive complexity metrics significantly improves AUC and Nag. $R^2$ measures across all systems.

In order to test the logistic regression model's prediction ability, we compute precision, recall and F-measure using the out-of-sample bootstrap validation (1,000 times) and report them in Table 3. The comparison between SBCCM and BMM shows that SBCCM is having higher F-measure in all 10 datasets. SBCCM achieves an improvement in F-measure up to 14% and not less than 6% (average 9%) over the BMM. Figure 3 shows the F-measure values distribution of the 1000 iterations of out-of-sample bootstrap validation for all datasets of SBCCM and BMM. In all systems, SBCCM has a larger

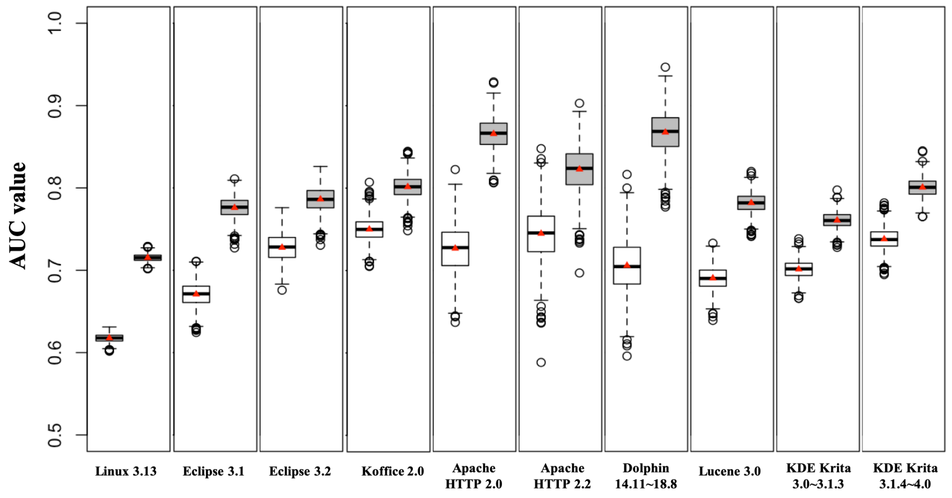

**Figure 2. Logistic regression models AUC distribution of the 1000 out of sample bootstrap (Gray boxplots are SBCCM models, white boxplots are BMM and red triangles indicate mean values).**

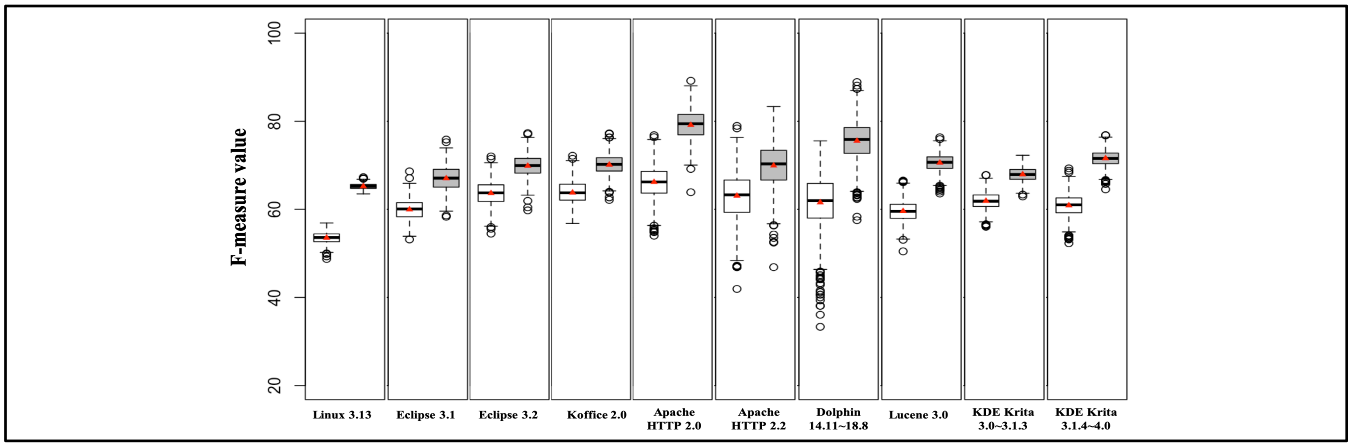

**Figure 3. Logistic regression models F-measure distribution of the 1000 out of sample bootstrap (Gray boxplots are SBCCM models, white boxplots are BMM and red triangles indicate mean values).**

median and mean than the BMM. In addition, no sharable IQR across systems. To measure the significance of the differences, we use a Wilcoxon signed-rank test. The results reveal that the differences are significant ($p < 0.001$) across all datasets. The higher F-measure values for SBCCM include an increase in both recall (average 12%, range 6%-17%) and precision (average 6%, range 2%-13%) in all systems.

Across all systems, slice-based cognitive complexity features significantly improve defect classification F1, recall and precision.

Beside defect prediction, identifying the most difficult to comprehend program elements can be a valuable aid in deciding schedules and choosing appropriate programmers for a project. It is not always feasible to improve on the cognitive complexity of a difficult program element, as some domains are inherently complex. In such situations, we should allocate enough time for individuals to understand the material with minimal error. Additionally, programmer experience plays a larger role in such situations and the programmer's familiarity with the application domain and type of implementation should be considered when setting time constraints.

Moreover, program slicing allows for fine-grained cognitive complexity driven inspection of the source code, by focusing and prioritizing the preventive maintenance activity on these fine-grained parts of the system that might require additional comprehension and maintenance effort during future system maintenance.

## VII. Threats To Validety

**Construct validity.** We used an automated bug linking process, which may introduce false positives in the linked set. This may arise because undetected defects in the considered interval are labeled clean, as defective commits are detected and fixed in 100-300days [79, 80]. To overcome such issue, we tried to cover a large span of time period an average of 20 months across systems.

The slicing process of *srcSlice* is performed using the *srcML* [60, 61] format for source code which provides direct access to abstract syntactic information. While this approach is inter procedural and highly scalable, it might not match the accuracy of generating a complete PDG/SDG. However, a previous study [56] compared *srcSlice's* accuracy with a heavyweight slicing tool and shows that *srcSlice* produces reliable accuracy given its speed and lightweight approach.

**External validity**. We studied 10 datasets that represent varying application domains and with different characteristics (defect rates, size, language, #files, etc.) to make our dataset general and representative. However, it is unclear how well they generalize to closed source software, which may have different behavior. Our approach only requires software metrics that can be computed in a standard way by publicly available tools and all our data will be made publicly available. Replication using closed source systems may prove fruitful.

**Internal validity.** We validate our model stability using out-of-sample bootstrap which been recently shown to provide the least bias and most stable performance estimates across measures in defect prediction [64]. While 100 repetition found to be sufficient [64], we repeated the experiment 1,000 times to ensure that the results converge, and found consistent results.

## VIII. Conclusions and future work

We empirically examine the usefulness of cognitive complexity slice-based metrics in the context of defect prediction. Our findings from an evaluation of 10 datasets covers parts of the version histories of open source systems show that 94% of the investigated metrics are statistically significant in relation to defects. Cognitive complexity metrics have significant impact on defect classification measured by AUC, $R^2$, F1, recall and precision. Slice-based metrics allows for fine-grained cognitive complexity driven inspection of the source code, by focusing and prioritizing the preventive maintenance activity on these parts of the code that require additional comprehension and maintenance effort during future maintenance. The approach can also be practically applied, as the slicing approach used is scalable to large system. Running it on the largest system (Linux) takes less than 10 minutes on a typical desktop machine. Future effort will be devoted to replicate our analyses on closed source software which might exhibit different behaviors. Furthermore, we will provide a replication package, which includes data for both the defects and metrics used for our experiment, to allow other researchers to compare our results.

## References

[1] M. D'Ambros, M. Lanza, and R. Robbes, "An extensive comparison of bug prediction approaches," in 2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010), 2010, pp. 31–41.

[2] R. Moser, W. Pedrycz, and G. Succi, "A Comparative Analysis of the Efficiency of Change Metrics and Static Code Attributes for Defect Prediction," in Proceedings of the 30th International Conference on Software Engineering, New York, NY, USA, 2008, pp. 181–190.

[3] F. Rahman and P. Devanbu, "How, and why, process metrics are better," in 2013 35th International Conference on Software Engineering (ICSE), 2013, pp. 432–441.

[4] D. D. Nucci, F. Palomba, G. D. Rosa, G. Bavota, R. Oliveto, and A. D. Lucia, "A Developer Centered Bug Prediction Model," IEEE Trans. Softw. Eng., vol. 44, no. 1, pp. 5–24, Jan. 2018.

[5] F. Khomh, M. D. Penta, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change- and fault-proneness," Empir. Softw. Eng., vol. 17, no. 3, pp. 243–275, Jun. 2012.

[6] S. E. S. Taba, F. Khomh, Y. Zou, A. E. Hassan, and M. Nagappan, "Predicting Bugs Using Antipatterns," in 2013 IEEE International Conference on Software Maintenance, 2013, pp. 270–279.

[7] G. B. de Pádua and W. Shang, "Studying the Relationship Between Exception Handling Practices and Post-release Defects," in Proceedings of the 15th International Conference on Mining Software Repositories, New York, NY, USA, 2018, pp. 564–575.

[8] S. L. Abebe, V. Arnaoudova, P. Tonella, G. Antoniol, and Y. Guéhéneuc, "Can Lexicon Bad Smells Improve Fault Prediction?," in 2012 19th Working Conference on Reverse Engineering, 2012, pp. 235–244.

[9] F. Palomba, M. Zanoni, F. A. Fontana, A. D. Lucia, and R. Oliveto, "Smells Like Teen Spirit: Improving Bug Prediction Performance Using the Intensity of Code Smells," in 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2016, pp. 244–255.

[10] C. Chen, S. Lin, M. Shoga, Q. Wang, and B. Boehm, "How Do Defects Hurt Qualities? An Empirical Study on Characterizing a Software Maintainability Ontology in Open Source Software," in 2018 IEEE International Conference on Software Quality, Reliability and Security (QRS), 2018, pp. 226–237.

[11] S. Scalabrino, G. Bavota, C. Vendome, M. Linares-Vásquez, D. Poshyvanyk, and R. Oliveto, "Automatically Assessing Code Understandability: How Far Are We?," in Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering, Piscataway, NJ, USA, 2017, pp. 417–427.

[12] G. A. Campbell, Cognitive Complexity - A new way of measuring understandability. SonarSource SA, 2018.

[13] T. J. McCabe, "A Complexity Measure," IEEE Trans. Softw. Eng., vol. SE-2, no. 4, pp. 308–320, Dec. 1976.

[14] M.-A. D. Storey, K. Wong, and H. A. Müller, "How do program understanding tools affect how programmers understand programs?," Sci. Comput. Program., vol. 36, no. 2, pp. 183–207, Mar. 2000.

[15] M.- Storey, "Theories, methods and tools in program comprehension: past, present and future," in 13th International Workshop on Program Comprehension, 2005, pp. 181–191.

[16] N. Pennington, "Stimulus structures and mental representations in expert comprehension of computer programs," Cognit. Psychol., vol. 19, no. 3, pp. 295–341, Jul. 1987.

[17] B. Alqadi, "The Relationship Between Cognitive Complexity and the Probability of Defects," in 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2019, pp. 600–604.

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

[19] H. W. Alomari, M. L. Collard, and J. I. Maletic, "A Slice-Based Estimation Approach for Maintenance Effort," in 2014 IEEE International Conference on Software Maintenance and Evolution, 2014, pp. 81–90.

[20] 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. 2:1–2:27, Dec. 2007.

[21] S. Counsell, T. Hall, and D. Bowes, "A theoretical and empirical analysis of three slice-based metrics for cohesion," Int. J. Softw. Eng. Knowl. Eng., vol. 20, no. 05, pp. 609–636, Aug. 2010.

[22] A. Rahman, "Comprehension Effort and Programming Activities: Related? Or Not Related?," in Proceedings of the 15th International Conference on Mining Software Repositories, New York, NY, USA, 2018, pp. 66–69.

[23] 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.

[24] S. Black, S. Counsell, T. Hall, and D. Bowes, "Fault Analysis in OSS Based on Program Slicing Metrics," in 2009 35th Euromicro Conference on Software Engineering and Advanced Applications, 2009, pp. 3–10.

[25] K. Pan, S. Kim, and E. J. W. Jr, "Bug Classification Using Program Slicing Metrics," in 2006 Sixth IEEE International Workshop on Source Code Analysis and Manipulation, 2006, pp. 31–42.

[26] S. Black, S. Counsell, T. Hall, and P. Wernick, "Using Program Slicing to Identify Faults in Software," in Beyond Program Slicing, Dagstuhl, Germany, 2006.

[27] F. Rahman and P. Devanbu, "Ownership, Experience and Defects: A Fine-grained Study of Authorship," in Proceedings of the 33rd International Conference on Software Engineering, New York, NY, USA, 2011, pp. 491–500.

[28] E. Shihab, C. Bird, and T. Zimmermann, "The Effect of Branching Strategies on Software Quality," in Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, New York, NY, USA, 2012, pp. 301–310.

[29] W. Shang, M. Nagappan, and A. E. Hassan, "Studying the relationship between logging characteristics and the code quality of platform software," Empir. Softw. Eng., vol. 20, no. 1, pp. 1–27, Feb. 2015.

[30] F. Zhang, F. Khomh, Y. Zou, and A. E. Hassan, "An Empirical Study of the Effect of File Editing Patterns on Software Quality," in 2012 19th Working Conference on Reverse Engineering, 2012, pp. 456–465.

[31] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, "An empirical study of the impact of modern code review practices on software quality," Empir. Softw. Eng., vol. 21, no. 5, pp. 2146–2189, Oct. 2016.

[32] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai, "Have Things Changed Now?: An Empirical Study of Bug Characteristics in Modern Open Source Software," in Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability, New York, NY, USA, 2006, pp. 25–33.

[33] M. Weiser, "Program Slicing," in Proceedings of the 5th International Conference on Software Engineering, Piscataway, NJ, USA, 1981, pp. 439–449.

[34] L. M. Ott and J. J. Thuss, "Slice based metrics for estimating cohesion," in [1993] Proceedings First International Software Metrics Symposium, 1993, pp. 71–81.

[35] J. M. Bieman and L. M. Ott, "Measuring Functional Cohesion," IEEE Trans Softw Eng, vol. 20, no. 8, pp. 644–657, Aug. 1994.

[36] T. M. Meyers and D. Binkley, "Slice-based cohesion metrics and software intervention," in 11th Working Conference on Reverse Engineering, 2004, pp. 256–265.

[37] 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 2010 36th EUROMICRO Conference on Software Engineering and Advanced Applications, 2010, pp. 283–286.

[38] K. J. Ottenstein and L. M. Ottenstein, "The Program Dependence Graph in a Software Development Environment," in Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, New York, NY, USA, 1984, pp. 177–184.

[39] D. Liang and M. J. Harrold, “Slicing Objects Using System Dependence Graphs,” in Proceedings of the International Conference on Software Maintenance, Washington, DC, USA, 1998, pp. 358–.

[40] Y. Yang et al., “Are Slice-Based Cohesion Metrics Actually Useful in Effort-Aware Post-Release Fault-Proneness Prediction? An Empirical Study,” IEEE Trans. Softw. Eng., vol. 41, no. 4, pp. 331–357, Apr. 2015.

[41] J. Siegmund et al., “Measuring Neural Efficiency of Program Comprehension,” in Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, New York, NY, USA, 2017, pp. 140–150.

[42] T. Klemola, “A Cognitive model for complexity metrics,” in 4th International ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (ECOOP 2000), Sophia Antipolis, 2000, pp. 12–16.

[43] N. Cowan, “The magical number 4 in short-term memory: a reconsideration of mental storage capacity,” Behav. Brain Sci., vol. 24, no. 1, pp. 87–114; discussion 114-185, Feb. 2001.

[44] M. Weiser, “Programmers Use Slices when Debugging,” Commun ACM, vol. 25, no. 7, pp. 446–452, Jul. 1982.

[45] V. Arnaoudova, L. Eshkevari, R. Oliveto, Y. G. Guéhéneuc, and G. Antoniol, “Physical and conceptual identifier dispersion: Measures and relation to fault proneness,” in 2010 IEEE International Conference on Software Maintenance, 2010, pp. 1–5.

[46] E. Enslen, E. Hill, L. Pollock, and K. Vijay-Shanker, “Mining source code to automatically split identifiers for software analysis,” in 2009 6th IEEE International Working Conference on Mining Software Repositories, 2009, pp. 71–80.

[47] S. L. Abebe, V. Arnaoudova, P. Tonella, G. Antoniol, and Y. G. Guéhéneuc, “Can Lexicon Bad Smells Improve Fault Prediction?,” in 2012 19th Working Conference on Reverse Engineering, 2012, pp. 235–244.

[48] S. Scalabrino, M. Linares-Vásquez, D. Poshyvanyk, and R. Oliveto, “Improving code readability models with textual features,” in 2016 IEEE 24th International Conference on Program Comprehension (ICPC), 2016, pp. 1–10.

[49] B. Sharif and J. I. Maletic, “An Eye Tracking Study on camelCase and Under_Score Identifier Styles,” in Proceedings of the 2010 IEEE 18th International Conference on Program Comprehension, Washington, DC, USA, 2010, pp. 196–205.

[50] R. P. L. Buse and W. R. Weimer, “Learning a Metric for Code Readability,” IEEE Trans. Softw. Eng., vol. 36, no. 4, pp. 546–558, Jul. 2010.

[51] W. Kintsch, “Toward a Model of Text Comprehension and Production,” 2005.

[52] J. Rilling and T. Klemola, “Identifying Comprehension Bottlenecks Using Program Slicing and Cognitive Complexity Metrics,” in Proceedings of the 11th IEEE International Workshop on Program Comprehension, Washington, DC, USA, 2003, pp. 115–.

[53] N. E. Gold, A. M. Mohan, and P. J. Layzell, “Spatial complexity metrics: an investigation of utility,” IEEE Trans. Softw. Eng., vol. 31, no. 3, pp. 203–212, Mar. 2005.

[54] J. K. Chhabra and V. Gupta, “Evaluation of Object-oriented Spatial Complexity Measures,” SIGSOFT Softw Eng Notes, vol. 34, no. 3, pp. 1–5, May 2009.

[55] G. K. Rajbahadur, S. Wang, Y. Kamei, and A. E. Hassan, “The Impact of Using Regression Models to Build Defect Classifiers,” in Proceedings of the 14th International Conference on Mining Software Repositories, Piscataway, NJ, USA, 2017, pp. 135–145.

[56] H. W. Alomari, M. L. Collard, J. I. Maletic, N. Alhindawi, and O. Meqdadi, “srcSlice: very efficient and scalable forward static slicing: SRCSLICE: VERY EFFICIENT AND SCALABLE FORWARD STATIC SLICING,” J. Softw. Evol. Process, vol. 26, no. 11, pp. 931–961, Nov. 2014.

[57] C. D. Newman, T. Sage, M. L. Collard, H. W. Alomari, and J. I. Maletic, “srcSlice: A Tool for Efficient Static Forward Slicing,” in Proceedings of the 38th International Conference on Software Engineering Companion, New York, NY, USA, 2016, pp. 621–624.

[58] H. W. Alomari, M. L. Collard, and J. I. Maletic, “A Very Efficient and Scalable Forward Static Slicing Approach,” in 2012 19th Working Conference on Reverse Engineering, Kingston, ON, Canada, 2012, pp. 425–434.

[59] S. Horwitz, T. Reps, and D. Binkley, “Interprocedural Slicing Using Dependence Graphs,” in Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation, New York, NY, USA, 1988, pp. 35–46.

[60] M. L. Collard, M. J. Decker, and J. I. Maletic, “Lightweight Transformation and Fact Extraction with the srcML Toolkit,” in Proceedings of the 2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation, Washington, DC, USA, 2011, pp. 173–184.

[61] M. L. Collard, J. I. Maletic, and B. P. Robinson, “A lightweight transformational approach to support large scale adaptive changes,” in 2010 IEEE International Conference on Software Maintenance, 2010, pp. 1–10.

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

[63] 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.

[64] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, “An Empirical Comparison of Model Validation Techniques for Defect Prediction Models,” IEEE Trans. Softw. Eng., vol. 43, no. 1, pp. 1–18, Jan. 2017.

[65] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, “Automated Parameter Optimization of Classification Techniques for Defect Prediction Models,” in 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), 2016, pp. 321–332.

[66] M. Hervé, RVAideMemoire: Testing and Plotting Procedures for Biostatistics. 2019.

[67] D. J. Best and D. E. Roberts, “Algorithm AS 89: The Upper Tail Probabilities of Spearman’s Rho,” J. R. Stat. Soc. Ser. C Appl. Stat., vol. 24, no. 3, pp. 377–379, 1975.

[68] M. Hollander and D. Wolfe, Nonparametric Statistical Methods, 2nd Edition. Wiley-Interscience, 1999.

[69] F. E. H. Jr, Regression Modeling Strategies: With Applications to Linear Models, Logistic and Ordinal Regression, and Survival Analysis, 2nd ed. 2015 edition. Cham Heidelberg New York: Springer, 2015.

[70] J. E. Jackson, A User’s Guide to Principal Components. Hoboken, N.J: Wiley-Interscience, 2003.

[71] F. E. H. Jr and with contributions from C. D. and many others, Hmisc: Harrell Miscellaneous. 2018.

[72] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, “SMOTE: Synthetic Minority Over-sampling Technique,” J. Artif. Intell. Res., vol. 16, pp. 321–357, Jun. 2002.

[73] “DMwR-package: Functions and data for the book ‘Data Mining with R’ in DMwR: Functions and data for ‘Data Mining with R.’” [Online]. Available: https://rdrr.io/cran/DMwR/man/DMwR-package.html. [Accessed: 14-Jan-2019].

[74] C. Tantithamthavorn, A. E. Hassan, and K. Matsumoto, “The Impact of Class Rebalancing Techniques on the Performance and Interpretation of Defect Prediction Models,” IEEE Trans. Softw. Eng., p. 1, Jan. 2018.

[75] F. E. H. Jr, rms: Regression Modeling Strategies. 2018.

[76] S. Wu and P. Flach, A scored AUC Metric for Classifier Evaluation and Selection. 2005.

[77] N. J. D. Nagelkerke, “A note on a general definition of the coefficient of determination,” Biometrika, vol. 78, no. 3, pp. 691–692, Sep. 1991.

[78] F. Wilcoxon, “Individual Comparisons by Ranking Methods,” Biom. Bull., vol. 1, no. 6, pp. 80–83, 1945.

[79] S. Kim and E. J. Whitehead Jr., “How Long Did It Take to Fix Bugs?,” in the 2006 International Workshop on Mining Software Repositories, New York, NY, USA, 2006, pp. 173–174.

[80] M. Tan, L. Tan, S. Dara, and C. Mayeux, “Online Defect Prediction for Imbalanced Data,” in Proceedings of the 37th International Conference on Software Engineering - Volume 2, Piscataway, NJ, USA, 2015, pp. 99–108.