Chapter 4

Identifying Linux Bug Fixing Patches

4.1 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 fixing some versions for longterm support. The primary 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. Such 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 small number of changes to the code. This process, however, puts an extra burden on the subsystem maintainers and thus necessary bug fixing patches could be missed. Thus, a technique that could automatically label a commit as a bug fixing patch would be

valuable.

In the literature, there are furthermore many studies that require the identification of links between commits and bugs. These include work on empirical study of software changes [54, 81], bug prediction [32, 59], bug localization [15, 49, 52, 71], and many more. 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.

Some of these limitations have also been observed by Bird et al. [5] who performed an empirical study that show 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 chapter, 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 code change to identify bug fixing patches. We use an analysis plus classification framework that 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.

A challenge for our approach is to obtain appropriately labeled training data. 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 negative data, i.e., non bug fixing patches. To address this problem, we propose a new approach that integrates two learning algorithms via 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 0.601 and 0.875 while those of the key-word based approach are 0.613 and 0.603. 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 45.11% recall while maintaining similar precision.

4.2 Background

Linux is an open-source operating system that 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 end users, 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 the current versions of Suse Enterprise Linux, Red Hat Enterprise Linux and Oracle Unbreakable Linux.

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 Chapter, we refer to both 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 email address. These patches are

[1] linux-2.6.39/Documentation/stable kernel rules.txt
[2] linux-2.6.39/Documentation/SubmittingPatches.txt

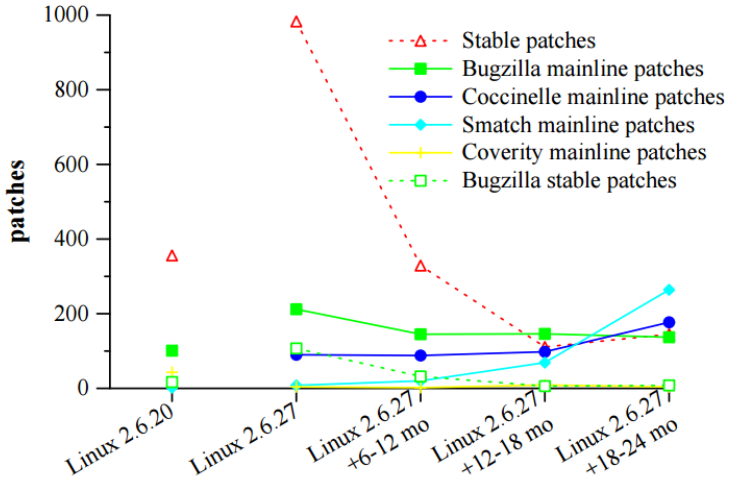

Figure 4.1: Various kinds of patches applied to the stable kernels 2.6.20 and 2.6.27 and to the mainline kernel in the same time period.

then reviewed by the maintainers of the stable kernels before being integrated into the code base.

A comparison, shown in Figure 5.5 of the patches accepted into the mainline kernel with those 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, shows a gap between the set of bug fixing patches accepted into the mainline as compared to the stable kernels. Specifically, we consider the mainline patches that mention Bugzilla, or that mention a bug finding tool (Coccinelle [60], Smatch [3] or Coverity [4]). We also include the number of patches mentioning Bugzilla that are included in the stable kernel. These amount to at best around half of the Bugzilla patches. Fewer than 5 patches for each of the considered bug finding tools were included in 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.

[3] http://smatch.sourceforge.net/
[4] http://coverity.com/

4.3 Approach

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 4.2.

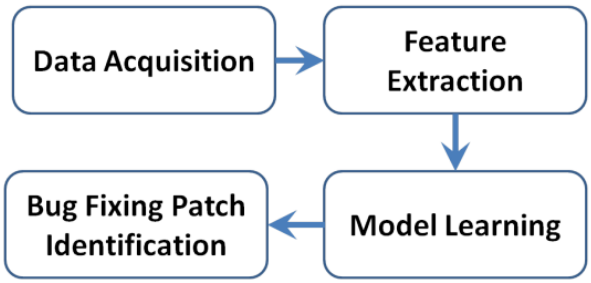

Figure 4.2: Overall Framework

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 hidden in the mass of many other commits that do not perform bug fixing. There are a variety of non bug fixing commits including those that 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 also can help a lot. Many bug fixes involve a single location change, boolean operators in if and loop expressions, etc. Many non-bug fixing commits involve substantially many lines of code, etc. To obtain a good collective discriminative features we would 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 algorithms together. The final step is the application of our model on 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. As representatives of bug fixing patches, we may use the patches that have already been applied to Linux stable versions, as well as patches that are known to be bug fixing patches, such as those 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 can then be 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 subsections.

4.3.1 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. An example is shown in Figure 4.3. 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 4.3, consists

[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 4.3: A bug fixing patch, applied to stable kernel Linux 2.6.27

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 code change of a patch, as illustrated by lines 17-29 of Figure 4.3, appears in the format generated by the command diff, using the "unified context" notation [50]. 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 4.3) and then contains a series of hunks describing the changes (lines 21-29 of Figure 4.3). 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 a single one. The example in Figure 4.3 contains only a single 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.

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. For each of these bags of words, we perform stop-word removal and stemming [74]. Stop words, such as, is, are, am, would, etc, are used very frequently in almost all documents thus they provide little power in discriminating if a commit is a bug fixing patches or not. Stemming reduces a word to its root; for example, eating, ate, and eaten, are all reduced to the root word eat. Stemming is employed to group 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 words, where each word is a root word and not a stop word. We call this information the textual facts that represent the commit.

To better understand the effect of a patch, we have incorporated a parser of patches into our data acquisition tool [61]. 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, counting these separately and ignoring their subterms. For example, in the patch in Figure 4.3, the change is detected to involve a function call, i.e. the call 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 function headers, loops, conditionals, and function calls that include removed or added code. We call these the code facts that represent the commit.

4.3.2 Feature Extraction

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 4.1 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} to 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. 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 or the number of times it appears in a commit log as its corresponding feature value. We also consider two composite families of words each conveying a similar meaning:

Table 4.1: 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, has to, dont, fault, need to, error, have to, remove
F_{55}	The word "warning" exists in the commit log
$W_1 - W_n$	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).

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.

4.3.3 Model Learning

We propose a solution that integrates two classification algorithms: Learning from Positive and Unlabeled Examples (LPU) [45][6] and Support Vector Machine (SVM) [12].[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 performs semi-supervised classification. 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.

LPU tends to learn a weaker discriminative model than SVM. This is because LPU 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 a combination of LPU and SVM. First, we use LPU to rank how far an unlabeled data point

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

[7] http://svmlight.joachims.org

is from the positive training data (in descending order). For this, we sort the data points based on their LPU scores, 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 sequence of steps in our model learning process is shown in Figure 4.4.

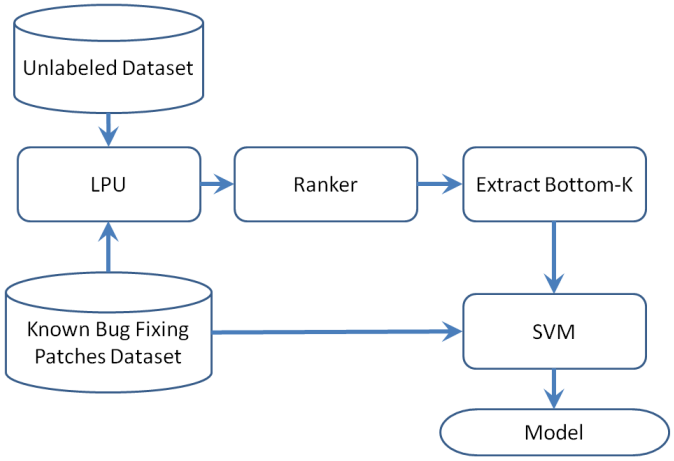

Figure 4.4: Model Learning

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.

4.3.4 Bug Fix Identification

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 fea-

ture values. These feature values are then fed to the learned model as described in Section 4.3.3. Based on these features, the model then assigns either one of the following two labels to a particular commit: bug-fixing patch or non bug-fixing patch. [8]

4.4 Evaluation

4.4.1 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,[9] released in February 2007 and maintained until August 2007, and Linux 2.6.27,[10] released in October 2008 and maintained until December 2010. We have taken only those patches that refer somewhere to C code, and where the 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,[11] which was most actively used prior to 2009, and the open source tools Coccinelle [60] and Smatch,[12] which have been most actively used since 2008 and 2009, respectively [62]. The Coverity patches are collected by searching for patches that mention Coverity in the log message. The Coccinelle and Smatch patches are

[8] We use the analogy of black, white and grey in the remaining parts of the paper
[9] http://www.kernel.org/pub/scm/linux/kernel/git/stable/linux-2.6.20.y
[10] http://www.kernel.org/pub/scm/linux/kernel/git/stable/linux-2.6.27.y
[11] http://www.coverity.com
[12] http://smatch.sourceforge.net

Table 4.2: Properties of the considered black datasets. LOC refers to the complete patch size, including 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 4.3: Properties of the considered grey dataset, broken down by Linux version. LOC refers to the complete patch size, including both the log and the changed code.

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

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 is somewhat impure, in that it contains some patches that also represent simple refactorings, as Coccinelle targets such changes as well as bug fixes. 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. These are also taken from the complete set of patches between April 2005 and August 2011. Table 4.2 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 version 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 4.3 summarizes various properties of the data sets.

4.4.2 Research Questions & Evaluation Metrics

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

RQ1: *Is our approach effective in identifying bug fixing patches as compared to the existing keyword-based method?*

We evaluate the effectiveness of our approach as compared with existing keyword-based method. We consider the following two criteria:

Criteria 1: Precision and Recall on Sampled Data. We randomly sample 500 commits and manually assign labels to them, i.e., each commit is labeled as being either a bug fixing patch or not. We compare human assigned labels with the labels assigned by each bug fix identification approach, and compute the associated precision and recall to evaluate the effectiveness of the approach [74].

Criteria 2: Accuracy on Known Black Data. We take commits that have been identified by Linux developers as bug fixing patches. We split this dataset into ten 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.

In the first criteria, our goal 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 these sampled data points. In the second criteria, we would like 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 varying the parameter k on the results?*

Our algorithm takes in 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 this value k to be 0.9 $\times$ the number of "black" data that are known bug fixing patches. We would like to vary this number and investigate its impact on the performance.

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 [13] of all the features that we have. A variant of Fisher score reported in [9] and implemented in LibSVM[13] is computed. Fisher score and its variants have been frequently used to identify important features [8].

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 solve this problem, researchers in the machine learning community have investigated semi-supervised learning solutions. Many of these techniques still required a number of negatively labeled data points. However, LPU [45], 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 or whether our hybrid approach could

[13] `http://www.csie.ntu.edu.tw/~cjlin/libsvm/`

Table 4.4: Precision and Recall Comparison

Approach	Precision	Recall
Ours	0.601	0.875
Keyword	0.613	0.603

improve the results of LPU.

4.5 Evaluation Results & Discussion

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

4.5.1 Effectiveness of Our Approach

We compare our approach to the keyword-based approach used in the literature [32, 54]. The result of the comparisons using the two criteria are discussed below.

Precision and Recall on Sampled Data. The precision and recall of our approach as compared to those of the keyword-based approach are shown in Table 4.4. We notice that 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 0.603 to 0.875; this is a relative improvement of 45.11%.

To combine precision and recall, we also compute the F-measure [74], which is a harmonic mean of precision and recall. The F-measure is often used as a unified measure to evaluate whether an improvement in recall outweighs the reduction in precision (and vice versa). The F-measure has a parameter β that measures the importance of precision over recall. The formula is:

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

In the case that precision and recall are equally important β is set to one. This would compute what is known as F1. If β is set higher than 1 then recall is preferred

Table 4.5: F-Measures Comparison

Approach	**F1**	**F2**	**F3**	**F5**
Ours	0.712	0.802	0.837	0.860
Keyword	0.608	0.605	0.604	0.600
Rel.Improvement	17.11%	32.56%	38.58%	43.33%

Table 4.6: Comparison of $Accuracy^{Black}$ Scores

Approach	**Accuracy**^{Black}
Ours	0.945
Keyword	0.772

over precision; similarly, if β is set lower than 1 then precision is preferred over recall.

In the setting 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). This is the case as missing bug fixing patch could potentially cause system errors and even expose security holes. There are also other studies that recommend setting β equal to 2, e.g. [82].

In Table 4.6 we also compute the different F-measures using different values of β. We notice that for all values of β our approach has better results as compared to those of keyword-based approach. The F1, F2, F3, and F5 scores are improved by 17.11%, 31.56%, 38.58%, and 43.33% respectively.

From the 500 randomly sampled commits, we notice that a very small number of commits that are bug fixing patches contains a reference to Bugzilla. Thus, identifying bug fixing patches is not trivial. Also, as shown in Table 4.4, about 40% of bug fixing patches do not contain the keywords considered in previous work [54, 32].

Accuracy on Known Black Data. Table 4.6 shows the $Accuracy^{Black}$ score of our approach as compared to that of keyword-based approach.

From the result, we note that our approach can increase $Accuracy^{Black}$ from 0.772 to 0.945, a 22.4% increase. The above results show that our approach is effective in identifying bug fixing patches as compared to the keyword-based approach used in existing studies.

Table 4.7: Effect of Varying k on Performance. TP = True Positive, FN = False Negative, FP = False Positive, TN = True Negative.

k	**TP**	**FN**	**FP**	**Prec.**	**Recall**	**F2**
0.75	176	8	186	0.486	0.957	0.801
0.80	172	12	166	0.509	0.935	0.801
0.85	168	16	146	0.535	0.913	0.800
0.90	161	23	107	0.601	0.875	0.802
0.95	133	51	68	0.662	0.723	0.710

The known black data is unbiased as we do not label it ourselves. However, we can not compute the number of false positives, as all our test data are black.

The high accuracy of the keyword-based approach is due to the large number of Bugzilla patches in our 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 fixing related commits.

4.5.2 Effects of Varying Parameter k

When we vary the parameter k (as a proportion of the number 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 4.7.

We notice that as we increase the value of k the number of false negatives (FN) increases, while the number of false positives (FP) decreases. 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 "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 would reduce the number of cases where "white" data are labeled "black" (i.e., a reduction in false positives). We illustrate this in Figure 4.5.

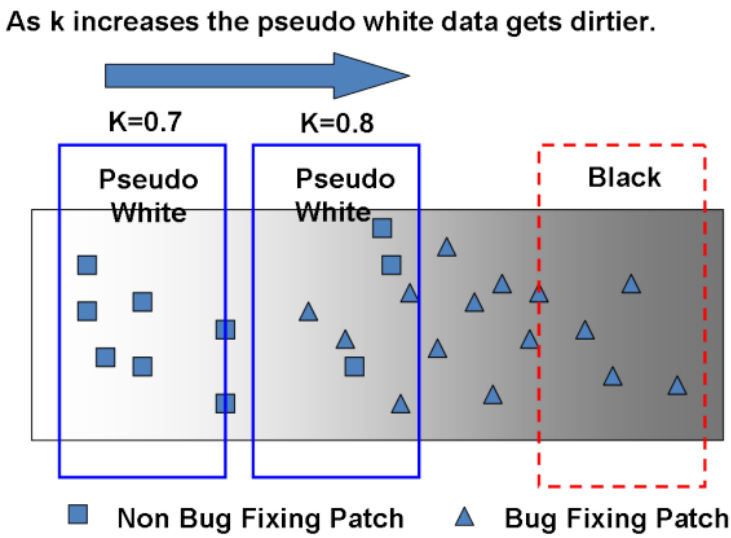

Figure 4.5: Effect of Varying K. The pseudo white data is the bottom k commits that we treat as a proxy to non bug fixing patches. The three boxes corresponding to pseudo white (2 of them) and black represent the aggregate features of the respective pseudo-white and black data in our training set respectively. The squares and triangles represent test data points whose labels (i.e., bug fixing patches or not) are to be predicted.

4.5.3 Best Features

We report the top 20 features sorted based on their Fisher scores in Table 4.8. We note that among the top-20 features there are both textual and code features. This highlight 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 [54, 32, 81], 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.

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 7th). Next include features such as the number of lines added and deleted (ranked 11th), the number of

Table 4.8: Top-20 Most Discriminative Features Based on Fisher Score

Rank	Feature Desc.	Fisher Score
1	http	0.030
2	blackfin	0.023
3	bug.cgi	0.021
4	show	0.019
5	fix	0.015
6	bugzilla.kernel.org	0.014
7	F_18 (i.e., # lines removed)	0.014
8	commit	0.013
9	upstream	0.012
10	unifi	0.012
11	F_{20} (i.e., # lines added & removed)	0.011
12	id	0.011
13	F_{38} (i.e., # boolean operators removed)	0.011
14	checkpatch	0.011
15	F_{44} (i.e., # assignments removed)	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)	0.009
20	F_{48} (i.e., # function calls added)	0.008

boolean operators added (ranked 13th), the number of assignments removed (ranked 15th), 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).

4.5.4 Our Approach versus LPU

We have run LPU on our dataset and found that the results of using LPU alone are not good. The comparison of our results and LPU alone is shown in Table 4.9.

We notice that the precision of LPU is slightly higher than that of our approach; however, the reported recall is much less than ours. 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 78%, 151.4%, 179.0%, and 197.6% for F1, F2, F3, and F5 respectively.

The $accuracy^{Black}$ of our approach and that of LPU alone is comparable. Notice

Table 4.9: Comparisons with LPU

Approach	**Precision**	**Recall**	**F1**
Ours	0.601	0.875	0.712
LPU Only	0.650	0.283	0.400
Rel.Improvement	-7.5%	209.2%	78%

Approach	**F2**	**F3**	**F5**	**Accuracy**^{Black}
Ours	0.802	0.837	0.860	0.944
LPU Only	0.319	0.300	0.289	0.942
Rel.Improvement	151.4%	179.0%	197.6%	0.2%

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 is 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.

4.5.5 Threats to Validity

Threats to internal validity relate to the relationship between the independent and dependent variables in the study. One relevant threat to internal validity in our study is experimenter bias. In the study, we have personally labeled each commits as a bug fixing patch or as a non bug fixing patch. This labeling might introduce some experimenter bias. However, we have tried to ensure that we label the commits correctly, according to our substantial experience with Linux code [40, 60, 61]. Also, we have labeled the commits before seeing the results of our identification approach, to minimize this bias.

Threats to external validity relate 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 believe it is still a good sample size. We plan to reduce this threat to external validity in the future by investigating an even larger number of manually labeled commits. We have also only investigated patches in Linux. We believe our approach can be easily applied to identify bug fixing patches in other systems. We leave the investigation as to whether our approach remains effective for other systems as future work.

Threats to construct validity relate to the appropriateness of the evaluation criteria. We use the standard measures precision, recall, and F-measure [51] to evaluate the effectiveness of our approach. Thus, we believe there is little threat to construct validity.

4.6 Chapter Conclusion

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 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. To do so, we first extract features from the commits that describe those code changes and commit logs 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 improved recall, with a relative improvement of 45.11%.