Predicting Method Crashes with Bytecode Operations

Sunghun Kim
Hong Kong University of
Science and Technology
hunkim@cse.ust.hk

Thomas Zimmermann
Microsoft Research
Redmond, USA
tzimmer@microsoft.com

Rahul Premraj
VU University, Amsterdam
The Netherlands
rpremraj@cs.vu.nl

Nicolas Bettenburg
Queen's University
Kingston, Canada
nicbet@cs.queensu.ca

Shivkumar Shivaji
University of California,
Santa Cruz, USA
shiv@soe.ucsc.edu

ABSTRACT

Software monitoring systems have high performance overhead because they typically monitor all processes of the running program. For example, to capture and replay crashes, most current systems monitor all methods; thus yielding a significant performance overhead. Lowering the number of methods being monitored to a smaller subset can dramatically reduce this overhead. We present an approach that can help arrive at such a subset by reliably identifying methods that are the most likely candidates to crash in a future execution of the software. Our approach involves learning patterns from features of methods that previously crashed to classify new methods as crash-prone or non-crash-prone. An evaluation of our approach on two large open source projects, ASPECTJ and ECLIPSE, shows that we can correctly classify crash-prone methods with an accuracy of 80–86%. Notably, we found that the classification models can also be used for cross-project prediction with virtually no loss in classification accuracy. In a further experiment, we demonstrate how a monitoring tool, RECRASH could take advantage of only monitoring crash-prone methods and thereby, reduce its performance overhead and maintain its ability to perform its intended tasks.

Categories and Subject Descriptors:
D2.5 [*Software Engineering*]: Testing and Debugging; D2.7 [*Software Engineering*]: Distribution, Maintenance, and Enhancement; D2.8 [*Software Engineering*]: Metrics

General Terms: Measurement, Reliability, Experimentation

1. INTRODUCTION

> *"When you have a million users, it is amazing what will crash [. . .]" – Joel Spolsky [46]*

Imagine we were able to prevent crimes by predicting them in advance — in the science-fiction short story *Minority Report* [14], a special police department called *Precrime* uses this idea to identify potential suspects and closely monitor their behavior in order to prevent impending crimes. This has been the driving force behind the line of research that we present in this paper: unexpected behaviour and crashes in software systems can be similarly avoided or captured by monitoring the behaviour of constituent methods and modules.

Many capture and replay techniques exist that monitor software systems to perform a variety of tasks, such as locating failures [11, 18, 38], reproducing crashes [5, 10], and profiling executions [22, 39, 47]. However, most of these monitoring tools are impractical in deployment due to their significant performance overhead. For example, Valgrind's slow down factor is more than 30 times [39]. Even state-of-the-art lightweight monitoring approaches such as RECRASH have a performance overhead of up to 60% [5]. The main cause for such high performance overhead is the vast number of subjects being monitored. Most approaches monitor all processes of the running program as there is currently no way to foretell which methods are likely to crash. The parallel in Minority Report would be that *Precrime* tries to prevent crime by putting every citizen of the nation under surveillance.

In this paper, we investigate whether methods that run a high risk to crash (henceforth referred to as *crash-prone* methods) can be predicted in advance. If crash-prone methods can be identified before the crash occurs, existing capture-based techniques can be adapted to monitor only parts of a software system. In practice, this will mean that only high risk methods will undergo byte code instrumentation. Resulting key benefits from such adaptations include reductions in monitoring overhead.

Our approach predicts crash-prone methods by extracting bytecode features, which includes *operation codes*, or opcodes, in order to represent the execution sequence of instructions from the bytecode of the method bodies. The intuition is that certain bytecode sequences are more likely to lead to crashes than others. We then train a classifier to learn patterns from sequential opcodes, in order to classify methods as crash-prone or non-crash-prone (i.e., less likely to crash). A classifier can be simply described as a mathematical function that assigns a label (e.g., crash-prone) to an instance based on its set of features as inputs. We empirically validate our approach on two large open-source projects and achieve a classification accuracy of 80–86%.

At this stage, it is important to note that while our approach appears similar to that adopted for classifying defect-prone entities, we solve a different problem: instead of defects, we predict crashes. *Crashes are effects of defects* — a defect in method `foo` may lead to a crash in the same method or another method `bar`. Also, not all defects manifest themselves as crashes. Crashes are also more similar to each other in that our analysis revealed 35-40% crashes are `NullPointerExceptions`, while defects are more diverse and often unique. Hence, existing defect prediction models may not

ISEC'13, Feb. 21-23, 2013 New Delhi, India
Copyright © 2013 ACM ISBN yyy-y-yyyy-yyyy-y/02/13 ...$10.00.

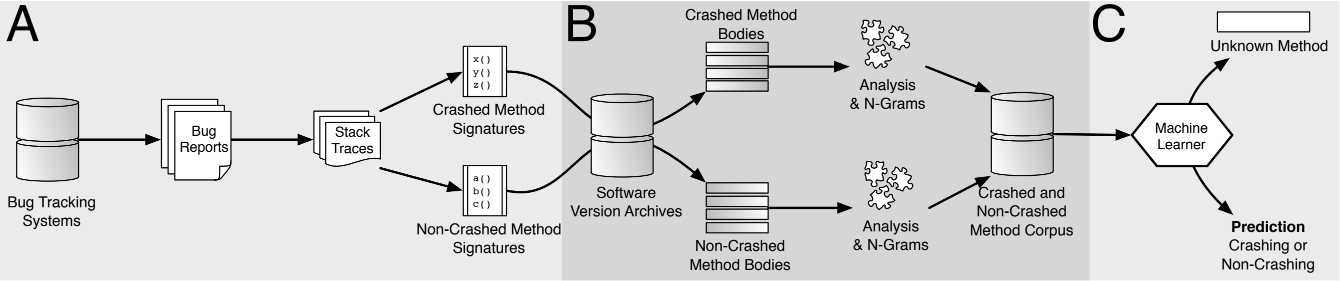


Figure 1: Summary of our approach: (A) Collection of historical crash data. (B) Extraction of machine learning features from bytecode. (C) Building a machine learner and classifying unseen methods into crash-prone or not.

perform well in identifying crash-prone methods. This calls for a fresh approach to identify crash-prone methods, which is the main contribution of this work. Rather than looking at the complexity of code, we believe that for crash prediction one should focus on what the code does and what operations it executes. While most defect prediction models are built on top of source code (because this is the unit where developers fix bugs), we instead analyze bytecode to predict crashes. This has the advantage that we can extent predictions beyond in-house code, e.g., for third-party libraries, and potentially replay crashes that involve those.

We also demonstrate the benefits of our approach by comparing the run-time overhead when running RECRASH [5] (a tool that monitors the program's execution to reproduce a failure) first, monitoring *all* methods to simulate classical approaches, and second, when monitoring *only methods predicted as crash-prone*. We observed a noticeable decrease in the run-time overhead of RECRASH at virtually no cost to its performance (crash reproduction rate).

We summarize the main contributions of our work as follows:

1. Analysis of the distribution of crashed methods and exceptions thrown from two large open-source projects (Section 2).
2. Novel use of opcodes to train a classifier to identify crash-prone methods; classifiers built with opcodes perform better than classifiers built with complexity metrics for predictions within projects (Sections 3, 4, and 5).
3. Demonstration that monitoring only crash-prone methods can reduce performance overhead of monitoring tools such as RECRASH at virtually no cost to their utility (Section 6).

We close this paper with a discussion of related work (Section 7), threats to validity (Section 8), and the consequences and conclusions that can be drawn from this research (Section 9).

2. CRASHED METHODS

Figure 1 presents an overview of the approach that we have adopted for our research. The first step is to identify methods that are known to have crashed in the past (Figure 1, part A). This section provides the details on how we identified the methods and presents the summary statistics of data collected from our subject projects.

Methods do crash when they cannot fulfill required specifications due to errors. Crashes manifest themselves as thrown exceptions—a functionality in many modern programming languages for error handling and its separation from method logic. When an exception is encountered, developers can access all active subroutine calls of the program as a report that is referred to as a *stack trace*. Figure 2 shows a sample stack trace from ASPECTJ. Typically, stack traces have the following parts:

1. The *exception, error,* or *assertion* that has been observed.
2. An optional exception or error *message*.
3. A *list of methods* that were active on the *call stack* when the exception occurred.

As can be seen in Figure 2, the information in a stack trace points to methods that are potential sources of error. It is no wonder that they are much sought after by developers during bug fixing [6]. Often, stack traces can be found in bug reports filed by reporters along with the bug's description. We used such bug reports as the primary source of data for our research.

2.1 Subject Projects

For our study, we use the following two open-source projects:

1. **ASPECTJ:** An aspect-oriented extension to the Java programming language that facilitates modularization of crosscutting concerns. Among other tools, it includes a compiler that weaves together existing program code with aspect code.
2. **ECLIPSE:** A large open-source integrated-development environment developed by IBM in 2001 and actively maintained since then.

We considered all bug reports from beginning (September 2002) to July 2008, for ASPECTJ that totalled to 1,885 bug reports; and from beginning (October 2001) to July 2008, for the ECLIPSE project that totaled to 41,999 bug reports.

2.2 InfoZilla

In order to extract stack trace information from bug reports (including their discussions and attachments) from the subject projects, we use the InfoZilla tool [7]. InfoZilla uses a set of regular expressions based on the following model to identify and extract stack traces:

```
( [MODIFIER]?[EXCEPTION]) ([:] [MESSAGE])?
( [at][METHOD][(] [FILE][:] [LINE][)] )*
```

Manual inspection of the extracted stack traces in earlier work [7] showed that InfoZilla can perform reliably with an accuracy of 98.5%. The output of the tool provides us with knowledge about the type of exception that was thrown when the program error occurred, the list of subroutines that were active when the crash occurred, the name and path of the source code files containing the respective methods, and the location of the methods' source code in that file.

A total of 34,970 stack traces were collected from the 43,884 bug reports of both projects. We then filtered duplicated or non-ASPECTJ/non-ECLIPSE stack traces:

Table 1: Distribution of top-5 exceptions in ASPECTJ and ECLIPSE.

Project	Observed Frequency		Exception thrown
ASPECTJ (366)	166	(45.36%)	`lang.NullPointerException`
	50	(13.66%)	`aspectj.weaver.BCException`
	35	(9.56%)	`lang.ClassCastException`
	27	(7.38%)	`lang.IllegalStateException`
	21	(5.34%)	`lang.ArrayIndexOutOfBoundsException`
ECLIPSE (17,910)	6,932	(38.70%)	`java.lang.NullPointerException`
	1,183	(6.61%)	`java.lang.IllegalArgumentException`
	952	(5.31%)	`org.eclipse.swt.SWTException`
	948	(5.29%)	`java.lang.ClassCastException`
	582	(3.25%)	`java.lang.ArrayIndexOutOfBoundsException`

1. *Remove duplicate stack traces:* For the purpose of this study, we adopted a definition for duplicate stack traces which is different from that of developers. The reason is that we predict crashes, i.e., the *effects* of defects, and some defects result in crashes at multiple locations. Developers can decide that two stack traces are duplicates because the comprising bug reports are marked as duplicates (=the same defect, will be fixed only once). In contrast, for our purpose we have to instrument every crash location, even if it is the same defect. Thus we considered only those stack traces that contained exactly the same exceptions and order of crashed methods in the stack frames as duplicates (=the same effect). Removing such duplicate stack traces left us with 23,435 unique stack traces.

2. *Remove non-ASPECTJ and non-ECLIPSE stack traces:* Although the bug reports used to extracted these stack traces were filed in the bug databases of the two projects, many reported crashes were caused by external API and not the projects themselves, such as crashes produced by the JAVA virtual machine. To consider only ASPECTJ and ECLIPSE related crashes, we further filtered the 23,435 stack traces to include only those beginning with either `org.eclipse.*` or `org.aspectj.*`. This left us with 17,910 stack traces for ECLIPSE and 366 for ASPECTJ, which we used for the remainder of our study.

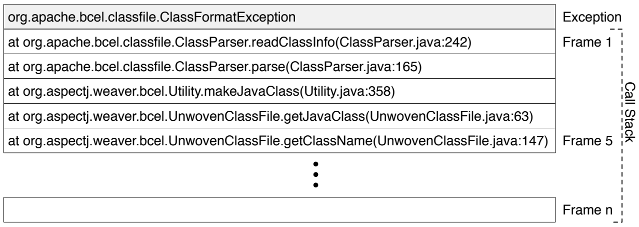
2.3 Identifying Methods that Crashed

A stack trace includes multiple frames (see Figure 2), each containing information on the method such as the file name, method name and line number. We consider the first (top-most) frame in the stack as the crashed method because the exception/crash occurred in this method. [1] To exemplify, in the stack trace presented in Figure 2, the `readClassInfo()` method, defined in `ClassParser.java`, crashed when executing the source code at line `242`.

In total, we identified 209 unique methods that crashed in ASPECTJ and 8,621 unique methods in ECLIPSE. We consider these methods as crash-prone methods because the exception occurred in these methods. More formally, for the experiments in this paper, we define a method as "crash-prone" if it crashed at least once in our dataset.

3. OUR APPROACH

Having identified crashed methods from the two projects, we now continue by retrieving the method bodies from their respective version archives. We then extract features from their bytecode presentation that will constitute the data to train our classifier (Figure 1, part B). This section elaborates on the steps we followed to collect the data needed to perform our experiments.



```
org.apache.bcel.classfile.ClassFormatException
at org.apache.bcel.classfile.ClassParser.readClassInfo(ClassParser.java:242)
at org.apache.bcel.classfile.ClassParser.parse(ClassParser.java:165)
at org.aspectj.weaver.bcel.Utility.makeJavaClass(Utility.java:358)
at org.aspectj.weaver.bcel.UnwovenClassFile.getJavaClass(UnwovenClassFile.java:63)
at org.aspectj.weaver.bcel.UnwovenClassFile.getClassName(UnwovenClassFile.java:147)
```

Figure 2: Stack trace extracted from ASPECTJ bug #59208.

3.1 Method Body Extraction

The information on the crashed methods (file name, method name and line number) in the stack traces allows us to trace back to the method body, and in turn, extract the features to train our classifier. While this task appears trivial at first, we must exert caution to ensure that the features are collected from the correct version of the method, i.e., the one that actually crashed, else we risk training the classifier with incorrect inputs. For example, if a user reports that a method `foo` from ECLIPSE 2.0 (build 40013) crashed, we must strictly extract features from method `foo` in ECLIPSE 2.0 (build 40013).

To match the version of the software, we extracted the value of the version field in the bug report. The relevant method body was then extracted from the project's corresponding version. For example, for a crash reported while using version "1.5.2" in ASPECTJ, we use `aspectj-1.5.2.jar` to extract the method body with the ASM framework [40]. We excluded stack traces found in bug reports with no version numbers reported from our analysis.

The validity of our data was further increased by matching the line numbers from the stack trace to those in the program archives. This was done because JAVA supports method overriding and hence, two or more methods can share the same name leading to ambiguity when matching method names only. Comparing line numbers ensures that the extracted method bodies are the ones that crashed. If a method in a specified version archive is not found, we excluded it from our analysis. We also excluded methods whose line numbers in the stack frame and program archive did not match.

We downloaded ASPECTJ[2] and ECLIPSE[3] versions from their respective home pages to extract the crashed method bodies. In total, 18 versions of ASPECTJ and 25 versions of ECLIPSE were available.

[1] We use the top-most frame because this paper focusses on crash prediction (recall the discussion of differences between defects and crashes in the introduction). The defect that caused the crash can be somewhere below in the stack trace or not be included at all.

[2] `http://www.eclipse.org/aspectj/downloads.php`

[3] `http://www.eclipse.org/eclipse/downloads.php`

```
 0   ldc <Class Analysis>
 2   invokevirtual desiredAssertionStatus() :boolean
 5   ifne 12
 8   iconst_1
 9   goto 13
12   iconst_0
13   putstatic Analysis.assertionsDisabled :boolean
16   return
```

Figure 3: Opcode example from `org.aspectj.Main.class`

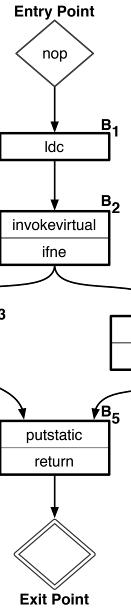


Figure 4: Example basic blocks and control flow graphs of opcodes in Figure 3.

3.2 Feature Extraction

After locating the method bodies of the crashed methods, we extracted features that are available in their bytecode representation using the FindBugs framework [21]. We extract features from the entire method body (and not just the line that crashed) to take into account the surrounding region of the crash. Figure 3 shows the sample bytecode of an initialization method for static fields.

A method's bytecode consists of operation codes (*opcodes*) that represent the execution sequence of instructions. Opcodes have the advantage of representing low-level semantics of the code and the resulting data is likely to contain less noise than that extracted directly from source code. We hypothesize that certain opcode sequences are definite indicators of crash-prone methods.

The FindBugs framework that we used to extract features from the bytecode first performs static analysis to determine *basic blocks* and the control flow among them. Basic blocks are groups of consecutive instructions that can be sequentially executed without halting program execution or a branch. A basic block is entered at its first instruction and is left at the last. After obtaining information about basic blocks, we can trace the control flow of information by inspecting the execution order and branching behavior among the identified blocks.

Figure 4 presents the basic blocks and control flow of information obtained from the opcode sequence example in Figure 3. First, a `nop` instruction is inserted before the first block (B_1) to allow constructing a location representing the entry to the method. Next, we obtain five separate sequences of opcodes that can be executed without halting and branching and group as basic blocks. After determining their execution order and branching behavior, a control flow graph is generated as in Figure 4. Basic blocks form the nodes of the graph. Whenever a block B_2 follows a block B_1 in the execution order of the method, the control flow graph contains a directed edge from B_1 to B_2.

Table 2: Bytecode features extracted for this paper

		Count	
Features	Abbreviation	ASPECTJ	ECLIPSE
Meta data	`m`	7	7
Single opcode	`op1`	92	148
2-grams opcode	`op2`	681	1,938
3-grams opcode	`op3`	864	4,938
First opcode in blocks	`ops`	80	131
Last opcode in blocks	`ope`	58	80
Exceptions thrown	`e`	26	120
Method names	`n`	114	1,597

We consider opcode sequences within a basic block as a series of words and extract n-grams from them. N-grams are essentially subsequences of *n* items in a given sequence. We extract single opcodes (1-gram), two consecutive opcodes (2-grams), and three opcode combinations (3-grams) and use them as features. Using up to three opcode combinations allows us to investigate whether sequences of n-grams are better indicators of crash-prone methods than single opcodes in isolation. As an example, B_2 in Figure 4 has `invokevirtual` and `ifne` opcode. From this block we extract features, `invokevirtual` (single opcode), `ifne` (single opcode), and `invokevirtual ifne` (2-grams. There are no 3-gram features in B_2. In addition, the first and last opcodes of each basic block are used as features.

For the n-gram extraction, it is important to use basic block information instead of the entire list of opcodes. Even though two opcodes are consecutive, there is no guarantee that the two opcodes are executed sequentially. For example, in Figure 3 the opcodes `goto 13` (in Line 9) and `iconst_0` are consecutive, but they will never be executed sequentially. If we just use opcode n-grams without basic block considerations, we may include meaningless opcode combinations.

The attributes of basic blocks are good feature candidates. Basic block metadata such as the number of basic blocks, the number of edges between basic blocks, and size of basic blocks indicate properties that are used as features. Some opcodes can include exception targets such as `athrow`. We used these strings as features. All extracted features are summarized in Table 2.

We also use class and methods names. As Schröter et al. [43] discussed, crashes do not occur uniformly in methods. It is possible that methods in some packages or some classes would crash more (or less) frequently than other methods. Features extracted from method names capture these properties.

In summary, we identify methods as crashed and non-crashed. From each method, we extract features using extraction techniques described in this section. The extracted features along with the labels, crashed or non-crashed compose a corpus. The corpus is used by our experiments to evaluate the prediction model.

4. EXPERIMENTAL SETUP

In this section, we describe the setup adopted to perform our experiments.

4.1 Bayesian Networks

We used Bayesian networks in the Weka implementation [49] to classify methods as crash-prone or non-crash-prone (see Figure 1, part C). Our decision to use Bayesian networks is due to its good performance when dealing with a large number of variables with much variance in values [23]. In our case, we have many opcode n-grams with highly varying values. It is possible that other classifiers may perform as well as Bayesian networks, however investigating the most suitable classifier is beyond the scope of this paper.

4.2 Training and Test Sets

A classifier has to learn from both, positive and negative examples; in our case, crash-prone and non-crash-prone methods respectively. The previous sections elucidate how we identified crash-prone methods as those that are known to have crashed. We consider non-crash-prone methods to be the complement set of crash-prone methods.

To train and test our classifier, we created a corpus of methods for each project using *undersampling*, which is commonly used in the literature [2, 32, 45]; we randomly sampled non-crash-prone methods equal in number to the known crash-prone methods in the project. As a result, our corpora consisted a total of 418 methods for ASPECTJ and 17,242 methods for ECLIPSE.

4.3 Within and Cross-Project Classification

Using this data, we adopted two experimental setups to conduct our experiments:

- **Within-project classification:** In this setup, we used data from the same project to both, train and test the Bayesian network. The setup allows us to investigate how well bytecode features from a project can be used to classify crash-prone and non-crash-prone methods in the same project. We applied 10-fold cross-validation in these experiments. In such a setup, the data is divided equally into 10 folds and the instances in each fold are classified in turn using the remaining nine to train the model. The reported values in table 3 are the average values of accuracy, precision, recall, and F-measure over the 10 runs.

- **Cross-project classification:** In this setup, we trained the Bayesian network with the corpus from one project and tested it on the corpus from the other project. The reported values in table 4 are the resulting accuracy, precision, recall, and F-measure after training on one project and testing on another. Earlier in Section 3, we mentioned that bytecode features have the advantage of not being project-specific; hence patterns indicative of crashes are likely to be generalizable across projects. This setup allows us to investigate whether bytecode features can indeed be used across projects for at least select types of classification purposes.

4.4 Baseline

We compared the classifier built from our bytecode features with a classifier built from complexity metrics commonly used for traditional defect prediction [9]. We computed the metrics directly on the bytecode with the SandMark tool [12, 13]. To ensure comparability of the results, we used the same experimental setup for both complexity metrics and bytecode features, that is, we used the same splits and the same classifier (Bayesian networks).

For the experiments we used the following complexity metrics:

- Size of Method (in Bytes)
- Number of Conditional Statements
- Number of Scalar Locals
- Number of Vector Locals
- Length of Local Identifiers
- McCabe Complexity [31]
- Munson/Khoshgoftaar: Data Structure Complexity [35]
- Harrison/Magel: Nesting Level Complexity [20]
- Halstead complexity measures: Length, Vocabulary, Volume, Difficulty, Effort, Volume + Effort [19]

4.5 Evaluation

Applying a machine learner to our problem can result in four possible outcomes: the classifier predicts (1) a crash-prone method as crash-prone ($cp \rightarrow cp$); (2) a crash-prone method as non-crash-prone ($cp \rightarrow cr$); (3) a non-crash-prone method as crash-prone ($cr \rightarrow cp$); and (4) a non-crash-prone method as non-crash-prone ($cr \rightarrow cr$). These outcomes can be then used to evaluate the classification with the following four measures [2, 27, 44]:

- **Accuracy:** the number of methods correctly classified as crash-prone ($N_{cp\rightarrow cp}$) or non-crash-prone ($N_{cr\rightarrow cr}$) divided by the total number of methods classified. This is a good overall measure of classification performance.

$$Accuracy = \frac{N_{cp\rightarrow cp} + N_{cr\rightarrow cr}}{N_{cp\rightarrow cp} + N_{cp\rightarrow cr} + N_{cr\rightarrow cp} + N_{cr\rightarrow cr}} \tag{1}$$

- **Precision:** the number of methods correctly classified as crash-prone ($N_{cp\rightarrow cp}$) over the number of all methods classified as crash-prone.

$$Precision\ P(cp) = \frac{N_{cp\rightarrow cp}}{N_{cp\rightarrow cp} + N_{cr\rightarrow cp}} \tag{2}$$

- **Recall:** the number of methods correctly classified as crash-prone ($N_{cp\rightarrow cp}$) over the total number of crash-prone.

$$Recall\ R(cp) = \frac{N_{cp\rightarrow cp}}{N_{cp\rightarrow cp} + N_{cp\rightarrow cr}} \tag{3}$$

- **F-measure:** a composite measure of precision *P(cp)* and recall *R(cp)* for crash-prone methods.

$$F\text{-}measure\ F(cp) = \frac{2 * P(cp) * R(cp)}{P(cp) + R(cp)} \tag{4}$$

Of these measures, the F-measure is the most important because it indicates overall classification performance as a single value taking both precision and recall into account. The same measures can be correspondingly defined for non-crash-prone methods.

5. CLASSIFICATION RESULTS

We now turn to discussing the results from both experimental setups in this section. All measures of evaluation are reported in Tables 3 and 4.

5.1 Within-Project Classification

We observed an F-measure of 86.4% for ASPECTJ and 79.5% for ECLIPSE for classifying crash-prone methods; both measures are very high. The precision and recall values for crash-prone and non-crash-prone methods for both projects are also very high (see Table 3). These results strongly suggest that bytecode features can be used to reliably distinguish between the two classes of methods.

Table 3: Classification results from within-project setup (Bayesian networks with K2, 10-fold cross-validation).

			Crash-prone methods			non-crash-prone methods		
Project	Feature Set	Accuracy	Precision	Recall	F-measure	Precision	Recall	F-measure
ASPECTJ	Bytecode features	**0.859**	**0.838**	**0.891**	**0.864**	**0.883**	**0.828**	**0.855**
	Complexity metrics	0.805	0.800	0.813	0.806	0.810	0.797	0.803
ECLIPSE	Bytecode features	**0.799**	**0.812**	**0.778**	**0.795**	**0.787**	**0.819**	**0.803**
	Complexity metrics	0.716	0.714	0.720	0.717	0.718	0.712	0.715

Table 4: Classification results from cross-project setup. (Bayesian networks with K2)

				Crash-prone methods			non-crash-prone methods		
Project	Training Data	Feature Set	Accuracy	Precision	Recall	F-measure	Precision	Recall	F-measure
ASPECTJ	ECLIPSE	Bytecode features	0.750	0.686	**0.922**	0.787	**0.881**	0.578	0.698
		Complexity metrics	**0.796**	**0.779**	0.828	**0.803**	0.817	**0.766**	**0.790**
ECLIPSE	ASPECTJ	Bytecode features	**0.721**	**0.751**	**0.664**	**0.705**	**0.699**	**0.779**	**0.737**
		Complexity metrics	0.713	0.745	0.648	0.693	0.689	0.778	0.731

For both ASPECTJ and ECLIPSE the values of all evaluation measures are lower for complexity metrics than the values for bytecode features (between 3.4 and 9.9 percentage points).

The precision and recall values can be further improved when using the classifier, but it typically involves trading off one for the other, i.e., an increase in precision often results in a decrease in recall and vice versa. The improvement can be made by adjusting the *threshold* value of probability, which ranges from [0–1], in the classifier. This value acts as boundary to determine the membership class of an instance. Typically, the threshold value is set to 0.5 as was the case for the results presented in Table 3. But when the threshold value is increased or decreased, it often results in an improvement in either precision or recall. For example, if the threshold value is set to 0, the recall value of crash-prone methods will increase to 1, while the precision value will fall. On the other hand, if the threshold value is increased to 1, precision may be very high but only at the cost of fall in recall.

In Figure 5, we plot the corresponding precision and recall values by varying the threshold in both projects. Overall, the precision and recall curves display high values; it is possible to achieve values > 85% for both, precision and recall, at certain threshold values. Note that the curve for ASPECTJ is not as smooth in comparison to ECLIPSE because of the relatively fewer number of methods.

Such a graph can be of much help when developing monitoring tools with high performance overhead—in a calibration phase, one can select the ideal threshold value for the project to adjust performance overhead depending on the use-case. For example, if the monitoring tool requires higher precision, then threshold values corresponding to the upper-left portions of the curve are more suitable. In such setups, the performance overhead will decrease because fewer methods will be monitored. Vice versa, if performance overhead is not a major concern, then choosing a threshold value that results in higher recall may be more desirable. This graph can be used to estimate future performance based on past performance.

Overall, the high values for accuracy and F-measure show that our approach is promising for crash-prone method prediction and that it can be reliably used to complement monitoring tools to narrow down the number of methods to monitor.

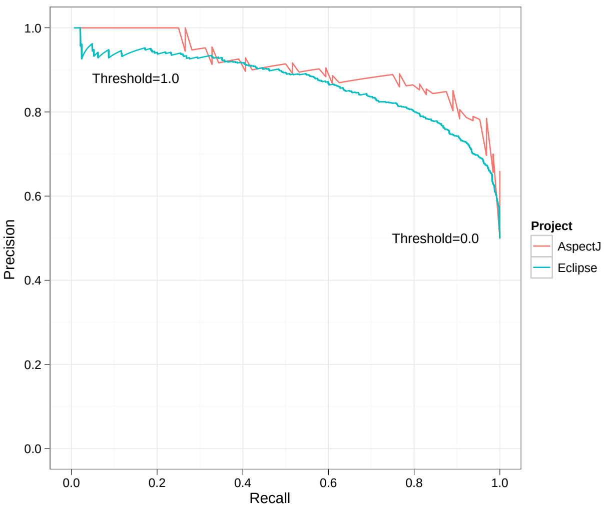


Figure 5: Precision and recall graph of method crash prediction of ASPECTJ and ECLIPSE.

5.2 Cross-Project Classification

Next, we examine how the classifier performs on classifying methods from one project, when trained on methods from another. Usually data in the same project share common characteristics and so, it is reasonable to expect that classification accuracy will be high. However, past research has found classification models to show low performance when reused across different projects. For example, Zimmermann et al. [51] noted that defect prediction models do not yield good results from used across different projects. Also, Kim et al. [27] found that classifying changes as clean or buggy yields high accuracy within the same project, but not across projects. However, as opposed to these very specialized models, we extract features from relatively unified bytecode, hence it may be possible that the classifier may perform well even across projects.

The results from our cross-project experiments are reported in Table 4. We focus first on the values of F-measure since it combines both, precision and recall. The values for crash-prone methods are 78.7% for ASPECTJ and 70.5% for ECLIPSE, which is similar to those in the within-project setup (86.4% and 80.3% respectively). The F-measure value for non-crash-prone methods is also comparable in the case of ECLIPSE, but drops noticeably for ASPECTJ. A closer look at precision and recall values, shows that for non-crash-prone methods the recall dropped significantly from 82.8% within project to 57.8% across projects.

Surprisingly the values for complexity metrics remained fairly stable between the within-project and cross-project setups. For ASPECTJ, complexity metrics performed slightly better than bytecode features (except for recall for crash-prone methods and precision for non-crash-prone methods). This might be due to the fact that complexity features are quite general whereas the bytecode features are more specific. For ECLIPSE however, the bytecode features performed better than complexity metrics.

These results warrant further research in the area of using bytecode features for identifying crash-prone methods and perhaps other topics too. A very encouraging consequence of these results is that projects with limited data of their own can leverage data (bytecode features and complexity) from other projects to address their own concerns without too much loss in prediction quality.

5.3 Significant Features

We have shown that our approach yields high classification accuracy of crash-prone methods in both, within-project and cross-project experimental setups. A natural question that follows is *what are the common and most influential features that determine a method to be crash-prone or non-crash-prone?* This section focuses on answering this question.

Most influential factors can be investigated using the concept of gain ratio [1]. Gain ratio improves upon information gain, which is a well known measure of the degree of influence exerted by the feature to arrive at a classification. However, information gain can be skewed by features that have a large number of distinct values. Gain ratio plays the same role as information gain, but instead provides a normalized measure of a feature's contribution to a classification, and thus avoiding the problem of numerous distinct values.

We investigated our classification model using gain ratio and present the results in Table 5. The table lists opcode sequences with the highest gain ratio, i.e., ones that cast the highest degree of influence for classification. These features provide the most gain with respect to the values of the attribute. The gain is reported in the column *Gain ratio* in the table. The higher the gain ratio, the more importance the opcode pattern has when identifying crash-prone methods.

Some opcode patterns in Table 5 are common between ASPECTJ and ECLIPSE, such as the `aload` opcode that loads a local reference from a local variable [29]. Since `aload` accesses local variables, it has more chance to crash with exceptions such as `NullPointerException`. `invokevirtual` is another common top feature that invokes an instance method. This indicates that methods with that make calls to many other methods are more crash-prone.

Note that while Table 5 suggests that one likely cause of crashes is the existence of an `aload` operation, it is important to avoid any wrong interpretations [50]. The result does *not* mean that developers should not use this opcode (in fact, developers should continue writing code without worrying about bytecode operations). Rather the result has implication on monitoring systems such as RECRASH. It shows that methods with `aload` are more likely to crash, thus monitoring systems should pay more attention to those methods. The approach in this paper provides an automated way to predict crash-prone locations with high accuracy.

5.4 Methods with declared thrown exceptions and Crashes

The JAVA language provides a very extensive set of methods for exception handling. A method can throw exceptions by adding the `throws` declaration to its signature. A common understanding is that if a method with declared thrown exceptions, there is a higher chance it would crash. One may argue that a classifier can simply predict all methods with declared thrown exceptions as crash-prone and yield good results.

Table 5: Top 5 opcodes sequences from ASPECTJ and ECLIPSE that are predictors of crash-prone methods.

Project	Gain ratio	Feature
ASPECTJ	0.26	`op1: aload`
	0.24	`op1: astore`
	0.23	`ope: aload`
	0.22	`ope: ifle`
	0.22	`op1: ifle`
ECLIPSE	0.14	`op3: aload aload iload`
	0.13	`ops: aload`
	0.12	`ops: jsr`
	0.12	`op2: invokevirtual astore`
	0.12	`op2: checkcast aload`

We investigated this issue by analyzing our data to observe the types of methods that crashed. Only 30% of crashed methods from ECLIPSE and 44% from ASPECTJ threw exceptions. The remaining 70% of crashed methods are not declared as throwing exceptions. Most crashes found are runtime exceptions. It is less common to see developers throw runtime exceptions in their programs. This indicates the methods without declared thrown exceptions also have a high chance to crash.

Next, to assess the effect of the throwable-related features on our classifier, we removed all such features including the 'exception thrower' meta data and the opcode 'athrow', and then reevaluated our model. The classification accuracy stood comparable at 84.0% as opposed to 85.9% using all data for ASPECTJ and 78.0% as opposed to 79.9% for ECLIPSE. The higher accuracy for ASPECTJ may be attributed to the small data set. The F-measures also remained comparable for both projects. Overall, these results show that the classifier depends little on features related to 'throwable'.

6. EXPERIMENTS WITH RECRASH

RECRASH is a framework that monitors and reproduces software crashes [5]. It operates in two phases: monitoring phase and test generation phase. In short, the former phase keeps track of all method calls and arguments passed during the execution of a program and stores this information. If a crash occurs, RECRASH uses this information to generate unit tests that can reliably reproduce the crash. The tests then allow developers to investigate the exact cause for the crash and fix it.

In this section, we focus on comparing the performance overhead of running RECRASH in its original form and when strictly monitoring only those methods classified as crash-prone by our classifier. We further check whether monitoring only crash-prone methods reduces the ability of RECRASH to reproduce crashes.

For our experiments with RECRASH, we train our classifier using crash-prone and non-crash-prone methods from ECLIPSE and use it to classify methods from a different project called SVNKit. We chose SVNKit for our experiments because of our previous experience in using it with RECRASH and being able to reproduce three real crashes from its execution [5]. In addition, SVNKit is the only standalone application used in RECRASH [5] while others are a plug-in (JSR), a library (JDT), and a toy program (BST). Note that SVNKit is completely independent of the ECLIPSE project and so, these experiments qualify as cross-project validation.

SVNKit has a total of 2,347 methods of which 625 (27%) were classified as crash-prone by our classifier; the remaining methods

Table 6: User times of SVNKit processes with RECRASH and RECRASH+. Numbers in parenthesis indicate increase in execution time using original time as the baseline.

	User time (in seconds)		
Process	Original	RECRASH	RECRASH+
`checkout`	2.28	2.92 (28%)	2.63 (15%)
`update`	1.15	1.36 (18%)	1.24 (8%)

were classified as non-crash-prone. Having identified crash-prone methods from SVNKit, we use RECRASH to strictly monitor them while executing two SVNKit processes: `checkout` and `update`.

6.1 Performance Overhead

We first present the comparison of performance overhead of RECRASH in its original form and when monitoring crash-prone methods only. For brevity, we henceforth refer to the latter setup as RECRASH+.

In Table 6, we summarize the user times (measured in seconds using the UNIX `time` command) of the two SVNKit processes in three different setups. First, we run the individual processes solely on the system (column *Original* in Table 6), second, we run the processes alongside RECRASH (column *RECRASH*), and lastly, we run the processes alongside RECRASH+ (column *RECRASH+*) and measure the user times of each setup.

Running RECRASH in parallel with `checkout` increased the process's user time by 28% from 2.28 to 2.92 seconds; for `update`, it increased by 18%. In comparison, running RECRASH+ increased the user time of the processes by only 15% and 8% respectively. RECRASH+ significantly reduced the monitoring overhead from 28% to 15%, and 18% to only 8%.

In terms of absolute numbers, the improvements of 0.29 (from 2.92 to 2.63) and 0.12 seconds (from 1.36 to 1.24) may look insignificant. However, when software runs continuously such as long-running desktop and server programs, these improvements make a big difference.

Note that the overhead can be further reduced by changing the threshold value in the classifier to vary the number of monitored methods as previously shown in Figure 5.

In this experiment, the SVNKit commands were executed to check out and update a mock SVN repository[4].

6.2 Reproducing Crashes

We observed a definite reduction in performance overhead when using RECRASH+. However, what remains to be investigated is whether the reduction in overhead affects the ability of RECRASH to reproduce crashes.

In Table 7, we present three crashes from SVNKit that have been identified in previous work [5] and successfully reproduced using RECRASH. The table also lists the methods listed in the stack trace at the time of each crash. RECRASH uses this list of methods and arguments passed to them to reproduce the crash. Note that crashes from other subject programs, i.e., ECLIPSE, JSR, and BST [5] were excluded from our evaluation because (a) only a subset of tests in ECLIPSE and JSR reproduced the crash and (b) BST is a toy subject program with 10 methods and 200 lines of code The last column in Table 7 indicates whether the methods were classified crash-prone by our classifier and if they would be monitored using RECRASH+. Crash-prone methods are indicated using ✔ (monitored using RECRASH+) and non-crash-prone methods are indicated using ✘.

Table 7: Stack trace frames from crashes in SVNKit. Crash-prone methods are indicated using ✔ (monitored using RECRASH+) and non-crash-prone methods are indicated using ✘.

Crash	Methods in stack trace	Crash-prone
Crash 1	`org...SVNCommandLine.getURL()`	✘
	`org...CheckoutCommand.run()`	✔
	`org...SVN.main()`	✔
Crash 2	`com...UserAuthNone.start()`	✔
	`com...Session.connect()`	✔
	`org...SVNJSchConnector.connect()`	✔
	`org...SVNJSchConnector.open()`	✔
	`org...SVNConnection.open()`	✘
	`org...openConnection()`	✔
	`org...testConnection()`	✘
	`org...CheckoutCommand.run()`	✔
	`org...SVN.main()`	✔
Crash 3	`org...InfoCommand.run()`	✔
	`org...SVN.main()`	✔

RECRASH+ monitors 11 (79%) of all 14 methods listed in the table. Even though some methods were not monitored, each of the three crashes could be reproduced because RECRASH+ monitored the *calling* crash-prone methods in the stack traces. For example, for Crash 1, RECRASH+ did not monitor the crashing method `org...getURL()`, but it monitored the calling method `org...run()`. A unit test was generated for the latter which called `org...getURL()` and passed the same argument to it as when the crash occurred, which in turn reproduced the crash in `org...getURL()`. Crashes 2 and 3 were similarly reproduced. However, RECRASH+ may be unable to reproduce the crash when the crashing method is not monitored and the cause behind the crash is non-deterministic.

Despite this limitation, RECRASH+ performed very well: it reduced performance overhead substantially and reproduced all three crashes in SVNKit. Recall that the crash-prone methods in SVNKit were identified by a classifier trained using data from ECLIPSE.

7. RELATED WORK

Our work is closely related to *defect prediction*, which is fairly often studied in software engineering, for example [8,25,37,41]. A study that is close to ours is by Mizuno and Kikuno [34]: they applied a spam filter on source code to classify files as defect-prone. While they focused only on single words, our approach takes sequences of bytecode operations into account and further abstracts specific variables names (through the opcodes). Also crashes are fairly different from regular defects. For example, in ASPECTJ and ECLIPSE the number of crashes in files correlates only very weakly with the number of defects (Spearman correlation values are between 0.10 and 0.20). This suggests that for crash prediction novel models like ours are needed.

Our work is also related to *software reliability* [30,36], which is defined as "the probability of failure-free software operation for a specified period of time in a specified environment" [3]. For our study, failure-free operation means that a software does not crash. For the prediction of software reliability, software measurements are typically used, for example product metrics (in particular complexity), product management metrics, process metrics, fault and failure metrics, and many more [42]. In contrast to these approaches, we use n-grams of bytecode operations and basic meta data extracted by the FindBugs framework.

[4] `http://amock.googlecode.com/svn/trunk/src`

Ganapathi et al. conducted an *empirical study* of crashes in the Windows XP Kernel [16]. They found that most OS crashes are caused by poorly written device drivers. Several reliability monitoring approaches have been proposed that help to collect crashes from the field [17, 28]. In addition, capture and replay techniques can locate failures, reproduce crashes, and monitor the performance of a system [5, 10, 11, 18, 22, 24, 38, 47]. However, they typically require a large runtime overhead and thus are too expensive to be used. With the approach presented in this paper, capture and replay can be focused on only crash-prone methods and hence reduce the runtime overhead.

Previously, some static analysis techniques have been proposed to predict *crash prone methods*. For example, JSR308 [15] tries to prevent `NullPointerExceptions` using static analysis and user annotations. However, such techniques are limited to predict only specific types of exceptions and/or usually yield high false positive rates. For example, all object references are potential `Null` pointer exception points, but most of them do not throw an exception. In contrast, our approach learns from features of methods that previously crashed to classify new methods as likely to crash or not. Information from previously crashed methods is collected from stack traces posted in bug reports and source code in projects' archives. Kim et al. proposed a prediction model to prioritize crashes using machine-learning techniques [26]. The focus of the earlier work [26] is on crash triage, i.e., deciding which crashes should be fixed. In contrast, the focus of our paper is on predicting crash-prone methods.

8. THREATS TO VALIDITY

We identified the following threats to validity to our work:

- *Integrity of stack traces.* Stack traces in bug reports are copied and pasted by the bug reporter. Occasional clerical errors may lead to incorrect stack traces in the report. However, we expect such errors to be negligible in number and have virtually no impact on our results.

- *Projects may not be representative.* Two projects were used in this paper which were chosen because we had access to their bug reports from which the stack traces were extracted. However, these projects are not representative of all software systems and hence, we cannot currently generalize the results of our study across all projects.

- *Projects are open source.* Both projects used are open source. So they might not be representative of closed-source projects. Having said that, it is important to note that the analyzed projects have substantial industrial participation too.

- *Incomplete crash data.* We could only collect stack traces from bug reports. It is possible that other methods crashed too, but they were not reported. Thus, our analysis does not treat them as crash-prone methods.

- *Selecting non-crash-prone methods.* Methods that are not reported to have crashed in the analyzed bug reports were treated as non-crash-prone. But we cannot guarantee that these methods will not crash in the future. This could make our corpus noisy and affect the classification accuracy.

9. CONSEQUENCES AND CONCLUSIONS

Capture and replay techniques are very useful to prevent or reproduce failures, but they have typically associated with high performance overhead. In this work, we show that this problem can be addressed leveraging known crashes in the project to identify crash-prone methods and monitor only them, and not the entire system.

Our results have shown that monitoring crash-prone methods reduced the performance overhead of RECRASH significantly at almost no cost to its original objectives, reproducing crashes. Hence, one of the main consequences of this work is giving a new direction to design of capture and replay tools that leverage a project's history to intelligently monitor running processes. We believe our approach can be used with most capture and replay techniques, substantially reducing their overhead.

To the best of our knowledge, our research is also the first to use a project's history to identify crash-prone methods. We used opcodes extracted from the bytecode to train a classifier to classify the methods. We also demonstrated that the use of opcodes for such tasks can allow classification models to be reliably used across projects. We expect that future approaches will see software crash history not only as information for debugging, but also as a source to predict future crashes.

In the future, we plan to expand our analysis to other projects and analyze crash history from automated crash report systems and identifying common features of methods that crashed. In its current form our technique is most useful for analyses that consider methods in isolation, in future work we plan to investigate interactions between methods. Furthermore we plan to use crash reports in automated crash report systems such as Dr. Watson [33], Apple crash reporter [4], and Talkback [48] to mark crash prone methods, since automated crash reports are more reliable and complete than manual reports.

10. REFERENCES

[1] E. Alpaydin. *Introduction to machine learning*. MIT press, 2004.

[2] E. Alpaydin. *Introduction to Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2004.

[3] ANSI/IEEE. Standard glossary of software engineering terminology. STD-729-199, 1996.

[4] Apple Crash Reporter, 2007. http://developer.apple.com/technotes/ tn2004/tn2123.html.

[5] S. Artzi, S. Kim, and M. D. Ernst. Recrash: Making software failures reproducible by preserving object states. In *ECOOP '08*, pages 542–565. Springer-Verlag, 2008.

[6] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann. What makes a good bug report? In *Proceedings of the 16th International Symposium on Foundations of Software Engineering*, November 2008.

[7] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim. Extracting structural information from bug reports. In *MSR '08: Proceedings of the Fifth International Workshop on Mining Software Repositories*, May 2008.

[8] G. D. Boetticher, T. Menzies, T. Ostrand, and G. H. Ruhe, editors. *Proceedings of the 4th International Workshop on Predictor Models in SE (PROMISE 2008)*, 2008.

[9] C. Catala and B. Diri. A systematic review of software fault prediction studies. *Expert Systems with Applications*, 36(4):7346–7354, May 2009.

[10] J.-D. Choi and H. Srinivasan. Deterministic replay of Java multithreaded applications. In *SPDT '98: Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, pages 48–59, 1998.

[11] J. Clause and A. Orso. A Technique for Enabling and Supporting Debugging of Field Failures. In *Proceedings of the 29th IEEE and ACM SIGSOFT International Conference*

on Software Engineering (ICSE 2007), pages 261–270, Minneapolis, Minnesota, May 2007.

[12] C. Collberg. Sandmark: A tool for the study of software protection algorithms. http://sandmark.cs.arizona.edu/.

[13] C. Collberg, G. Myles, and M. Stepp. An empirical study of java bytecode programs. *Software: Practice and Experience*, 37(6):581–641, 2007.

[14] P. K. Dick. *Minority Report*. Science fiction short stories. Gollancz, New York, NY, USA, 2002.

[15] M. D. Ernst and D. Coward. JSR 308: Annotations on Java types. `http://pag.csail.mit.edu/jsr308/`, Oct. 17, 2006.

[16] A. Ganapathi, V. Ganapathi, and D. A. Patterson. Windows xp kernel crash analysis. In *Proceedings of the 20th Conference on Systems Administration (LISA)*, pages 149–159. USENIX, 2006.

[17] A. Ganapathi and D. Patterson. Crash data collection: a windows case study. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 280–285, 2005.

[18] D. Geels, G. Altekar, S. Shenker, and I. Stoica. Replay debugging for distributed applications. In *USENIX-ATC'06*, pages 27–27, 2006.

[19] M. H. Halstead. *Elements of Software Science*. Operating and programming systems series. Elsevier Science Inc., New York, NY, USA, 1977.

[20] W. A. Harrison and K. I. Magel. A complexity measure based on nesting level. *SIGPLAN Notices*, 16:63–74, March 1981.

[21] D. Hovemeyer and W. Pugh. Finding bugs is easy. In *OOPSLA '04*, pages 132–136. ACM, 2004.

[22] Java Profiler - JProfiler, 2007. http://www.ej-technologies.com/products/jprofiler/overview.html.

[23] F. Jensen and T. Nielsen. *Bayesian networks and decision graphs*. ASA, 2001.

[24] S. Joshi and A. Orso. SCARPE: A Technique and Tool for Selective Record and Replay of Program Executions. In *Proceedings of the International Conference on Software Maintenance (ICSM 2007)*, Paris, France, October 2007.

[25] T. M. Khoshgoftaar and E. B. Allen. Ordering fault-prone software modules. *Software Quality Control*, 11(1):19–37, 2003.

[26] D. Kim, X. Wang, S. Kim, A. Zeller, S. C. Cheung, and S. Park. Which crashes should i fix first?: Predicting top crashes at an early stage to prioritize debugging efforts. *IEEE Trans. Softw. Eng.*, 37:430–447, May 2011.

[27] S. Kim, J. E. James Whitehead, and Y. Zhang. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering*, 34(2):181–196, 2008.

[28] P. L. Li, M. Ni, S. Xue, J. P. Mullally, M. Garzia, and M. Khambatti. Reliability assessment of mass-market software: Insights from windows vista®. In *ISSRE '08*, pages 265–270, 2008.

[29] T. Lindholm and F. Yellin. *The Java(TM) Virtual Machine Specification (2nd Edition)*. Prentice Hall PTR, April 1999.

[30] M. R. Lyu, editor. *Handbook of Software Reliability Engineering*. McGraw-Hill, 1995.

[31] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, 1976.

[32] T. Menzies, B. Turhan, A. Bener, G. Gay, B. Cukic, and Y. Jiang. Implications of ceiling effects in defect predictors. In *Proceedings of International Workshop on Predictor Models in Software Engineering*, pages 47–54, 2008.

[33] Microsoft Online Crash Analysis. `http://oca.microsoft.com`. Last accessed 2008-01-16.

[34] O. Mizuno and T. Kikuno. Training on errors experiment to detect fault-prone software modules by spam filter. In *ESEC-FSE '07*, pages 405–414. ACM, 2007.

[35] J. C. Munson and T. M. Khoshgoftaar. Measurement of data structure complexity. *Journal of Systems and Software*, 20:217–225, March 1993.

[36] J. D. Musa. *Software Reliability Engineering*. Osborne/McGraw-Hill, 1998.

[37] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 452–461, New York, NY, USA, 2006.

[38] S. Narayanasamy, G. Pokam, and B. Calder. BugNet: Continuously recording program execution for deterministic replay debugging. In *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture*, pages 284–295, 2005.

[39] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Notices*, 42(6):89–100, 2007.

[40] ObjectWeb Consortium. ASM - Home Page. `http://asm.objectweb.org/`. Last accessed 2008-01-16.

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

[42] Reliability Analysis Center. Introduction to software reliability: A state of the art review. Reliability Analysis Center (RAC), 1996.

[43] A. Schröter, N. Bettenburg, and R. Premraj. Do stack traces help developers fix bugs? In *MSR'10: Proceedings of the 7th International Working Conference on Mining Software Repositories*, pages 118–121, 2010.

[44] S. Scott and S. Matwin. Feature engineering for text classification. In *Proc. 16th International Conf. on Machine Learning*, pages 379–388, 1999.

[45] Y. Shin, A. Meneely, L. Williams, and J. Osborne. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *Software Engineering, IEEE Transactions on*, 37(6):772 –787, nov.-dec. 2011.

[46] J. Spolsky. *Joel on Software: And on Diverse and Occasionally Related Matters That Will Prove of Interest to Software Developers, Designers, and Managers, and to Those . . . Ill-Luck, Work with Them in Some Capacity*, chapter Hard-assed Bug Fixin'. APress, US, 2004.

[47] J. Steven, P. Chandra, B. Fleck, and A. Podgurski. jRapture: A capture/replay tool for observation-based testing. In *ISSTA '00*, pages 158–167, 2000.

[48] Talkback Reports. http://talkback-public.mozilla.org. Last accessed 2008-01-24.

[49] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kaufmann, 2nd edition, 2005.

[50] A. Zeller, T. Zimmermann, and C. Bird. Failure is a four-letter word: A parody in empirical research. In *Proceedings of the 7th International Conference on Predictive Models in Software Engineering*, September 2011.

[51] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy. Cross-project defect prediction. In *ESEC/FSE '09*, pages 91–100. ACM, 2009.