Identifying Linux Bug Fixing Patches

Yuan Tian[1], Julia Lawall[2], and David Lo[1]
[1]*Singapore Management University, Singapore*
[2]*INRIA/LIP6-Regal, France*
{*yuan.tian.2012,davidlo*}*@smu.edu.sg, Julia.Lawall@lip6.fr*

***Abstract*—In the evolution of an operating system there is a continuing tension between the need to develop and test new features, and the need to provide a stable and secure execution environment to users. A compromise, adopted by the developers of the Linux kernel, is to release new versions, including bug fixes and new features, frequently, while maintaining some older "longterm" versions. This strategy raises the problem of how to identify bug fixing patches that are submitted to the current version but should be applied to the longterm versions as well. The current approach is to rely on the individual subsystem maintainers to forward patches that seem relevant to the maintainers of the longterm kernels. The reactivity and diligence of the maintainers, however, varies, and thus many important patches could be missed by this approach.**

In this paper, we propose an approach that automatically identifies bug fixing patches based on the changes and commit messages recorded in code repositories. We compare our approach with the keyword-based approach for identifying bug-fixing patches used in the literature, in the context of the Linux kernel. The results show that our approach can achieve a 53.19% improvement in recall as compared to keyword-based approaches, with similar precision.

I. Introduction

For an operating system, reliability and continuous evolution to support new features are two key criteria governing its success. However, achieving one is likely to adversely affect the other, as supporting new features entails adding new code, which can introduce bugs. In the context of Linux development, these issues are resolved by regularly releasing versions that include new features, while periodically designating some versions for longterm support. Development is carried out on the most recent version, and relevant bug fixes are backported to the longterm code.

A critical element of the maintenance of the longterm versions is thus the identification of bug fixing patches. In the Linux development process, contributors submit patches to subsystem maintainers, who approve the submissions and initiate the process of integrating the patch into the coming release. A maintainer may also forward the patch to the maintainers of the longterm versions, if the patch satisfies various guidelines, such as fixing a real bug, and making only a few changes to the code. This process, however, puts an extra burden on the subsystem maintainers, implying that necessary bug fixing patches could be missed. Thus, a technique that automatically labels a commit as a bug fixing patch would be valuable.

In the literature, there are furthermore many studies that rely on identifying links between commits and bugs. These include work on empirical study of software changes [21], [29], bug prediction [14], [22], and bug localization [17], [20], [27]. All of these studies employ a *keyword-based* approach to infer commits that correspond to bug fixes, typically relying on the occurrence of keywords such as "bug" or "fix" in the commit log. Some studies also try to link software repositories with a Bugzilla by the detection of a Bugzilla number in the commit log. Unfortunately these approaches are not sufficient for our setting because:

1) Not all bug fixing commit messages include the words "bug" or "fix"; indeed, commit messages are written by the initial contributor of a patch, and there are few guidelines as to their contents.
2) Linux development is mostly oriented around mailing lists, and thus many bugs are found and resolved without passing through Bugzilla.

A similar observation was made by Bird et al. [4], who performed an empirical study that showed bias could be introduced due to *missing* linkages between commits and bugs.

In view of the above limitations, there is a need for a more refined approach to automatically identify bug fixing patches. In this work, we perform a dedicated study on bug fixing patch identification in the context of the Linux kernel. The results of our study can also potentially benefit studies that require the identification of bug fixes from commits. We propose a combination of text analysis of the commit log and code analysis of the change description to identify bug fixing patches. We use an analysis *plus* classification framework which consists of:

1) The extraction of basic "facts" from the text and code that are then composed into features.
2) The learning of an appropriate model using machine learning and its application to the detection of bug fixing commits.

In a typical classification task, appropriately labeled training dataset is available. However this is not the case in our setting. For positive data, *i.e.*, bug fixing patches, we can use the patches that have been applied to previous Linux longterm versions, as well as patches that have been developed based on the results of bug-finding tools. There is, however, no corresponding set of independently labeled

978-1-4673-1067-3/12/$31.00 © 2012 IEEE

negative data, *i.e.*, non bug fixing patches. To address this problem, we propose a new approach that integrates ranking and classification.

We have tested our approach on commits from the Linux kernel code repository, and compare our results with those of the keyword-based approach employed in the literature. We can achieve similar precision with improved recall; our approach's precision and recall are 0.537 and 0.900 while those of the keyword based approach are 0.519 and 0.588. Our contributions are as follows:

1) We identify the new problem of finding bug fixing patches to be integrated into a Linux "longterm" release.
2) We propose a new approach to identifying bug fixing patches by leveraging both textual and code features. We also develop a suitable machine-learning approach that performs ranking and classification to address the problem of unavailability of a clean negative dataset (i.e., non bug-fixing patches).
3) We have evaluated our approach on commits in Linux and show that our approach can improve on the keyword-based approach by up to 53.19% in terms of recall while maintaining similar precision.

Section II provides more information on Linux longterm and stable kernels. Section III gives an overview of our bug fixing patch identification framework. Section IV describes the data acquisition and feature extraction processes. Section V describes the learning and application of discriminative models to detect bug fixing patches. Our experiments are discussed in Section VI. We discuss interesting issues in Section VII. Related work is presented in Section VIII. We conclude and discuss future work in Section IX.

II. Linux Longterm and Stable Kernels

The Linux operating system is widely used across the computing spectrum, from embedded systems, to desktop machines, to servers. From its first release in 1994 until the release of Linux 2.6.0 in 2003, two versions of the Linux kernel were essentially maintained in parallel: stable versions for users, receiving only bug-fixing patches over a number of years, and development versions, for developers only, receiving both bug fixes and new features. Since the release of Linux 2.6.0, there has been only a single version, which we refer to as the *mainline* kernel, targeting both users and developers, which includes both bug fixes and new features as they become available. Since 2005, the rate of these releases has been roughly one every three months.

The current frequent release model is an advantage for both Linux developers and Linux users because new features become quickly available and can be tested by the community. Nevertheless, some kinds of users value stability over support for new functionalities. Nontechnical users may prefer to avoid frequent changes in their working environment, while companies may have a substantial investment in software that is tuned for the properties of a specific kernel, and may require the degree of security and reliability that a well-tested kernel provides. Accordingly, Linux distributions often do not include the latest kernel version. For example, the current stable Debian distribution (squeeze) and the current Ubuntu Long Term Support distribution (lucid) rely on the Linux 2.6.32 kernel, released in December 2009. For industrial users, the same kernel is at the basis of Suse Enterprise Linux, Red Hat Enterprise Linux and Oracle Unbreakable Linux.

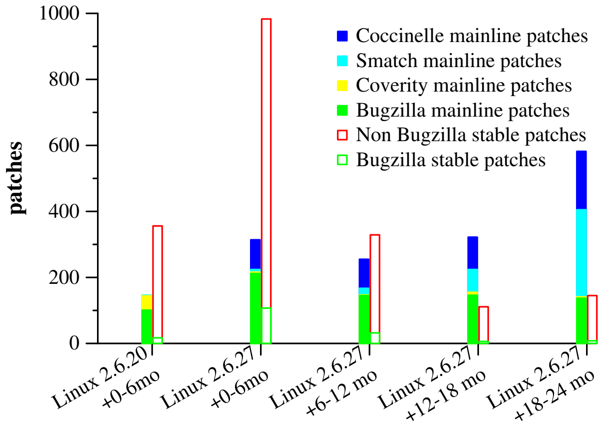

Figure 1. Various kinds of patches applied to the stable kernels 2.6.20 and 2.6.27 and to the mainline kernel in the same time period

In recognition of the need for a stable kernel, the Linux development community maintains a "stable" kernel in parallel with the development of the next version, and a number of "longterm" kernels that are maintained over a number of years. For simplicity, in the rest of this paper, we refer to all of these as stable kernels. Stable kernels only integrate patches that represent bug fixes or new device identifiers, but no large changes or additions of new functionalities.[1] Such a strategy is possible because each patch is required to perform only one kind of change.[2] Developers and maintainers may identify patches that should be included in the stable kernels by forwarding the patches to a dedicated e-mail address. These patches are then reviewed by the maintainers of the stable kernels before being integrated into the code base.

Figure 1 compares a very conservative approximation of the number of bug-fixing patches accepted into the mainline kernel (left, solid bars) with the number of patches accepted into the stable kernels Linux 2.6.20, maintained between February 2007 and August 2007, and Linux 2.6.27, maintained between October 2008 and December 2010 (right, open bars). Bug-fixing patches are approximated as those where the log message mentions a bug-finding tool (Coccinelle [23], Smatch,[3] or Coverity[4]) or where it mentions

[1] linux-2.6.39/Documentation/stable_kernel_rules.txt
[2] linux-2.6.39/Documentation/SubmittingPatches.txt
[3] http://smatch.sourceforge.net/
[4] http://www.coverity.com/

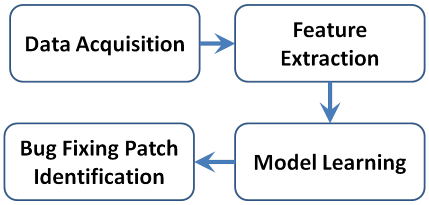

Figure 2. Overall Framework

Bugzilla. Stable patches are separated into those that do not mention Bugzilla and those that do. In any of the considered 6-month periods, at least twice as many patches mentioning Bugzilla were integrated into the mainline kernel as into the stable kernel. And after 1 year, the number of patches based on bug-finding tools in the mainline kernel outstripped the number of patches integrated into the stable kernel. Indeed, fewer than 5 patches based on each of the considered bug-finding tools were integrated into the stable kernel in each of the considered time periods. While it is ultimately the stable kernel maintainers who decide whether it is worth including a bug-fixing patch in a stable kernel, the very low rate of propagation of the considered types of bug-fixing patches from the mainline kernel to the stable kernels suggests that automatic identification of bug-fixing patches could be useful.

III. Overall Framework

Our approach is composed of the following steps: data acquisition, feature extraction, model learning, and bug-fixing patch identification. These steps are shown in Figure 2.

The data acquisition step extracts commits from Linux code repository. Some of these commits represent bug fixing patches while others do not. Not all bug fixing patches are well marked in Linux code. Furthermore, many of these bug fixes are not recorded in Bugzilla. Thus, they are simply hidden in the mass of other commits that do not perform bug fixing. Non bug fixing commits may perform code cleaning, feature addition, performance enhancement, etc.

The feature extraction component then reduces the dataset into some potentially important facets. Each commit contains a textual description along with code elements that are changed by the commit. The textual description can provide hints whether a particular commit is fixing a bugs or is it only trying to clean up some bad coding style or poor programming practice. Code features can also help identify the kind of patch. Many bug fixes involve a change at a single location, while many non-bug fixing commits involve substantially more lines of code. To obtain a good collective discriminative features we need to leverage both text and code based features.

Next, the extracted features are provided to a model learning algorithm that analyzes the features corresponding to bug fixing patches and tries to build a model that discriminates bug fixing patches from other patches. Various algorithms have been proposed to learn a model given a sample of its behavior. We consider some popular classification algorithms (supervised and semi-supervised) and propose a new framework that merges several of them. The final step is the application of our model to the unlabeled data to obtain a set of bug fixing patches.

A challenge in our work is to obtain adequate training data, consisting of known bug fixing patches and known non bug fixing patches. For the former, we may use the patches that have already been applied to Linux stable versions, as well as patches that are derived from the use of bug finding tools or that refer to Bugzilla. But there is no comparable source of labeled non bug fixing patches. Accordingly, we propose a hybrid machine learning algorithm, that first uses a ranking algorithm to identify a set of patches that appear to be quite distant from the set of bug fixing patches. These patches are then considered to be a set of known non bug fixing patches. We then use a supervised classification algorithm to infer a model that can discriminate bug fixing from non bug fixing patches in the unlabeled data.

We describe the details of our framework in the following two sections. In Section IV, we describe our approach to collect data and to extract features from the collected data, corresponding to the first two blocks in Figure 2. In Section V, we describe our new framework that integrates ranking (via semi-supervised classification) and supervised classification.

IV. Data Acquisition & Feature Extraction

A. Data Acquisition

Linux development is managed using the version control system git.[5] Git makes available the history of changes that have been made to the managed code in the form of a series of *patches*. A patch is a description of a complete code change, reflecting the modifications that a developer has made to the source code at the time of a commit. Figure 3 shows an example. A patch consists of two sections: a log message, followed by a description of the code changes. Our data acquisition tool collects information from both of these sections. The collected information is represented using XML, to facilitate subsequent processing.

The log message of a patch, as illustrated by lines 1-16 of Figure 3, consists of a commit number (SHA-1 code), author and date information, a description of the purpose of the patch, and a list of names and emails of people who have been informed of or have approved of the patch. The data acquisition tool collects all of this information.

The change description of a patch, as illustrated by lines 17-29 of Figure 3, appears in the format generated by the command `diff`, using the “unified context” notation [18]. A change may affect multiple files, and multiple code fragments within each file. For each modified file, the `diff` output first indicates the file name (lines 17-20 of Figure 3)

[5] http://git-scm.com/

```
1  commit 45d787b8a946313b73e8a8fc5d501c9aea3d8847
2  Author: Johannes Berg <johannes.berg@intel.com>
3  Date:  Fri Sep 17 00:38:25 2010 +0200
4
5   wext: fix potential private ioctl memory content leak
6
7   commit df6d02300f7c2fbd0fbe626d819c8e5237d72c62 upstream.
8
9   When a driver doesn’t fill the entire buffer, old
10  heap contents may remain, ...
11
12  Reported-by: Jeff Mahoney <jeffm@suse.com>
13  Signed-off-by: Johannes Berg <johannes.berg@intel.com>
14  Signed-off-by: John W. Linville <linville@tuxdriver.com>
15  Signed-off-by: Greg Kroah-Hartman <gregkh@suse.de>
16
17 diff --git a/net/wireless/wext.c b/net/wireless/wext.c
18 index d98ffb7..6890b7e 100644
19 --- a/net/wireless/wext.c
20 +++ b/net/wireless/wext.c
21 @@ -947,7 +947,7 @@ static int ioctl_private_iw_point(...
22        } else if (!iwp->pointer)
23               return -EFAULT;
24
25 -      extra = kmalloc(extra_size, GFP_KERNEL);
26 +      extra = kzalloc(extra_size, GFP_KERNEL);
27        if (!extra)
28               return -ENOMEM;
29
```

Figure 3. A bug fixing patch, applied to stable kernel Linux 2.6.27

and then contains a series of *hunks* describing the changes (lines 21-29 of Figure 3). A hunk begins with an indication of the affected line numbers, in the old and new versions of the file, which is followed by a fragment of code. This code fragment contains *context lines*, which appear in both the old and new versions, *removed lines*, which are preceded by a − and appear only in the old version, and *added lines*, which are preceded by a + and appear only in the new version. A hunk typically begins with three lines of context code, which are followed by a sequence of zero or more removed lines and then the added lines, if any, that replace them. A hunk then ends with three more lines of context code. If changes occur close together, multiple hunks may be combined into one. The example in Figure 3 contains only one hunk, with one line of removed code and one line of added code.

Given the different information in a patch, our data acquisition tool records the boundaries between the information for the different files and the different hunks. Within each hunk, it distinguishes between context, removed, and added code. It does not record file names or hunk line numbers.

B. Feature Extraction

1) Analysis of the Text: A commit log message describes the purpose of the change, and thus can potentially provide valuable information as to whether a commit represents a bug fix. To mechanically extract information from the commit logs, we represent each commit log as a bag of words. In these words, we perform stop-word removal and stemming [19]. Stop words, such as, “is”, “are”, “am”, “would”, etc, are used very frequently in almost *all* documents, and thus they provide little discriminative power. Stemming reduces a word to its root; for example, “writing” and “writes”, could all be reduced to “write”. Stemming groups together words that have the same meaning but only differ due to some grammatical variations. This process can potentially increase the discriminative power of root words that are good at differentiating bug fixing patches from other commits, as more commits with logs containing the root word and its variants can potentially be identified and associated together after stemming is performed.

At the end of this analysis, we represent each commit as a bag of root words. We call this information the *textual facts* that represent the commit.

2) Analysis of the Code: To better understand the effect of a patch, we have also incorporated a parser of patches into our data acquisition tool [24]. Parsing patch code is challenging because a patch often does not represent a complete, top-level program unit, and indeed portions of the affected statements and expressions may be missing, if they extend beyond the three lines of context information. Thus, the parsing is necessarily approximate. The parser is independent of the line-based − and + annotations, only focusing on the terms that have changed. In the common case of changes in function calls, it furthermore detects arguments that have not changed and ignores their subterms. For example, in the patch in Figure 3, the change is detected to involve a function call, *i.e.* to `kmalloc`, which is replaced by a call to `kzalloc`. The initialization of `extra` is not included in the change, and the arguments to `kmalloc` and `kzalloc` are detected to be identical.

Based on the results of the parser, we collect the numbers of various kinds of constructs such as loops, conditionals, and function calls that include removed or added code. We call these the *code facts* that represent the commit.

3) Feature Engineering: Based on the *textual* and *code* facts extracted as described above, we pick interesting features that are compositions of several facts (e.g., the difference between the number of lines changed in the minus and plus hunks, etc.). Table I presents some features that we form based on the facts. Features F_1 to F_{52} are those extracted from code facts. The other features (i.e., features F_{53}-F_{55} and features W_1 to W_n) are those extracted from textual facts.

For code features, we consider various program units changed during a commit including, files, hunks, loops, ifs, contiguous code segments, lines, boolean operators, etc. For many of these program units, we consider the number of times they are added or removed; and also, the sum and difference of these numbers. Our initial investigation suggests that often bug fixing patches, and other commits (e.g., feature additions, performance enhancements, etc) have different value distributions for these code features.

For text features, we consider stemmed non-stop words appearing in the logs as features. For each feature corresponding to a word, we take its frequency (i.e., number of times it appears in a commit log) as its corresponding feature value. We also consider two composite families of

Table I
EXTRACTED FEATURES

ID	Feature
F_1	Number of files changed in a commit.
F_2	Number of hunks in a commit.
F_3	#Loops Added
F_4	#Loops Removed
F_5	$\|F_3$ - $F_4\|$
F_6	F_3 + F_4
F_7	$F_{13} > F_{14}$
$F_8 - F_{12}$	Similar to F_3 to F_7 for #Ifs
$F_{13} - F_{17}$	Similar to F_3 to F_7 for #Contiguous code segments
$F_{18} - F_{22}$	Similar to F_3 to F_7 for #Lines
$F_{23} - F_{27}$	Similar to F_3 to F_7 for #Character literals
$F_{28} - F_{32}$	Similar to F_3 to F_7 for #Paranthesized expressions
$F_{33} - F_{37}$	Similar to F_3 to F_7 for #Expressions
$F_{38} - F_{42}$	Similar to F_3 to F_7 for #Boolean operators
$F_{43} - F_{47}$	Similar to F_3 to F_7 for #Assignments
$F_{48} - F_{52}$	Similar to F_3 to F_7 for #Function calls
F_{53}	One of these words exists in the commit log {robust, unnecessary, improve, future, anticipation, superfluous, remove unused}
F_{54}	One of these words exists in the commit log {must, needs to, remove, has to, don't, fault, error, have to, need to}
F_{55}	The word "warning" exists in the commit log
W_1 *to* W_n	Each feature represents a stemmed non-stop word in the commit log. Each feature has a value corresponding to the number of times the word appears in the commit (i.e., term frequency).

words each conveying a similar meaning: one contains words that are likely to relate to performance improvement, feature addition, and clean up; another contains words that are likely to relate to a necessity to fix an error. We also consider the word "warning" (not stemmed) as a separate textual feature.

V. MODEL LEARNING & BUG FIX IDENTIFICATION

A. *Model Learning*

We propose a solution that integrates two classification algorithms: Learning from Positive and Unlabeled Examples (LPU) [16][6] and Support Vector Machine (SVM) [9].[7] These learning algorithms take in two datasets: training and testing, where each dataset consists of many data points. The algorithms each learn a model from the training data and apply the model to the test data. We first describe the differences between these two algorithms.

LPU [16] performs *semi-supervised* classification [7], [40]. Given a positive dataset and an unlabelled dataset, LPU builds a model that can discriminate positive from negative data points. The learned model can then be used to label data with unknown labels. For each data point, the model outputs a score indicating the likelihood that the unlabeled data is positive. We can rank the unlabeled data points based on this score.

SVM on the other hand performs *supervised* classification. Given a positive dataset and a negative dataset, SVM builds a model that can discriminate between them. While LPU only requires the availability of datasets with positive labels, SVM requires the availability of datasets with both positive and negative labels.

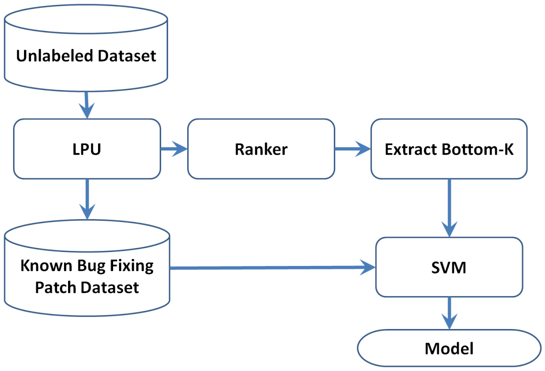

Figure 4. Model Learning

LPU tends to learn a weaker discriminative model than SVM because it takes in only positive and *unlabeled* data, while SVM is able to compare and contrast positive and negative data. To be able to classify well, we propose to combine LPU and SVM. First, we use LPU to rank how far an unlabeled data point is from the positive training data. For this, we sort the data points based on their LPU scores (in descending order), indicating the likelihood of a data point being positive. The bottom k data points, where k is a user-defined parameter, are then taken as a proxy for the negative data. These negative data along with the positive data are then used as the input to SVM. The steps in our model learning process are shown in Figure 4.

In the problem of identifying bug fixing patches, each data point is a commit. We have a set of positive data points, *i.e.*, bug fixing patches, and a set of unlabeled data points, *i.e.*, arbitrary commits. We first apply LPU to sort commits such that bug fixing patches are listed first and other patches, which may correspond to innocuous changes, performance improvements or feature additions, are listed later. According to this ordering, the bottom k commits are likely to be non-bug fixing patches. We then take the bottom k commits to be a proxy of a dataset containing non-bug fixing patches. We use the original bug fixing patch dataset and this data to create a model using SVM.

B. *Bug Fix Identification*

Once we have learned the model, for bug fix identification, we apply the same feature extraction process to a test dataset with unknown labels. We then represent this test dataset by a set of feature values. These feature values are then fed to the learned model as described in Section V-A. Based on these features, the model then assigns either one of the following two labels to each commit: bug-fixing or non bug-fixing.

VI. EXPERIMENT

We first describe the datasets used for our evaluation, and then present a number of research questions. Then we present experimental results that answer these questions.

[6] http://www.cs.uic.edu/ liub/LPU/LPU-download.html
[7] http://svmlight.joachims.org/

Table II
PROPERTIES OF THE CONSIDERED BLACK DATASETS. LOC (LINES OF CODE) INCLUDES BOTH THE LOG AND THE CHANGED CODE.

Source	Dates	# patches	LOC
Stable 2.6.20	02.2007-08.2007	409	29K
Stable 2.6.27	10.2008-12.2010	1534	116K
Coverity	05.2005-06.2011	478	22K
Coccinelle	11.2007-08.2011	825	54K
Smatch	12.2006-08.2011	721	31K
Bugzilla	08.2005-08.2011	2568	275K

Table III
PROPERTIES OF THE CONSIDERED GREY DATASET, BY LINUX VERSION.

Source	Dates	# patches
2.6.20-2.6.21	02.2007-04.2007	3415
2.6.21-2.6.22	04.2007-07.2007	3635
2.6.22-2.6.23	07.2007-10.2007	3338
2.6.23-2.6.24	10.2007-01.2008	4639
2.6.24-2.6.25	01.2008-04.2008	6110
2.6.25-2.6.26	04.2008-07.2008	5069

A. Dataset

Our algorithm requires as input "black" data that is known to represent bug-fixing patches and "grey" data that may or may not represent bug-fixing patches. The "grey" data may contain both "black" data and "white" data (i.e., non bug-fixing patches).

As there is no a priori definition of what is a bug-fixing patch in Linux, we have created a selection of black data sets from varying sources. One source of black data is the patches that have been applied to existing stable versions. We have considered the patches applied to the stable versions Linux 2.6.20,[8] released in February 2007 and maintained until August 2007, and Linux 2.6.27,[9] released in October 2008 and maintained until December 2010. We have taken only those patches that include C code, and where this code is not in the Documentation section of the kernel source tree. Another source of black data is the patches that have been created based on the use of bug finding tools. We consider uses of the commercial tool Coverity,[10] which was most actively used prior to 2009, and the open source tools Coccinelle [23] and Smatch,[11] which have been most actively used since 2008 and 2009, respectively [25]. The Coverity patches are collected by searching for patches that mention Coverity in the log message. The Coccinelle and Smatch patches are collected by searching for patches from the principal users of these tools, which are the second author of this paper and Dan Carpenter, respectively. The Coccinelle data may contain both bug fixes and simple refactorings. The Coverity and Smatch patches should contain only bug fixes. All three data sets are taken from the complete set of patches between April 2005 and August 2011. Our final source of black data is the set of patches that mention Bugzilla, taken from the same time period. Table II summarizes various properties of these data sets.

The grey data is taken as the complete set of patches that have been applied to the Linux kernel between versions 2.6.20 and 2.6.26. To reduce the size of the dataset, we take only those patches that can apply without conflicts to the Linux 2.6.20 code base. Table III summarizes various properties of the data sets.

B. Research Questions

Our study addresses the following four research questions (RQ1-RQ4). In RQ1, we investigate the effectiveness of our approach. Factors that influence its effectiveness are investigated in RQ2 and RQ3. Finally, RQ4 investigates the benefit of our hybrid classification model.

RQ1: *Does our approach identify bug fixing patches as well as the existing keyword-based method?* To evaluate the effectiveness of our approach as compared with existing keyword-based methods, we consider the following criteria:

Criterion 1: Precision and Recall on Sampled Data. We randomly sample 500 commits and manually label each as a bug fix that could go to stable or not. We then compare the human-assigned labels with the labels assigned by each bug fix identification approach, and compute the associated precision and recall [19].

Criterion 2: Accuracy on Known Black Data. We take commits that have been identified by Linux developers as bug fixing patches and split this dataset into 10 equal sized groups. We train on 9 groups and use one group to test. We evaluate how many of the bug fixing patches are correctly labeled. The process is iterated 10 times. For each iteration we compute the number of bug fixing patches that are correctly identified (we refer to this as accuracy^{Black}) and report the average accuracy.

The goal of the first criterion is to estimate the accuracy of our approach on some sampled data points. One of the authors is an expert on Linux development and has contributed many patches to Linux code base. This author manually assigned labels to the sampled data points. The goal of the second criterion is to address the experimenter bias existing in the first criteria. Unfortunately, we only have known black data. Thus, we evaluate our approach in terms of its accuracy in labeling black data as such.

RQ2: *What is the effect of the parameter k on the results?* Our algorithm has one parameter, k, which specifies the number of bottom ranked commits that we take as a proxy of a dataset containing non-bug fixing patches. As a default value in our experiments, we fix k to be 0.9 × the number

[8] http://www.kernel.org/pub/scm/linux/kernel/git/stable/linux-2.6.20.y
[9] http://www.kernel.org/pub/scm/linux/kernel/git/stable/linux-2.6.27.y
[10] http://www.coverity.com/
[11] http://smatch.sourceforge.net/

of "black" data that are known bug fixing patches. We vary this number and investigate its impact on the result.

RQ3: *What are the best features for discriminating if a commit is a bug fixing patches?* Aside from producing a model that can identify bug fixing patches, we are also interested in finding discriminative features that could help in distinguishing bug fixing patches and other commits. We would like to identify these features out of the many textual and code features that we extract from commits.

We create a *clean dataset* containing all the known black data, the manually labeled black data, and the manually labeled white data. We then compute the Fisher score [10] of all the features that we have. Specifically, we compute a variant of the Fisher score reported in [8] and implemented in LibSVM[12]. The Fisher score and its variants have been frequently used to identify important features [6].

RQ4: *Is our hybrid approach (i.e., ranking + supervised classification using LPU+SVM) more effective than a simple semi-supervised approach (i.e., LPU)?* Our dataset only contains positively labeled data points (i.e., bug fixing patches). To address this sort of problem, machine-learning researchers have investigated semi-supervised learning solutions. Many of these techniques still required a number of negatively labeled data points. However, LPU [16], which is one of the few semi-supervised classification algorithms with an implementation available online, only requires positively labeled and unlabeled data points.

Our proposed solution includes a ranking and a supervised classification component. The ranking component makes use of LPU. Thus it is interesting to investigate if the result of using LPU alone is sufficient and whether our hybrid approach improves the results of LPU.

C. Experimental Results

We present our experimental results as answers to the four research questions: RQ1-RQ4.

1) RQ1: Effectiveness of Our Approach: We compare our approach to the keyword-based approach used in the literature [14], [21]. The result of the comparisons using the two criteria are discussed below.

Precision and Recall on Sampled Data. Table IV compares the precision and recall of our approach to those of the keyword-based approach. Our precision is comparable with that of the keyword-based approach. On the other hand, we increase the recall of the keyword-based approach from 0.588 to 0.900; this is an improvement of 53.19%.

To combine precision and recall, we also compute the F-measure [19], which is a harmonic mean of precision and recall. The F-measure is often used to evaluate whether an improvement in recall outweighs a reduction in precision (and vice versa). The F-measure has a parameter β that measures the importance of precision over recall. The formula is:

Table IV
PRECISION AND RECALL COMPARISON

Approach	Precision	Recall
Ours	0.537	0.900
Keyword	0.519	0.588

Table V
F-MEASURES COMPARISON

Approach	F1	F2	F3	F5
Ours	0.673	0.793	0.843	0.877
Keyword	0.551	0.572	0.580	0.585
Improvement	22.05%	38.51%	45.39%	50.07%

$$\frac{(\beta^2+1)\times precision \times recall}{(\beta^2 \times precision) + recall}$$

If precision and recall are equally important, β is set to one. This computes what is known as F1. If beta is set higher than 1, then recall is preferred over precision; similarly, if beta is set lower than 1 then precision is preferred over recall.

In the context of bug fixing patch identification, recall (*i.e.*, not missing any bug fixing patch) is more important than precision (*i.e.*, not reporting wrong bug fixing patch). Missing a bug fixing patch could potentially cause system errors and even expose security holes.[13] For these cases, a standard IR book [19], recommend setting β equal to 3 or 5. Other studies recommend setting β to 2 [30].

Table V shows the F-measures for the different values of β. For all values of β our approach has better results than the keyword-based approach. The F1, F2, F3, and F5 scores are improved by 22.05%, 38.51%, 45.39%, and 50.07% respectively.

From the 500 randomly sampled commits, we notice that a very small number of the commits that are bug fixing patches reference Bugzilla. Thus identifying these patches are not trivial. Also, as shown in Table IV, about 40% of bug fixing patches do not contain the keywords considered in previous work [14], [21].

Accuracy on Known Black Data. For the given data, our approach increases $accuracy^{Black}$ by 22.4% as compared to the keyword-based approach, from 0.772 to 0.945. These results show that our approach is effective in identifying bug fixing patches as compared to keyword-based approach used in existing studies.

The known black data is unbiased as we do not label it ourselves. However, this experiment does not provide any information about the rate of false positives, as all our known test data are black.

[12] http://www.csie.ntu.edu.tw/∼cjlin/libsvmtools

[13] Recall is more important in other related settings. Manning *et al.* describe scenarios where recall is more important than precision, e.g., paralegals and intelligence analysts potentially looking for violations of a legal clause or for terrorists [19].

Table VI
EFFECT OF VARYING k ON PERFORMANCE. TP = TRUE POSITIVE, FN = FALSE NEGATIVE, FP = FALSE POSITIVE, TN = TRUE NEGATIVE.

k	TP	FN	FP	TN	Prec.	Recall	F2
0.75	156	4	206	134	0.431	0.975	0.778
0.80	152	8	186	154	0.450	0.950	0.777
0.85	149	11	165	175	0.475	0.931	0.781
0.90	144	16	124	216	0.537	0.900	0.793
0.95	117	43	84	256	0.582	0.731	0.696

The high accuracy of the keyword-based approach is due to the large number of Bugzilla patches in the *clean* bug fixing patch dataset. In practice, however, most bug fixing patches are not in Bugzilla – these bug fixing patches are hidden in the mass of other non bug fix related commits.

2) RQ2: Effects of Varying Parameter k: When we vary the parameter k, as a proportion of the amount of "black" data, the number of false positives and false negatives changes. The results of our experiments with varying values for k is shown in Table VI.

As we increase the value of k the number of false negatives (FN) increases, while the number of false positives (FP) decreases. Indeed, as we increase the value of k, the "pseudo-white" data (*i.e.*, the bottom k commits in the sorted list after ranking using LPU) gets "dirtier" as more "black" data are likely to be mixed with the "white" data in it. Thus more and more "black" data are wrongly labeled as "white" (*i.e.*, an increase in false negatives). However, the white data are still closer to the "dirty" "pseudo-white" data than to the black data. Also, more and more borderline "white" data are "closer" to the "dirtier" "pseudo-white" data than before. This reduces the number of cases where "white" data are labeled "black" (*i.e.*, a reduction in false positives). We illustrate this in Figure 5.

3) RQ3: Best Features: From the bug reports, we extract thousands of features corresponding to the presence or absence of words in commit logs and the various code facts. We report the top 20 features sorted based on their Fisher scores in Table VII.

We note that among the top-20 features there are both textual and code features. This highlights the usefulness of combining both textual features in commit logs and code features in changed code to predict bug fixing patches. We notice however that the Fisher score is low (the highest possible value is 1), which highlights that one feature alone is not sufficient to discriminate positive from negative datasets (*i.e.*, bug fixing patches versus other commits).

Some keywords used in previous approaches [14], [21], [29], e.g., fix, bugzilla, etc., are also included in the top-20 features. Due to tokenization some of these features are split into multiple features, e.g., http, bug.cgi, and bugzilla.kernel.org. The word blackfin, which is the name of a family of microprocessors, is in the top 20 as many commits containing this keyword are non bug fixing patches.

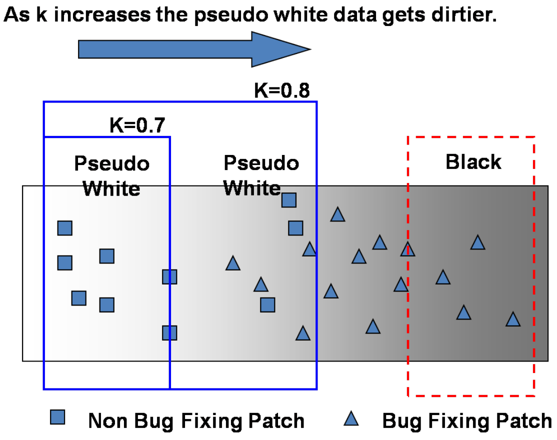

Figure 5. Effect of Varying k. The pseudo white data is the bottom k commits that we treat as a proxy to non bug fixing patches. The three boxes, corresponding to pseudo white (2 of them) and black data, represent the aggregate features of the respective pseudo-white and black data in our training set, respectively. The squares and triangles represent test data points whose labels (*i.e.*, bug fixing patches or not) are to be predicted.

Table VII
TOP-20 MOST DISCRIMINATIVE FEATURES BASED ON FISHER SCORE

Rank	Feature Desc.	Fisher Score
1	http	0.032
2	bug.cgi	0.022
3	blackfin	0.021
4	show	0.019
5	bugzilla.kernel.org	0.015
6	fix	0.015
7	commit	0.014
8	F_{18} (*i.e.*, # lines removed)	0.014
9	upstream	0.013
10	id	0.012
11	F_{20} (*i.e.*, # lines added & removed)	0.011
12	unifi	0.011
13	F_{38} (*i.e.*, # boolean operators removed)	0.010
14	F_{44} (*i.e.*, # assignments removed)	0.010
15	checkpatch	0.010
16	spell	0.010
17	F_{46} (*i.e.*, # assign. removed & added)	0.009
18	F_{37} (*i.e.*, # boolean operators added)	0.009
19	F_6 (*i.e.*, # loops added & removed)	0.008
20	F_{48} (*i.e.*, # function calls added)	0.008

Many other features in the list are code features; these include the number of times different program elements are changed by a commit. The most discriminative code element is the number of lines of code being deleted (ranked 8th). Next come features such as the number of lines added and deleted (ranked 11th), the number of boolean operators added (ranked 13th), the number of assignments removed (ranked 14th), the number of assignments added and removed (ranked 17th), the number of boolean operators added (ranked 18th), the number of loops added and removed (ranked 19th), and the number of function calls made (ranked 20th).

Table VIII
COMPARISONS WITH LPU.
PREC. = PRECISION, ACC. = ACCURACY^{Black}

Approach	Prec.	Recall	F1
Ours	0.537	0.900	0.673
LPU Only	0.563	0.281	0.375
Improvement	-4.5%	219.9%	79.4%

Approach	F2	F3	F5	Acc.
Ours	0.793	0.843	0.877	0.944
LPU Only	0.313	0.296	0.287	0.942
Improvement	153.7%	184.7%	205.9%	0.2%

4) RQ4: Our Approach versus LPU: We have run LPU on our dataset and found that the results of using LPU alone are not good. Results are shown in Table VIII.

The precision of LPU alone is slightly higher than that of our approach, but the reported recall is much lower. Our approach can increase the recall by more than 3 times (*i.e.*, 200% improvement). When we trade off precision and recall using F-measure, we notice that for all β our approach is better than LPU by 79.4%, 153.7%, 184.7%, and 205.9% for F1, F2, F3, and F5 respectively.

The $accuracy^{Black}$ values of our approach and that of LPU alone are comparable. Notice that the black data in $accuracy^{Black}$ are similar to one another, with many having the terms Bugzilla, http, etc. The black data in the 500 random sample are more challenging and better reflect the black data that are often hidden in the mass of other commits.

The above highlights the benefit of our hybrid approach of combining ranking and supervised classification to address the problem of unavailability of negative data points (*i.e.*, the non bug fixing patches) as compared to a simple application of a standard semi-supervised classification approach. In our approach, LPU is used for ranking to get a pseudo-negative dataset and SVM is used to learn the discriminative model.

VII. DISCUSSION

Threats to Validity. As with other empirical studies there are a number of threats to the validity of our results.

Threats to internal validity corresponds to the relationship between the independent and dependent variables in the study. One such threat in our study is experimenter bias, as we have personally labelled each commits as a bug fixing patch or as a non bug fixing patch. This labelling might introduce some experimenter bias. However, we have tried to ensure that we label the commits correctly, according to our experience with Linux code [15], [23], [24]. Also, we have labelled the commits before seeing the results of our identification approach, to minimize this bias.

Threats to external validity refers to the generalizability of the result. We have manually checked the effectiveness of our approach over 500 commits. Although 500 is not a very large number, we still believe it is a good sample size. Past studies, e.g., [1], [5], [38], investigate a similar amount of manually labeled data. We plan to reduce this threat of external validity in the future by investigating an even larger number of manually labeled commits. We have also only investigated patches in Linux. Although we analyze one system, it is large and contains diverse components. Linux is one of the largest open source project in terms of code size, number of contributors, and configurations. Thus we believe that its development is worth study. The size of Linux (LOC) is indeed larger than the sum of that of systems investigated in related prior studies, *e.g.*, [38]. We believe our approach can be easily applied to identify bug fixing patches in other systems, but leave this to future work.

Threats to construct validity deals with the appropriateness of the evaluation criteria. We use the standard measures precision, recall, and F-measure [19] to evaluate the effectiveness of our approach. Thus, we believe there is little threat to construct validity.

Automated Tuning of k. If there exist manual user labels on some representative samples of commits, then we can use this information to automatically tune the value of k. As a default initial value, k could be set as 0.9 × the number of black data. The performance of a classifier based on this initial value could be evaluated based on the manual user labels. k could be reduced or increased to improve recall and precision on the input representative sample of commits. The best value of k is one that allows learning from the most "white" example data points without mixing too many "black" data points with the "white" ones.

In an additional experiment, we take 250 of the 500 manually labeled commits and use it to fine tune k. The best value of k to optimize $F1$ is again 0.9. For $F2$, at $k = 0.9$, its score is not much different than that of the other values of k in the range of 0.75-0.9 that we try. We still notice that in terms of $F2$, taking either 0.75, 0.8, 0.85, or 0.9 does not impact $F2$ much (less than 0.03 difference).

Benefit of Including Unlabeled Data. Labeling is expensive and time consuming. On the other hand, unlabeled data can be obtained easily. As there are many variants of bug fixing and non bug fixing patches, many labeled data are needed to characterize all these variants. Producing these labels would cost much time and effort. Unlabeled data already contain many different variants of bug fixing and non bug fixing patches. In this study, we leverage unlabeled data and show that they can be used to identify bug fixing patches well.

Characteristics of Non-Bug Fixing Patches. From the non-bug fixing patches that we label and inspect, we notice that they fall into several categories:

1) Performance enhancement commits. A commit of this type often tries to make something run faster. Often this performance issue is not a bug that need to be patched in a stable version.
2) Code clean up. A commit of this type improves the

structure of the code without changing its semantics, to make the code easier to read, etc. Stable versions target users, not developers, so readability is not a priority.

3) Feature addition or removal. A commit of this type adds a new feature or remove an old one. Users of stable versions are likely to find such changes undesirable.
4) Warnings. Warnings are not errors. The code can still run well even in the existence of these warnings. Eliminating such a warning does not fix a bug.

VIII. Related Work

We describe some studies that are related to our approach. We start with those that are most related, and then consider some that are marginally related.

Identification of Bug Fixes. A number of studies have searched for keywords such as "bug" and "fix" in log messages to identify bug fixing commits [11], [14], [21], [29]. Our approach, on the other hand, is not based on a fixed set of keywords. Rather, we automatically infer keywords that are good at discriminating bug fixing patches from other commits. Furthermore, we consider not only commit logs, but also some features extracted from the changes made to the source code. We then built a discriminative machine learning model that is used to classify commits as either bug fixing or not. Experimental results show that our approach can identify more bug fixing patches with similar precision but improved recall, as compared to a fixed-keyword based approach.

Bird *et al.* have observed that the lack of clearly identified bug fixing patches itself has caused potential bias in many prior studies [4]. Thus we believe our approach could not only benefit Linux stable release maintainers but also help other studies involving the analysis of bug fixes.

There are two recent studies that are related to ours. Wu *et al.* propose ReLink which links bug reports to their associated commits [38]. ReLink only captures tracked bugs; bugs described only in mailing lists, etc. are mentioned as future work. Our work considers a different problem and does not require the availability of bug reports, which may be absent or incomplete. Bird *et al.* propose Linkster which integrates information from various sources to support manual link recovery [5]. Our approach is based on machine learning for more automation.

Studies on Bug Reports. There have been a number of studies that analyze bug reports [2], [3], [33], [13], [17], [20], [27], [28], [32], [34], [37]. Similar to these studies we also analyze textual data found in software artifacts. We focus on commit logs made by developers while these studies focus on bug reports made by users.

Bug localization tries to locate methods that are responsible for a bug given the corresponding bug report [17], [20], [27]. Bug localization approaches require as input linkages between code responsible for the bugs and the bug reports. Obtaining these linkages is often hard due to poor commit log comments. Our approach of inferring bug fixing patches could potentially help infer these links.

Anvik *et al.* investigate the problem of automatic bug triaging, which tries to recommend developers suitable to fix a bug [2]. With more bug fixing patches identified, possibly a better recommendation of suitable developers could be made.

Other Studies on Textual Software Engineering Data. There have been other studies that analyze software and its related textual artifacts. Closest to our approach, is the work by Antoniol et al. which analyzes change requests and classify them as either bugs or enhancements [1]. Change requests contain text with additional fields such as severity and are written by ordinary users. Commits considered in this work contains both text and code. They are written by experts, but do not include fields such as severity. Thus, the type of input is different.

There are many other studies. For example, Tan *et al.* analyze source code comments to find concurrency bugs [35]. Zhong *et al.* infer program specifications from API documentation [39]. Gottipati *et al.* build an effective search engine over software forum posts [12]. Wang et al. extract paraphrases of technical terms from bug reports [36]. There are also a number of techniques that trace requirements expressed in natural language, including the work of Port *et al.* [26], Sultanov *et al.* [31], etc.

IX. Conclusion & Future Work

Linux developers periodically designate a release as being subject to longterm support. During the support period, bug fixes applied to the mainline kernel need to be back ported to these longterm releases. This task is not trivial as developers do not necessarily make explicit which commits are bug fixes, and which of them need to be applied to the longterm releases. To address this problem, we propose an automated approach to infer commits that represent bug fixing patches. Our approach first extracts features from the commits that describe those code changes and log messages that can potentially distinguish bug fixing patches from regular commits. A machine learning approach involving ranking and classification is employed. Experiments on Linux commits show that we can improve on the existing keyword-based approach, obtaining similar precision and improving recall by 53.19%.

In the future, we plan to further improve the accuracy of our approach. We also plan to apply our approach to work in bug prediction and related areas that suffer from bias due to unidentified bug fixes.

References

[1] G. Antoniol, K. Ayari, M. D. Penta, F. Khomh, and Y.-G. Guéhéneuc. Is it a bug or an enhancement?: a text-based approach to classify change requests. In *CASCON*, 2008.

[2] J. Anvik, L. Hiew, and G. Murphy. Who should fix this bug? In *ICSE*, pages 361–370, 2006.

[3] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann. What makes a good bug report? In *SIGSOFT '08/FSE-16*, pages 308–318, 2008.

[4] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. T. Devanbu. Fair and balanced?: bias in bug-fix datasets. In *ESEC/SIGSOFT FSE*, pages 121–130, 2009.

[5] C. Bird, A. Bachmann, F. Rahman, and A. Bernstein. Linkster: enabling efficient manual inspection and annotation of mined data. In *SIGSOFT FSE*, pages 369–370, 2010.

[6] Y.-W. Chang and C.-J. Lin. Feature ranking using linear svm. *Journal of Machine Learning Research - Proceedings Track*, 3:53–64, 2008.

[7] O. Chapelle, B. Scholkopf, and A. Zien. *Semi-Supervised Learning*. MIT Press, 2006.

[8] Y.-W. Chen and C.-J. Lin. Combining SVMs with various feature selection strategies. In I. Guyon, S. Gunn, M. Nikravesh, and L. Zadeh, editors, *Feature extraction, foundations and applications*. Springer, 2006.

[9] C. Cortes and V. Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995.

[10] T. Cover and J. Thomas. *Elements of Information Theory*. Wiley-Interscience, 2006.

[11] V. Dallmeier and T. Zimmermann. Extraction of bug localization benchmarks from history. In *ASE*, pages 433–436, 2007.

[12] S. Gottipati, D. Lo, and J. Jiang. Finding relevant answers in software forums. In *ASE*, 2011.

[13] N. Jalbert and W. Weimer. Automated duplicate detection for bug tracking systems. In *proceedings of the International Conference on Dependable Systems and Networks*, pages 52–61, 2008.

[14] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller. Predicting faults from cached history. In *ICSE*, pages 489–498, 2007.

[15] J. L. Lawall, J. Brunel, R. R. Hansen, H. Stuart, G. Muller, and N. Palix. WYSIWIB: A declarative approach to finding protocols and bugs in Linux code. In *DSN*, pages 43–52, June 2009.

[16] X. Li and B. Liu. Learning to classify text using positive and unlabeled data. In *IJCAI*, 2003.

[17] S. Lukins, N. Karft, and E. Letha. Source code retrieval for bug localization using latent dirichlet allocation. In *WCRE*, 2008.

[18] D. MacKenzie, P. Eggert, and R. Stallman. *Comparing and Merging Files With Gnu Diff and Patch*. Network Theory Ltd, Jan. 2003. Unified Format section, http://www.gnu.org/software/diffutils/manual/html_node/Unified-Format.html.

[19] C. Manning, P. Raghavan, and M. Schutze. *Introduction to Information Retrieval*. Cambridge University Press, 2009.

[20] A. Marcus, A. Sergeyev, V. Rajlich, and J. Maletic. An information retrieval approach to concept location in source code. In *WCRE*, pages 214–223, 2004.

[21] A. Mockus and L. G. Votta. Identifying reasons for software changes using historic databases. In *ICSM*, pages 120–130, 2000.

[22] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Recurring bug fixes in object-oriented programs. In *ICSE (1)*, pages 315–324, 2010.

[23] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller. Documenting and automating collateral evolutions in Linux device drivers. In *EuroSys*, pages 247–260, 2008.

[24] Y. Padioleau, J. L. Lawall, and G. Muller. Understanding collateral evolution in Linux device drivers. In *EuroSys*, pages 59–71, 2006.

[25] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller. Faults in Linux: Ten years later. In *ASPLOS*, pages 305–318, 2011.

[26] D. Port, A. P. Nikora, J. H. Hayes, and L. Huang. Text mining support for software requirements: Traceability assurance. In *HICSS*, pages 1–11, 2011.

[27] S. Rao and A. Kak. Retrieval from software libraries for bug localization: a comparative study of generic and composite text models. In *MSR*, 2011.

[28] P. Runeson, M. Alexandersson, and O. Nyholm. Detection of duplicate defect reports using natural language processing. In *ICSE*, pages 499–510, 2007.

[29] J. Sliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *MSR*, 2005.

[30] K. Small, B. Wallace, C. Brodley, and T. Trikalinos. The constrained weight space svm:learning with ranked features. In *ICML*, 2011.

[31] H. Sultanov, J. H. Hayes, and W.-K. Kong. Application of swarm techniques to requirements tracing. *Requir. Eng.*, 16(3):209–226, 2011.

[32] C. Sun, D. Lo, S.-C. Khoo, and J. Jiang. Towards more accurate retrieval of duplicate bug reports. In *ASE*, 2011.

[33] C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. Khoo. A discriminative model approach for accurate duplicate bug report retrieval. In *ICSE*, pages 45–54, 2010.

[34] A. Sureka and P. Jalote. Detecting duplicate bug report using character n-gram-based features. In *APSEC*, pages 366–374, 2010.

[35] L. Tan, Y. Zhou, and Y. Padioleau. aComment: mining annotations from comments and code to detect interupt related concurrency bugs. In *ICSE*, pages 11–20, 2011.

[36] X. Wang, D. Lo, J. Jiang, L. Zhang, and H. Mei. Extracting paraphrases of technical terms from noisy parallel software corpora. In *Proc. of the 47th Annual Meeting of the Assoc. for Computational Linguistics (ACL)*, 2009.

[37] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun. An approach to detecting duplicate bug reports using natural language and execution information. In *ICSE*, pages 461–470, 2008.

[38] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung. Relink: recovering links between bugs and changes. In *SIGSOFT FSE*, pages 15–25, 2011.

[39] H. Zhong, L. Zhang, T. Xie, and H. Mei. Inferring resource specifications from natural language API documentation. In *ASE*, pages 307–318, 2009.

[40] X. Zhu and A. Goldberg. *Introduction to Semi-Supervised Learning*. Morgan & Claypool Publishers, 2009.