2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering

Defect Prediction: Accomplishments and Future Challenges

Yasutaka Kamei
Principles of Software Languages Group (POSL)
Kyushu University, Fukuoka, Japan
Email: kamei@ait.kyushu-u.ac.jp

Emad Shihab
Dept. of Computer Science and Software Engineering
Concordia University, Montréal, Canada
Email: eshihab@encs.concordia.ca

***Abstract*—As software systems play an increasingly important role in our lives, their complexity continues to increase. The increased complexity of software systems makes the assurance of their quality very difficult. Therefore, a significant amount of recent research focuses on the prioritization of software quality assurance efforts. One line of work that has been receiving an increasing amount of attention for over 40 years is software defect prediction, where predictions are made to determine where future defects might appear. Since then, there have been many studies and many accomplishments in the area of software defect prediction. At the same time, there remain many challenges that face that field of software defect prediction. The paper aims to accomplish four things. First, we provide a brief overview of software defect prediction and its various components. Second, we revisit the challenges of software prediction models as they were seen in the year 2000, in order to reflect on our accomplishments since then. Third, we highlight our accomplishments and current trends, as well as, discuss the game changers that had a significant impact on software defect prediction. Fourth, we highlight some key challenges that lie ahead in the near (and not so near) future in order for us as a research community to tackle these future challenges.**

I. Introduction

If you know your enemies and know yourself, you will not be imperiled in a hundred battles [89]. This is the quote by Sun Tzu (c. 6th century BCE), who was a Chinese general, military strategist, and author of the book The Art of War, an immensely influential ancient Chinese book on military strategy. This quote is the one of the principle of empirical software engineering. To know your *enemies* (i.e., defects) and *yourself* (i.e., software systems) and win *battles* (i.e., leading a project to success conclusion), one needs to investigate a large amount of research on Software Quality Assurance (SQA). SQA can be broadly defined as the set of activities that ensure software meets a specific quality level [16].

As software systems continue to play an increasingly important role in our lives, their complexity continues to increase; making SQA efforts very difficult. At the same time, the importance of SQA efforts is of paramount importance. Therefore, to ensure high software quality, software defect prediction models, which describe the relationship between various software metrics (e.g., SLOC and McCabe's Cyclomatic complexity) and software defects, have been proposed [57, 95]. Traditionally, software defect prediction models are used in two ways: (1) to predict where defects might appear in the future and allocate SQA resources to defect-prone artifacts (e.g., subsystems and files) [58] and (2) to understand the effect of factors on the likelihood of finding a defect and derive practical guidelines for future software development projects [9, 45].

Due to its importance, defect prediction work has been at the focus of researchers for over 40 years. Akiyama [3] first attempted to build defect prediction models using size-based metrics and regression modelling techniques in 1971. Since then, there have been a plethora of studies and many accomplishments in the software defect prediction area [23]. At the same time, there remain many challenges that face software defect prediction. Hence, we believe that it is a perfect time to write a Future of Software Engineering (FoSE) paper on the topic of software defect prediction.

The paper is written from a budding university researchers' point of view and aims to accomplish four things. First, we provide a brief overview of software defect prediction and its various components. Second, we revisit the challenges of software prediction models as they were seen in the year 2000, in order to reflect on our accomplishments since then. Third, we highlight the accomplishments and current trends, as well as, discuss the game changers that had a significant impact on the area of software defect prediction. Fourth, we highlight some key challenges that lie ahead in the near (and not so near) future in order for us as a research community to tackle these future challenges.

Target Readers. The paper is intended for researchers and practitioners, especially masters and PhD students and young researchers. As mentioned earlier, the paper is meant to provide background on the area, reflect on accomplishments and present key challenges so that the reader can quickly grasp and be able to contribute to the software defect prediction area.

Paper Organization. The paper is organized as follows. Section II overviews the area of defect prediction models. Section III revisits the challenges that existed in the year 2000. Section IV describes current research trends and presents game changers, which dramatically changed impacted the field of defect prediction. Section V highlights some key challenges for the future of defect prediction. Section VI draws conclusions.

978-1-5090-1855-0/16 $31.00 © 2016 IEEE
DOI 10.1109/SANER.2016.56

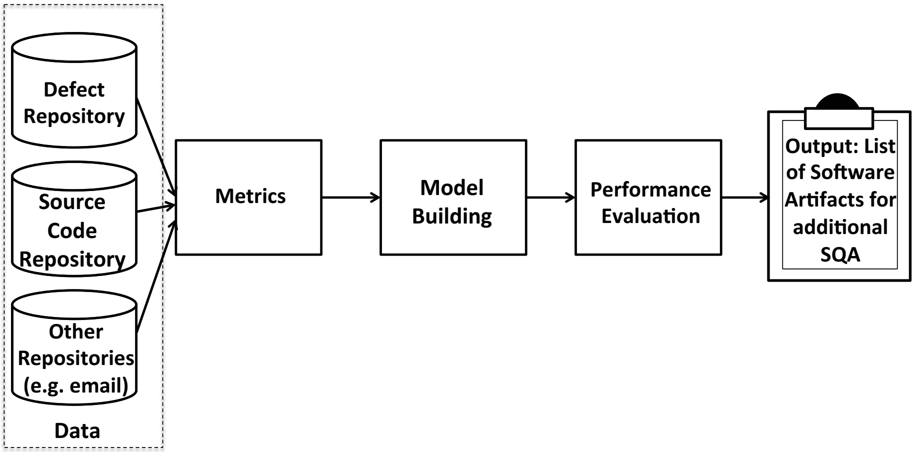


Fig. 1. Overview of Software Defect Prediction [78]

II. Background

As mentioned earlier, the main goal of most software defect prediction studies is (1) to predict where defects might appear in the future and (2) to quantify the relative importance of various factors (i.e., independent variables) used to build a model. Ideally, these predictions will correctly classify software artifacts (e.g., subsystems and files) as being defect-prone or not and in some cases may also predict the number of defects, defect density (the number of defects / SLOC) and/or the likelihood of a software artifact including a defect [78].

Figure 1 shows an overview of the software defect prediction process. First, data is collected from software repositories, which archive historical data related to the development of the software project, such as source code and defect repositories. Then, various metrics that are used as independent variables (e.g., SLOC) and a dependent variable (e.g., the number of defects) are extracted to build a model. The relationship between the independent variables and dependent variable is modeled using statistical techniques and machine learning techniques. Finally, the performance of the built model is measured using several criteria such as precision, recall and AUC (Area Under the Curve) of ROC (the Receiver Operating Characteristic). We briefly explain each of the four aforementioned steps next.

A. Data

In order to build software defect prediction models, we need a number of metrics that make up the independent and dependent variables. Large software projects often store their development history and other information, such as communication, in software repositories. Although the main reason for using these repositories is to keep track of and record development history, researchers and practitioners realize that this repository data can be used to extract software metrics. For example, prior work used the data stored in the source control repository to count the number of changes made to a file [95] and the complexity of changes [24], and used this data to predict files that are likely to have future defects.

Software defect prediction work generally leverages various types of data from different repositories, such as (1) source code repositories, which store and record the source code and development history of a project, (2) defect repositories, which track the bug/defect reports or feature requests filed for a project and their resolution progress and (3) mailing list repositories, which track the communication and discussions between development teams. Other repositories can also be leveraged for defect prediction.

B. Metrics

When used in software defect prediction research, metrics are considered to be independent variables, which means that they are used to perform the prediction (i.e., the predictors). Also, metrics can represent the dependent variables, which means they are the metrics being predicted (i.e., these can be pre- or post-release defects). Previous defect prediction studies used a wide variety of independent variables (e.g., process [24, 25, 37, 57, 59, 67], organizational [9, 61] or code metrics [30, 38, 51, 60, 85, 95]) to perform their predictions. Moreover, several different metrics were used to represent the dependent variable as well. For example, previous work predicted different types of defects (e.g., pre-release [59], post-release [92, 95], or both [82, 83]).

C. Model Building

Various techniques, such as linear discriminant analysis [30, 65, 71], decision trees [39], Naive Bayes [54], Support Vector Machines (SVM) [13, 36] and random forest [20], are used to build defect prediction models. Each of the aforementioned techniques have their own benefits (e.g., they provide models that are robust to noisy data and/or provide more explainable models).

Generally speaking, most defect prediction studies divide the data into two sets: a training set and a test set. The training set is used to train the prediction model, whereas the testing set is used to evaluate the performance of the prediction model.

D. Performance Evaluation

Once a prediction model is built, its performance needs to be evaluated. Performance is generally measured in two ways: *predictive power* and *explanative power*.

Predictive Power. Predictive power measures the *accuracy* of the model in predicting the software artifacts that have defects. Measures such as precision, recall, f-measure and AUC-ROC, which plots the the false positive rate on the x-axis and true positive rate on the y-axis over all possible classification thresholds, are commonly-used in defect prediction studies.

Explanative Power. In addition to measuring the predictive power, explanatory power is also used in defect prediction studies. Explanative power measures how well the variability in the data is explained by the model. Often the R^2 or deviance explained measures are used to quantify the explanative power. Explanative power is particularly useful since it enables us to measure the variability explained by each independent variable in the model, providing us with a ranking as to which independent variable is most "useful".

III. Past Challenges: The Early 2000s

In order to grasp the level of accomplishments, it is necessary to look back and examine the challenges that the software defect prediction community faced in the past. In particular, we look back to the early 2000s (2000-2002), when Fenton *et al.* [16] published the seminal survey on software defect prediction. To enhance readability, we organize this section along the four dimensions used in Section II, i.e., data, metrics, modelling and performance evaluation. Figure 2 shows the overview of past challenges, current trends and future challenges in defect prediction studies.

A. Data

Past Challenge 1: Lack of Availability and Openness. In the year 2000, one of the main challenges facing all data-driven approaches (including software defect prediction) was the lack of data availability. Software defect prediction was done only within several select, and perhaps forward-thinking, companies using their industrial data. However, since software companies almost never want to disclose the quality of their software, researchers could rarely obtain the datasets used in previous studies, which was a major challenge.

Validating this challenge, a survey of software defect prediction papers published between the years 2000-2010 [78] showed that 13 out of 14 defect prediction papers published between the years 2000-2002 conducted their studies using datasets that are collected from industrial projects.[1] Obviously, these dataset were not shared. The other paper that used open source software data (the Apache Web server project) [12] never made their datasets publicly available. This shows that back in the early 2000s, data availability and openness was a real challenge facing the defect prediction community.

Past Challenge 2: Lack of Variety in Types of Repositories. In addition to the lack of availability of data, the variety of the data was very limited as well. In the year 2000, most papers used data from source code and defect repositories, because those repositories provided the most basic information for building defect prediction models. For example, all 14 defect prediction studies between the years 2000 and 2002 used source code and defect repositories only [78]. Clearly, this shows that not only was data scarce, it was also very difficult to come up with different types of data.

Past Challenge 3: Lack of Variety of Granularity. In addition to data availability and variety, the granularity of the data was another issue that faced software defect prediction work in the early 2000s. The prediction unit (granularity) heavily depends on the data that is collected from the software repositories and used to build the defect prediction models. There are different levels of granularity such as the subsystem [93], file [11] or function [37] level.

In the early 2000s, the majority of studies were performed at the subsystem or file levels. Only one paper [90] of the 14 defect prediction studies between 2000 and 2002 performed its prediction at the function level [78]. The main reason for most studies performing their prediction at high levels of granularity is that repository data is often given at the file level and can be easily abstracted to the subsystem level. Although performing predictions at the subsystem and file levels may lead to better performance results [76, 95], the usefulness of the defect prediction studies becomes less significant (i.e., since more code would need to be inspected at high levels of abstraction).

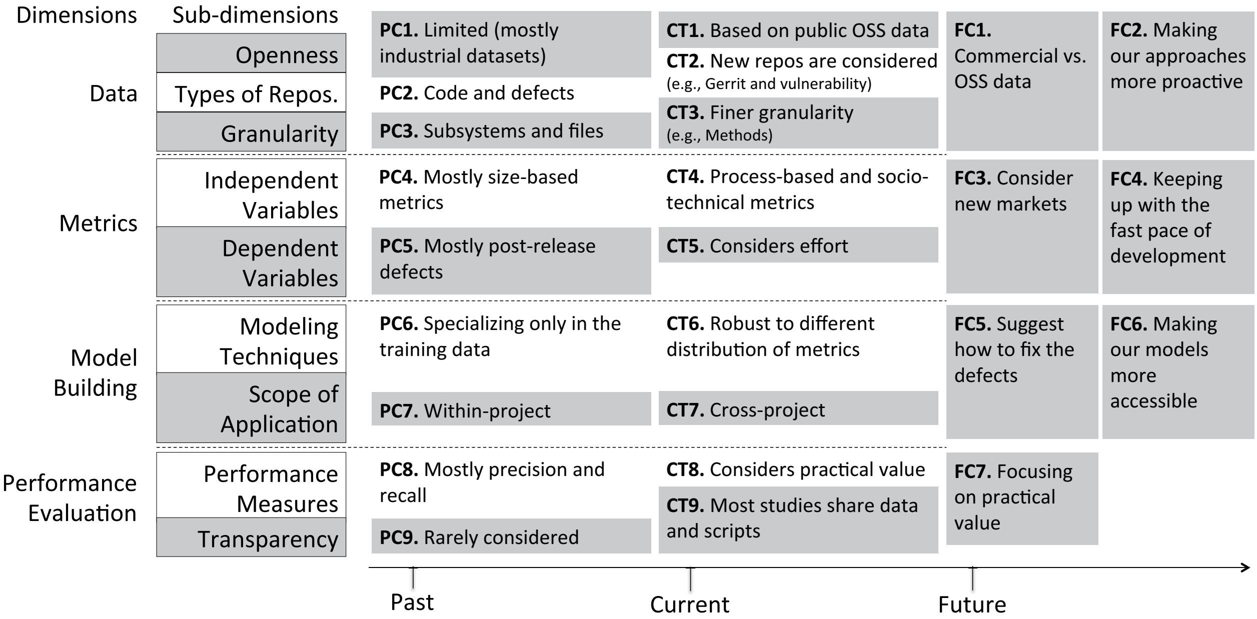
Software defect prediction studies in the early 2000s suffered from a lack of data availability, variety and granularity.

B. Metrics

Due to the limitations on data in the early 2000s, there were several implications on the metrics and the type of metrics that were used in software defect prediction models.

Past Challenge 4: Lack of Variety of Independent Variables —Size-Based Metrics. Size-based metrics (i.e., product metrics) are metrics that are directly related to or derived from the source code (e.g., complexity or size). In the early 2000s, a large body of defect prediction studies used product metrics to predict defects. The main idea behind using product metrics is that, for example, complex code is more likely to have defects. However, as Fenton and Neil mentioned [17] in their future of software engineering paper in 2000, while size-based metrics correlated to the number of defects, they are poor predictors of defects (i.e., there is no linear relationship between defect density and size-based metrics). Furthermore, several studies found that size-based metrics, especially code complexity metrics, tend to be highly correlated with each other and with the simple measure of Lines of Code (LOC) [17].

[1] Those 14 defect prediction papers are selected by (1) searching for papers related to software defect prediction in the top software engineering venues (e.g., TSE, ICSE and FSE), (2) reading through the titles and abstracts of each paper to classify whether or not the papers are actually related to software defect prediction (details can be found in the Appendix of [78]).



Dimensions	Sub-dimensions	Past	Current	Future	
Data	Openness	**PC1.** Limited (mostly industrial datasets)	**CT1.** Based on public OSS data	**FC1.** Commercial vs. OSS data	**FC2.** Making our approaches more proactive
	Types of Repos.	**PC2.** Code and defects	**CT2.** New repos are considered (e.g., Gerrit and vulnerability)		
	Granularity	**PC3.** Subsystems and files	**CT3.** Finer granularity (e.g., Methods)		
Metrics	Independent Variables	**PC4.** Mostly size-based metrics	**CT4.** Process-based and socio-technical metrics	**FC3.** Consider new markets	**FC4.** Keeping up with the fast pace of development
	Dependent Variables	**PC5.** Mostly post-release defects	**CT5.** Considers effort		
Model Building	Modeling Techniques	**PC6.** Specializing only in the training data	**CT6.** Robust to different distribution of metrics	**FC5.** Suggest how to fix the defects	**FC6.** Making our models more accessible
	Scope of Application	**PC7.** Within-project	**CT7.** Cross-project		
Performance Evaluation	Performance Measures	**PC8.** Mostly precision and recall	**CT8.** Considers practical value	**FC7.** Focusing on practical value	
	Transparency	**PC9.** Rarely considered	**CT9.** Most studies share data and scripts		

Fig. 2. Overview of Past Challenges, Current Trends and Future Challenges in Defect Prediction Studies (PC1: Past Challenge 1, CT1: Current Trend 1, and FC1: Future Challenge 1).

Past Challenge 5: Lack of Variety of Dependent Variables —Post-Release Defects. Generally speaking, the majority of defect prediction studies predicted post-release defects. In fact, between 2000 and 2002, 13 of 14 defect prediction studies used post-release defects as a dependent variable [78]. Although the number of post-release defects is important and measures the quality of the released software, it is not the only criteria for software quality. The fact that so many studies focused on the prediction of post-release defects is a limitation of software defect prediction work.

In the early 2000s, software defect prediction mainly leveraged size-based metrics as independent variables and post-release defects as a dependent variable.

C. Model building

Past Challenge 6: Specializing Only in the Training Data. In the early 2000s, linear regression and logistic regression models were often used as modeling techniques due to their simplicity. Defect prediction models assume that the distributions of the metrics in the training and testing datasets are similar [86]. However, in practice, the distribution of metrics can vary among releases, which may cause simple modeling techniques such as linear and logistic regression models to specialize only in the training data and perform poorly in its prediction on the testing data.

Past Challenge 7: Building Models for Projects in the Initial Development Phases.

In the early 2000s, most software defect prediction studies trained their models on data from the same project, usually from early releases. Then, the trained models were used to predict defects in future releases. In practice however, training data may not be available for projects in the initial development phases or for legacy systems that have not archived historical data. How to deal with projects that did not have prior project data was an open challenge in the early 2000s.

In the early 2000s, software defect prediction studies had to deal with the lack of project data and the fact that metrics may not have similar distributions.

D. Performance Evaluation

Past Challenge 8: Lack of Practical Performance Measures. Once the prediction models are built, one of the key questions is how well do they perform. In the early 2000s, many defect prediction studies empirically evaluated their performance using standard statistical measures such as precision, recall and model fit (measured in R^2 or deviance explained). Such standard statistical measures are fundamental criteria to know how well defect prediction models predict and explain defects. However, in some cases other (and more practical criteria) need to be considered. For example, how much effort is needed to address a predicted defect or the impact of a defect may need to be taken into consideration.

Past Challenge 9: Lack of Transparency/Repeatability. In the early 2000s, due to the lack of availability and openness of the datasets, other studies were not able to repeat the findings of prior studies. Such lack of repeatability misses the critiques of current and new research [77] and [18]. Hence, comparing the performance of a technique with prior techniques was nearly impossible in the early 2000s.

In the early 2000s, defect prediction studies lacked practical performance evaluation measures and transparency.

IV. Current Trends

The challenges faced in the early 2000s were the focus of the research that followed. Solutions to many of the challenges mentioned in Section III were proposed and major advancements were accomplished. In this section, we highlight some of the *current* trends in the area of software defect prediction. Furthermore, we discuss - what we call game changers - that had a significant impact on the accomplishments in the software defect prediction area. Similar to the previous sections, we organize the trends along the four dimensions, data, metrics, models and performance evaluation.

A. Data

Current Trend 1: Availability and Openness. Once the software defect prediction community realized that data availability and openness is a key factor to its success, many defect prediction studies started sharing not only their data, but even their analysis scripts. For example, in 2004, NASA MDP (metrics data program) shared 14 datasets that are measured during the software development of NASA projects through the PROMISE repository (we will discuss the PROMISE repository later in this section) [53]. The NASA datasets were some of the first public datasets from industrial software projects to be shared in the defect prediction domain. Similarly, Zimmermann *et al.* [95], D'Ambros *et al.* [11], Jureczko *et al.* [27], Kamei *et al.* [31] and many others shared their open source data. In fact, many conferences now have special tracks to facilitate the sharing of datasets and artifacts. For example, the ESEC/FSE conference [15] now has a replication package track that encourages authors to share their artifacts. The MSR conference now has a dedicated data showcase track that focuses on the sharing of datasets that can be useful for the rest of the community.

Reflecting back, the current trend of data sharing and openness have in many ways helped alleviate the challenge that existed in the early 2000s. That said, not all data is being shared; our community needs to continue to nourish such efforts in order to make data availability and openness a non-existing issue.

Current Trend 2: Types of Repositories. In addition to using the traditional repositories such as defect and code repositories, recent studies have also explored other types of repositories. For example, Meneely and Williams [49] leveraged the vulnerabilities database (e.g., the National Vulnerability Database and the RHSR security metrics database) to examine the effect of the "too many cooks in the kitchen" phenomena on software security vulnerabilities. Other studies such as the study by Lee *et al.* [41] leveraged Mylyn repositories, which capture developer interaction events. McIntosh *et al.* [45] leveraged Gerrit data, which facilitates a traceable code review process for git-based software projects to study of the impact of modern code review practices on software quality. Other types of repositories are also being used, especially as software development teams become more appreciative of the power of data analytics.

Game Changer 1: OSS projects. The open source initiative, founded in 1998 is an organization dedicated to promoting open source software (OSS), describes *open source software as software that can be freely used, changed, and shared (in modified or unmodified form) by anyone* [66]. Nowadays, there is no end of the number of active OSS projects available online supported by a wide range of communities.

The proliferation of OSS projects is considered a game changer because it opened up the development history of many long-lived, widely-deployed software systems. The number of papers using OSS projects is rapidly and steadily growing over time [78].

Another way to access rich data sources is to cooperate with commercial organization (e.g., ABB Inc, [43] and Avaya [55]). However, in many cases, commercial organization are not willing to give access to the details of its data sources due to confidentiality reasons. Academic projects (e.g., course projects) have also been used, however, they tend to not be as rich since they do not have real customers, and are often developed by small groups. In short, OSS projects provide rich, extensive, and readily available software repositories, which had a significant impact on the software defect prediction field.

Reflecting back, the current trends show strong growth in the different types of repositories being used. We believe that exploring new repositories will have a significant impact on the future of software defect prediction since it will facilitate better and more effective models and allow us to explore the impact of different types of phenomena on software quality.

Current Trend 3: Variety of Granularity. Due to the benefits of performing predictions at a finer granularity, recent defect prediction studies have focused on more fine-grained levels, such as method level [19, 26, 37] and change level predictions [4, 31, 36]. For example, Giger *et al.* [19] empirically investigate whether or not defect prediction models at the method-level work well. Another example of work that aims to perform predictions at a fine granularity is the work on change-level prediction, which aims to predict defect introducing changes (i.e., commits). The advantage of predicting defect introducing changes, compared to subsystems or files is that a change is often much smaller, can be easily assigned and contains complete information about a single change (which is important if a fix spans multiple files for example).

Similar to change-level prediction, Hassan and Holt [25] propose heuristics to create the Top Ten List of subsystems that are most susceptible to defects. Their approach dynamically updates the list to reflect the progress of software development when developers modify source code. Kim *et al.* [37] built on the work by Hassan and Holt to propose BugCache, which caches locations that are likely to have defects based on locality of defects (e.g., defects are more likely to occur in recently added/changed entities). When source code is modified, the cached locations are dynamically updated to reflect that the defect occurrences directly affect the cached locations.

The current trends have realized that the practical value of the predictions decrease as the abstraction level increases (i.e., since more code would need to be inspected at high levels of abstraction). Recently, studies have focused more on performing predictions at a finer level of granularity, e.g., at the method-level and change-level.

B. Metrics

Current Trend 4: Variety of Independent Variables. In addition to using size-based metrics (i.e., product metrics), recent software defect prediction used process metrics, which measure the change activity that occurred during the development of a new release, to build highly accurate defect prediction models [24, 25, 37, 57, 59, 67]. The idea behind using process metrics in defect prediction is that the process used to develop the code may lead to defects, hence the process metrics may be a good indicator of defects. For example, if a piece of code is changed many times or by many people, this may indicate that it is more likely to be defect-prone. One of the main advantages of process metrics is that process metrics are independent of the programming language, so process metrics are easier to expand to other languages than product metrics.

Although much of the current research used process metrics to predict defects, other metrics have been proposed in the literature. For example, studies explored design/UML metrics [7, 14, 64] (which capture the design of the software system), social metrics [5] (which combine dependency data from the code and from the contributions of developers), organizational metrics [55] (which capture the geographic distribution of the development organization, e.g., the number of sites that modified a software artifact) and ownership metrics [6] (which measure the level of ownership of a developer to a software artifact). In addition, there are studies that use textual information (e.g., identifiers and comments in source code) as independent variables [36, 87].

Reflecting back, we see a very clear evolution in the way software defect prediction work uses metrics. Initially, most metrics were deigned to help improve the prediction accuracy. However, more recently, studies have used defect prediction to examine the impact of certain phenomena, e.g., ownership, on code quality. Such studies have pushed the envelope in metric design and contributed significantly to the body of empirical work on software quality in general.

Current Trend 5: Variety of Dependent Variables. In the early 2000s, most studies used post-release defects as a dependent variable [7, 12]. More recently however, many studies started to focus on different types of dependent variables that span much more than post-release defects [10, 80, 82, 88]. For example, Shihab *et al.* [82] focused on predicting defects that break pre-existing functionality (breakage defects) and defects in files that had relatively few pre-release changes (surprise defects). Meneely and Williams [49] built models to predict software security vulnerabilities. Garcia *et al.* [88] predicted blocking defects, which block other defects from being fixed. Furthermore, researchers have proposed to perform predictions at the change-level, focusing on predicting defect-inducing changes [36, 56]. The prediction can be conducted at the time when the change is submitted for unit test (private view) or integration. Such immediate feedback ensures that the design decisions are still fresh in the minds of developers [31].

Game Changer 2: The PROMISE repository. The PROMISE repository is a research data repository for software engineering research datasets and offers free and long-term storage for research datasets [53]. To date, the PROMISE repository contains more than 45 datasets related to defect prediction. The PROMISE repository started to share the samples of the Metrics Data Program, which was run by NASA for collecting static code measures in 2002, as version 1 in 2004. The PROMISE repository is currently in version 4 and it stores more than one terabyte of data.
The PROMISE repository is a game changer, because it facilitated the sharing of data sets across researchers. This dramatically speeds up the progress of defect prediction research. The repository is popular among software defect prediction researchers.

Reflecting back, it seems that many of the recent studies have realized that not all defects are equal and that there are important defects that are not post-release defects. The recent trends show that different types of dependent variables are being considered, which take into account different stakeholders and different timing (e.g., releases vs. commits). For example, the prediction of blocking defects clearly shows that helping the developers in the main goal, which is different from the traditional defect prediction studies which mainly focused on the customer as the main stakeholder. The prediction of defect-inducing changes shows that predictions are made early on, comparing with the traditional studies which have their drawbacks (i.e., predictions are made too late in the development cycle).

C. Model building

Current Trend 6: Treating Uncertainty in Model Inputs or Outputs. As we mentioned in Section III, many defect prediction models assume that the distributions of the metrics in the training and testing datasets are similar [86]. However, in practice, the distribution of metrics can vary among releases. To deal with projects that do not have similar distributions in their training and testing datasets, recent defect prediction studies have used ensemble techniques. One of the frequently-used ensemble techniques is a random forest classifier that consists of a number of tree-structured classifiers [20, 29, 47]. New objects are classified from an input vector that is composed of input vectors on each tree in the forest. Each tree casts a vote at the input vector by providing a classification.

The forest selects the classification that has the most votes over all trees in the forest.

The main advantages of random forest classifiers are that they generally outperform simple decision trees algorithms in terms of prediction accuracy and random forest classifiers are more resistant to noise in data [44].

Current Trend 7: Building Cross-Project Defect Prediction Models for Projects with Limited Data.

The majority of studies in the early 2000s focused on within-project defect prediction. This means that they used data from the same project to build their prediction models. However, one major drawback with such an approach that was highlighted by previous studies [7, 94] is the fact that within-project defect prediction requires sufficient historical data, for example, data from past releases. Having sufficient historical data can be a challenge, especially for newer projects and legacy projects. Hence, more recently, defect prediction studies have worked on cross-project defect prediction models, i.e., models that are trained using historical data from other projects [28, 50, 62, 69, 70, 86, 91, 94], in order to make defect prediction models available for projects in the initial development phases, or for legacy systems that have not archived historical data [7].

In the beginning, cross-project defect prediction studies showed that the performance of cross-project predictions is lower than the within-project predictions. However, recent work has shown that cross-project models can achieve performance similar to that of within-project models. For example, in their award winning work, Zhang *et al.* [91] propose a context-aware rank transformation method to preprocess predictors and address the variations in their distributions, and showed that their cross-project models achieve performance that rivals within-project models. Recent work [69, 70] has also focused on secure data sharing to address the privacy concerns of data owners in cross-project models. For example, Peters *et al.* [69] studied the approaches that prevent the disclosure of sensitive metrics values without a significant performance degradation of cross-project models.

Reflecting back, thanks to cross-project defect prediction models, the recent defect prediction models are now available for projects in their initial development phases and for legacy systems that have not archived historical data. At this point, the research community has come a long way with cross-project defect prediction, however, many questions still remain. For example, the lack of availability of industrial data leaves open the question of whether models built using data from open source projects would apply to industrial projects. At this stage, cross-project defect prediction remains as an open research area.

D. *Performance evaluation*

Current Trend 8: Practical Performance Measures. In the early 2000s, most studies used traditional performance evaluation techniques such as precision and recall. More recently defect prediction studies have focused on more practical performance evaluations. To consider evaluation in more practical settings, recent work has considered the effort required to address the predicted software artifacts [29, 48, 52, 72, 73]. For example, recent work by Kamei *et al.* [29] evaluates common defect prediction findings (e.g., process metrics vs. product metrics and package-level prediction vs. file-level prediction) when effort is considered. Mende and Koschke [48] compared strategies to include the effort treatment into defect prediction models. Menzies *et al.* [52] argue that recent studies have not been able to improve defect prediction results since their performance is measured as a tradeoff between the probability of false alarms and probability of detection. Therefore, they suggest changing the standard goal to consider effort, i.e., to finding the smallest set of modules that contain most of the defects.

Furthermore, recent defect prediction studies have also conducted an interview to better understand the defect prediction models and derive practical guidelines for developing high quality software. For example, Shihab *et al.* [82] ask the opinions of the highly experienced quality manager in the project about their prediction results. Based on their opinions, the authors conclude that the defect prediction models should not only predict defect locations, but also detect patterns of changes that are suggestive of a particular type of defect and recommend appropriate remedies.

Reflecting back, we see that more recent studies have focused on the practical value of software defect prediction and on trying to evaluate their models in realistic settings. We strongly support such research and see this trend continuing to grow since software defect prediction models are starting to be used in industry (e.g., [79]).

Game Changer 3: SZZ algorithm. Śliwerski *et al.* proposed the SZZ algorithm [84] that extracts whether or not a change introduces a defect from Version Control Systems (VCSs). The SZZ algorithm links each defect fix to the source code change introducing the original defect by combining information from the version archive (such as CVS) with the defect tracking system (such as Bugzilla).

The SZZ algorithm is a game changer, because it provided a new data source for defect prediction studies. Without the SZZ algorithm, we would not be able to determine when a change induces a defect and conduct empirical studies on defect prediction models. When developers submit their revision to add functionality and /or fix defects to VCSs in their project, they enter comments (e.g., fix defect #1000) related to their revision in the log message. However, there is no comments to detect that a change induces a defect (e.g., introducing defects), because developers have no intention of introducing defects and introduce them wrongly. At the time of writing this paper (September 2015), the paper [84] is cited by more than 400 times according to Google Scholar†.

†https://goo.gl/lUiGbR

Current Trend 9: Transparency/Repeatability. As mentioned earlier, the software engineering community as a whole has realized the value of making studies as transparent as possible. For example, recent defect prediction studies have kept transparency for their studies, then generated a lively discussion (e.g., critique) of the results of the studies and led to new findings. For example, Shepperd *et al.* [77] derive some comments on the NASA software defect datasets (e.g., the dataset contains several erroneous and implausible entries) and share cleaned NASA defect datasets. Then, Ghotra *et al.* [18] revisit the findings of Lessmann *et al.* [42] that used original NASA datasets. In contrary to prior results [42], Ghotra *et al.* show that there are statistically significant differences in the performance of defect prediction models that are trained using different classification techniques. Such value is made possible thanks to the fact that the NASA projects made their datasets available.

Reflecting back, we see a very healthy and progressive trend where software defect prediction studies are becoming more transparent. Such changes will only make our findings more practical and will encourage us to advance the science (not only the engineering) behind the area of software defect prediction since it allows us to repeat and scrutinize assumptions of prior studies.

V. Challenges for the Near (and not so near) Future

The field of software defect prediction has made many accomplishments in the recent years. However, many challenges remain and (we believe) will pop up in the future due to changes in technology, data and the increasingly important role software systems continue to play.

Future Challenge 1: Commercial vs. OSS Data. As Section IV shows, many researchers make use of the dataset collected from open source software projects. The main reason for using data from open source projects is that these projects archive long and practical development history and make their data publicly available. However, the generality of our findings and techniques to non open source software projects (e.g., commercial projects) is not studied in depth; in part due to the lack of availability of data.

To solve this challenge, we need to make more partnership with industrial partners and have access to their repositories. The authors had some success starting projects with industry and our experience shows that starting such partnerships is easier than one might think. Especially with the type of big data and data analysis in general, companies are starting to realize that there is value in mining their data. That said, the industrial partners need to see some value with what the researchers are doing with their data, otherwise they may lose interest.

On the positive side, many industrial projects have already shifted to modern software development environment (e.g., Git and Gerrit). That is, it is easy to apply our tools to their projects without much effort. In short, the most important thing is to have the will to start a collaboration with industrial projects. When we continue to demonstrate the value of data in software repositories and the benefits of MSR techniques for helping practitioners in their daily activities, practitioners are more likely to contact us and consider using our technique in practice.

Game Changer 4: Statistical and Machine Learning Tools (Weka and R).

The majority of defect prediction studies rely heavily on statistical and machine learning (ML) techniques. Some of the most popular statistical and ML tools used in software defect prediction studies are R and WEKA. WEKA [22] is a tool developed by the Machine Learning Group at University of Waikato. Weka is a collection of machine learning algorithms for data mining tasks that can be applied directly to a dataset. R [1] is an open source language and environment for statistical analysis.

Weka and R are game changers, because both of them provide a wide variety of data pre-processing, statistical (linear and nonlinear modelling, classical statistical tests and classification) and support for graphical techniques. They are also open source and are highly extensible. Therefore, Weka and R are commonly used in defect prediction studies [78].

Future Challenge 2: Making our Approaches More Proactive. When it comes to software defect prediction, many of our techniques thus far have been reactive in nature. What that means is that we observe the software development process and then use this data to predict what will happen post-release. However, in many cases practitioners would like to have predictions happen much sooner, e.g., before or as soon as they commit their changes. Doing so would make our approaches more proactive in a sense, since it will allow us to perform our predictions as the development is happening rather than waiting till it has completed.

Several studies have already started to work in this area, using metrics from the design stage [76] and performing change-level defect predictions [31, 36]. However, there remains much work to do in this area. We may be able to devise tools that not only predict risky areas or changes, but also generate tests (and possibly fixes) for these risky areas and changes. We can also devise techniques that proactively warn developers, even before they modify the code, that they are working with risky code that has had specific types of defects in the past.

Future Challenge 3: Considering New Markets. The majority of software defect prediction studies used code and/or process metrics to perform their predictions. To date, this has served the community well and has helped us advance the state-of-the-art in software defect prediction. However, comparing with the year 2000, our environment has changed dramatically. Therefore, we need to tackle the challenges that new environments raise.

One example of new markets that we should tackle is mobile application fields. We use personal smart phone every day during moving and update applications from online stores (e.g., Google Play and App Store). Mobile applications play a significant role in our daily life and these applications have different characteristics compared to conventional applications that we studied in the past. These mobile application stores allow us to gain valuable user data that, till today, has not been leveraged in the area of software defect prediction. Few studies have leveraged this data to improve testing [32, 33]. Moreover this data can be leveraged to help us understand what impacts users and in which way the user is impacted. Such knowledge can help us build better and more accurate models. For example, we may be able to build the models that predict specific types of defects (e.g., energy defects and performance defects) using the complaint of users [10, 34]. We anticipate the use of user data in software defect prediction models to be an area of significant growth in the future.

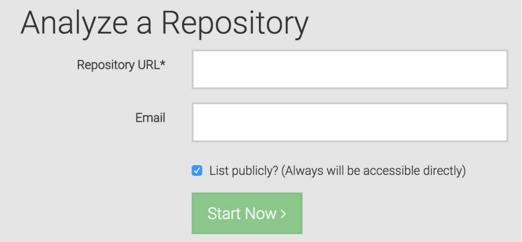
Future Challenge 4: Keeping Up with the Fast Pace of Development. In today's fast changing business environment, the recent trend of software development is to reduce the release cycle to days or even hours [74]. For example, the Firefox project changed their release process to a rapid release model (i.e., a development model with a shorter release cycle) and releases over 1,000 improvements and performance enhancements with version 5.0 in 3 months [35]. IMVU, which is an online social entertainment website, deploys new code fifty times a day on average [2].

We need to think about how we can integrate our research into continuous integration. For example, O'Hearn suggested that commenting on code changes at review time makes a large difference in helping developers as opposed to producing the defect list from batch-mode analysis because such commenting does not ask them to make a context switch to understand and act on an analysis report [68].

The majority of quality assurance research focused on defect prediction models that identify defect-prone modules (i.e., files or packages) at release-level like batch-mode analysis [21, 24, 43, 58]. Those studies use the datasets collected from previous releases to build a model and derive the list of defect-prone modules. Such models require practitioners to remember the rationale and all the design decisions of the changes to be able to evaluate if the change introduced a defect.

To solve the problem that O'Hearn pointed out, we can focus on Just-In-Time (JIT) Quality Assurance [31, 36, 56], which performs predictions at the change level. JIT defect prediction models aim to be an earlier step of continuous quality control because it can be invoked as soon as a developer commits code to their private or to the team's workspace.

There remains much work to do in this area. We still need to evaluate how to integrate JIT models into the actual contentious integration process. For example, we can devise the approaches that suggest how much effort developers spend to find and fix defects based on the probability of prediction (e.g., while JIT models predict that this change includes defects with 80% of probability, the developer should work on the change for an additional 30 minutes to find the defects).



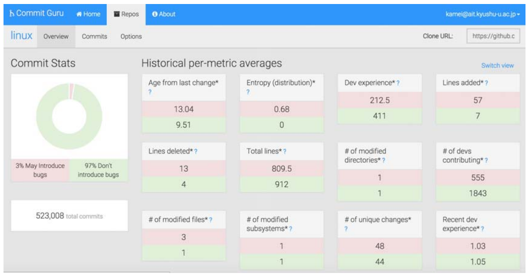
Analyze a Repository

Repository URL*

Email

☑ List publicly? (Always will be accessible directly)

Fig. 3. Adding a Repository in Commit Guru



Clone URL: https://github.c

Commit Stats

3% May Introduce bugs | 97% Don't introduce bugs

523,008 total commits

Historical per-metric averages

Age from last change* ? 13.04 9.51	Entropy (distribution)* ? 0.68 0	Dev experience* ? 212.5 411	Lines added* ? 57 7
Lines deleted* ? 13 4	Total lines* ? 809.5 912	# of modified directories* ? 1 1	# of devs contributing* ? 555 1843
# of modified files* ? 3 1	# of modified subsystems* ? 1 1	# of unique changes* ? 48 44	Recent dev experience* ? 1.03 1.05

Fig. 4. Commit Statistics of the Linux Kernel Project in Commit Guru

Future Challenge 5: Knowing How to Fix the Defects. The main purpose of defect prediction models thus far has mainly been two-fold: (1) to predict where defects might appear in the future and allocate SQA resources to defect-prone artifacts (e.g., subsystems and files) and (2) understand the effect of factors on the likelihood of finding a defect. Although such models have proven to be useful in certain contexts to derive practical guidelines for future software development projects, how to fix the defects that are flagged remains to be an open question.

Therefore, we envision that future defect prediction models will not only predict where the defects are, they will also provide information on how to best fix these defects. In the future, we need to understand what kind of defects happen and why such defects happen. Such information can be leveraged by developers to locate and fix the predicted defect. One area of research that might be useful here is the area of automated program repair [40, 46, 63], which produces candidate patches. Automated program repair can be combined with defect prediction to automatically generate patches for the predicted defects.

Future Challenge 6: Making our Models More Accessible. Recently, software engineering research papers are providing replication packages (e.g., including dataset, tool/script and readme files) with their studies. The main reason for doing so is (1) to ensure the validity and transparency of the experiments and (2) to allow others to replicate their studies and/or compare the performance of new models with the original models using the same datasets. The general belief is that we always want to improve the accuracy of our approaches.

However, in practice, we need to consider how easy our proposed models and replication packages are to use by other researchers and practitioners. When we just want to use a replication package, some types of replication packages are not suitable: (e.g., the README files are inaccurate and additional set up is needed to produce the same environment). In that case, we may give up on using these techniques due to time investment needed.

Furthermore, it would be great to provide tools that can be easily accessed via REST-APIs and original scripts for the people who want to integrate our tools into their projects. For example, Commit Guru [75] provides a language agnostic analytics and prediction tool that identifies and predicts risky software commits (Figure 3 and Figure 4). It is publicity available via the web. The tool simply requires a URL of the Git repository that you want to analyze (Figure 3). Its source code is freely available under the MIT license.[2] In short, we need to make our models and techniques simple and extendable, and where applicable, provide tools so that others can easily use our techniques.

Future Challenge 7: Focusing on Effort. From the year 2000 [17], we have one argument that we need to evaluate our prediction models in a practical setting (e.g., how much effort do defect prediction models reduce for code review?) instead of only improving precision and recall. Recent studies tried to tackle such problems when considering effort [8, 29, 48, 72]. Such studies use LOC or churn as a proxy for effort. However, our previous studies [81] show that using a combination of LOC, code and complexity metrics provides a better prediction of effort than using LOC alone. In the future, researchers need to examine what is the best way to measure effort in effort-aware defect prediction models. Such research can have a significant impact on the future applicability of defect prediction work.

VI. Conclusion

The field of software defect prediction has been well-researched since it was first proposed. As our paper showed, there have been many papers that explored different types of data and their implications, proposed various metrics, examined the applicability of different modeling techniques and evaluation criteria.

The field of software defect prediction has made a series of accomplishments. As our paper highlighted, there have been many papers that addressed challenges related to data (e.g., the recent trend of data sharing and openness have in many ways helped alleviate the challenge that existed in the early 2000s), metrics (e.g., studies have used defect prediction to examine the impact of certain phenomena, e.g., ownership, on code quality), model building (e.g., thanks to cross-project defect prediction models, the recent defect prediction models are now available for projects in the initial development phases) and performance evaluation (e.g., we see a very healthy and progressive trend where software defect prediction studies are becoming more accurate and transparent.).

At the same time, particular initiatives and works have had a profound impact on the area of software defect prediction, which we listed as game changers. For example, OSS projects providing rich, extensive, and readily available software repositories; the PROMISE repository providing SE researchers with a common platform to share datasets; the SZZ algorithm automatically extracting whether or not a change introduces a defect from VCSs, which in turn dramatically accelerated the speed of research, especially for JIT defect prediction; tools such as Weka and R providing a wide variety of data pre-processing and making available common statistical and ML techniques.

That said, there remain many future challenges for the field that we also highlight. For example, we need to tackle that the generality of our findings and the applicability of our techniques to non-open source software projects (e.g., commercial projects). We also need to consider new markets (e.g., mobile applications and energy consumption) in the domain of software quality assurance, as well as many other future challenges that may impact the future of software defect prediction.

This paper only provides the authors perspective based on their preferences and experiences. The aim of the paper is to provide the readers with an understanding of prior defect prediction studies and highlight some key challenges for the future of defect prediction.

Acknowledgment

We would like to thank Ahmed E. Hassan, Thomas Zimmerman and Massimiliano Di Penta (the co-chairs of Leaders of Tomorrow: Future of Software Engineering) for giving us the opportunity to write our vision paper on the topic of software defect prediction. We would also like to thank the reviewers and Dr. Naoyasu Ubayashi for their constructive and fruitful feedback. The first author was partially supported by JSPS KAKENHI Grant Numbers 15H05306 and 25540026.

References

[1] The R project. http://www.r-project.org/.

[2] B. Adams. On Software Release Engineering. ICSE 2012 technical briefing.

[3] F. Akiyama. An example of software system debugging. In *IFIP Congress (1)*, pages 353–359, 1971.

[4] L. Aversano, L. Cerulo, and C. Del Grosso. Learning from bug-introducing changes to prevent fault prone code. In *Proc. Int'l Workshop on Principles of Software Evolution (IWPSE'07)*, pages 19–26, 2007.

[5] C. Bird, N. Nagappan, H. Gall, B. Murphy, and P. Devanbu. Putting it all together: Using socio-technical networks to predict failures. In *Proc. Int'l Symposium on Software Reliability Engineering (ISSRE'09)*, pages 109–119, 2009.

[6] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu. Don't touch my code!: examining the effects of ownership on software quality. In *Proc. European Softw.*

[2] It can be downloaded at https://github.com/CommitAnalyzingService/CAS_Web (front-end) and https://github.com/CommitAnalyzingService/CAS_CodeRepoAnalyzer (back-end).

Eng. Conf. and Symposium on the Foundations of Softw. Eng. (ESEC/FSE'11), pages 4–14, 2011.

[7] L. C. Briand, W. L. Melo, and J. Wust. Assessing the applicability of fault-proneness models across object-oriented software projects. *IEEE Trans. Softw Eng*, 28:706–720, 2002.

[8] G. Canfora, A. D. Lucia, M. D. Penta, R. Oliveto, A. Panichella, and S. Panichella. Defect prediction as a multiobjective optimization problem. *Softw. Test., Verif. Reliab.*, 25(4):426–459, 2015.

[9] M. Cataldo, A. Mockus, J. A. Roberts, and J. D. Herbsleb. Software dependencies, work dependencies, and their impact on failures. *IEEE Trans. Softw. Eng.*, 99(6):864–878, 2009.

[10] T.-H. Chen, M. Nagappan, E. Shihab, and A. E. Hassan. An empirical study of dormant bugs. In *Proc. Int'l Working Conf. on Mining Software Repositories (MSR'14)*, MSR 2014, pages 82–91, 2014.

[11] M. D'Ambros, M. Lanza, and R. Robbes. An extensive comparison of bug prediction approaches. In *Proc. Int'l Working Conf. on Mining Software Repositories (MSR'10)*, pages 31–41, 2010.

[12] G. Denaro and M. Pezzè. An empirical evaluation of fault-proneness models. In *Proc. Int'l Conf. on Softw. Eng. (ICSE'02)*, pages 241–251, 2002.

[13] K. O. Elish and M. O. Elish. Predicting defect-prone software modules using support vector machines. *Journal of Systems and Software*, 81:649–660, 2008.

[14] A. Erika and C. Cruz. Exploratory study of a UML metric for fault prediction. In *Proc. Int'l Conf. on Softw. Eng. (ICSE'10) - Volume 2*, pages 361–364, 2010.

[15] ESEC/FSE 2015. Research sessions. http://esec-fse15.dei.polimi.it/research-program.html.

[16] N. E. Fenton and M. Neil. A critique of software defect prediction models. *IEEE Trans. Softw. Eng.*, 25:675–689, 1999.

[17] N. E. Fenton and M. Neil. Software metrics: Roadmap. In *Proc. Conf. on The Future of Software Engineering (FoSE)*, pages 357–370, 2000.

[18] B. Ghotra, S. McIntosh, and A. E. Hassan. Revisiting the impact of classification techniques on the performance of defect prediction models. In *Proc. Int'l Conf. on Softw. Eng. (ICSE'15)*, pages 789–800, 2015.

[19] E. Giger, M. D'Ambros, M. Pinzger, and H. C. Gall. Method-level bug prediction. In *Proc. the Int'l Symposium on Empirical Softw. Eng. and Measurement (ESEM'12')*, pages 171–180, 2012.

[20] L. Guo, Y. Ma, B. Cukic, and H. Singh. Robust prediction of fault-proneness by random forests. In *Proc. Int'l Symposium on Software Reliability Engineering (ISSRE'04)*, pages 417–428, 2004.

[21] T. Gyimothy, R. Ferenc, and I. Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Trans. Softw. Eng.*, 31:897–910, 2005.

[22] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The WEKA data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1):10–18, 2009.

[23] 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.*, 38(6):1276–1304, 2012.

[24] A. E. Hassan. Predicting faults using the complexity of code changes. In *Proc. Int'l Conf. on Softw. Eng. (ICSE'09)*, pages 78–88, 2009.

[25] A. E. Hassan and R. C. Holt. The top ten list: Dynamic fault prediction. In *Proc. the 21st IEEE Int'l Conf. on Software Maintenance*, pages 263–272, 2005.

[26] H. Hata, O. Mizuno, and T. Kikuno. Bug prediction based on fine-grained module histories. In *Proc. Int'l Conf. on Softw. Eng. (ICSE'12)*, pages 200–210, 2012.

[27] M. Jureczko and L. Madeyski. Towards identifying software project clusters with regard to defect prediction. In *Proc. Int'l Conf. on Predictor Models in Softw. Eng. (PROMISE'10)*, pages 9:1–9:10, 2010.

[28] Y. Kamei, T. Fukushima, S. McIntosh, K. Yamashita, N. Ubayashi, and A. E. Hassan. Studying just-in-time defect prediction using cross-project models. *Empirical Softw. Eng.*, pages 1–35, 2015.

[29] 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. Int'l Conf. on Software Maintenance (ICSM'10)*, pages 1–10, 2010.

[30] Y. Kamei, A. Monden, S. Matsumoto, T. Kakimoto, and K. Matsumoto. The effects of over and under sampling on fault-prone module detection. In *Proc. Int'l Symposium on Empirical Softw. Eng. and Measurement (ESEM'07)*, pages 196–204, 2007.

[31] 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.*, 39(6):757–773, 2013.

[32] H. Khalid, M. Nagappan, and A. E. Hassan. Examining the relationship between Findbugs warnings and end user ratings: A case study on 10,000 Android apps. *IEEE Software*, In Press.

[33] H. Khalid, M. Nagappan, E. Shihab, and A. E. Hassan. Prioritizing the devices to test your app on: A case study of Android game apps. In *Proc. Int'l Symposium on Foundations of Software Engineering (FSE'14)*, pages 610–620, 2014.

[34] H. Khalid, M. Nagappan, E. Shihab, and A. E. Hassan. What do mobile app users complain about? *IEEE Software*, 32(3):70–77, 2015.

[35] F. Khomh, B. Adams, T. Dhaliwal, and Y. Zou. Understanding the impact of rapid releases on software quality — the case of Firefox. *Empirical Softw. Eng.*, 20:336–373, 2015.

[36] S. Kim, E. J. Whitehead, Jr., and Y. Zhang. Classifying software changes: Clean or buggy? *IEEE Trans. Softw.*

Eng., 34(2):181–196, 2008.

[37] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller. Predicting faults from cached history. In *Proc. Int'l Conf. on Softw. Eng. (ICSE'07)*, pages 489–498, 2007.

[38] A. G. Koru and H. Liu. Building defect prediction models in practice. *IEEE Software*, 22:23–29, 2005.

[39] A. G. Koru and H. Liu. An investigation of the effect of module size on defect prediction using static measures. In *Proc. Int'l Workshop on Predictor Models in Softw Eng. (PROMISE'05)*, pages 1–5, 2005.

[40] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Proc. Int'l Conf. on Softw. Eng. (ICSE'12)*, pages 3–13, 2012.

[41] T. Lee, J. Nam, D. Han, S. Kim, and H. P. In. Micro interaction metrics for defect prediction. In *Proc. European Softw. Eng. Conf. and Symposium on the Fundations of Softw. Eng. (ESEC/FSE'11)*, pages 311–321, 2011.

[42] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Trans. Softw. Eng.*, 34:485–496, 2008.

[43] 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. Int'l Conf. on Softw. Eng. (ICSE'06)*, pages 413–422, 2006.

[44] L. Marks, Y. Zou, and A. E. Hassan. Studying the fix-time for bugs in large open source projects. In *Proc. Int'l Conf. on Predictive Models in Softw. Eng. (PROMISE'11)*, pages 11:1–11:8, 2011.

[45] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan. The impact of code review coverage and code review participation on software quality: A case study of the Qt, VTK, and ITK projects. In *Proc. Int'l Working Conf. on Mining Software Repositories (MSR'14)*, pages 192–201, 2014.

[46] S. Mechtaev, J. Yi, and A. Roychoudhury. DirectFix: Looking for simple program repairs. In *Proc. Int'l Conf. on Softw. Eng. (ICSE'15)*, pages 448–458, 2015.

[47] T. Mende and R. Koschke. Revisiting the evaluation of defect prediction models. In *Proc. Int'l Conf. on Predictor Models in Softw. Eng. (PROMISE'09)*, pages 1–10, 2009.

[48] T. Mende and R. Koschke. Effort-aware defect prediction models. In *Proc. European Conf. on Software Maintenance and Reengineering (CSMR'10)*, pages 109–118, 2010.

[49] A. Meneely and L. Williams. Strengthening the empirical analysis of the relationship between Linus' Law and software security. In *Proc. Int'l Symposium on Empirical Softw. Eng. and Measurement (ESEM'10)*, pages 9:1–9:10, 2010.

[50] T. Menzies, A. Butcher, D. Cok, A. Marcus, L. Layman, F. Shull, B. Turhan, and T. Zimmermann. Local versus global lessons for defect prediction and effort estimation. *IEEE Trans. Softw. Eng.*, 39(6):822–834, 2013.

[51] T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *IEEE Trans. Softw. Eng.*, 33:2–13, 2007.

[52] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener. Defect prediction from static code features: current results, limitations, new approaches. *Automated Software Engineering*, 17:375–407, 2010.

[53] T. Menzies, M. Rees-Jones, R. Krishna, and C. Pape. The PROMISE repository of empirical software engineering data, 2015.

[54] T. Menzies, B. Turhan, A. Bener, G. Gay, B. Cukic, and Y. Jiang. Implications of ceiling effects in defect predictors. In *Proc. Int'l Workshop on Predictor Models in Softw. Eng. (PROMISE'08)*, pages 47–54, 2008.

[55] A. Mockus. Organizational volatility and its effects on software defects. In *Proc. Int'l Symposium on Foundations of Softw. Eng. (FSE'10)*, pages 117–126, 2010.

[56] A. Mockus and D. M. Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5(2):169–180, 2000.

[57] 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. Int'l Conf. on Softw. Eng. (ICSE'08)*, pages 181–190, 2008.

[58] J. C. Munson and T. M. Khoshgoftaar. The detection of fault-prone programs. *IEEE Trans. Softw. Eng.*, 18(5):423–433, 1992.

[59] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *Proc. Int'l Conf. on Softw. Eng. (ICSE'05)*, pages 284–292, 2005.

[60] N. Nagappan and T. Ball. Using software dependencies and churn metrics to predict field failures: An empirical case study. In *Proc. Int'l Symposium on Empirical Softw. Eng. and Measurement (ESEM'07)*, pages 364–373, 2007.

[61] N. Nagappan, B. Murphy, and V. Basili. The influence of organizational structure on software quality: an empirical case study. In *Proc. Int'l Conf. on Softw. Eng. (ICSE'08)*, pages 521–530, 2008.

[62] J. Nam, S. J. Pan, and S. Kim. Transfer defect learning. In *Proc. Int'l Conf. on Softw. Eng. (ICSE'13)*, pages 382–391, 2013.

[63] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. SemFix: Program repair via semantic analysis. In *Proc. Int'l Conf. on Softw. Eng. (ICSE'13)*, pages 772–781, 2013.

[64] A. Nugroho, M. Chaudron, and E. Arisholm. Assessing UML design metrics for predicting fault-prone classes in a java system. In *Proc. Int'l Working Conf. on Mining Software Repositories (MSR'10)*, pages 21 –30, 2010.

[65] N. Ohlsson and H. Alberg. Predicting fault-prone software modules in telephone switches. *IEEE Trans. Softw. Eng.*, 22(12):886–894, 1996.

[66] Open Source Initiative. Welcome to the open source

initiative. http://opensource.org/.

[67] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Predicting the location and number of faults in large software systems. *IEEE Trans. Softw. Eng.*, 31(4):340–355, 2005.

[68] Peter O'Hearn. Moving fast with software verification. http://www0.cs.ucl.ac.uk/staff/p.ohearn/Talks/Peter-CAV.key.

[69] F. Peters, T. Menzies, L. Gong, and H. Zhang. Balancing privacy and utility in cross-company defect prediction. *IEEE Trans. Softw. Eng.*, 39(8), 2013.

[70] F. Peters, T. Menzies, and L. Layman. LACE2: better privacy-preserving data sharing for cross project defect prediction. In *Proc. Int'l Conf. on Software Engineering (ICSE'15)*, pages 801–811, 2015.

[71] M. Pighin and R. Zamolo. A predictive metric based on discriminant statistical analysis. In *Proc. Int'l Conf. on Softw. Eng. (ICSE'97)*, pages 262–270, 1997.

[72] F. Rahman, D. Posnett, and P. Devanbu. Recalling the "imprecision" of cross-project defect prediction. In *Proc. Int'l Symposium on the Foundations of Softw. Eng. (FSE'12)*, pages 61:1–61:11, 2012.

[73] F. Rahman, D. Posnett, A. Hindle, E. Barr, and P. Devanbu. BugCache for inspections: Hit or miss? In *Proc. European Softw. Eng. Conf. and Symposium on the Foundations of Softw. Eng. (ESEC/FSE'11)*, pages 322–331, 2011.

[74] RELENG2015. 3rd International Workshop on Release Engineering 2015. http://releng.polymtl.ca/RELENG2015/html/index.html.

[75] C. Rosen, B. Grawi, and E. Shihab. Commit Guru: Analytics and risk prediction of software commits. In *Proc. European Softw. Eng. Conf. and Symposium on the Foundations of Softw. Eng. (ESEC/FSE'15)*, pages 966–969, 2015.

[76] A. Schröter, T. Zimmermann, and A. Zeller. Predicting component failures at design time. In *Proc. Int'l Symposium on Empirical Softw. Eng. (ISESE'06)*, pages 18–27, 2006.

[77] M. Shepperd, Q. Song, Z. Sun, and C. Mair. Data quality: Some comments on the NASA software defect datasets. *IEEE Trans. Softw. Eng.*, 39(9):1208–1215, 2013.

[78] E. Shihab. *An Exploration of Challenges Limiting Pragmatic Software Defect Prediction*. PhD thesis, Queen's University, 2012.

[79] E. Shihab, A. E. Hassan, B. Adams, and Z. M. Jiang. An industrial study on the risk of software changes. In *Proc. Int'l Symposium on Foundations of Softw. Eng. (FSE'12)*, pages 62:1–62:11, 2012.

[80] E. Shihab, A. Ihara, Y. Kamei, W. M. Ibrahim, M. Ohira, B. Adams, A. E. Hassan, and K. Matsumoto. Studying re-opened bugs in open source software. *Empirical Softw. Eng.*, 5(18):1005–1042, 2013.

[81] E. Shihab, Y. Kamei, B. Adams, and A. E. Hassan. Is lines of code a good measure of effort in effort-aware models? *Inf. Softw. Technol.*, 55(11):1981–1993, 2013.

[82] E. Shihab, A. Mockus, Y. Kamei, B. Adams, and A. E. Hassan. High-impact defects: a study of breakage and surprise defects. In *Proc. European Softw. Eng. Conf. and Symposium on the Foundations of Softw. Eng. (ESEC/FSE'11)*, pages 300–310, 2011.

[83] Y. Shin, R. Bell, T. Ostrand, and E. Weyuker. Does calling structure information improve the accuracy of fault prediction? In *Proc. Int'l Working Conf. on Mining Software Repositories (MSR'09)*, pages 61–70, 2009.

[84] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *Proc. Int'l Workshop on Mining Software Repositories (MSR'05)*, pages 1–5, 2005.

[85] R. Subramanyam and M. S. Krishnan. Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. *IEEE Trans. Softw. Eng.*, 29(4):297–310, 2003.

[86] B. Turhan, T. Menzies, A. B. Bener, and J. Di Stefano. On the relative value of cross-company and within-company data for defect prediction. *Empirical Softw. Eng.*, 14:540–578, 2009.

[87] B. Ujhazi, R. Ferenc, D. Poshyvanyk, and T. Gyimothy. New conceptual coupling and cohesion metrics for object-oriented systems. In *Proc. Int'l Working Conf. Source Code Analysis and Manipulation (SCAM'10)*, pages 33–42, 2010.

[88] H. Valdivia Garcia and E. Shihab. Characterizing and predicting blocking bugs in open source projects. In *Proc. Int'l Working Conf. on Mining Software Repositories (MSR'14)*, pages 72–81, 2014.

[89] Wikiquote. Sun Tzu. https://en.wikiquote.org/wiki/Sun_Tzu.

[90] W. E. Wong, J. R. Horgan, M. Syring, W. Zage, and D. Zage. Applying design metrics to predict fault-proneness: a case study on a large-scale software system. *Software: Practice and Experience*, 30(14), 2000.

[91] F. Zhang, A. Mockus, I. Keivanloo, and Y. Zou. Towards building a universal defect prediction model. In *Proc. Int'l Working Conf. on Mining Software Repositories (MSR'14)*, pages 182–191, 2014.

[92] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. A. Vouk. On the value of static analysis for fault detection in software. *IEEE Trans. Softw. Eng.*, 32:240–253, 2006.

[93] T. Zimmermann and N. Nagappan. Predicting subsystem failures using dependency graph complexities. In *Proc. Int'l Symposium on Software Reliability (ISSRE'07)*, pages 227 –236, 2007.

[94] 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. European Softw. Eng. Conf. and Symposium on the Foundations of Softw. Eng. (ESEC/FSE'09)*, pages 91–100, 2009.

[95] T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for Eclipse. In *Proc. Int'l Workshop on Predictor Models in Softw. Eng. (PROMISE'07)*, pages 1–7, 2007.