USING DATA MINING TECHNIQUES TO IMPROVE SOFTWARE RELIABILITY

BY

ZHENMIN LI

B.E., Tsinghua University, China, 1999
M.E., Tsinghua University, China, 2001

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2006

Urbana, Illinois
Reliability has become ever important. Unfortunately, software errors continue to be frequent and account for the major causes of system failures. Further, detecting and fixing bugs is one of the most time-consuming and difficult tasks in software development. In order to facilitate the procedure, it would be highly beneficial if we can first analyze and understand the bug characteristics, and then detect the bugs automatically. The huge amount of analysis data in large software such as source code and documents, however, renders a tedious and difficult task on developers to analyze them.

This dissertation proposes a novel approach that applies data mining techniques to extract information in large software and exploit such extracted information for bug detection. Thanks to the distinguished characteristics in data mining that can efficiently handle a huge amount of data, this approach can also efficiently discover useful information from large software code and documents.

To understand the bug characteristics, this dissertation studies them from bug databases for two large and popular open source projects. The bug databases contain more than 300,000 bug reports, which is impossible to study all of them manually. In order to extract useful information from such a huge amount of data, this dissertation proposes applying text classification and information retrieval techniques to automatically classify the bugs from different dimensions, namely root causes, impacts, and software components. The study shows that this approach can help developers analyze and understand bug characteristics in large software efficiently, and facilitate testing and bug detection so as to improve reliability. Furthermore, this study has discovered several new interesting findings about bug characteristics
that can provide useful guideline for related research.

One of the findings in bug characteristic study is that semantic error is the major root cause of bugs in modern software. Semantic bugs are application specific and so it requires knowledge about the application to detect them. To address this problem, this dissertation proposes using data mining technique to automatically detect software bugs. To demonstrate this approach, this dissertation presents two automatic bug detection tools, including PR-Miner that extracts programming rules and detects violations, and CP-Miner that detects copy-pasted code and related bugs.

One of bug detection tools proposed in this dissertation for large software using data mining techniques is PR-Miner. Programs usually follow many implicit programming rules, most of which are too tedious to be documented manually by programmers. When these rules are violated, bugs can be easily introduced. Therefore, it is highly desirable to automatically extract such rules and also to automatically detect violations. Previous work in this direction focuses on simple function-pair-based programming rules and additionally requires programmers to provide rule templates. PR-Miner uses frequent itemset mining to efficiently extract implicit programming rules from large software code, requiring little effort from programmers and no prior knowledge of the software. Benefiting from data mining, PR-Miner can extract programming rules in general forms that can contain multiple program elements of various types. In addition, this dissertation also proposes an efficient algorithm to automatically detect violations to the extracted programming rules, which are strong indications of bugs. The evaluation with large software code shows that PR-Miner can efficiently extract thousands of general programming rules and detect violations within minutes. Moreover, PR-Miner has detected many violations to the extracted rules, which are potential bugs.

To further demonstrate the approach, this dissertation proposes another tool to identify copy-pasted code and detect related bugs. Copy-pasted code is very common in large software because programmers prefer reusing code via copy-paste in order to reduce programming
effort. However, copy-pasting is prone to introducing bugs. Unfortunately, it is challenging to efficiently identify copy-pasted code in large software. Existing copy-paste detection tools are either not scalable to large software, or cannot handle small modifications in copy-pasted code. Furthermore, few tools are available to detect copy-paste related bugs. In order to address these problems, this dissertation proposes CP-Miner that uses frequent sequence mining to efficiently identify copy-pasted code in large software, and detects copy-paste related bugs. In order to further understand copy-paste in system software, this dissertation also analyzes some interesting characteristics of copy-paste in Linux and FreeBSD.
To my parents, Mingying and Guanying
Acknowledgments

This dissertation would not have been possible without the support of many people.

My first and foremost thank goes to my dear adviser, Yuanyuan Zhou, for her support, patience, and encouragement throughout my graduate studies. She has always been my major source of inspiration and motivation. Her dedication and diligence has set a long-lasting role model for me. Most importantly, she has taught me more than a lot ever, not only research also career and life, since I started working with her four years ago.

I would like to thank the other members of my dissertation committee, Dr. Jiawei Han, Dr. Ravi Iyer, Dr. James Larus, and Dr. Chengxiang Zhai, for their invaluable guidance and comments.

Thanks also go to all members in the Opera group, with so many talented graduate students and alumni. Special thanks to Zhifeng Chen, Shan Lu, Feng Qin, Lin Tan, Pin Zhou, and Qingbo Zhu, whom I happily collaborated with in several projects through my five years of graduate study and research. They offered me delightful discussions and unremitting assistance. I would also like to thank Weihang Jiang, Xiao Ma, Joseph Tucek and Spiros Xanthos, who inspired me and provided valuable comments on my projects and papers.

I would also like to express my gratitude to Professor Sarita Adve, Professor Jennifer Hou, Professor Darko Marinov and Professor Josep Torrellas, who provided me with insightful comments and constructive criticisms on my research work. I am grateful to them for teaching me how to do research and enforce strict validations for each research result.

I could not thank enough my first and most important teachers in my life, my dearest parents. They used to and will always support me no matter what I have chosen to do. As
a small token of my love for them, I dedicate this dissertation to my parents. I would also like to thank my brothers, who always encourage me and take care of my parents in China and compensate a lot for my absence.
# Table of Contents

**List of Tables** ......................................................... xii

**List of Figures** ...................................................... xiii

**Chapter 1 Introduction** ............................................. 1

1.1 Characteristics of Software Bugs .................................. 2

1.1.1 Things Have Changed .............................................. 2

1.1.2 New Findings ..................................................... 5

1.2 Detecting Software Bugs ........................................... 6

1.2.1 Detecting Semantic Bugs ....................................... 6

1.3 Mining Information from Large Software ......................... 9

1.3.1 Bug Classification ............................................... 9

1.3.2 Mining for Programming Rules ................................. 10

1.3.3 Mining for Copy-pasted Code and Related Bugs .......... 11

1.4 Dissertation Contributions ....................................... 12

1.5 Outline ............................................................... 14

**Chapter 2 Background of Data Mining** .......................... 15

2.1 Frequent Itemset Mining ........................................... 15

2.2 Frequent Sequence Mining ........................................ 16

**Chapter 3 Empirical Study of Bug Characteristics** .......... 19

3.1 Methodology ........................................................ 19

3.1.1 Bug Sources ..................................................... 19

3.2 Bug Classification and Analysis .................................. 21

3.2.1 Bug Categories ................................................ 21

3.2.2 Automatic Classification ...................................... 22

3.2.3 Studying the Trend ............................................. 25

3.2.4 Studying the Complexity of Fixing ......................... 25

3.2.5 Measuring the Correlation ................................... 26

3.3 Root Cause Analysis ............................................... 26

3.4 Impact Analysis .................................................... 29

3.5 Bugs in Different Components .................................... 31

3.6 Automatic Classification ......................................... 32

3.7 Difficulty of Bug Fixing ......................................... 34
### 3.8 Security Related Bugs

- Page: 36

### 3.9 Concurrency Bugs

- Page: 39

### 3.10 Summary

- Page: 40

### Chapter 4 PR-Miner

- Page: 43

#### 4.1 Overview

- Page: 43
  - 4.1.1 Programming Rules
  - Page: 43
  - 4.1.2 Contributions of PR-Miner
  - Page: 47

#### 4.2 Design of PR-Miner

- Page: 49
  - 4.2.1 Main Idea
  - Page: 49
  - 4.2.2 Extracting Programming Patterns
  - Page: 52
  - 4.2.3 Generating Programming Rules
  - Page: 56
  - 4.2.4 Detecting Violations to Extracted Rules
  - Page: 60

#### 4.3 Evaluation

- Page: 64
  - 4.3.1 Experiment Setup
  - Page: 64
  - 4.3.2 Extracting Implicit Programming Rules
  - Page: 64
  - 4.3.3 Detecting Violations
  - Page: 67
  - 4.3.4 Time and Space Overheads
  - Page: 70

#### 4.4 Summary

- Page: 71

### Chapter 5 CP-Miner

- Page: 73

#### 5.1 Overview

- Page: 73
  - 5.1.1 State of the Art
  - Page: 76
  - 5.1.2 Contributions of CP-Miner
  - Page: 76

#### 5.2 Background in Identifying Copy-pasted Code

- Page: 78

#### 5.3 Design of CP-Miner

- Page: 79
  - 5.3.1 Identifying Copy-pasted Code
  - Page: 79
  - 5.3.2 Detecting Copy-paste Related Bugs
  - Page: 88

#### 5.4 Methodology

- Page: 90

#### 5.5 CP-Miner Basic Results

- Page: 92
  - 5.5.1 Overall Results
  - Page: 92
  - 5.5.2 False Alarms
  - Page: 94
  - 5.5.3 Time and Space Overheads
  - Page: 95
  - 5.5.4 Comparison with CCFinder
  - Page: 96
  - 5.5.5 Effects of Threshold Settings
  - Page: 96
  - 5.5.6 Another Bug Example
  - Page: 98

#### 5.6 Statistics of Copy-paste in OS code

- Page: 98
  - 5.6.1 Copy-pasted Segment Size and Granularity
  - Page: 99
  - 5.6.2 Modifications in Copy-pasted Segments
  - Page: 102
  - 5.6.3 Copy-pasted Code across Modules
  - Page: 102
  - 5.6.4 Copy-paste Evolution
  - Page: 106

#### 5.7 Summary

- Page: 107
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Categories of three dimensions</td>
<td>22</td>
</tr>
<tr>
<td>3.2</td>
<td>Subcategories of memory and semantic bugs</td>
<td>23</td>
</tr>
<tr>
<td>3.3</td>
<td>Correlation between root causes and impacts in Mozilla</td>
<td>30</td>
</tr>
<tr>
<td>3.4</td>
<td>Correlation between root causes and software components in Mozilla</td>
<td>31</td>
</tr>
<tr>
<td>3.5</td>
<td>Distribution of root causes based on automatic classification</td>
<td>33</td>
</tr>
<tr>
<td>3.6</td>
<td>Distribution of impacts based on automatic classification</td>
<td>33</td>
</tr>
<tr>
<td>3.7</td>
<td>Correlation between root causes and severity/impacts for security related bugs</td>
<td>38</td>
</tr>
<tr>
<td>4.1</td>
<td>Example of parsing a function</td>
<td>55</td>
</tr>
<tr>
<td>4.2</td>
<td>Software evaluated in our experiments</td>
<td>64</td>
</tr>
<tr>
<td>4.3</td>
<td>The number of closed rules extracted by PR-Miner</td>
<td>65</td>
</tr>
<tr>
<td>4.4</td>
<td>Violations detected by PR-Miner</td>
<td>68</td>
</tr>
<tr>
<td>4.5</td>
<td>Execution time and memory space of PR-Miner</td>
<td>71</td>
</tr>
<tr>
<td>5.1</td>
<td>Identifier mapping in the example in Figure 5.1</td>
<td>88</td>
</tr>
<tr>
<td>5.2</td>
<td>Software evaluated in our experiments</td>
<td>90</td>
</tr>
<tr>
<td>5.3</td>
<td>The number of copy-pasted segments and $CP_{Coverage}$</td>
<td>92</td>
</tr>
<tr>
<td>5.4</td>
<td>Errors reported by CP-Miner</td>
<td>94</td>
</tr>
<tr>
<td>5.5</td>
<td>Execution time and memory space of CP-Miner</td>
<td>95</td>
</tr>
<tr>
<td>5.6</td>
<td>$CP_{Coverage}$ comparison between CP-Miner and CCFinder</td>
<td>96</td>
</tr>
<tr>
<td>5.7</td>
<td>Distribution of copy-paste granularity</td>
<td>102</td>
</tr>
<tr>
<td>5.8</td>
<td>Copy-paste code within a module and across modules</td>
<td>105</td>
</tr>
</tbody>
</table>
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>A semantic bug in Linux 2.6.11</td>
<td>7</td>
</tr>
<tr>
<td>3.1</td>
<td>Distribution of root causes with impacts</td>
<td>27</td>
</tr>
<tr>
<td>3.2</td>
<td>Distribution of impacts</td>
<td>29</td>
</tr>
<tr>
<td>3.3</td>
<td>Distribution of bugs in software components</td>
<td>31</td>
</tr>
<tr>
<td>3.4</td>
<td>Trend of bugs with different root causes</td>
<td>34</td>
</tr>
<tr>
<td>3.5</td>
<td>Patch size, response time and resolution time in Mozilla</td>
<td>35</td>
</tr>
<tr>
<td>3.6</td>
<td>Security related bugs collected from NVD</td>
<td>37</td>
</tr>
<tr>
<td>3.7</td>
<td>Concurrency bugs in Mozilla</td>
<td>39</td>
</tr>
<tr>
<td>4.1</td>
<td>An example of function-pair rule</td>
<td>43</td>
</tr>
<tr>
<td>4.2</td>
<td>An example for complex programming rule</td>
<td>44</td>
</tr>
<tr>
<td>4.3</td>
<td>An example of variable correlation</td>
<td>45</td>
</tr>
<tr>
<td>4.4</td>
<td>An example of a programming rule involving multiple functions</td>
<td>46</td>
</tr>
<tr>
<td>4.5</td>
<td>Flowchart of PR-Miner</td>
<td>52</td>
</tr>
<tr>
<td>4.6</td>
<td>Algorithm \texttt{ClosedRules}</td>
<td>59</td>
</tr>
<tr>
<td>4.7</td>
<td>Checking the call paths for pruning false violations</td>
<td>63</td>
</tr>
<tr>
<td>4.8</td>
<td>Distribution of rule support in the Linux code</td>
<td>66</td>
</tr>
<tr>
<td>4.9</td>
<td>Distribution of rule size in the Linux code</td>
<td>67</td>
</tr>
<tr>
<td>4.10</td>
<td>Effects of parameter settings on violation detection</td>
<td>69</td>
</tr>
<tr>
<td>5.1</td>
<td>An example of a copy-paste related error detected by CP-Miner</td>
<td>74</td>
</tr>
<tr>
<td>5.2</td>
<td>Flowchart of identifying copy-pasted segments</td>
<td>81</td>
</tr>
<tr>
<td>5.3</td>
<td>An example of hashing statements</td>
<td>83</td>
</tr>
<tr>
<td>5.4</td>
<td>Effects of \texttt{min_size} on \texttt{CP_Coverage}</td>
<td>97</td>
</tr>
<tr>
<td>5.5</td>
<td>Effects of \texttt{UnchangeRatioThreshold} on errors reported</td>
<td>97</td>
</tr>
<tr>
<td>5.6</td>
<td>Another example of copy-paste-related error detected by CP-Miner</td>
<td>99</td>
</tr>
<tr>
<td>5.7</td>
<td>Size distribution of copy-pasted segments</td>
<td>100</td>
</tr>
<tr>
<td>5.8</td>
<td>Copy-paste group size distribution</td>
<td>101</td>
</tr>
<tr>
<td>5.9</td>
<td>Distribution of identifiers changed in copy-pasted segments</td>
<td>103</td>
</tr>
<tr>
<td>5.10</td>
<td>Copy-pasted code in different modules</td>
<td>104</td>
</tr>
<tr>
<td>5.11</td>
<td>Copy-pasted code in Linux and FreeBSD through various versions</td>
<td>106</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Reliability has become increasingly important, especially for server applications that require high availability. System failures can degrade system performance, crash systems and corrupt important data, which would significantly reduce system availability and lead to huge loss in productivity and business. According to a report by Gartner Group the average cost of an hour of downtime for a financial company is more than $6 million \cite{sco98}. Software bugs in critical systems have caused airplane crashes, shut down nuclear reactors, and resulted in many other disasters \cite{der05, huc04}.

Unfortunately, software bugs continue to be frequent. A number of studies \cite{gra86, gra90, li92, sco99, ogp03, ogp03, vi84} on different types of systems have shown that software bugs are one of the major causes of system failures. For Tandem systems, Gray’s report in 1986 shows that software bugs caused 25% of system failures \cite{gra86}. A report in 2000 shows that software errors account for more than 40% of system failures \cite{ms00}. According to the National Institute of Standards and Technology (NIST), software bugs cost the U.S. economy about $59.5 billion annually, approximately 0.6% of the gross domestic product \cite{nat02}. In order to improve software quality, most software companies put a lot of effort on software testing and debugging. An exploratory study shows that the attempts to reduce the number of delivered errors are estimated to account for 50–80% of the development and maintenance effort \cite{ves85}.
1.1 Characteristics of Software Bugs

To design effective tools for improving software quality requires a good understanding of software error characteristics in representative software. Such characteristics include bug root causes, impact, resolution time, and correlations among them. For example, if many bugs are caused by simple typos or copying-and-pasting, software development tools can provide more support to help detect these automatically. In testing, developers can focus on bugs based on the severity of the impact so that resource can be utilized more effectively [RUCH99].

Many previous empirical studies, including a few classic ones [BP84, CKC91, End75, Gla81, OW84, SC91, SC92, TLN78], have been performed more than ten years ago to understand the characteristics of software bugs. For example, several researchers [End75, Gla81, TLN78] have studied software errors occurring during software development, testing and validation phases. Sullivan and Chillarege [SC91, SC92] analyzed error type, defect type and error trigger distribution for shipped code of two IBM database management products and one IBM operating system. They found that undefined state errors dominated but did not have high impact on availability, while memory allocation errors, pointer errors, and synchronization errors had high impact. These studies provide useful insights and guidelines for software engineering tool designers and reliable system builders.

1.1.1 Things Have Changed

Over the last ten years, however, many factors in software development have significantly changed. As most previous studies were conducted using old software that was developed under the old development environment (e.g. without modern debugging tools) with traditional development paradigms and software architectures, it is unclear whether their results still apply. In addition, rising issues such as security concerns and multithread-related problems are not well studied in previous work.

Specifically, the following changes in software and development motivate a new study
with modern software to answer those newly rising issues as well as to provide new answers to old questions:

- **More Effective Modern Debugging Tools:** In order to facilitate debugging, quite a few bug detection tools such as Purify [HJ92] and Valgrind [NS03] have been recently proposed and widely used. An interesting question is whether they are helpful in minimizing the number of memory bugs in released code. To answer this question requires a new empirical study of bugs from some modern software developed after those debugging tools became available around 1997. If these tools become a standard practice in software testing and validation, there should be fewer memory bugs in modern software than that reported before for traditional software, and the diagnosis time for these types of bugs should be much shorter than that for other types of bugs. If the empirical results with modern software indicate otherwise, it may imply that these tools are not effectively used by programmers due to either too many false positives or to other reasons. Furthermore, if the major root cause of software bugs has changed, what kind of detection tools do we need now?

- **Software Architecture Shift:** Due to the recent advance in hardware, many modern computer systems, especially server systems, are all configured with multi-processors. As a result, many software architectures are multi-threaded or multi-processed to exploit the parallelism provided by hardware. These trends will continue especially since, with the continuous shrinking of transistor size and increasing chip area density, multithreaded/multicore (multiple processors on one chip or CMP) architecture is becoming a mainstream technology. For example, the Intel Pentium processor has already provided hyper-threading and dual-core capabilities. Multicore technology is also embraced by other chip vendors including AMD, IBM and Sun. To exploit these technologies, increasingly more software is becoming multi-threaded. An interesting question is whether modern software has more concurrency bugs due to this software
architecture shift? Do these bugs cause severe impacts on systems?

- **Emphasis on User-Friendly Interface**: In order to provide friendly user interface, graphical user interfaces (GUIs) have become one of the major components in many systems. Although GUIs have become more complex and widely used, GUI testing techniques still significantly lag behind [Mem02]. Therefore, GUI-related bugs may have become more pervasive and dominant. Further, since GUI modules and their development process are quite different from other modules, do they have different root causes?

- **Rising Security Concerns**: Over recent years, security is becoming increasingly important as many malicious users exploit software vulnerabilities to tamper system integrity, steal confidential data, and make systems unavailable [Cow03, Pay02]. Unfortunately, previous work has not addressed how software errors affect system security. It is unclear how many reported bugs are related to security, and what types of security-related bugs there are, and how fast these security-related bugs are fixed.

- **New Software Development Paradigm**: Recently more and more software is developed using the open source paradigm that allows programmers from the Internet to read, redistribute, and modify the source code. For quality control, most open source software (OSS) projects usually allow only a small set of experienced developers to check code changes into the main branch of the software. The OSS development paradigm enables software to evolve at a much faster pace. Some preliminary experiences seem to indicate that this process produces better software than the traditional closed model, in which only a few programmers can see the source and everybody else can only use the binary code. This is why the OSS paradigm is also endorsed by many industrial companies including IBM and Sun. For example, recently Sun’s Solaris source code has also become open to public. An interesting question is whether OSS actually takes a much shorter time to fix bugs.
1.1.2 New Findings

To understand the effects of the above new factors on software errors, it is necessary to perform an empirical study on bugs in modern software. To this end, this dissertation analyzes bug characteristics in two large and popular OSS projects, Mozilla and Apache Web Server, each of which contains up to 4 million lines of code and about 90 release versions developed over the last 8–10 years. The bugs are first classified in three dimensions, root causes, impacts and software components. Further, this study also analyzes the statistical correlations among these dimensions, which has never been systematically studied before. In addition, this study also analyzes hundreds of security related bugs and concurrency bugs in order to understand the characteristics of these emerging types of bugs.

The study on these two large projects shows that memory bugs only account for 12.2–16.3%, much less than the 28–38% reported in previous studies [SC91, SC92], indicating that memory bugs are becoming less pervasive due to the available techniques to automatically detect them. In contrast, semantic bugs are the major root causes, accounting for 81.1–86.7%, and their percentages increases with the maturity of software. Moreover, such semantic bugs also have severe impacts on system availability as memory related bugs, contributing to 42.9–44.2% of crashes. Furthermore, it takes a longer time to diagnose and fix semantic bugs, almost twice as memory related bugs. Our results suggest that more effort should be put into automatically detecting and diagnosing semantic bugs.

Additionally, careless programming causes many bugs. For example, some simple bugs such as typos account for 9.4–9.8% of semantic bugs. Although their fixes are usually simple, it takes a longer time to diagnose them than to diagnose memory bugs, which suggests that software development environments such as Microsoft Visual Studio and Eclipse should provide more support to avoid them in the early stages of software development.
1.2 Detecting Software Bugs

Identifying and fixing software bugs is one of the most time-consuming and difficult tasks to reduce the errors in software. In order to improve programmers’ efficiency, quite a few debugging techniques and tools have been proposed recently. The approaches can be classified into two categories: dynamic checking and static checking. Some examples of dynamic tools are Purify [HJ92], Valgrind [NS03], DIDUCE [HL02], Eraser [SBN+97], and CCured [NMW02]. They have more accurate information but can introduce overheads during execution. Moreover, most of them can only find bugs on the execution paths, so they depend on good test cases that are usually hard to provide. In contrast, static tools, including explicit model checking [ECC01, MPC02, SD95] and program analysis [CLL02, EA03, HCXE02], do not incur any overheads during execution. Further, they can find bugs that may not occur in the common execution paths.

However, most static tools require significant involvement of programmers to build checking models, write specifications, and annotate programs. It is difficult to build good checking models and specifications since it requires that programmers understand the target system thoroughly. Further, it is a tedious and time-consuming task for the programmers to write such specifications and annotations. Therefore, most of the existing bug detection tools are based on some general checkers that are applicable to any applications. For example, a program should never dereference a null pointer; a program should release memory that it no longer needs. Based on such well-known rules, quite a few memory-related bug detection tools have been proposed, which are quite effective in detecting the well-known bugs such as null pointer dereference, memory leak, buffer overrun, and so on.

1.2.1 Detecting Semantic Bugs

Unfortunately, the existing detection tools with such checkers can detect only a small fraction of bugs in modern software. The root causes of most bugs cannot be detected because
the dominant bugs are application specific. The empirical study of bug characteristics in Chapter 3 shows that memory related bugs in modern software are not the major ones now, accounting for only 16.3% in Mozilla and 12.2% in Apache; in contrast, the major root causes are semantic errors such as missing features, missing cases, wrong control flows, typos, and some other wrong functionality implementation, which account for 81.1% and 86.7% in total in Mozilla and Apache, respectively. Most of the semantic bugs violate the rules that should be obeyed only in a specific application. For example, in Linux, the correct procedure to probe and initialize a SCSI device includes three steps: first call `scsi_host_alloc` to allocate a data structure for the device driver, then call `scsi_add_host` to register the device with the SCSI stack, and finally scan the host by `scsi_scan_host`. Any violation of this procedure would cause potential errors. Figure 1.1 shows the bug missing the last step `scsi_scan_host` in the latest version of Linux. This bug cannot be detected by the existing tools without the knowledge of SCSI device drivers in Linux.

Therefore, in order to detect semantic bugs, the major ones in modern software, the checkers should be adaptive to the targeted applications. It indicates that it is necessary to obtain specifications for different applications instead of using general checkers. However, it is too tedious and time-consuming for programmers to write such specifications for different applications. It would be desirable to automatically generate such application-specific
checking models and specifications.

**Specification Generation**  A few recent studies have proposed to extract information for specification generation from source code [ABL02, ECC01, KTB+06a]. The work by Engler, Chen and Chou [ECC01] has conducted a preliminary investigation in this direction. They proposed a method to extract programmer MUST and MAY beliefs from source code and then use them to check for contradictions that indicate potential bugs. MUST beliefs are always held in programs, which mostly come from the general rules aforementioned in any applications. On the other hand, MAY beliefs are only observed in source code of a specific application and they may not be true due to coincidence. Since programmers should follow them as rules in programming, we call them **programming rules** in this dissertation. To extract the programming rules, Engler et al. used some programmer-specified rule templates such as “function \(a\) must be paired with function \(b\)”\(^\r\). The two arguments, \(a\) and \(b\), will be fit by passing all plausible pairs that are selected based on statistical analysis and weighted by naming conventions such as the substrings \(\text{“lock”}\) and \(\text{“unlock”}\). Therefore, the extracted rules are restricted by the templates and the weights also come from the rules for general applications, which discriminate the application-specific factors. For example, the programming rule violated by the bug in Linux shown in Figure 1.2 contains three function calls and one related variable, so it cannot be extracted by the function-pair templates, and the function calls would get a low weight by naming conventions.

While the work above is inspiring and proposes a promising direction to generate specifications from source code, it extracts only the programming rules restricted by templates that depend on some apriori knowledge from programmers. It is not effective enough to extract information from source code and detect application-specific bugs. Therefore, it is necessary to propose some general approaches in order to extract the information for specification generation and bug detection *automatically* from large software with little human effort.
In order to address the issues above, this dissertation project focuses on two challenges: how to automatically extract application-specific information for bug detection from large software document and source code, and how to apply such information to bug detection efficiently.

1.3 Mining Information from Large Software

The huge amount of analysis data in large software such as source code and documents, however, renders a tedious and difficult task on developers to analyze them. In order to extract information from such large software and detect bugs, this dissertation proposes to apply data mining techniques for analysis of source code and documents. This approach is the first one to apply data mining techniques in bug detection.

Data mining, also known as knowledge discovery in databases (KDD), has been developed at a rapid pace in recent years due to the wide availability of voluminous data and the imminent need for extracting useful information and knowledge from them. Traditional methods of data analysis dependent on human handling cannot scale to huge sizes of data sets. In contrast, data mining techniques are suitable and efficient to extract information from large amount of data sets.

1.3.1 Bug Classification

Most previous empirical studies on bug characteristics were based on a small set (usually with 60–500) of software bugs for the entire software revolution, which may result in a large experimental error and misleading results. For example, some bugs may not be included in the sampled set but could account for a significant fraction in the whole bug database of the evaluated software. Moreover, small size datasets also make it difficult to study the bug trend with the software evolution because some time intervals under studied may only contain very few or no bug samples.
Fortunately, most of the open-source projects maintain bug tracking systems to track bugs and code changes. Some of them contain hundreds of thousands of bug reports, which are a valuable asset for us to understand the bug characteristics. However, it is challenging to obtain useful information from such large datasets written in human natural languages. For example, Bugzilla database for Mozilla contains more than 300,000 bug reports [moz05]. It is impractical to verify every bug report by manual investigation. Text mining techniques such as document categorization and information retrieval is a suitable approach to analyze such large datasets that are written in human natural language. Using text mining, it can relieve heavy human work to classify and analyze such huge amount of data.

This dissertation proposes using natural language text classification and information retrieval techniques to automatically classify a large number of bug reports. Such a large dataset enables us to provide more accurate results such as trends of bug types with software evolution and the complexity of bug fixing, which are usually difficult to draw representative and accurate results from small datasets.

### 1.3.2 Mining for Programming Rules

As discussed in Section 1.2.1, a large portion of software bugs violate the application-specific programming rules. To detect such bugs, we need to provide the knowledge about the specific applications to the checkers, but unfortunately, most of such knowledge is not well documented.

To address this problem, this dissertation proposes a new approach called PR-Miner that uses data mining techniques to extract programming rules from large software source code. The main idea to extract application-specific information from source code is to identify the frequent programming patterns existing in programs. PR-Miner first converts the source code into a database, and then apply frequent pattern mining algorithm on it so that we can find the frequent patterns that can infer programming rules. Due to the scalability of data mining algorithms, this approach can perform efficiently on large software. The results
show that PR-Miner can efficiently extract useful information for bug detection, and can also efficiently detect application-specific bugs that cannot be detected by other existing tools.

### 1.3.3 Mining for Copy-pasted Code and Related Bugs

Copy-pasted code is very common in large software because programmers prefer reusing code via copy-paste in order to reduce programming effort. However, copy-pasting is prone to introducing bugs because errors in code can be propagated by copy-pasting and the programmer can easily make mistakes in modification after copy-pasting. For example, some recent work shows that a significant portion of operating system bugs concentrate in copy-pasted code [CYC+01].

A major reason that copy-paste is error-prone is that programmers may easily incorrectly modify or forget to modify identifiers (variables, functions, and types) consistently throughout the piece of pasted code. Such inconsistent modification also violates the intent of programmers, and therefore it can be considered as a specific type of violation to the programming rule that modification within copy-pasted code should be consistent.

Unfortunately, it is challenging to efficiently identify copy-pasted code in large software. Existing copy-paste detection tools are either not scalable to large software, or cannot handle small modifications in copy-pasted code. Furthermore, few tools are available to detect copy-paste related bugs.

To address these problems, this dissertation proposes a novel approach called CP-Miner. CP-Miner first applies data mining techniques to efficiently identify copy-pasted code in large software and then compare the copy-pasted code to detect copy-paste related bugs that contains inconsistent modification.
1.4 Dissertation Contributions

This dissertation proposes using data mining techniques for bug detection to improve software reliability. Specifically, this approach uses data mining techniques to extract useful information from source code and detect bugs, and uses text mining techniques to analyze bug reports in large software.

1. **Empirical Study of Bug Characteristics:** To understand the effects of the new factors on software bugs, we analyze bug characteristics in two large and popular OSS projects, Mozilla and Apache Web Server, each of which contains up to 4 million lines of code and about 90 release versions developed over the last 8–10 years. We first manually examine 362 randomly selected bugs from the bug databases and study the bug distribution in three dimensions, root causes, impacts and software components. Furthermore, this dissertation also studies the statistical correlations among these dimensions, which has never been systematically studied before (to the best of our knowledge). In addition, this dissertation also studies 257 security related bugs and 90 concurrency bugs to understand the characteristics of these emerging types of bugs.

To validate that the analysis results from the sampled datasets are representative, this study uses *natural language text classification* and *information retrieval* techniques to **automatically classify a large number (around 29,000)** of bug reports. Such a large dataset enables us to provide more accurate results such as trends of bug types with software evolution and the complexity of bug fixing, which are usually difficult to draw representative and accurate results from small datasets.

2. **PR-Miner:** In order to automatically extract implicit programming rules and also to automatically detect violations, some previous work focuses on simple function-pair based programming rules and additionally requires programmers to provide rule templates as we described in Section 13.2. PR-Miner uses a data mining technique
to efficiently extract implicit programming rules from large software code, requiring little effort from programmers and no prior knowledge of the software. Benefiting from frequent itemset mining, PR-Miner can extract programming rules in general forms (without being constrained by any fixed rule templates) that can contain multiple program elements of various types such as functions, variables and data types. In addition, this dissertation also proposes an efficient algorithm to automatically detect violations to the extracted programming rules, which are strong indications of bugs.

This dissertation evaluates PR-Miner with large software, including Linux, PostgreSQL Server and the Apache Web Server, with 84K–3M LOC each, shows that PR-Miner can efficiently extract thousands of programming rules and detect violations within a few minutes. Moreover, PR-Miner has detected hundreds of violations to the extracted rules. Among the top 60 violations, 16 have been confirmed as bugs in the latest version of Linux, 6 in PostgreSQL and 1 in Apache. Most of them violate complex application-specific programming rules and are thereby difficult for previous tools to detect.

3. **CP-Miner**: CP-Miner uses data mining techniques to efficiently identify copy-pasted code in large software including operating system code, and also detects copy-paste related bugs. It requires no modification or annotation to the source code of software being analyzed. It takes less than 20 minutes for CP-Miner to identify 190,000 copy-pasted segments in Linux and 150,000 in FreeBSD. Moreover, CP-Miner has detected 28 copy-paste related bugs in the latest version of Linux and 23 in FreeBSD. In addition, we analyze some interesting characteristics of copy-paste in Linux and FreeBSD, including the distribution of copy-pasted code across different length, granularity, modules, degrees of modification, and various software versions.
1.5 Outline

The remainder of this dissertation is organized as follows. Chapter 2 introduces the background of data mining used in our work. Chapter 3 describes the empirical study of bug characteristics. Chapter 4 and Chapter 5 focus on PR-Miner and CP-Miner, respectively. Chapter 6 discusses the issues in applying data mining to software reliability and the current limitations with PR-Miner and CP-Miner. Chapter 7 presents related work, and Chapter 8 summarizes this dissertation and discusses the future research direction.

The materials in some chapters have been published as conference papers or journal articles. The materials in Chapter 5 have been presented in [LLMZ04] and [LLM06], and the materials in Chapter 4 have been presented in [LZ05].
Chapter 2

Background of Data Mining

CP-Miner and PR-Miner are based on frequent pattern mining, which is an association analysis technique to discover frequent patterns in a database. Frequent mining is an active research topic in data mining. It has broad applications, including mining motifs in DNA sequences, analysis of customer shopping behavior, etc. In our work, we use two types of frequent pattern mining: frequent itemset mining and frequent sequent mining.

2.1 Frequent Itemset Mining

PR-Miner is based on frequent itemset mining. The goal of frequent itemset mining is to efficiently find frequent itemsets in a large database, where an itemset is a set of items. In a database composed of a large number of itemsets, if a sub-itemset (subset of an itemset) is contained in more than a specified threshold (called min_support) of itemsets, it is considered frequent. The number of occurrences of a sub-itemset \( A \) is denoted as its support. The itemset that contains \( A \) is called its supporting itemset. For example, in an itemset database \( D \):

\[
D = \{\{a, b, c, d, e\}, \{a, b, d, e, f\}, \{a, b, d, g\}, \{a, c, h, i\}\}
\]

The support of sub-itemset \( \{a, b, d\} \) is 3, and its supporting itemsets are \( \{a, b, c, d, e\} \), \( \{a, b, d, e, f\} \) and \( \{a, b, d, g\} \). If min_support is specified as 3, the frequent sub-itemsets for \( D \)
are \{a\}:4, \{b\}:3, \{d\}:3, \{a, b\}:3, \{a, d\}:3, \{b, d\}:3 and \{a, b, d\}:3, where the numbers are the supports of the corresponding sub-itemsets.

Frequent itemset mining can be classified into two categories: the Apriori-like approach \cite{AS94} and FP-tree-based (Frequent Pattern tree based) approach \cite{HPY00}. Compared with the Apriori-like approach that needs multiple database scans, the FP-tree-based approach only needs two database scans, and thereby is more efficient.

To solve the frequent itemset mining problem, quite a few algorithms have been proposed. PR-Miner uses a FP-tree-based mining algorithm called FPclose \cite{GZ03}, which is one of the most efficient frequent itemset mining algorithms. Instead of generating the complete set of frequent sub-itemsets, FPclose mines only the closed sub-itemsets. A closed sub-itemset is the sub-itemset whose support is different from that of its super-itemsets. In the example above, the frequent sub-itemsets \{b\}, \{d\}, \{a, b\}, \{a, d\} and \{b, d\} are not closed since their supports are the same as their super-itemset \{a, b, d\}. FPclose only generates the closed sub-itemsets \{a\}:4 and \{a, b, d\}:3 as result. This can significantly improve time and space performance since it can avoid generating exponential number of frequent sub-itemsets.

After all closed frequent sub-itemsets are mined from an itemset database, association rules can be generated. An association rule can be denoted as \(X \Rightarrow Y\) with confidence \(c\) and support \(s\), where \(X\) and \(Y\) are itemsets. The meaning of the rule is that if an itemset contains \(X\), it also contains \(Y\) with probability of \(c\) \cite{AS94, HGN00}. Association rules allow violation detection. If the confidence is very high, say \(99\%\), the itemset that contains only \(X\) but not \(Y\) violates the rule, indicating a potential outlier. Due to space limitation, we do not describe the details of the FPclose algorithm since they can be found in \cite{GZ03}.

### 2.2 Frequent Sequence Mining

Other than frequent itemset mining, frequent sequence mining is also an important topic in frequent pattern mining. The major difference is that the items have order and can
be duplicated in a transaction in the database for frequent sequence mining [AS95]. For example, the sequential pattern $xy$ is different from $yx$. Accordingly, there are also closed frequent sequence mining algorithms such as $CloSpan$ [YHA03].

Similarly, a subsequence is considered frequent when it occurs in at least $min\_support$ in the sequence database. A subsequence is not necessarily contiguous in an original sequence. A sequence that contains a given subsequence is called a supporting sequence of this subsequence.

For example, a sequence database $D$ has five sequences:

$$D = \{\text{abcd, abce, agbc, abij, aklc}\}$$

The number of occurrences of subsequence $abc$ is 4, and sequence $agbc$ is one of $abc$’s supporting sequences. If $min\_support$ is specified as 4, the frequent subsequences are $\{a: 5, b: 4, c: 5, ab: 4, ac: 5, bc: 4, abc: 4\}$, where the numbers are the supports of the subsequences.

CP-Miner uses a recently proposed frequent subsequence mining algorithm called $CloSpan$ ($Closed\ Sequential\ Pattern\ Mining$) [YHA03], which outperforms most previous algorithms. CloSpan mainly consists of two stages: (1) using a depth-first search procedure to generate a candidate set of frequent subsequences that includes all the closed frequent subsequences; and (2) pruning the non-closed subsequences from the candidate set. The computational complexity of CloSpan is $O(n^2)$ if the maximum length of frequent sequences is constrained by a constant.

There are two main ideas in CloSpan to improve the mining efficiency. The first idea is based on an obvious observation that if a sequence is frequent, then all of its subsequences are frequent. For example, if a sequence $abc$ is frequent, all of its subsequences $\{a, b, c, ab, ac, bc\}$ are frequent. CloSpan recursively produces a longer frequent subsequence by concatenating every frequent item to a shorter frequent subsequence that has already been obtained in the
previous iterations.

To better explain this idea, let us consider an example. Let \( L_n \) denote the set of frequent subsequences with length \( n \). In order to get \( L_n \), we can join the sets \( L_{n-1} \) and \( L_1 \). For example, suppose we have already computed \( L_1 \) and \( L_2 \) as shown below. In order to compute \( L_3 \), we can first compute \( L'_3 \) by concatenating a subsequence from \( L_2 \) and an item from \( L_1 \):

\[
L_1 = \{a, b, c\}; \\
L_2 = \{ab, ac, bc\}; \\
L'_3 = L_2 \times L_1 = \{abc, abb, abc, aca, acb, acc, bca, bcb, bcc\}
\]

For greater efficiency, CloSpan does not join the sequences in set \( L_2 \) with all the items in \( L_1 \). Instead, each sequence in \( L_2 \) is concatenated with only the frequent items in its suffix database. In our example, for the frequent sequence \( ab \) in \( L_2 \), its suffix database is \( D_{ab} = \{ced, cef, ch, ijc\} \), and only \( c \) is the frequent item, so \( ab \) is only concatenated with \( c \) and then we get a longer sequence \( abc \) that belongs to \( L'_3 \).

The second idea is used for efficiently evaluating whether a concatenated subsequence is frequent or not. It tries to avoid searching through the whole database. Instead, it only checks with certain suffixes. In the above example, for each sequence \( s \) in \( L'_3 \), CloSpan checks whether it is frequent or not by searching the suffix database \( D_s \). If the number of its occurrences is greater than \( \text{min\_sup} \), \( s \) is added into \( L_3 \), which is the set of frequent subsequences of length 3. CloSpan continues computing \( L_4 \) from \( L_3 \), \( L_5 \) from \( L_4 \), and so on until no more subsequences can be added into the set of frequent subsequences.

Recently we have used CloSpan to detect block correlations in storage systems [LCSZ04], which demonstrates that CloSpan can efficiently analyze a large amount of system trace data.
Chapter 3

Empirical Study of Bug Characteristics

Software failures greatly reduce system reliability. As software becomes more and more complex, there is an urgent need to explore more effective software testing and debugging tools and software engineering methods to minimize the number of bugs that escape into production runs. According to a report from NIST [NIST02], improvements in software testing could eliminate about $22.2 billions of business loss in US caused by software errors annually. Since some bugs still slip through even the strictest testing, system designers also need to provide fault tolerant mechanisms to recover from these inevitable software failures.

3.1 Methodology

3.1.1 Bug Sources

To discover what has changed now, we study bugs from two large widely-used new OSS projects, Mozilla and Apache HTTP Server. We choose these two projects mainly for the following reasons:

(1) They are new in two aspects. They are developed by the new open-source development paradigm and are newly developed over the recent 8 to 10 years, which is after many empirical studies were conducted. Yet, they are stable and mature, and have evolved long enough to allow for the study of bug characteristics through the software lifetime. In particular, they have been developed around 8 years after the release of many bug detection tools (around 1997) to allow us to study the impact of bug detection tools on memory bugs.
(2) They are two of the most successful OSS projects. Mozilla was rated as the best of 2003 by PCWorld [bes], Apache is the most widely-used web server, used by 71.0% of near 53 million sites polled in a survey [net05].

(3) They are large modern software applications. The Mozilla suite contains 4 million LOC and there are about 90 releases developed over the last 8 years, while Apache HTTP Server contains 310 thousand LOC and around 90 releases developed over 10 years.

(4) They have large well-maintained Bugzilla databases [apa05, moz05], which contain hundreds of thousands of well-structured bug reports reported by many users as well as developers.

Randomly Collecting Bugs In our study, we focus on the characteristics of software errors that manifest at run time, that is, excluding new feature requests, compile-time errors, configuration errors, environmental errors, and software maintenance. To ensure correct classification, we only study fixed runtime bugs whose root causes can be identified from reports because unfixed bugs may be invalid and root causes described in the reports can be wrong. In this way, we randomly select 548 fixed bug reports from the Mozilla Bugzilla database [moz05]. As not every report describes a runtime bug, we manually investigate them and narrow down to 264 runtime bugs, and then classify them manually. We study bug characteristics based on these manual labeled data, and further use them to train and evaluate automatic classifiers for the whole bug database as described in Section 3.2.2. Similarly, we randomly select 209 fixed bug reports from the Apache Bugzilla database [apa05], and then manually classify 98 runtime bugs. All bugs collected are reported since 1998 for Mozilla and since 2002 for Apache, when Bugzilla was adopted by these two applications. In the rest of this chapter, we use the general name “bugs” to refer to fixed runtime bugs.

Collecting Security Bugs We collect all of the 193 security vulnerabilities in Mozilla and all of the 64 in Apache Web Server kept in National Vulnerability Database (NVD) [NVD05b].
3.2 Bug Classification and Analysis

In this section, we first describe the categories of bug types. We then describe how to automatically classify bugs into these categories, how we study the trend and the correlation of different categories.

3.2.1 Bug Categories

We classify bugs in three dimensions, **Root Cause**, **Impact** and **Software component**. According to root causes, bugs can be classified into three disjoint categories, **Memory**, **Concurrency**, and **Semantic**, whose definition as well as the definition of impact categories, is shown in Table 3.1. Memory bug and semantic bugs are further classified into sub-categories as shown in table 3.2.

This classification allows for the study of distribution of some important types of bugs, e.g. memory, concurrency, and semantic. In addition, by comparing with the distribution of similar types of bugs in previous work [CC00, CKC91, End75, SC91, SC92], we study the change of bug characteristics in modern OSS. Combining classification with other techniques, such as, trend study, and measuring the correlation, we can answer interesting questions, such as (1) whether the number of concurrency bugs decreases and (2) what root causes lead to more crashes now. To estimate the statistical errors caused by sampling, we compute the confidence intervals with 95% confidence level for distribution of each category.

**Classifying Security Vulnerabilities** Security vulnerabilities are manually classified in three dimensions. The root cause dimension is the same as that of general bugs. Based on impact, vulnerabilities are classified into four categories, **confidentiality** (unauthorized disclosure of information), **integrity** (unauthorized modification), **availability** (disruption of service), and **access** (unauthorized access). The third dimension is the NVD severity, which contains three levels, **High**, **Medium**, and **Low**, as defined in [nvd05a].
<table>
<thead>
<tr>
<th>Dimension</th>
<th>Category</th>
<th>Description</th>
<th>Abbr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Root</td>
<td>Memory</td>
<td>Bugs caused by improper handling of memory objects.</td>
<td>Mem</td>
</tr>
<tr>
<td></td>
<td>Concurrency</td>
<td>Bugs that happen only in multi-threading (or multi-processes) environment, including data race, deadlock, and synchronization.</td>
<td>Con</td>
</tr>
<tr>
<td>Cause</td>
<td>Semantic</td>
<td>Inconsistent with the original design requirements or the programmers’ intention. We consider all bugs as Semantic bugs unless they are already classified as Memory bugs or Concurrency bugs.</td>
<td>Sem</td>
</tr>
<tr>
<td>Impact</td>
<td>Hang</td>
<td>Program keeps running but does not respond.</td>
<td>Hang</td>
</tr>
<tr>
<td></td>
<td>Crash</td>
<td>Program halts abnormally.</td>
<td>Crash</td>
</tr>
<tr>
<td></td>
<td>Data Corruption</td>
<td>Mistakenly change user data.</td>
<td>Corrupt</td>
</tr>
<tr>
<td></td>
<td>Performance Degradation</td>
<td>Functions correctly but runs/responds slowly.</td>
<td>Perf</td>
</tr>
<tr>
<td></td>
<td>Incorrect Functionality</td>
<td>Not behave as expected.</td>
<td>Func</td>
</tr>
<tr>
<td>Software Component</td>
<td>Core</td>
<td>Bugs related to core functionality implementations.</td>
<td>Core</td>
</tr>
<tr>
<td></td>
<td>GUI</td>
<td>Bugs related to graphical user interfaces.</td>
<td>GUI</td>
</tr>
<tr>
<td></td>
<td>Network</td>
<td>Bugs related to network environment and network communication.</td>
<td>Network</td>
</tr>
<tr>
<td></td>
<td>I/O</td>
<td>Bugs related to I/O handling.</td>
<td>I/O</td>
</tr>
</tbody>
</table>

Table 3.1: Categories of three dimensions  
Some categories and definitions are borrowed from BugBench [LLQ*05].

3.2.2 Automatic Classification

The whole bug databases contain hundreds of thousands of bug reports, so it may result in large statistical variances in distribution analysis by sampling only hundreds of bugs.

To verify the analysis results from the sampled datasets, we propose a novel method to automatically classify a large number of bugs, and then study bug characteristics based on such a large dataset. Specifically, we apply text classification and information retrieval techniques on the bug reports to automatically classify 29,000 bugs. Such a large dataset also enables us to perform study on trend and complexity of fixing. Our method consists of the following steps:
<table>
<thead>
<tr>
<th>Category</th>
<th>Sub-category</th>
<th>Description</th>
<th>Abbr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Bug</td>
<td>Memory Leak</td>
<td>Failures to release unused memory.</td>
<td>MLK</td>
</tr>
<tr>
<td></td>
<td>Uninitialized Memory Read</td>
<td>Read memory data before it is initialized.</td>
<td>UMR</td>
</tr>
<tr>
<td></td>
<td>Dangling Pointer</td>
<td>Pointers still keep freed memory addresses.</td>
<td>Dangling</td>
</tr>
<tr>
<td></td>
<td>NULL Pointer Dereference</td>
<td>Dereference of a null pointer.</td>
<td>NULL</td>
</tr>
<tr>
<td></td>
<td>Overflow</td>
<td>Illegal access beyond the buffer boundary.</td>
<td>Overflow</td>
</tr>
<tr>
<td></td>
<td>Double Free</td>
<td>One memory location is freed twice.</td>
<td>2Free</td>
</tr>
<tr>
<td>Semantic Bug</td>
<td>Missing Features</td>
<td>A feature is supposed to be but is not implemented.</td>
<td>MissF</td>
</tr>
<tr>
<td></td>
<td>Missing Cases</td>
<td>A case in a functionality is not implemented.</td>
<td>MissC</td>
</tr>
<tr>
<td></td>
<td>Corner Cases</td>
<td>Some boundary cases are considered incorrectly or ignored.</td>
<td>CornerC</td>
</tr>
<tr>
<td></td>
<td>Wrong Control Flow</td>
<td>The control flow is incorrectly implemented.</td>
<td>CtrlFlow</td>
</tr>
<tr>
<td></td>
<td>Exception Handling</td>
<td>Do not have proper exception handling.</td>
<td>Except</td>
</tr>
<tr>
<td></td>
<td>Processing</td>
<td>Processing such as evaluation of expressions and equations is incorrect.</td>
<td>Process</td>
</tr>
<tr>
<td></td>
<td>Typo</td>
<td>Typographical mistakes.</td>
<td>Typo</td>
</tr>
<tr>
<td></td>
<td>Other Wrong Functionality Implementation</td>
<td>Any other semantic bug that does not meet the design requirement.</td>
<td>FuncImpl</td>
</tr>
</tbody>
</table>

Table 3.2: Subcategories of memory and semantic bugs

Some definitions are borrowed from BugBench [LLQ+05] and a book [Bei90].

- **Preprocessing** In Bugzilla databases, each bug report may contain the following information: bug ID, summary, time, status, reporter, assignee, severity, bug description, discussion comments, test cases, attachments, etc. We include most of the information except time and attachments because time is irrelevant for our classification, and the major contents in attachments are source code that is hard for current classifiers to use. We represent bug documents in word level, called *bag-of-words* approach. Each word in bug documents is parsed into an index. Each bug document is represented by a vector.

- **Training** We use the manually-labeled bugs as a training set to produce classification models for different categories of bugs. We use several different classifier learning
methods, including support vector machine (SVM) [Vap95], Winnow, Perceptron, and Naive Bayes [RZ98]. We choose the best one based on their accuracy.

We randomly divide the whole sampling dataset into two halves: training set for learning and tuning, and test set for accuracy evaluation. Applying the classification methods on the training set, we can get several models for different categories. Since each method has some parameters, we explore the entire parameter space and use n-fold (n = 5) cross validation [Joa02] to find out the best method with the best parameter setting based on the accuracy metrics described in Section 3.2.2. Additionally, to avoid the errors of accuracy evaluation caused by tuning, we use only the training set for tuning and use the test set for accuracy evaluation. Therefore, the test set does not affect parameter tuning, and so accuracy evaluation based on the test set could be representative for the whole dataset.

- **Evaluating Accuracy** To evaluate how good the classification models are, we can measure the prediction accuracy. Four different types of prediction results are possible from a binary classifier:

<table>
<thead>
<tr>
<th>Actual Class</th>
<th>Predicted Class</th>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>True Positive ($T_+$)</td>
<td>False Negative ($F_-$)</td>
<td></td>
</tr>
<tr>
<td>No</td>
<td>False Positive ($F_+$)</td>
<td>True Negative ($T_-$)</td>
<td></td>
</tr>
</tbody>
</table>

We use three metrics to evaluate accuracy, which are Precision ($P = \frac{T_+}{T_+ + F_+}$), Recall ($R = \frac{T_+}{T_+ + F_-}$), and $F_1$, which is an even combination of precision and recall ($F_1 = \frac{2PR}{P+R}$). When precision and recall are equally important, $F_1$ can be used. For bug classification, the accuracy goal is both of high precision and recall, so we use $F_1$ as the accuracy metric in learning parameter tuning.

- **Applying Classification Model** After we obtain classification models for each category, we apply them on the whole database to predict which categories a bug probably belongs to. Specifically, we use the 548 sampled fixed bug reports as training and test
data to learn a model, and apply it on the whole database to identify runtime bugs. Among all 75,519 fixed bugs in the whole database for Mozilla, 28,928 bugs are identified as runtime bugs with precision of 89.6% in the test set. Then the classification model for each bug category is applied on the 28,928 runtime bugs.

Some bug categories such as concurrency bugs only constitute a very small percentage of all reported bugs. Therefore, random sampling cannot provide enough data for classifier training. To solve this problem, we apply information retrieval techniques [JWR00] on the bug database to retrieve the specific category of bugs.

3.2.3 Studying the Trend

To study the trend of different categories of bugs, including memory, concurrency, semantic, and security bugs, we calculate the number and relative percentage of bugs in each category reported in each year. Some previous work investigates the trend of bug density over releases [OW02], but it does not study the trend of different categories of bugs. Our analysis method on a large dataset enables us to study the change of different categories over time.

3.2.4 Studying the Complexity of Fixing

It is interesting to look at how long it takes to respond to a bug report and to fix it for different categories. Toward this end, we examine both response time and resolution time. The response time of a bug is the time between a bug is reported and a bug is taken care of. We approximate this time by the difference between the report time and the time a report is replied by others. The resolution time is the time between a bug is reported and it is marked as fixed when the fix is checked in to code base. A bug report’s resolution time encompasses its response time.

In addition, we also study size-related complexity of fixing, which is the number of different lines of code (changed, added, and deleted) to fix different types of bugs.
In order to help above study of bug fixing, we also collect the bug “priority” information from Bugzilla. The priority is a level number: 1 as the most and 5 as the least prioritized. They are assigned by developers before and during fixing based on urgency and difficulty.

### 3.2.5 Measuring the Correlation

To study the correlation between two categories in different dimensions, we use a statistical metric called $lift$. The $lift$ of category $A_i$ in dimension $A$ and category $B_j$ in dimension $B$, $lift(A_i, B_j)$, is calculated as $\frac{P(A_i B_j)}{P(A_i) P(B_j)}$, where $P(A_i B_j)$ is the probability that a bug belongs to both category $A_i$ and $B_j$. If $lift(A_i, B_j)$ is equal to 1, it means $P(A_i B_j) = P(A_i) P(B_j)$, which indicates that category $A_i$ and $B_j$ are not correlated. If it is greater than 1, category $A_i$ and $B_j$ are positively correlated, which means that if a bug belongs to $A_i$, it is more likely to also belongs to $B_j$.

### 3.3 Root Cause Analysis

We first present the analysis results based on sampled datasets in Section 3.3 and 3.4, and then present the results based on automatic classification in Section 3.6 to confirm that the results from sampled datasets are representative.

In this section, we analyze the root causes of bugs in modern software and compare our results with those in the previous studies [SC91, SC92], so that we can understand some important bug characteristic changes.

Figure 3.1 summarizes the distribution of bugs with different root causes and their corresponding impacts. From these figures, we can observe the following:

**Memory Bugs Have Decreased.** Figure 3.1(a) shows that memory bugs account for a relatively small fraction of all bugs, 16.3% in Mozilla and 12.2% in Apache. These percentages are much lower than the 28–38% reported in previous work [SC91, SC92]. Since debugging tools are known to help detect/avoid memory bugs, this reduction probably benefits from
using these tools in recent years.

Figure 3.1(b) shows that among memory bugs, NULL pointer dereference is a major cause, accounting for 37.2–41.7% in the memory categories, and most of them resulting in a system crash. Memory leak is another major cause, accounting for 16.7–30.2% of memory bugs, which is much more than the 8% reported previously [SC92]. This may be because memory leaks are relatively more difficult to detect without tools since their impacts may be “silent” within some time. Since most memory bugs can be detected by the existing tools such as Purify, Valgrind and Coverity [cov05], our results indicate that these debugging tools have not been used in development with their full capacity yet.

Semantic Bugs Are Dominant Root Causes. Figure 3.1(a) shows that the dominant
root causes are semantic errors in both applications, accounting for 81.1% in Mozilla and 86.7% in Apache. These results are much more than the 57–62% reported in a previous study [SC92] (bugs excluding memory bugs and concurrency bugs). Therefore, semantic bugs not only remain to be the dominant root causes, but also seem to have become more dominant.

One possible reason may be that most semantic bugs are application specific and are different from memory bugs which are general for any applications. Thus a programmer can easily introduce semantic bugs due to a lack of thorough understanding of the system, its requirements or its specifications. Additionally, it is harder to automatically detect semantic bugs because they are more application specific. The percentage increase of semantic bugs may be attributed to the decrease of memory bugs. Further, the previous work [SC92] is based on commercial software, while our study is based on OSS whose specifications are not well defined as commercial software, and hence resulting in more semantic bugs in OSS.

In order to further understand what are the major causes among semantic bugs, in Figure 3.1(c), we show the breakdown of semantic bugs into subcategories. We see that most semantic bugs are caused by wrong functionality implementation that does not meet the design requirements. In addition, missing features and missing cases also account for a large portion of semantic bugs, which is consistent with the previous study [CKC91]. Since knowledge about the target system is critical for avoiding and detecting such semantic bugs, these results suggest that it would be beneficial to develop techniques to automatically extract specifications from programs, similar to Daikon [ECGN01] and our previous work [LZ05].

Interestingly, there are quite a few simple semantic bugs. For example, typo errors account for 9.4–9.8% of the semantic bugs. It indicates that careless programming is still causing many bugs, suggesting that the development environment should provide some tools for programmers to check for simple errors such as typos and copy-and-paste related bugs (as our previous work [LLMZ04] does).

In order to further understand concurrency bugs, instead of randomly sampling, we use
information retrieval technique to obtain possible concurrency bugs; the results are presented in Section 3.9.

### 3.4 Impact Analysis

In this section, we study the distribution of impacts and the correlation between impacts and root causes.

**Dominant Impacts**  Figure 3.2 summarizes the distribution of different impacts with the corresponding root causes. It shows that incorrect functionality is the dominant impact; indeed, the percentage of incorrect functionality is 64.3–69.4%, much larger than the 35% reported in the previous work [SC91]. This is because the percentage of semantic bugs has significantly increased in modern software and most of them cause incorrect functionality as shown in Figure 3.1(a).

![Figure 3.2: Distribution of impacts](image)

In contrast, because memory bugs have decreased substantially, the severe impacts on availability, including crashes and hangs, have also been reduced to 18.4–22.0%. However, although the percentage of these impacts is reduced, they still account for a considerable portion and can significantly compromise availability. Thus, recovery techniques are still needed to provide highly available services.
Correlations Between Causes and Impacts  Figure 3.2 also shows that the major cause of crashes is memory bugs, accounting for 53.8–57.1%, which is similar to what has been found in the previous work [SC91]. Semantic bugs are clearly more likely to cause incorrect functionality, accounting for 96.6–98.5%, which is also consistent with the previous studies. However, among the crashes, 42.9–44.2% are contributed from semantic bugs, which is higher than that reported in the previous work [SC91]. It indicates that although most semantic bugs result in incorrect functionality, they are also one of the important factors of unavailability.

In order to further understand the correlation between causes and impacts, we show the correlation metric lift in Table 3.3. Not surprisingly, here we see that hanging has an extremely strong correlation with concurrency bugs, while crashing has a strong correlation with memory bugs. The (incorrect) functionality impact has a relatively strong correlation with semantic bugs, though no specific subcategory of semantic bug has an exceptionally high correlation. Interestingly, although semantic bugs are overall negatively correlated with crashing and hanging, some specific semantic bugs are positively correlated with them.

<table>
<thead>
<tr>
<th>Impact</th>
<th>Memory</th>
<th>Concurrency</th>
<th>Semantic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hang</td>
<td>0.77</td>
<td>9.43</td>
<td>0.77</td>
</tr>
<tr>
<td>Crash</td>
<td>3.31</td>
<td>0.73</td>
<td>0.55</td>
</tr>
<tr>
<td>Func.</td>
<td>0.11</td>
<td>0.65</td>
<td>1.19</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Impact</th>
<th>Memory Subcategories</th>
<th>MLK</th>
<th>UMR</th>
<th>NULL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hang</td>
<td>0.00</td>
<td>0.00</td>
<td>2.06</td>
<td></td>
</tr>
<tr>
<td>Crash</td>
<td>0.78</td>
<td>3.05</td>
<td>5.08</td>
<td></td>
</tr>
<tr>
<td>Func.</td>
<td>0.23</td>
<td>0.30</td>
<td>0.00</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Impact</th>
<th>Semantic Subcategories</th>
<th>MissF</th>
<th>MissC</th>
<th>CornerC</th>
<th>CtrlFlow</th>
<th>Except</th>
<th>Process</th>
<th>Typo</th>
<th>FuncImpl</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hang</td>
<td></td>
<td>0.00</td>
<td>2.20</td>
<td>0.00</td>
<td>2.36</td>
<td>5.08</td>
<td>0.00</td>
<td>0.00</td>
<td>0.35</td>
</tr>
<tr>
<td>Crash</td>
<td></td>
<td>0.22</td>
<td>0.68</td>
<td>1.13</td>
<td>0.73</td>
<td>1.95</td>
<td>0.00</td>
<td>0.73</td>
<td>0.38</td>
</tr>
<tr>
<td>Func.</td>
<td></td>
<td>1.31</td>
<td>1.31</td>
<td>1.01</td>
<td>1.08</td>
<td>0.70</td>
<td>1.21</td>
<td>0.86</td>
<td>1.33</td>
</tr>
</tbody>
</table>

Table 3.3: Correlation between root causes and impacts in Mozilla
Categories with too few examples are not shown due to statistical insignificance.
3.5 Bugs in Different Components

![Distribution of bugs in software components]

Figure 3.3: Distribution of bugs in software components

<table>
<thead>
<tr>
<th>Component</th>
<th>Memory</th>
<th>Concurrency</th>
<th>Semantic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core</td>
<td>1.84</td>
<td>1.06</td>
<td>0.83</td>
</tr>
<tr>
<td>GUI</td>
<td>0.31</td>
<td>1.09</td>
<td>1.14</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Component</th>
<th>Memory Subcategories</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core</td>
<td>MLK</td>
</tr>
<tr>
<td></td>
<td>1.71</td>
</tr>
<tr>
<td>GUI</td>
<td>0.29</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Component</th>
<th>Semantic Subcategories</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core</td>
<td>MissF</td>
</tr>
<tr>
<td></td>
<td>0.75</td>
</tr>
<tr>
<td>GUI</td>
<td>1.32</td>
</tr>
</tbody>
</table>

Table 3.4: Correlation between root causes and software components in Mozilla

In this section, we study where the dominant bugs are located in software components, including core, GUI, network, I/O, and others. To the best of our knowledge, there is no previous work on studying bug characteristics from this perspective. Since the modern software tends to emphasize friendly user interfaces, we are particularly interested in studying characteristics of GUI bugs. Due to the lags of GUI testing techniques, we believe that
characterization of GUI bugs is very important.

**Bug distribution in software components** Figure 3.3 shows the distribution of bugs within different components and their impacts. As we can see, GUI modules are critical for software reliability in modern graphical interface software, such as Mozilla. GUI modules account for more than half of bugs in Mozilla and also cause around 30% of Mozilla crashes. Unfortunately, GUI testing techniques still lag far behind now. Future research should pay more attention to GUI related testing and debugging.

**Root causes of GUI bugs** The correlations between software components and root causes for Mozilla are shown in Table 3.4. Interestingly, the results indicate that bugs within GUI and core modules have quite different root causes. The major root cause of bugs in core modules is memory related, while that of GUI bugs is semantic and concurrency bugs. Such difference is likely because the GUI modules and their development process are quite different from other modules. Further, GUI bugs are correlated to the sub-categories of missing features and other wrong functionality implementation that are application specific. The results indicate that the existing debugging tools aiming at memory bugs are unsuitable for GUI bugs, while study on application-specific semantic bugs can be helpful.

### 3.6 Automatic Classification

This section presents our results with automatic classification so that we can confirm that the results of distribution based on the sampled bugs in previous sections are representative. Some further improvement on the classification accuracy could be done such as exploiting higher level information to represent documents and optimizing the queries in information retrieval for small categories as described in Section 3.2.2.
Distribution of Root Causes  Table 3.5 shows the distribution of bugs with different root causes. Compared with the results using sampled bugs in Section 3.3, the percentage of memory and semantic bugs are similar, which indicates that the distribution results based on sampled bugs and a large dataset are consistent.

<table>
<thead>
<tr>
<th>Cause</th>
<th>Percentage</th>
<th>Classification Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory</td>
<td>13.2%</td>
<td>0.736</td>
</tr>
<tr>
<td>Semantic</td>
<td>78.0%</td>
<td>0.874</td>
</tr>
</tbody>
</table>

Table 3.5: Distribution of root causes based on automatic classification

Precision, recall, and F1 are defined in Section 3.2.2.

Trend of Root Causes  In order to understand the trend of bug distribution along software evolution, we plot the relative percentages of memory and semantic bugs in Figure 3.4. Although software becomes mature and stable, semantic bugs that are specific to applications still remain dominant. Semantic bugs increase gradually with the maturity of software, while memory bugs decrease gradually. However, both types have not changed too much, compared with concurrency bugs that are decreasing significantly (Section 3.9). Bug detection tools such as Purify only affect the relative percentage of memory bugs a little during this period, because they had become available before 1999 and are used during these years.

Distribution of Impacts  Table 3.6 shows the distribution of bugs with different impacts. Compared with the results using sampled bugs in Section 3.3, the distribution is similar.

<table>
<thead>
<tr>
<th>Impact</th>
<th>Percentage</th>
<th>Classification Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hang</td>
<td>2.5%</td>
<td>0.714</td>
</tr>
<tr>
<td>Crash</td>
<td>13.0%</td>
<td>0.852</td>
</tr>
<tr>
<td>Data corruption</td>
<td>1.8%</td>
<td>0.400</td>
</tr>
<tr>
<td>Performance degradation</td>
<td>2.2%</td>
<td>0.500</td>
</tr>
<tr>
<td>Incorrect Functionality</td>
<td>75.2%</td>
<td>0.860</td>
</tr>
</tbody>
</table>

Table 3.6: Distribution of impacts based on automatic classification
3.7 Difficulty of Bug Fixing

In this section, we study the difficulty in fixing different bugs. Mockus and Herbsleb [MFH02] studied the resolution time in OSS, but they only analyzed the relation between resolution time and fixing priority and verified the intuition that the bugs with higher priority are fixed more quickly. Differently, we analyze how difficult it is to fix bugs of different causes, especially in the context of with or without debugging tools. In order to understand other factors besides resolution time, we also analyze the size of patches that can indicate how complex to fix the bugs, which has never been studied before as far as we know. Our analysis is based on the automatic classification data so that the statistical results can be more representative with less variance.

Patch Size  Figure 3.5(a) shows that it needs 106 LOC to fix a bug on average. Considering different root causes, it is relatively simple to fix memory bugs. The patch for a memory bug is around 71 LOC, while each semantic bug needs 111 LOC. The reason is that memory bugs are usually caused by careless programming and it is usually unnecessary to re-implement the functionality. Interestingly, considering different impacts, performance bugs are more complex to fix. The reason is that it usually needs restructuring in order to fix performance problems.
Response Time and Resolution Time  Figure 3.5 (b) shows that the average response time and resolution time are 16 days and 110 days in Mozilla, respectively. Considering the bugs with different root causes, it takes the shortest time, 62 days only, to fix memory bugs. One reason is that nowadays memory bugs are relatively easy to fix because quite a few debugging tools are effective for identifying the root causes of memory bugs. In contrast, semantic bugs generally take a much longer time (117 days) to fix, because there are relatively fewer tools to help detect semantic bugs automatically. For some semantic bugs, the long resolution time may be also caused by their low fixing priority.

Debugging Tools and Resolution time  Most of the available debugging tools used by developers aim at memory bugs. Section 3.3 showed that memory bugs are decreasing likely because of using bug detection tools. Among the 43 memory bugs we manually verified in Mozilla, the programmers explicitly indicated that they used Purify for debugging in six reports. The average resolution time of these six bugs is 14 days, much shorter than the overall memory bugs, 62 days, which indicates that bug detection tools can significantly facilitate diagnosing root causes, thus speed up bug fixing.
3.8 Security Related Bugs

Security is becoming increasingly important recently. However, previous work has not addressed how software errors affect security. To study the security related bugs, we collect 193 security vulnerabilities in Mozilla and 64 in Apache HTTP Server from NVD \[NVD05b\]. In this section we would like to understand (1) whether security bugs are increasing, (2) what types of bugs cause vulnerability, and (3) whether security related bugs are fixed faster than other bugs.

**Are Security Related Bugs Increasing?** Figure 3.6(a) shows the numbers of vulnerabilities along time. We also normalize the numbers to relative percentage by comparing with the total fixed bugs reported in Bugzilla during the corresponding year. The trend shows that the numbers of vulnerabilities are increasing for both Mozilla and Apache. Although there are some exceptional points in some years, such as Mozilla in 2003\footnote{Data in 2005 only contain bug reports until July.}, the normalized percentages are increasing significantly over the recent years. It demonstrates that security issue is becoming *increasingly important* for both client and server software.

**What Types of Bugs Lead to Vulnerabilities?** Figure 3.6(b) shows the distribution of root causes and severity for security bugs. Surprisingly, memory bugs account for only 8.8–17.2%, while semantic bugs cause 71.9–83.9% of vulnerabilities. This finding is against the belief that buffer overflows are the most common form of security vulnerability \[CWP^+00\]. The reason may be that most buffer overflows have been detected and fixed before release due to the available debugging tools recently.

The different distribution of impacts for Mozilla and Apache shown in Figure 3.6(c) indicates that vulnerabilities have different impacts on client and server systems. For client systems, most security bugs result in unauthorized accesses, while for Apache Web Server, both availability and unauthorized accesses are the major vulnerabilities.
than the other types of bugs, and they are likely to cause unavailability and unauthorized accesses because systems can be intruded by exploiting buffer overflows. Thus besides the response time for security related bugs is longer than the overall bugs; especially, the

![Graph](image1.png)

(a) Trend

![Graph](image2.png)

(b) Causes and severity

![Graph](image3.png)

(c) Impacts and causes

(c) Response and resolution time for Mozilla

Figure 3.6: Security related bugs collected from NVD

The correlation metric lift shown in Table 3.7 indicates that memory bugs are more severe than the other types of bugs, and they are likely to cause unavailability and unauthorized accesses because systems can be intruded by exploiting buffer overflows. Thus besides reducing the number of buffer overflows in source code using bug detection tools, it is also important to prevent attackers from exploiting them. StackGuard is such a tool that can protect against stack smash attacks. In addition, semantic bugs are less severe than memory bugs, and are likely to cause confidentiality and integrity problems.

Are Security Bugs Fixed Faster? Figure 3.6(d) shows the average response and resolution time for security bugs compared with all the 264 sampled bugs in Mozilla. Surprisingly, the response time for the security related bugs is longer than the overall bugs; especially, the
response time for the security bugs with high severity is much longer than the other bugs, so is their resolution time. The response time is longer could be because the communication between developers is hidden from the publicly accessible forum in Bugzilla for security concerns.

Since the deviation for high-severity bugs is much larger than others, we doubted that there are some cases with long resolution time. We found that four security bugs with high severity took more than one year to fix. These bugs are extremely difficult to fix, which makes the average resolution time longer than the others. We should also note that these long response and resolution time do not imply less attention from developers. Actually, the average priority level of security bugs is 1.52 (the smaller level, the higher priority), the most prioritized among all bug categories we have studied. It is possible that, since very critical, these security bugs’ fixing and patch validation are done in a very careful way, which extend the diagnosis and fixing period. The response and resolution time for the security bugs with medium and low severity are shorter than other bugs, which means that they are fixed faster. Our results indicate that although security bugs are important, some of them need to take a long time to fix, which can lead to a long vulnerability window. Therefore, before the patch is available, we should use some techniques to detect attacks and to prevent attacks from exploiting the vulnerability. For example, recently Joshi et al. [JKDC05] proposed using vulnerability-specific predicates to detect when vulnerabilities are triggered.

Table 3.7: Correlation between root causes and severity/impacts for security related bugs

<table>
<thead>
<tr>
<th>Cause</th>
<th>Severity</th>
<th>Impact</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>High</td>
<td>Medium</td>
</tr>
<tr>
<td>Mem</td>
<td>1.80/1.62</td>
<td>0.55/2.91</td>
</tr>
<tr>
<td>Con</td>
<td>0.00/0.00</td>
<td>1.65/8.00</td>
</tr>
<tr>
<td>Sem</td>
<td>0.87/0.85</td>
<td>1.09/0.52</td>
</tr>
<tr>
<td>Others</td>
<td>0.58/1.19</td>
<td>1.27/0.00</td>
</tr>
<tr>
<td></td>
<td>Confidentiality</td>
<td>Integrity</td>
</tr>
<tr>
<td>Mem</td>
<td>0.00/0.00</td>
<td>0.00/0.65</td>
</tr>
<tr>
<td>Con</td>
<td>6.40/4.92</td>
<td>0.00/7.11</td>
</tr>
<tr>
<td>Sem</td>
<td>1.11/1.18</td>
<td>1.19/1.08</td>
</tr>
<tr>
<td>Others</td>
<td>1.65/0.82</td>
<td>1.82/0.00</td>
</tr>
</tbody>
</table>

The first number is for Mozilla, and the second one is for Apache Web Server.
3.9 Concurrency Bugs

Because concurrency bugs account for only a small portion in bug report databases, it is difficult to study their characteristics by randomly sampling hundreds of bug reports. Therefore, we use information retrieval techniques to obtain potential concurrency bugs, and then manually verify these potential concurrency bugs. In this way, we have collected 90 concurrency bugs from the top of retrieval results for Mozilla. The larger available dataset allows us to study the characteristics of concurrency bugs, which has never been done before.

Are Concurrency Bugs Increasing? Figure 3.7(a) shows the changes of concurrency bugs over time in Mozilla. Unexpectedly, the number of concurrency bugs increased only in the first two years (1999-2000) but sharply decreased later. To show the relative percentage, we normalize the number by all fixed bugs in the corresponding year (which is not absolute percentage because we only collect part of concurrency bugs). The relative percentage has the same trend except in 2005. Considering the evolution of Mozilla, the number and relative percentage of concurrency bugs increased, and then decreased as the software became stable, which is different from the trend of memory and semantic bugs shown in Figure 3.4.

![Figure 3.7: Concurrency bugs in Mozilla](image)

However, the previous studies [CC00, Gra86] indicated the perception that the relative
percentage of Heisenbugs (transient bugs) should increase when the software becomes stable because the other non-transient bugs are fixed but the transient bugs are still remaining in the software. Our finding about the trend of concurrency bugs does not follow this perception. A possible explanation is that the hard-to-reproduce concurrency bugs are underreported. Before the software became stable (the first two years), most of the concurrency bugs that could be easily reproduced were reported and fixed, and the other concurrency bugs that were difficult to reproduce were not reported and remained in software. When we verified the concurrency bugs, we found that most of them could not be reproduced by developers with an acceptable probability, say more than once out of ten times. Therefore, when the software became stable, even though the software still contained many concurrency bugs, the failures caused by them were unlikely to be reported by users or to be fixed by developers because they were difficult to reproduce.

**What Are the Impacts from Concurrency Bugs?** Figure 3.7(b) shows the distribution of impacts from concurrency bugs. Compared with all bugs shown in Figure 3.2, concurrency bugs can cause much more severe impacts on systems, which is consistent with previous work [SC92]. Specifically, 42.2% of concurrency bugs can cause hanging due to synchronization errors and deadlocks, 10 times higher than all bugs. Further, 55.5% of concurrency bugs lead to fail-stop failures (crashes and hangs), which can be detected and recovered by generic recovery techniques.

### 3.10 Summary

This empirical study investigates the impacts of new factors on software errors and studies the bug characteristics in two large modern OSS projects. It uses text classification and information retrieval techniques to automatically classify tens of thousands of bugs and classifies bugs from different dimensions and investigates the correlation between categories.
This empirical study has discovered several interesting findings:

- Memory bugs only account for 12.2–16.3%, much less than the 28–38% reported in previous studies [SC91, SC92], indicating memory bugs are becoming less pervasive due to the available techniques to automatically detect them. Our results also show bug detection tools can effectively reduce the diagnosis and resolution time of memory bugs. However, we also found that there still exist many simple memory bugs such as memory leaks and NULL pointer dereferences, indicating memory bug detection tools have not been used at their full capacity.

- Semantic bugs are the major root causes, accounting for 81.1–86.7%, and their percentages increases with the maturity of software. Moreover, they also have severe impacts on system availability, contributing to 42.9–44.2% of crashes. Additionally, it takes a longer time to diagnose and fix semantic bugs, almost twice as memory bugs. Our results suggest that more effort should be put into automatically detecting and diagnosing semantic bugs.

- Careless programming causes many bugs. For example, some simple bugs such as typos account for 9.4–9.8% of semantic bugs. Although their fixes are usually simple, it takes a longer time to diagnose them than to diagnose memory bugs, which suggests that software development environments such as Microsoft Visual Studio and Eclipse should provide more support to avoid them in the early stages of software development.

- Most bugs result in incorrect functionality due to the dominance of semantic bugs. In contrast, only 14.3–19.1% bugs, mostly memory bugs, results in crashes. In addition to memory bugs, semantic bugs related to corner cases and exception handling are also highly likely to cause crashes.

- Our results show that security bugs are increasing significantly over time in terms of
number and relative percentage. Among different root causes of security vulnerabilities, memory related bugs contribute for only 8.8–17.2% but are usually severe, while semantic bugs are the dominant cause, accounting for 71.9–83.9%. Surprisingly, we found that some security bugs with high severe impacts took more than one year to fix after being reported, which means that system security can be easily compromised during such a long vulnerability window.

- GUI bugs have become the major ones in graphical interface software, accounting for 52.7% of bugs in Mozilla, and resulting in 28.8% of all crashes. Furthermore, most GUI bugs are caused by semantic errors, which indicates that designing good GUI test cases with good coverages is probably the only way to significantly reduce the number of GUI bugs.

- Concurrency bugs account for a small portion of bug reports, probably because they are underreported. We found that 55.5% of them cause hangs or crashes, which means that most of them are benign faults (fail stop) so that most failures caused by them should be able to recover using simple generic techniques such as restart or rollback and reexecute.

Overall, these results not only provide software development with a good understanding about software bugs, but also provide useful guidelines for related research such as bug detection and testing.
Chapter 4

PR-Miner

4.1 Overview

4.1.1 Programming Rules

Programs usually follow many implicit programming rules. A simple example of a programming rule is the function call pair of \texttt{lock} and \texttt{unlock}: a call to \texttt{lock} should be followed by a call to \texttt{unlock} later. Besides such a well-known programming rule, there are many other implicit rules in large software. For example, as shown in Figure 4.1 PostgreSQL, a well-known open-source database server, contains an implicit programming rule that a call to \texttt{SearchSysCache} must be followed by \texttt{ReleaseSysCache}. The reason is that the function \texttt{SearchSysCache} returns a cache copy of a specified tuple; so after the caller finishes using the tuple, it must call \texttt{ReleaseSysCache} to release it so that this copy can be replaced by other data in the cache. This rule appears 209 times in PostgreSQL code. If some code violates this rule, it causes a memory leak in PostgreSQL’s buffer cache.

```
postgresql-8.0.1/src/backend/catalog/dependency.c:
1733 getRelationDescription (StringInfo buffer, Oid relid)
1734 {
1735    HeapTuple relTup;
1736 ......
1740    relTup = SearchSysCache(...);
1796    ReleaseSysCache(relTup);
1797 }
```

Figure 4.1: An example of function-pair rule

This rule appears 209 times in PostgreSQL code.
Many programming rules are much more complex. Some rule may contain more than two program elements, with each element being of a different type including function, variable or data type. For example, the programming rule shown in Figure 4.2, also extracted from PostgreSQL, contains four function calls and one variable. This rule specifies the correct procedure to replace a tuple. Specifically, it requires that, before replacing a tuple using `simple_heap_update`, the relation must be first opened by calling `heap_openr`, and then call `CatalogUpdateIndexes` to keep the index consistent. Furthermore, after `simple_heap_update`, the relation needs to be closed by calling `heap_close`. This rule appears 68 times in PostgreSQL.

Some complex rules may indicate variable correlations, i.e. these variables should be accessed together or modified in a consistent manner. For example, Figure 4.3 shows that, in the Linux code, the two variables `ic.command` and `ic.driver` should be accessed together. This rule appears 98 times in Linux.

Implicit programming rules such as those shown above are intrinsic features of programs and violations to these rules can result in software defects. Figure 4.4 shows such an implicit programming rule and a violation detected by PR-Miner from the latest version of Linux.
The rule shown in Figure 4.4(a) is for probing and initializing a SCSI device: the system should call `scsi_host_alloc` to allocate a data structure for the device driver, then call `scsi_add_host` to register the device with the SCSI stack, and finally scan the host by `scsi_scan_host`. This is the correct procedure for probing SCSI devices in Linux. This rule appears 27 times in Linux. However, there are two program locations missing `scsi_scan_host` in the latest version of Linux as shown in Figure 4.4(b), which are undetected defects in Linux (we reported these two bugs as well as others to the Linux developers and they are being fixed now).

Such implicit programming rules are useful information for software development. Unfortunately, they usually exist only in programmers’ minds as they are too tedious to be documented manually. In addition, rule maintenance is a hard task since some rules can change in new versions. Moreover, when the software scales up, the number of rules also increases significantly. As a result, most of them are undocumented, especially in open-source projects. Consequently, violations to these rules are easy for programmers to introduce, especially for new programmers who are unaware of these rules. Therefore, it is highly desirable if programming rules can be automatically extracted from existing source code. The extracted rules can thereafter be used as a specification to be referenced by programmers. In addition, it is also useful to automatically detect violations to these rules to make software more robust.

Previous work [ECC01] by Engler, Chen and Chou has conducted a preliminary inves-
Figure 4.4: An example of a programming rule involving multiple functions
The rule \{`scsi_host_alloc, scsi_add_host`\}⇒\{`scsi_scan_host`\} appears 27 times in different functions in Linux 2.6.11, one of which is shown in (a). PR-Miner detects two violations that miss the function `scsi_scan_host`, which are potential bugs. One of the violations is shown in (b).

The two arguments, \(a\) and \(b\), will be fit by passing all plausible \(a-b\) pairs that are selected based on statistical analysis and weighted by naming conventions such as the substrings “lock” and “unlock”.

While the work above is inspiring and proposes a promising direction, it extracts only \textit{pair-wise} programming rules. Our experimental results indicate that programming rules
with 2 elements only account for a small portion (14%) of all rules extracted by PR-Miner. In addition, their work requires programmers to give some particular rule templates such as “function \(a\) must be paired with function \(b\)”, which not only restricts the types of rules extracted but also needs some specific knowledge about the target software. Furthermore, programmers also need to give different weights to the functions and variables when fitting elements into templates.

To significantly advance the state-of-the-art, it would be beneficial if implicit programming rules, including complex ones, could be automatically extracted in a general form without requiring prior knowledge or rule templates from programmers. To do this, a naive method is to check every possible combination of program elements to see if they are frequently used together in the target software’s code. Obviously, for large software such as Linux that contains hundreds of thousands of functions and variables, such a naive method would result in exponential complexity. Therefore, an efficient method needs to be developed to achieve the goal.

### 4.1.2 Contributions of PR-Miner

This chapter presents a novel method called PR-Miner (Programming Rule Miner), that uses a data mining technique to automatically extract *general* programming rules from software code written in an industrial programming language such as C\(^1\) and detect violations with *little effort* from programmers. More specifically, our work makes the following two major contributions:

1. We propose a general method to automatically extract implicit programming rules from large software code. Benefiting from data mining techniques, PR-Miner can extract thousands of programming rules from software such as Linux with 3500 files and a total of 3 million lines of code within 1 minute. Compared with the previous work \[ECC01\] \[^{1}\]Similar to other automatic specification generation tools, we assume that the software has been reasonably well tested and runs correctly most of the time.
that extracts only function-pair based rules, PR-Miner extracts substantially more rules. This is because PR-Miner has substantially generalized Engler et al’s work in the following two aspects:

— **General method:** Our technique for extracting programming rules is more general. PR-Miner can automatically extract programming rules from software code without any prior knowledge about the software or requiring any annotation, templates or weight assignments from programmers. Additionally, by replacing the front-end parser, PR-Miner can be easily modified to work with programs written in other programming languages such as Java.

— **General rules:** The programming rules extracted by PR-Miner are more general. Since it does not limit the programming rules using any fixed templates, PR-Miner can extract rules in general forms and with multiple program elements of different types including functions, variables, data types, etc. As a result, it not only extracts simple pair-wise rules, but also extracts more complex rules like the examples shown above.

2. We also propose an efficient algorithm to detect violations to the extracted programming rules. PR-Miner has detected many violations to the extracted rules in the latest versions of Linux and PostgreSQL within 1 minute. Among the top 60 violations reported by PR-Miner, many of them have been confirmed as bugs, including 16 bugs in Linux, 6 bugs in PostgreSQL and 1 bug in Apache. These bugs are currently being fixed by corresponding developers after we reported. Most of these bugs are semantic bugs that violate complex programming rules that contain more than 2 elements and are thereby difficult for previous tools to detect.
4.2 Design of PR-Miner

PR-Miner has two major functionalities: automatically extracting implicit programming rules, and automatically detecting violations to the extracted programming rules. The flowchart of PR-Miner is shown in Figure 4.5. This section first gives the main idea of PR-Miner, and then presents how PR-Miner automatically extracts programming rules from source code, and how it detects rule violations and prunes false positives.

4.2.1 Main Idea

The high-level idea of PR-Miner in automatic rule extraction is to find associations among elements (e.g., function, variable, data type) by looking for elements that are frequently used together in source code. For example, calls to `spin_lock_irqsave` and `spin_unlock_irqrestore` in Linux appear together within the same function for more than 3600 times, which indicates that `spin_unlock_irqrestore` following `spin_lock_irqsave` is very likely to be an implicit programming rule. By identifying which elements are used together frequently in the source code, such correlated elements can be considered a programming rule with relatively high confidence. Of course, as described in the introduction, a naive implementation of this high-level idea is infeasible since it needs to examine all possible element combinations.

In order to efficiently find program element correlations, PR-Miner converts the problem into a frequent itemset mining problem by first parsing the software source code as shown in Figure 4.5. Each program element is hashed into a number, then a function definition is mapped into an itemset (a set of numbers), which is written as a row into the itemset database. As a result, the whole program is converted into a database that contains many itemsets. By mining this database using a frequent itemset mining algorithm such as `FPclose`, we can find the frequent sub-itemsets that appear for many times. These frequent sub-itemsets can then be used to infer programming rules.
For a frequent sub-itemset discovered by the mining algorithm, we call the set of the corresponding program elements a **programming pattern**, which indicates that the program elements are correlated and frequently used together. For example, \textit{FPclose} can find that \{\texttt{spin\_lock\_irqsave}, \texttt{spin\_unlock\_irqrestore}\} is a programming pattern since it appears in the Linux source code more than 3600 times.

Note that programming patterns are different from programming rules. For example, the above pattern may lead to one or two of the following programming rules:

\[
\{\texttt{spin\_lock\_irqsave}\} \Rightarrow \{\texttt{spin\_unlock\_irqrestore}\} \\
\{\texttt{spin\_unlock\_irqrestore}\} \Rightarrow \{\texttt{spin\_lock\_irqsave}\}
\]

The first rule says that whenever the program calls \texttt{spin\_lock\_irqsave}, it should also call \texttt{spin\_unlock\_irqrestore}, while the second says that whenever the program calls \texttt{spin\_unlock\_irqrestore}, it should also call \texttt{spin\_lock\_irqsave}. These two are different rules, and not all of them definitely hold, even if the pattern has appeared for many times.

Therefore, after programming patterns are extracted using the frequent itemset mining technique, PR-Miner needs to generate programming rules from the extracted patterns. The main idea of the rule generation process is to find the number of cases that contain the items on the left but not those on the right. For example, in the above example, we need to find out how many cases that \texttt{spin\_lock\_irqsave} appears but \texttt{spin\_unlock\_irqrestore} does not and vice versa. After generating the programming rules, PR-Miner stores them in specification files so that programmers can examine them and also use them later as references. Section 4.2.3 describes the rule generation process in detail.

After programming rules are generated, PR-Miner automatically detects violations in source code. It also automatically prunes false positives and ranks violations in the report so that programmers only need to examine top ranked violations. The violation detection process is based on the idea that a programming rule is usually followed in most cases and
violations occur only in a small percentage of cases. Section 4.2.4 describes the detection process in more detail.

Using closed frequent itemset mining algorithms such as FPclose provides PR-Miner several benefits: (1) Generality. Close frequent itemset mining algorithms do not limit the number of items in frequent sub-itemsets and also does not require any rule templates. Furthermore, the items in a frequent sub-itemset are not necessarily adjacent in the supporting itemset, i.e. they can be far apart from each other. (2) Time efficiency. Data mining algorithms such as FPclose are usually very efficient since they strive to avoid scanning data too many times by eliminating redundant computation as much as possible. Additionally, since FPclose generates only closed frequent itemsets, it can avoid generating an exponential number of sub-itemsets. (3) Space efficiency. From closed frequent sub-itemsets, we can find closed rules, rules that subsume many other rules with the same support. Take the itemset database \( D = \{ \{a, b, c, d, e\}, \{a, b, d, e, f\}, \{a, b, d, g\}, \{a, c, h, i\} \} \) described in Section 2.1 as an example. FPclose finds a closed frequent sub-itemset: \( \{a, b, d\} \):3. From the closed frequent sub-itemset we can have the following 6 closed rules with support 3:

\[
\{a\} \Rightarrow \{b, d\} \text{ with confidence } 3/4=75%
\]

\[
\{b\} \Rightarrow \{a, d\} \text{ with confidence } 100%
\]

\[
\{d\} \Rightarrow \{a, b\} \text{ with confidence } 100%
\]

\[
\{a, b\} \Rightarrow \{d\} \text{ with confidence } 100%
\]

\[
\{a, d\} \Rightarrow \{b\} \text{ with confidence } 100%
\]

\[
\{b, d\} \Rightarrow \{a\} \text{ with confidence } 100%
\]

Other rules are subsumed by the above closed rules. For example, the sub-rule \( \{a\} \Rightarrow \{b\} \) with confidence 75% is subsumed by the first closed rule. Using closed rules not only saves space, but also significantly reduces the number of rules that need to be examined or referenced by programmers. In addition, it also speeds up the violation detection process (See Section 4.2.4).
4.2.2 Extracting Programming Patterns

Parsing Source Code

The main purpose of parsing source code is to build an itemset database in order to convert the programming pattern extraction problem into a frequent itemset mining problem. PR-Miner does this by using a modified GCC compiler [St05] as the parser to convert each function definition into a set of numbers. The current prototype of PR-Miner only works for C, but it can be easily extended to other programming languages by replacing the GCC front ends.

In order to convert the source code into an itemset database, we need to address the following issues: (1) How to parse the source code? (2) What elements in the source code should be converted? (3) How to represent elements using numbers?

To parse the source code, PR-Miner first uses the GCC front end to obtain the intermediate representation. The intermediate representation is stored in a tree data structure, with each node representing various types of elements in source code including identifier name, data type name, keyword, operator, control structure, and so on. In order to convert a func-
tion to an itemset, PR-Miner traverses the representation tree of this function, and hashes each selected elements to a number. By combining the hash values of all selected elements in a function, this function is mapped to an itemset. Then the itemsets of all functions construct the itemset database to input to the mining algorithm $\textit{FPclose}$. The reason to convert a function instead of a basic block to an itemset is that most programming rules usually occur within the scope of a function. Of course, some rules can span across multiple functions. But mining these rules is much harder as it requires deeper inter-procedural analysis.

Not every program element in the intermediate representation is converted into a number because some elements can cause noise. For example, keywords and simple data types such as $\texttt{int}$ appear in almost every function. They are less likely lead to interesting programming rules. In addition, including them in the itemset would significantly increase the computation of frequent itemset mining. Therefore, PR-Miner does not hash such elements into numbers.

Furthermore, the same programming rule involving local variables may use different variable names at different code segments. For instance, in the example shown in Figure 4.4 the return value of calls to the same function $\texttt{scsi_host_alloc}$ can be assigned to different local variables such as $\texttt{host}$ and $\texttt{scsi_host}$. If we hash them into different numbers, the rule might be missed. In order to catch such kinds of rules, we need to use the common characteristics of these local variables such as their data types to represent them so that they are still hashed to the same number in the itemset database. For example, the local variable $\texttt{class_ref}$ in the code segment in Figure 4.2 is represented by the hash value of its data type $\texttt{Relation}$ instead of its name $\texttt{class_ref}$.

Another problem when hashing identifiers to numbers is name collisions. Different types of identifiers with the same name would be hashed to the same number, causing false positives in the generated frequent sub-itemsets. In order to eliminate such name collisions, PR-Miner hashes different types of identifiers into different values. To do that, PR-Miner first prefixes every identifier name based on its type, and then hashes the prefixed name to a number.
For example, a function call to `lock` would be prefixed with “F-” and then be hashed to a number corresponding to “F-lock”, while a global variable with the same name `lock` would be prefixed with “G-” and be hashed to a number corresponding to “G-lock”.

Similarly, different record structures may use the same name for their fields, which is quite common in large software. For example, the names, “next” and “prev” are commonly used as field names in many different structures. Such name collisions would result in false positives of frequent sub-itemsets. In order to differentiate fields of the same name but in different record structures, PR-Miner attaches the associated record type to every field name. For example, the fields `next` in the record types `tree` and `list` are considered as “D-tree.R-next” and “D-list.R-next”, respectively, and so they can be hashed into different numbers to avoid collision.

The hash function PR-Miner uses is “hashpjw” [ASU86], chosen for its low collision rate. Our experiments show that its collision rate is low enough for frequent sub-itemset mining. Additionally, if conflict-free mapping is needed, we can first parse the whole source code so that we can create a symbol table for all possible identifiers, and then convert elements into numbers based on their indexes to the symbol table. Since it takes one more pass of the source code and our hashing method already has a low collision rate, we do not use this method.

Table 4.1 shows how PR-Miner converts a function into an itemset. After parsing the source code, prefixing and hashing selected elements into numbers, PR-Miner converts the definition of function `twa.probe` into the itemset `{92, 39, 41, 68, 56, 36, ...}`.

Mining for Programming Patterns

After PR-Miner parses the source code and generates an itemset database, it applies the closed frequent itemset mining algorithm, `FPclose`, on the database to find closed frequent sub-itemsets. As we describe in Section 2.1, if a set of numbers appear together in any itemsets for more than a specified threshold number (`min_support`) of times, this sub-itemset
is considered frequent. Let us consider the example shown in Table 4.1. For simplicity, let
us denote these three functions as add, alloc and scan. The sub-itemset \{39, 68, 36, 92\} appears in totally 27 itemsets in the itemset database converted from Linux code. Suppose
that \textit{min\_support} is set as 15. \textit{FPclose} will find a frequent sub-itemset \{39, 68, 36, 92\} with
a support of 27, which means that the corresponding functions alloc, add and scan, and
the data type \texttt{Scsi\_Host} are used together for 27 times. Therefore, these four elements are
correlated with each other and are thereby outputted as a programming pattern, which is
then used to generate programming rules in the next step.

Since \textit{FPclose} generates only closed frequent itemsets whose support is larger than the
support of its super-itemset, it does not generate redundant sub-patterns with the same
support. In the above example, \{39, 68, 36\} is also a frequent sub-itemset. However, since
it is not closed, i.e., it is included in its super-itemset \{39, 68, 36, 92\} with the same support
27, we do not need to output it.

It is not enough to know only the closed programming patterns and their support values

<table>
<thead>
<tr>
<th>Source code</th>
<th>Preprocessed identifiers</th>
<th>Hash values</th>
</tr>
</thead>
<tbody>
<tr>
<td>linux-2.6.11/drivers/scsi/3w-9xxx.c: L1964 - 2088</td>
<td>T-Scsi_Host</td>
<td>92</td>
</tr>
<tr>
<td>int __devinit twa_probe(struct pci_dev *pdev,...) {</td>
<td>T-Scsi_Host</td>
<td>92</td>
</tr>
<tr>
<td>struct Scsi_Host *host = NULL;</td>
<td>F-scsi_host_alloc</td>
<td>39</td>
</tr>
<tr>
<td>host = scsi_host_alloc(&amp;driver_template, ...);</td>
<td>T-scsi_host_template</td>
<td>41</td>
</tr>
<tr>
<td>retval = scsi_add_host(host, &amp;pdev-&gt;dev);</td>
<td>T-pci_dev.R-dev</td>
<td>56</td>
</tr>
<tr>
<td>scsi_scan_host(host);</td>
<td>T-Scsi_Host</td>
<td>92</td>
</tr>
<tr>
<td>}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1: Example of parsing a function

The italic identifiers in source code are selected to analyze. They are prefixed with the
types as shown in the second column. Each preprocessed identifiers is then hashed to a
number. Only the last two digits of hash values are shown for simplicity.
(i.e., how many times the pattern occurs). It would be more helpful for programmers if we also record the functions in which each extracted pattern occurs. Such information is also needed later in violation detection in order to know which function violates an extracted rule. Unfortunately, the original algorithm *FPclose* and any other frequent itemset mining algorithms were not designed exactly for our purpose. They only output the support values for each discovered pattern but not their supporting itemsets. Therefore, we enhance *FPclose* to address this problem by also maintaining the supporting itemsets during the mining process. In the above example, PR-Miner outputs the closed frequent sub-itemset \{39, 68, 36, 92\} with the supporting itemset that corresponds to the 27 functions that contain this programming pattern.

### 4.2.3 Generating Programming Rules

As we explained briefly in Section 4.1, extracting only programming patterns is not enough because a pattern may lead to many different rules. Therefore, we also need to generate rules from patterns based on conditional probabilities.

#### A Naive Method

A naive method to generate programming rules from extracted patterns is to divide the items in each closed frequent sub-itemset into two parts and then calculate the confidence. In other words, from a closed frequent sub-itemset \( I \), we can compute the confidence for every possible association rules \( X \Rightarrow Y \), where \( X \) and \( Y \) are subsets of \( I \). The support of such a rule is equal to the support of \( I \), while the confidence of a rule is the conditional probability, i.e. \( \text{support}(I)/\text{support}(X) \), where \( \text{support}(X) \) is the number of occurrences of sub-itemset \( X \) in the itemset database, which also equals to the maximum support of any closed frequent itemset that contains \( X \). Basically, the confidence indicates the conditional probability that if \( X \) occurs, the likelihood for \( Y \) to occur. Rules with confidence smaller than a specified threshold (e.g. 90%) are pruned. And the remaining rules are outputted to...
the specification files to be examined and referenced by programmers.

Let us consider the above example again. After PR-Miner finds a programming pattern \{alloc, add, scan, Scsi\_Host\}. From this pattern, the naive method can generate 14 different possible rules by partitioning these three functions and the data type into 2 subsets in all possible ways such as \{add\}⇒\{alloc, scan, Scsi\_Host\}, and \{add, alloc\}⇒\{scan, Scsi\_Host\}, and so forth. All these rules have the support of 27. From the programming patterns discovered by FPclose, we know that the support for \{add\} is 37, and the support for \{add, alloc\} is 29. Therefore, the confidences for these 2 rules are 27/37 = 72.9% and 27/29 = 93.1%, respectively. The confidences for the other 12 rules can also be computed similarly. So if we set the confidence threshold to be 90%, the first rule \{add\}⇒\{alloc, scan, Scsi\_Host\} is pruned, while the second rule is outputted.

The biggest problem with the naive method is that it needs to examine all possible rules from each mined patterns. A programming pattern with \(k\) elements can generate up to \((2^k - 2)\) rules, which is impractical for long patterns. For example, our evaluation with large software code shows that some programming patterns are composed of more than 20 elements. Therefore, it is time and space inefficient to use this naive method to generate rules from patterns.

**Generating Closed Rules**

Instead of examining all possible programming rules from a mined pattern like in the naive method, PR-Miner examines only closed rules. As we explained in Section 4.1 it is enough to generate only closed rules since other rules are subsumed by closed rules.

To further reduce the number of outputted rules and speed up the generation as well as the violation detection processes, PR-Miner stores closed rules in **condensed format**. Formally, the condensed format for a closed frequent sub-itemset \(I\) is:

\[
I : s|\{C_1 : s_1|s_1 > s\} \ldots \{C_m : s_m|s_m > s\}
\]
where $C_1\ldots C_m$ are all subsets of $I$ whose supports ($s_1\ldots s_m$) are different from $I$’s. Obviously, $s_1\ldots s_m$ are all larger than $s$. Such condensed format can represent all the closed rules derived from $I$ and their confidences can be computed easily. For a closed rule $X \Rightarrow Y$ derived from $I$, if $X$ equals to $C_i$ (i.e. a subset of $I$ with a support larger than $I$), the confidence of the rule is $s/s_i$; otherwise, the confidence of the rule is 100%.

For example, suppose $FPclose$ extracts two closed frequent sub-itemsets: $\{a\} : 4$ and $\{a, b, d\} : 3$. The condensed format that represents all the closed rules derived from $\{a, b, d\}$ is

$$\{a, b, d\} : 3|\{a : 4\}$$

It explicitly expresses that the rule $\{a\} \Rightarrow \{b, d\}$ has confidence $3/4=75\%$, and also infers that any of the other 5 closed rules, such as $\{a, b\} \Rightarrow \{d\}$ has confidence 100%.

Now the rule generation problem becomes how to find out all of the subset $C_i$ that has a support $s_i$ larger than $s$. Since the support of $C_i$ is larger than $s$, it indicates that $C_i$ should be contained in another closed frequent sub-itemset (based on the definition of closed frequent sub-itemset). Since $C_i$ may include in multiple other closed frequent sub-itemset, PR-Miner needs to find the one with the maximum support.

To achieve this goal, PR-Miner uses a clever idea that converts this problem back to a frequent sub-itemset mining again. In other words, PR-Miner uses $FPclose$ one more time to find common sub-itemsets from frequent sub-itemsets generated by the first pass of $FPclose$. Doing such will find all common subsets among the closed frequent sub-itemsets generated in the first pass. Let $CommonSub$ denote all the common subsets generated by the second pass of $FPclose$. If a subset $C_i$ of $I$ is included in $CommonSub$, we can immediately find out which super-itemset of $C_i$ has the maximum support. The support of this super-itemset must be equal to the support of $C_i$ based on the definition of closed frequent sub-itemsets. We can easily prove this by contradiction. Note that the basic operation we need is to compute the common subsets for each pair of the closed frequent sub-itemsets. Therefore,
we can apply the frequent itemset mining algorithm again on the closed frequent sub-itemsets with minimum support of 2. Our algorithm ClosedRules for generating closed rules in condensed format is shown in Figure 4.6.

**Algorithm: ClosedRules(\(\mathcal{I}\))**

**Input:** \(\mathcal{I} = \{I_k|1 \leq k \leq n\}\),

\(I_k\) has 3 fields \(<F_k, s_k, E_k>\);

**Output:** The closed rules \(\mathcal{R}\) in condensed format.

1: Sort \(\mathcal{I}\) by supports in descending order such that \(s_1 \geq s_2 \geq ... \geq s_n\)

2: Mine common closed frequent sub-itemsets from \(\mathcal{I}\):

\(C \leftarrow \text{FPclose}(\{F_i|i = 1, 2, ..., n\}, 2)\),

where \(C = \{C_i|1 \leq i \leq m\}\) and \(C_i\) has 3 fields \(<F_i', s_i', E_i'>\)

3: for \(i = 1, 2, ..., m\)

4: Denote \(E_i' = \{i_j|1 \leq j \leq s_i'\}\)

5: for \(j = 2, 3, ..., s_i'\)

6: if \(s_{i_1} > s_{i_j}\)

7: Insert \(F_i' : s_{i_1}\) to sub-itemset \(I_{ij}\) in \(\mathcal{R}\)

---

**Figure 4.6:** Algorithm ClosedRules

Generating closed rules \(\mathcal{R}\) in condensed format from closed frequent itemsets \(\mathcal{I}\) mined from the first step explained in Section 4.2.2. The close frequent mining algorithm FPclose takes an itemset database and the minimum support threshold as input, and outputs the closed frequent sub-itemsets, each of which has three fields \(<F_i, s_i, E_i>\), where \(F_i\) is the frequent itemset itself, \(s_i\) is its support, \(E_i\) is the indexes of its supporting itemsets, and \(E_i\) is sorted in an ascending order. Similarly, \(<F_i', s_i', E_i'>\) have the same meanings but are generated by the second pass of FPclose (line 2) to a database that consists of all closed frequent sub-itemsets, i.e. \(\{F_i|i = 1, 2, ..., n\}\).

In the ClosedRules algorithm, it first sorts the frequent itemsets \(\mathcal{I}\) mined from FPclose (line 1) so that it can quickly locate the frequent itemset with the maximal support for any common sub-itemset. In line 2, it calls FPclose with minimum support of 2 to find out all common sub-itemsets \(\mathcal{C}\) from \(\mathcal{I}\). For each common sub-itemset \(C_i\) (line 3), ClosedRules inserts it with its support to the corresponding rule of condensed format as follows. \(E_i'\) includes the indexes of all \(C_i\)'s supporting itemsets in \(\mathcal{I}\). The first supporting itemset \(I_{i_1}\) has the maximum support for \(C_i\), because all indexes in \(E_i'\) are sorted based on their corresponding itemset's support. For the other supporting itemset \(I_{ij}\) (line 5), if its support
$s_{ij}$ is smaller than $s_{i1}$ (line 6), $C_i$ is inserted into the subset of the rule for the closed frequent itemset $I_{ij}$. This way, with only one pass it can insert $C_i$ into all rules that are super-itemsets of $C_i$ but have smaller support than $C_i$.

ClosedRules performs much better than the naive algorithm in terms of space and time, because it does not need to examine all possible rules generated from extracted programming patterns.

By calling ClosedRules on the closed frequent sub-itemsets that correspond to the extracted programming patterns, PR-Miner obtains the closed rules in the condensed format expressed in numbers, and then it maps the closed rules back to programming rules and stores them into a specification file. The programmers can then validate the programming rules so that later they can use them as specifications, and also new programmers can read them when they start coding to avoid mistakes.

Since PR-Miner extracts programming rules based on occurrences, some false positives may be introduced if some elements only coincidentally appear together for many times in the source code. However, a rule with larger supports can be more believable. Therefore, PR-Miner ranks the rules based on supports: programmers can examine those rules that are ranked in the top 100 or 500. Furthermore, as we explained early, rules with confidence lower than the specified threshold (e.g. 90%) are pruned. Additionally, some other ranking method such as giving weights for different elements as in Engler et al’s work [ECC01] can also be applied.

4.2.4 Detecting Violations to Extracted Rules

Based on the programming rules generated from the previous step, PR-Miner can find potential bugs by detecting violations to these rules. The main idea is that the programming rules usually hold for most cases and violations happen only occasionally. Take the potential bug detected by PR-Miner in Figure 4.4 as an example. The function call to scan should follow alloc and add as the programming rule indicates. This rule appears 27 times in Linux, but
there are 2 cases violating this rule because \texttt{scan} is missing.

As shown in Figure 4.5, PR-Miner first detects violations to the extracted programming rules, then prunes the false violations using inter-procedural analysis, and finally ranks the violations in the error report.

\textbf{Detecting Violations}

In order to detect violations to the programming rules, a naive method is to generate all possible programming rules and then check them upon the source code one by one. As we discussed in Section 4.2.3, there would be an exponential number of rules that need to be checked.

Fortunately, it is unnecessary to check all programming rules. First, if the rule has a low confidence, it is already pruned in the rule-generation step. In other words, if the confidence threshold is \( t \), any rules with confidence smaller than \( t \) are discarded. Second, if the rule has 100\% confidence, it indicates that there is no violation for the rule. Therefore, we only need to check the rules with confidence in the range \([ t, 100\% )\).

The main idea of the violation detection process is straightforward. For example, if a rule \( \{ a, b \} \Rightarrow \{ d \} \) has a support of 100 and \( \{ a, b \} \) has a support of 101, there is only one out of 101 cases that has \( \{ a, b \} \) but not \( \{ d \} \), which indicates that this case violates the rule \( \{ a, b \} \Rightarrow \{ d \} \). In other words, this case is likely to be a bug. But if \( \{ a, b \} \) has a support of 200, the rule \( \{ a, b \} \Rightarrow \{ d \} \) will be pruned as its confidence is only 50\%.

Since PR-Miner stores generated programming rules in the condensed format that explicitly indicates which rules have confidence less than 100\% but greater than the specified threshold \( t \), we can easily figure out which rules have violations. Even more efficiently, PR-Miner detects violations during the same process when it generates the programming rules by calling \textsc{ClosedRules} as shown in Figure 4.6. To do that, PR-Miner computes the confidence for the rule \( F'_i \Rightarrow (F'_i - F'_i) \) in the loop of line 5 as \( c = s_{ij}/s_{ii} \). If \( t \leq c < 1 \), it indicates that there are violations to this rule. The violations can be easily figured out by
comparing the supporting itemsets for the closed frequent sub-itemsets \( I_i \) and \( I_{ij} \) as follows. \( F_i \) contains the common sub-itemset \( F'_i \), but it does not contain \( (F_i \neq F'_i) \). It means that some supporting itemsets in \( E_i \) violate the rule \( F'_i \Rightarrow (F_i \neq F'_i) \). On the other hand, this rule is supported by the supporting itemsets \( E_{ij} \) for \( F_{ij} \). Therefore, the itemsets in \( E_i \) but not in \( E_{ij} \) violate this rule, and so the corresponding functions of the itemsets violate the programming rule.

### Pruning False Violations

The violation detection above can result in false positives if the elements in a programming rule span across multiple functions. The reason is that PR-Miner detects violations using only intra-procedural analysis because, as described in Section 4.2.2, each itemset in the database corresponds to a function definition. Suppose in an example with a function-pair (\texttt{lock} and \texttt{unlock}) rule, \texttt{unlock} is called inside a function \texttt{F} but \texttt{lock} is not. Instead, \texttt{F} calls another function \texttt{try\_lock} that calls \texttt{lock}. Without inter-procedural analysis, PR-Miner would report that \texttt{F} contains a violation of missing \texttt{lock}, even though \texttt{F} contains \texttt{lock} in its callee.

In order to prune the above false violations, PR-Miner performs an inter-procedural checking. It first checks the callees’ paths for each function that contains violations. For each violation of rule \( X \Rightarrow Y \) in function \( F \), PR-Miner checks whether every item \( y \in Y \) is in the functions \( F_1, \ldots, F_n \) called by \( F \). As shown in Figure 4.7(a), we can follow the calling path more deeply by checking the functions called by callees \( F_1, \ldots, F_n \) in \( F \). If the missing items are in any of the calling paths, it is a false violation. For time efficiency, PR-Miner limits the checking depth. Since PR-Miner outputs all function calls in each function definition as described in Section 4.2.2, it is easy to follow the calling path during checking.

Besides callees, PR-Miner also checks the callers to prune false positives. In the example above, there is also a violation in the function \texttt{try\_lock} because \texttt{lock} is in \texttt{try\_lock} but \texttt{unlock} is not in it. In order to prune such false violations, PR-Miner also checks whether
the missing items are in the caller functions’ paths as shown in Figure 4.7(b). In order to check the call path backwards, PR-Miner maintains a caller list for each function $F$ that consists of the indexes of the functions that call $F$. If the missing items for function $F$ are in the paths of all of its callers, it is a false violation.

**Ranking and Reporting Bugs**

After PR-Miner detects rule violations and prunes false positives, it ranks all remaining violations and reports them to programmers.

PR-Miner ranks the violations based on the confidence of the violated rules. Since a function may contain several violations, PR-Miner groups all violations of the same function together, and the violation with the highest confidence is assigned as the confidence of the violated function. The confidence of a violated function can be considered the possibility that the function has bugs. In the current version of PR-Miner, it simply ranks the bugs by the confidence. Because several functions may have the same violation, the potential bugs in these functions are strongly correlated. Therefore, some other advanced ranking schemes such as correlation ranking [KAYE04] can be used here to further improve the accuracy of our ranking function.
4.3 Evaluation

4.3.1 Experiment Setup

We have evaluated PR-Miner with the latest versions of Linux, PostgreSQL, and the Apache HTTP Server. The numbers of files, lines of code (LOC) and functions are shown in Table 4.2.

PR-Miner takes three parameters: \textit{min\_support}, the confidence threshold, and maximal checking path depth. By default, we set the \textit{min\_support} as 15, the confidence threshold as 90\%, and the maximal depth of call path as 3 for pruning false violations.

The parser for PR-Miner is GCC 3.3.4 [St05] with a small modifications. In our experiments, we run PR-Miner on an Intel Xeon 1.5 GHz machine with 4GB memory and Linux 2.4.20 system.

4.3.2 Extracting Implicit Programming Rules

Table 4.3 shows the number of closed rules discovered by PR-Miner in the evaluated software. Rules that have confidence lower than the threshold (90\%) are pruned automatically by PR-Miner and are thereby not included in the results reported in this section. From the closed rules, programmers can easily infer other rules subsumed by these closed rules. These closed rules can be classified into three categories: function-function (F-F) rules, variable-variable (V-V) rules, and function-variable (F-V) rules. F-F rules involve only functions, V-V rules involve only variables (including fields in structure) or their data types, while F-V rules involve both functions and variables.

\begin{table}[h]
\centering
\begin{tabular}{|l|c|c|c|c|}
\hline
Software & version & #C files & LOC & #functions \\
\hline
Linux & 2.6.11 & 3,538 & 3,037,403 & 73,607 \\
PostgreSQL & 8.0.1 & 409 & 381,192 & 6,964 \\
Apache & 2.0.53 & 160 & 84,724 & 1,912 \\
\hline
\end{tabular}
\caption{Software evaluated in our experiments.}
\end{table}
<table>
<thead>
<tr>
<th>Software</th>
<th>Total</th>
<th>F-F</th>
<th>V-V</th>
<th>F-V</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>32,283</td>
<td>1,075</td>
<td>8,883</td>
<td>22,325</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>6,128</td>
<td>379</td>
<td>687</td>
<td>5,062</td>
</tr>
<tr>
<td>Apache</td>
<td>283</td>
<td>33</td>
<td>92</td>
<td>158</td>
</tr>
</tbody>
</table>

Table 4.3: The number of closed rules extracted by PR-Miner
Notice that all F-V rules that contain more than two elements also include F-F and/or V-V rules as their sub-rules.

Our results show that a large number of implicit, undocumented programming rules can be effectively extracted from source code by PR-Miner without any priori knowledge or annotations/specifications from programmers. For example, PR-Miner extracts a total of 32,283 implicit, undocumented closed programming rules from Linux. It would be very difficult for programmers to manually specify these many programming rules. PR-Miner effectively relieves such burden from programmers by efficiently and automatically extracts such rules from source code.

The results also show that around 88.3–96.7% rules involve variables. For example, there are around 9000 V-V rules in Linux, along with a large number of variable correlations contained in F-V rules. Comparing with the previous studies such as Engler et al's work [ECC01] that do not consider rules about variable correlations, PR-Miner can extract substantially more programming rules.

**Supports of Programming Rules**

Figure 4.8 shows the support distribution of closed rules extracted by PR-Miner in Linux. As expected, the number of closed rules decreases when the corresponding support increases. The decreasing rate is approximately exponential from 15 to 80 (notice that Y-axis is in logarithmic scale). Since the rules with larger support are more “believable”, programmers can increase min_support to improve the quality of rules, or choose only those top ranked closed programming rules (Extracted rules are ranked by their supports).

The figure also shows that some rules have large supports, strongly validating our observation that programmers follow many implicit programming rules in writing software. For
example, there are 1442 rules with supports larger than 100 in Linux, and the rule with the largest support is the function pair of `spin_lock_irqsave` and `spin_unlock_irqrestore`, which has a support of 3656.

![Distribution of rule support in the Linux code](image)

Figure 4.8: Distribution of rule support in the Linux code
Notice that Y-axis is in logarithmic scale.

**Rule Size**

Each programming rule contains several elements such as functions, variables and data types. The number of elements in a rule is called the rule size. Figure 4.9 shows the distribution of rule size in Linux. Around 4200 closed rules contain only 2 elements, which accounts for 14% of all closed rules. On the other hand, 9% of the closed rules have even more than 10 elements. For example, PR-Miner found a rule that contains 12 program elements and appears 38 times in Linux, which is followed when the system registers for a PCMCIA device.

The above results, along with the results shown on Table 4.3, indicate the generality of PR-Miner over the previous work \cite{ECC01} because PR-Miner does not constrain the rule format or limit the number of elements in the rules to only 2.
4.3.3 Detecting Violations

PR-Miner has reported many violations of programming rules in the evaluated software. We have manually examined the top 60 violations to differentiate bugs from false positives. Confirmed bugs have been reported to the corresponding developer community and are currently being fixed by developers. The numbers of the verified bugs are shown in Table 4.4. Currently, we are still inspecting the violation report and more bugs will be confirmed. More specifically, we have validated 16 bugs in Linux, 6 bugs in PostgreSQL, and 1 in the Apache HTTP Server. Almost all of these bugs are semantic bugs instead of those simple bugs such as buffer overflow, data races, etc. and are thereby difficult to be detected by existing bug detection tools. In addition, most of these bugs violate complex rules that involve more than 2 elements, so it is difficult for the previous work [ECC01] to detect them.

Notice that we directly apply the programming rules mined by PR-Miner in violation detection without having programmers validate the extracted rules. Therefore, false programming rules might result in false positives in violation report. If programmers can validate those topped ranked rules and prune false rules, the number of false positives generated by PR-Miner in violation detection should be smaller than those presented in Table 4.4.

Even though our inter-procedural pruning method can prune a lot of false positives in violation report, many false positives still exist. Even for the strong function-pair rules with
high confidence such as lock-unlock, there are still a few violations that are false positives. For example, the function `spin_lock_bh` (in Linux `kernel/spinlock.c`) includes a call to `spin_lock_irqsave` but not `spin_unlock_irqrestore` because `spin_lock_bh` is to provide locking functionality and thereby does not need unlocking in it. Such false positives can be pruned if we can also conduct deeper inter-procedural analysis. In addition, combining with some dynamic checking methods would be helpful to further prune these false positives.

However, even these false positives are still useful for automatic specification and annotation of function interfaces. In the example above, we can know the unlock function should be called somewhere after calling `spin_lock_bh`. Therefore, we can automatically annotate the function `spin_lock_bh` with such assumption. There are many cases that are more complex than this example. For example, PR-Miner reports a violation to a rule that says: if fields `counter` and `len` in a structure `sk_buff` are modified, the function `kfree_skb` should be called. This rule appears 480 times in Linux. It indicates that if these two fields are accessed, some memory is allocated for the data structure of `sk_buff` and thereafter it should be freed. However, there is one violation of this rule in the function `skb_clone` in the file `net/core/skbuff.c` with confidence $\frac{480}{481} = 99.8\%$. Although it is a false violation, it still indicates that after the function `skb_clone` is called, `kfree_skb` also should be called later in order to free the memory; otherwise, it would cause memory leak. Therefore, this violation can be used to automatically annotate the interface of the function `skb_clone`.

<table>
<thead>
<tr>
<th>Software</th>
<th>Inspected (top 60)</th>
<th>Uninspected</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Bugs</td>
<td>Specification</td>
</tr>
<tr>
<td>Linux</td>
<td>16</td>
<td>20</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>Apache</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4.4: Violations detected by PR-Miner

We have inspected the top 60 violations in the violation report. The inspected violations are classified into 3 categories: real bugs that have been confirmed, potential usage for function interface annotation, and false positives. Uninspected means that we are unable to inspect them yet due to time limitation.
Sensitivity of Parameters  Figure 4.10 shows the effect of the parameter settings on the number of violated functions reported by PR-Miner. From Figure 4.10(a), we can see that the number of violated functions decreases when \textit{min\_support} increases. However, comparing with the exponential decreasing rate of the number of rules in Figure 4.8, it decreases much more slowly. The reason is that some violations of different rules may indicate the same error but with different confidence and support. This is desirable for debugging because even though the number of rules increases significantly when \textit{min\_support} decreases, the number of violated functions only increases almost linearly so that it will not increase the false positive rate too much. Furthermore, multiple violations that indicate the same potential bug can strengthen its confidence, so it would be helpful for ranking to combine these violations.

Similarly, the number of violated functions decreases linearly with the confidence threshold as shown in Figure 4.10(b). The results indicate that the programmer can adjust both parameters in order to generate a reasonable number of errors in the report and verify them. With a fixed confidence threshold, ranking the error reports by confidence can also relieve the programmer to verify the potential bugs.

False Positives  Although PR-Miner can effectively detect violations and potential bugs, it turns out that some of the reports are false. By analyzing the results, most of the false
positives are caused by the following reasons:

- **False programming rules** Currently, PR-Miner detects violations using all of the rules mined from source code without verification by programmers. Since some programming elements coincidentally appear together for many times, they may be incorrectly considered as programming patterns, which result in false rules. In order to eliminate the false positives in bug detection caused by such false rules, programming rules should be verified by programmers before they are used for violation detection.

- **Incomplete programming rules** When PR-Miner parses the source code and converts it into the data for mining, it does not keep the information of branch conditions. However, some rules are sensitive to branch conditions. For example, the rule `malloc ⇒ free` is not a complete rule because after `malloc` is called, if the allocation fails, it does not need to call `free`. The complete rule for this case should be: `malloc and return is not null ⇒ free`. In order to get the complete rules containing branch conditions, when PR-Miner converts source code, it should also tokenize the conditions. However, it is a challenging problem because it is hard to present the accurate information of a branch.

- **Inaccurate source code analysis** Since PR-Miner does not consider any path or context information, it can also cause false positives in violation detection. For example, for the rule `malloc ⇒ free`, `malloc` is called only in one of the branches upon some special condition, and in most of the other conditions, there is no need to call `free`. Further, the inter-procedure pruner in PR-Miner does not consider function pointers, so it cannot get the exact call graph when it does pruning.

### 4.3.4 Time and Space Overheads

PR-Miner can extract programming rules and detect violations in large software very efficiently. The execution time and space overhead is shown in Table 5.5.
It takes less than 1 minute for PR-Miner to extract more than 32,000 closed rules in Linux with more than 3500 files, and only several seconds for PostgreSQL and Apache. The results also show that PR-Miner can efficiently detect violations. For example, it takes less than 1 minute to detect violations.

PR-Miner is also space-efficient for rule extraction and violation detection. For example, it takes less than 500MB for Linux, 25MB for PostgreSQL and only 7MB for Apache. Therefore, PR-Miner is a practical method to extract programming rules from large software in just an ordinary PC machine.

<table>
<thead>
<tr>
<th>Software</th>
<th>Extracting rules</th>
<th>Detecting violations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time(s)</td>
<td>Space(MB)</td>
</tr>
<tr>
<td>Linux</td>
<td>42</td>
<td>441</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>5</td>
<td>25</td>
</tr>
<tr>
<td>Apache</td>
<td>1</td>
<td>7.3</td>
</tr>
</tbody>
</table>

Table 4.5: Execution time and memory space of PR-Miner

4.4 Summary

This chapter presents a general technique called PR-Miner that uses frequent itemset mining to efficiently and automatically extract implicit, undocumented programming rules and detect violations in large software code written in C with little efforts from programmers. The rules extracted by PR-Miner are in general forms, including both simple pair-wise rules and complex ones with multiple elements of different types.

We have evaluated PR-Miner with the latest versions of large software code including Linux, Apache HTTP Server and PostgreSQL with up to 3 million lines of code. PR-Miner takes only 1–42 seconds to extract more than 32,000 closed programming rules and also only 1–46 seconds to detect violations. In addition, PR-Miner has detected many violations to the extracted rules. Among the top 60 violations reported by PR-Miner, 16 bugs are confirmed in the latest version of Linux, 6 in PostgreSQL and 1 in Apache. Many of these bugs are
currently being fixed by developers after we reported them. Most of these bugs violate complex rules that contain more than 2 elements and are thereby difficult to be detected by previous tools.
Chapter 5
CP-Miner

5.1 Overview

Copying and pasting code is a common practice in software development. In order to reduce programming effort and shorten programming time, programmers prefer reusing a piece of code via copy-paste rather than rewriting similar code from scratch. Recent studies [Bak95, DRD99, KG03] have shown that a large portion of code is duplicated in software. For example, Kapser and Godfrey [KG03], using a copy-paste detection tool called CCFinder [KK02], found that 12% of the Linux file system code (279K lines) was involved in code cloning activity. Baker [Bak95] found that in the complete source of the X Window system (714K lines), 19% of the code was identified as duplicates.

Using abstractions such as functions and macros to remove this duplication might improve software maintenance; however, much duplication will likely remain, for two possible reasons. First, some changes are usually necessary, and copy-paste is much easier and faster than abstraction. Another reason is that functions may impose higher overhead. However, the psychological reasons for large percentage of existing copy-pasted code are beyond the scope of our work.

Copy-pasted code is prone to introducing errors. For example, Chou et al. [CYC+01] found that in a single source file under the Linux drivers/i2o directory, 34 out of 35 errors were caused by copy-paste. One of the errors was copied in 10 places and another in 24. They also showed that many operating system errors are not independent because programmers are ignorant of system restrictions in copy-pasted code. In our study, we have detected 28
Figure 5.1: An example of a copy-paste related error detected by CP-Miner.
This bug appears in `linux-2.6.6/arch/sparc64/prom/memory.c`. A similar bug is also
detected in file `/arch/sparc/prom/memory.c`.

copy-paste related bugs in the latest version of Linux and 23 in FreeBSD. Most of these bugs
were previously unreported.

A major reason why copy-paste introduces bugs is that programmers forget to modify
identifiers (variables, functions, types, etc.) consistently throughout the pasted code. This
mistake will be detected by a compiler if the identifier is undefined or has the wrong type.
However, these errors often slip through compile-time checks and become hidden bugs that
are very hard to detect.

Figure 5.1 shows an example of a bug detected by CP-Miner in the latest version of Linux
(2.6.6). We reported this bug to the Linux kernel community and it has been confirmed by
kernel developers [lin05]. In this example, the loop in lines 111–118 was copied from lines
92–99. In the new copy-pasted segment (lines 111–118), the variable `prom_phys_total` is
replaced with \textit{prom\_prom\_taken} in most of the cases except the one in line 117 (shown in bold font). As a result, the pointer \textit{prom\_prom\_taken[iter].theres\_more} incorrectly points to the element of \textit{prom\_phys\_total} instead of \textit{prom\_prom\_taken}. This bug is a semantic error, and therefore it cannot be easily detected by memory-related bug detection tools including static checkers [CLL+02, ECC01, EA03, HCXE02, MPC+02, SD95] or dynamic tools such as Purify [HJ92], Valgrind [SNF+], and CCured [NMW02]. Besides this bug, CP-Miner has also detected many other similar bugs caused by copy-paste in Linux, FreeBSD, PostgreSQL and Web Apache.

While one can imagine augmenting the software development tools and editors with copy-paste tracking, this support does not currently exist. Therefore, we are focusing on detecting likely copied and pasted code in an existing code base. Not all code segments identified by previous detection tools and our tool are really the results of copy-paste (even though we prune many of the false copy-pasted segments as described in Section 5.3.1), but for simplicity we refer likely-copy-pasted segments as copy-pasted segments.

It is a challenging task to efficiently extract copy-pasted code in large software such as an operating system. Even though some previous studies [Gri81, Jan88] have addressed the related problem of plagiarism detection, they are not suitable for detecting copy-pasted code. Those tools, such as the commonly used JPlag [PMP02], were designed to measure the degree of similarity between a pair of programs in order to detect cheating. If these tools were to be used to detect copy-pasted code in a single program without any modification, they would need to compare all possible pairs of code fragments. For a program with \(n\) statements, a total of \(O(n^3)\) pairwise comparisons would need to be performed. This complexity is certainly impractical for software with millions of lines of code such as Linux and FreeBSD. Of course, it is possible to modify these tools to identify copy-pasted code in single software, but the modification is not trivial and straightforward.

\footnote{Considering comparison between the pair of code fragments with \(k\) statements, there are \((n-k+1)\) different fragments. So there are \(\binom{n-k+1}{2} = O(n^2)\) possible pair comparisons. Since \(k\) can be \(1, 2, ..., \frac{n}{2}\), the total number of pairwise comparisons is \(O(n^3)\).}
5.1.1 State of the Art

So far, only a few tools have been proposed to identify copy-pasted code in a single program. Examples of such tools include Moss [Aik05, SWA03], Dup [Bak95], CCFinder [KKI02] and others [Bak92, BYM+98]. Most of these tools suffer from some or all of the following limitations:

- **Efficiency:** Most existing tools are not scalable to large software such as operating system code because they consume a large amount of memory and take a long time to analyze millions of lines of code.

- **Tolerance to modifications:** Most tools cannot deal with modifications in copy-pasted code. Some tools [DRD99, Joh94] can only detect copy-pasted segments that are exactly identical. Moreover, most of the existing tools do not allow statement insertions or modifications in a copy-pasted segment. Such modifications are very common in standard practice. Our experiments with CP-Miner show that about one third of copy-pasted segments contain insertion or modification of 1-2 statements.

- **Bug detection:** The existing tools cannot detect copy-paste related bugs. They only aim at detecting copy-pasted code and do not consider bugs associated with copy-paste.

5.1.2 Contributions of CP-Miner

This chapter presents CP-Miner, a tool that uses data mining techniques to efficiently identify copy-pasted code in large software including operating system code, and also detects copy-paste related bugs. It requires no modification or annotation to the source code of software being analyzed. Our work makes three main contributions:

1. **A scalable copy-paste detection tool for large software:** CP-Miner can efficiently find copy-pasted code in large software including operating system code. Our experimental results show that it takes less than 20 minutes for CP-Miner to detect
150,000–190,000 different copy-pasted segments that account for about 20–22% of the source code in Linux and FreeBSD (each with more than 3 million lines of code). Additionally, it takes less than one minute to detect copy-pasted segments in Apache web server and PostgreSQL, accounting for about 17–22% of total source code.

Compared to CCFinder [KKI02], CP-Miner is able to find 17–52% more copy-pasted segments because CP-Miner can tolerate statement insertions and modifications.

2. Detection of bugs associated with copy-paste: CP-Miner can detect copy-paste related bugs such as the one shown in Figure 5.1, most of which are hard to detect with existing static or dynamic bug detection tools. More specifically, CP-Miner has detected 28 potential bugs in the latest version of Linux, 23 in FreeBSD, 5 in Web Apache, and 2 in PostgreSQL. Most of these bugs had never been reported.

We have reported these bugs to the corresponding developers. So far five bugs have recently been confirmed and fixed by Linux developers, and one bug has been confirmed and fixed by Apache developers.

3. Statistical study of copy-pasted code distribution in operating system code:

Few previous studies have been conducted on the characteristics of copy-paste in large software. Our work analyzed some interesting statistics of copy-pasted code in Linux and FreeBSD. Our results indicate that (1) copy-pasted segments are usually not too large, most with 5–16 statements; (2) although more than 50% of copy-pasted segments have only two copies, a few (6.3–6.7%) copy-pasted segments are copied more than 8 times; (3) there is a significant number (11.3–13.5%) of copy-pasted segments at function granularity (copy-paste of an entire function); (4) most (65–67%) copy-pasted segments require renaming at least one identifier, and 23–27% of copy-pasted segments have inserted, modified, or deleted one statement; (5) different OS modules have very different copy-paste coverage: drivers, arch, and crypt have higher percentage of copy-paste than other modules in Linux; (6) as the operating system code evolves,
amount of copy-paste also increases, but the coverage percentage of copy-pasted code remains relatively stable over the recent versions of Linux and FreeBSD.

5.2 Background in Identifying Copy-pasted Code

Since copy-pasted code segments are usually similar to the original ones, detection of copy-pasted code involves detecting code segments that are identical or similar.

Previous techniques for copy-paste detection can be roughly classified into three categories: (1) string-based, in which the program is divided into strings (typically lines), and these strings are compared against each other to find sequences of duplicated strings [Bak95]; (2) parse-tree-based, in which pattern matching is performed on the parse-tree of the code to search for similar subtrees [BYM+98, KGD95]; (3) token-based, in which the program is divided into a stream of tokens and duplicate token sequences are identified [KKI02, PMP02].

Our tool, CP-Miner, is token-based. This approach has advantages over the other two. First, a string-based approach does not exploit any lexical information, so it cannot deal with simple modifications such as identifier renaming. Second, using parse trees can introduce false positives because two segments with identical syntax trees are not necessarily copy-pasted. This is because copy-paste is code-based rather than syntax-based, i.e., it reuses a piece of code rather than an abstract syntax structure.

Most previous copy-paste detection tools do not sufficiently address the limitations described in Section 5.1. Most of them consume too much time or memory to be scalable to large software, or do not tolerate modifications made in copy-pasted code. In contrast, CP-Miner can address both challenges.
5.3 Design of CP-Miner

CP-Miner has two major functionalities: detecting copy-pasted code segments, and finding copy-paste related bugs. It requires no modification to the source code of software being analyzed. The following two subsections describe the design for each functionality.

5.3.1 Identifying Copy-pasted Code

To detect copy-pasted code, CP-Miner first converts the problem into a frequent subsequence mining problem. It then uses an enhanced algorithm of CloSpan to find basic copy-pasted segments. Finally, it prunes false positives and composes larger copy-pasted segments. For convenience, we refer to a group of code segments that are similar to each other as a copy-paste group.

CP-Miner can detect copy-pasted segments efficiently because it uses frequent subsequence mining techniques that can avoid many unnecessary or redundant comparisons. To map our problem to a frequent subsequence mining problem, CP-Miner first maps a statement to a number, with similar statements being mapped to the same number. Then, a basic block (i.e., a straight-line piece of code without any jumps or jump targets in the middle) becomes a sequence of numbers. As a result, a program is mapped into a database of many sequences. By mining the database using CloSpan, we can find frequent subsequences that occur at least twice in the sequence database. These frequent subsequences are exactly copy-pasted segments in the original program. By applying some pruning techniques such as identifier mapping, we can find basic copy-pasted segments, which can then be combined with neighboring ones to compose larger copy-pasted segments.

CP-Miner is capable of handling modifications in copy-pasted segments for two reasons. First, similar statements are mapped into the same value. This is achieved by mapping all identifiers (variables, functions and types) of the same type into the same value, regardless of their actual names. This relaxation tolerates identifier renaming in copy-pasted segments.
Even though false positives may be introduced during this process, they are addressed later through various pruning techniques such as identifier mapping (described in Section 5.3.1). Second, we have enhanced the basic frequent subsequence mining algorithm, CloSpan, to support gap constraints in frequent subsequences. This enhancement allows CP-Miner to tolerate 1–2 statement insertions, deletions, or modifications in copy-pasted code. Insertions and deletions are symmetric because a statement deletion in one copy can also be seen as an insertion in the other copy. Modification is a special case of insertion. Basically, the modified statement can be treated as if both segments have a statement inserted.

Figure 5.2 illustrates the process of how CP-Miner identifies copy-pasted segments. The main steps of the process include:

1. Parsing source code: Parse the given source code and build a sequence database (a collection of sequences). In addition, information regarding basic blocks and block nesting levels are also passed to the mining algorithm.

2. Mining for basic copy-pasted segments: The enhanced frequent subsequence mining algorithm is applied to the sequence database to find basic copy-pasted segments.

3. Pruning false positives: Various techniques including identifier mapping are used to prune false positives.

4. Composing larger copy-pasted segments: Larger copy-pasted segments are identified by combining consecutive smaller ones. The combined copy-pasted segments are fed back to step (3) to prune false positives. This is necessary because the combined one may not be copy-pasted, even though each smaller one is.

Like other copy-paste detection tools, CP-Miner can only detect copy-pasted segments, but cannot tell which segment is original and which is copy-pasted from the original. Fortunately, this limitation is not a big problem because in most cases it is enough for programmers
to know what segments are similar to each other. Moreover, our bug detection method described in Section 5.3.2 does not rely on such differentiation. Additionally, if programmers really need the differentiation, navigating through RCS versions could help figuring out which segment is the original copy.

**Parsing Source Code**

The main purpose of parsing source code is to build a sequence database (a collection of sequences) in order to convert the copy-paste detection problem to a frequent subsequence mining problem. Comments are not considered normal statements in CP-Miner, and are thereby filtered by our parser. The current prototype of the CP-Miner parser only works for programs written in C or C++, but it is easy to modify it for other programming languages.

A statement is mapped to a number by first tokenizing its components such as variables, operators, constants, functions, keywords, etc. To tolerate identifier renaming in copy-pasted segments, identifiers of the same type are mapped into the same token. Constants are handled in the same way as identifiers: constants of the same type are mapped into the same token. However, operators and keywords are handled differently, with each one mapped
to a unique token. After all the components of a statement are tokenized, a hash value digest is computed using the “hashpjw” \cite{ASU86} hash function, chosen for its low collision rate. Figure 5.3 shows the hash value for each statement in the example shown in Figure 5.1 of Section 5.1. As shown in this figure, the statement in lines 93–94 and the statement in lines 112–113 have the same hash values.

After each statement is mapped, the program becomes a long number sequence. Unfortunately, the frequent subsequence mining algorithms need a collection of sequences (a sequence database) as described in Section 2.2, so we need a way to cut this long sequence into many short ones. One simple method is to use a fixed cutting window size (e.g., every 20 statements) to break the long sequence into many short ones. This method has two disadvantages. First, some frequent subsequences across two or more windows may be lost. Second, it is not easy to decide the window size: if it is too long, the mining algorithm would be very slow; if too short, too much information may be lost on the boundary of two consecutive windows.

Instead, CP-Miner uses a more elegant method to perform the cutting. It takes advantage of some simple syntax information and uses a basic programming block as the unit to break the long sequence into short ones. The idea for this cutting method is that a copy-pasted segment is usually either a part of a basic block or consists of multiple basic blocks. In addition, basic blocks are usually not too long to cause performance problems in CloSpan. By using a basic block as the cutting unit, CP-Miner can first find basic copy-pasted segments and then compose larger ones from smaller ones. Since different basic blocks have a different number of statements, their corresponding sequences also have different length. But this is not a problem for CloSpan because it can deal with sequences of different sizes. The example shown in Figure 5.3 is converted into the following collection of sequences:

(35487793)
......
(67641265)
(133872016, 133872016, 82589171)
<table>
<thead>
<tr>
<th>STATEMENT</th>
<th>HASH</th>
</tr>
</thead>
<tbody>
<tr>
<td>68 void __init prom_meminit(void)</td>
<td>35487793</td>
</tr>
<tr>
<td>69 {</td>
<td></td>
</tr>
<tr>
<td>92  for(iter=0; iter&lt;num_regs; iter++) {</td>
<td></td>
</tr>
<tr>
<td>93    prom_phys_total[iter].start_adr = prom_reg_memlist[iter].phys_addr;</td>
<td></td>
</tr>
<tr>
<td>94    prom_phys_total[iter].num_bytes = prom_reg_memlist[iter].reg_size;</td>
<td></td>
</tr>
<tr>
<td>95    prom_phys_total[iter].theres_more = &amp;prom_phys_total[iter+1];</td>
<td>67641265</td>
</tr>
<tr>
<td>96    &amp;prom_phys_total[iter+1];</td>
<td></td>
</tr>
<tr>
<td>112  for(iter=0; iter&lt;num_regs; iter++) {</td>
<td></td>
</tr>
<tr>
<td>113    prom_prom_taken[iter].start_adr = prom_reg_memlist[iter].phys_addr;</td>
<td></td>
</tr>
<tr>
<td>114    prom_prom_taken[iter].num_bytes = prom_reg_memlist[iter].reg_size;</td>
<td></td>
</tr>
<tr>
<td>115    &amp;prom_phys_total[iter+1];</td>
<td>67641265</td>
</tr>
<tr>
<td>133872016</td>
<td></td>
</tr>
<tr>
<td>133872016</td>
<td></td>
</tr>
<tr>
<td>82589171</td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.3: An example of hashing statements

......
(67641265)
(133872016, 133872016, 82589171)
......

Besides a collection of sequences, the parser also passes to the mining algorithm the source code information of each sequence. Such information includes (1) the nesting level of each basic block, which is later used to guide the composition of larger copy-pasted segments from smaller ones; (2) the file name and line number, which is used to locate the copy-pasted code corresponding to a frequent subsequence identified by the mining algorithm.

Mining for Basic Copy-pasted Segments

After CP-Miner parses the source code of a given program, it generates a sequence database with each sequence representing a basic block. At the next step, it applies the frequent
subsequence mining algorithm, CloSpan, on this database to find frequent subsequences with support value of at least 2, which corresponds to code segments that have appeared in the program at least twice. In the example shown in Figure 5.3, CP-Miner would find (133872016, 133872016, 82589171) as a frequent subsequence because it occurs twice in the sequence database. Therefore, the corresponding code segments in line 111–118 and line 92–99 are basic copy-pasted segments.

Unfortunately, the mining process is not as straightforward as expected. The main reason is that the original CloSpan algorithm was not designed exactly for our purpose, and nor were other frequent subsequence mining algorithms. Most existing algorithms including CloSpan have the following two limitations that we had to enhance CloSpan to make it applicable for copy-paste detection:

(1) **Adding gap constraints in frequent subsequences:** In most existing frequent subsequence mining algorithms, frequent subsequences are not necessarily contiguous in their supporting sequences. For example, sequence $abdec$ provides 1 support for subsequence $abc$, even though $abc$ does not appear contiguously in $abdec$. It is possible to have a large gap in the occurrence of a frequent subsequence in one of its supporting sequences. Hence, its corresponding code segment would have several statements inserted. Such segment is unlikely to be copy-pasted.

To address this problem, we modified CloSpan to add a gap constraint in frequent subsequences. CP-Miner only mines for frequent subsequences with a maximum gap not larger than a given threshold called $max$-gap. If the maximum gap of a subsequence in a sequence is larger than $max$-gap, this sequence is not “supporting” this subsequence. For example, for the sequence database $D = \{abced, abef, agchj, abijc, aklc\}$, the support of subsequence $abc$ is 1 if $max$-gap equals 0, and the support is 3 if $max$-gap equals 1.

The gap constraint with $max$-gap = 0 means that no statement insertion or deletions are allowed in copy-paste, whereas the gap constraint with $max$-gap = 1 or $max$-gap = 2 means that 1 or 2 statement insertions/deletions are tolerated in copy-paste.
(2) Matching frequent subsequences to copy-pasted segments: The original CloSpan algorithm outputs only frequent subsequences and their corresponding support values, but not their corresponding supporting sequences. To find copy-pasted code, we need to find the supporting sequences for each frequent subsequence.

We enhance CloSpan to address this problem. When CP-Miner generates a frequent subsequence, it maintains a list of IDs of its supporting sequences. In the above example, CP-Miner outputs two frequent subsequences: (67641265) and (133872016, 133872016, 82589171), each with their supporting sequence IDs, based on which the locations of the corresponding basic copy-pasted segments (file name and line numbers) can be identified.

Composing Larger Copy-pasted Segments

Since every sequence fed to the mining algorithm represents a basic block, a basic copy-pasted segment may only be a part of a larger copy-pasted segment. Therefore, it is necessary to combine a basic copy-pasted segment with its neighbors to construct a larger one, if possible.

The composition procedure is very straightforward. CP-Miner maintains a candidate set of copy-paste groups, which initially includes all of the basic copy-pasted segments that survive the pruning procedure described in Section 5.3.1. For each copy-paste group, CP-Miner checks their neighboring code segments to see if they also form a copy-paste group. If so, the two groups are combined together to form a larger one. This larger copy-paste group is checked against the pruning procedure. If it can survive the pruning process, it is added to the candidate set and the two smaller ones are removed. Otherwise, the two smaller ones still remain in the set and are marked as “non-expandable”. CP-Miner repeats this process until all groups in the candidate set are non-expandable.

Pruning False Positives

It is possible that copy-pasted segments discovered by the mining algorithm or the composition process may contain false positives. The main cause of false positives is the tokenization
of identifiers (variable/function/type) in order to tolerate identifier-renaming in copy-paste. Since identifiers of the same type are mapped into the same token, it is possible to identify false copy-pasted segments. For example, all statements similar to \( x = y + z \) would have the same hash value, which can introduce many false positives. To prune false positives, CP-Miner has applied several techniques to both of basic and composed copy-pasted segments. The pruning techniques include:

1. **Pruning unmappable segments:** This technique is used to prune false positives introduced by the tokenization of identifiers. This is based on the observation that if a programmer copy-pastes a code segment and then renames an identifier, he/she would most likely rename this identifier in all its occurrences in the new copy-pasted segment. Therefore, we can build an identifier mapping that maps old names in one segment to their corresponding new ones in the other segment that belongs to the same copy-paste group. In the example shown in Figure 5.3, variable \( \text{prom} \), \( \text{phys} \), \( \text{total} \) is changed into \( \text{prom} \), \( \text{prom} \), \( \text{taken} \) (except the bug on line 117).

   A mapping scheme is consistent if there are very few conflicts that map one identifier name to two or more different new names. If no consistent identifier mapping can be established between a pair of copy-pasted segments, they are likely to be false positives.

   To measure the amount of conflict, CP-Miner uses a metric called \( \text{ConflictRatio} \), which records the conflict ratio for an identifier mapping between two candidate copy-pasted segments. For example, if a variable \( A \) from segment 1 is changed into \( a \) in 75% of its occurrences in segment 2 but 25% of its occurrences is changed into other variables, the \( \text{ConflictRatio} \) of mapping \( A \rightarrow a \) is 25%. The \( \text{ConflictRatio} \) for the whole mapping scheme between these two segments are the weighted sum of \( \text{ConflictRatio} \) of the mapping for each unique identifier. The weight for an identifier \( A \) in a given code segment is the fraction of total identifier occurrences that are occurrences of \( A \). If \( \text{ConflictRatio} \) for two candidate copy-pasted segments is higher than a predefined threshold, these two code segments are filtered as false positives. In our experiments, we set the threshold to be 60%.
(2) **Pruning tiny segments:** Our mining algorithm may find tiny copy-pasted segments that consist of only 1-2 simple statements. If such a tiny segment cannot be combined with neighbors to compose a larger segment, it is removed from the copy-paste list. This is based on the observation that copy-pasted segments are usually not very small because programmers cannot save much effort in copy-pasting a simple tiny code segment.

CP-Miner uses the number of tokens to measure the size of a segment. This metric is more appropriate than the number of statements, because the length of statements is highly variable. If a single statement is very complicated with many tokens, it is still possible for programmers to copy-paste it.

To prune tiny segments, CP-Miner uses a tunable parameter called $min\_size$. If the number of tokens in a pair of copy-pasted segments is fewer than $min\_size$, this pair is removed.

(3) **Pruning overlapped segments:** If a pair of candidate copy-pasted segments overlap with each other, they are also considered false positives. CP-Miner stops extending the pair of copy-pasted segments once they overlap. For some program structures such as the `switch` statement that contain many pairs of self-similar segments, pruning overlapped segments can avoid most of the false positives in `switch` statements.

(4) **Pruning segments with large gaps:** Besides the mining procedure for basic copy-pasted segments, the gap constraint is also applied to composed ones. When two neighboring segments are combined, the maximum gap of the newly composed large segment may become larger than a predefined threshold, $max\_total\_gap$. If this is true, the composition is invalid. So the newly composed one is not added into the candidate set and the two smaller ones are marked as non-expandable in the set.

Of course, even after such rigorous pruning, false positives may still exist. However, we have manually examined 100 random copy-pasted segments reported by CP-Miner for Linux, and only a few false positives (8) are found. We can only manually examine each
identified copy-pasted segment because there are no traces that record programmers’ copy-paste operations during the development of the software.

**Computational Complexity of CP-Miner**

CP-Miner can extract copy-pasted code directly from a single software with total complexity of $O(n^2)$ in the worst case (where $n$ is the number of lines of code), and the optimizations further improve its efficiency in practice. For example, CP-Miner can identify more than 150,000 copy-pasted segments from 3–4 million lines of code in less than 20 minutes as shown in our results in Section 5.3.3. In CP-Miner, we break all of the large basic blocks into small blocks with at most 30 statements before feeding to the mining algorithm. Therefore, the search tree is at most with depth 30. With this constraint of search tree, the mining complexity of CP-Miner is $O(n^2)$ in the worst case. Furthermore, the optimizations described in Section 2.2 make it more efficient in both time and space overheads than the worst case.

**5.3.2 Detecting Copy-paste Related Bugs**

As we have mentioned in Section 5.1, the main cause of copy-paste related bugs is that programmers forget to modify identifiers consistently after copy-pasting. Once we get the mapping relationship between identifiers in a pair of copy-pasted segments (see Section 5.3.1), we can find the inconsistency and report these copy-paste related bugs. Table 5.1 shows the identifier mapping for the example described in Section 5.1.

<table>
<thead>
<tr>
<th>Identifiers in segment I (line 92-99)</th>
<th>Identifiers in segment II (line 111-118)</th>
</tr>
</thead>
<tbody>
<tr>
<td>iter (9)</td>
<td>iter (9)</td>
</tr>
<tr>
<td>num_reg (1)</td>
<td>num_reg (1)</td>
</tr>
<tr>
<td>prom_phys_total (4)</td>
<td>prom_prom_taken (3); prom_phys_total (1)</td>
</tr>
<tr>
<td>prom_reg_memlist (2)</td>
<td>prom_reg_memlist (2)</td>
</tr>
</tbody>
</table>

Table 5.1: Identifier mapping in the example in Figure 5.1. The number after each identifier indicates the number of occurrences.
For an identifier that appears more than once in a copy-pasted segment, it is consistent when it always maps to the same identifier in the other segment. Similarly, it is inconsistent when it maps itself to multiple identifiers. In Table 5.1, we can see that \texttt{prom\_phys\_total} is mapped inconsistently, because it maps to \texttt{prom\_prom\_taken} three times and \texttt{prom\_phys\_total} once. All the other variable mappings are consistent.

Unfortunately, inconsistency does not necessarily indicate a bug. If the amount of inconsistency is high, it may indicate that the code segments are not copy-pasted. Section 5.3.1 describes how we prune unmappable copy-pasted segments based on this observation.

Therefore, the challenge is to decide when an inconsistency is likely to be a bug instead of a false positive of copy-paste. To address this challenge, we need to consider the programmers’ intention. Our bug detection method is based on the following observation: if a programmer makes a change in a copy-pasted segment, the changed identifier is unlikely to be a bug. But if he/she changes an identifier in most places but forgets to change it in a few places, the unchanged identifier is likely to be a bug. In other words, “forget-to-change” is more likely to be a bug than an intentional “change”. For example, if in some cases, an identifier \texttt{A} is mapped into \texttt{a} and in other cases it is mapped into \texttt{a’} (both \texttt{a} and \texttt{a’} are different from \texttt{A}), it is unlikely to be a bug because programmers \emph{intentionally} change \texttt{A} to other names. On the other hand, if \texttt{A} is changed into \texttt{a} in most cases but remains unchanged only in a few cases, the unchanged places are likely to be bugs.

Based on the above observation, CP-Miner reexamines each non-expandable copy-paste group after running through the pruning and composing procedures. For each pair of copy-pasted segments, it uses a metric called \textit{UnchangedRatio} to detect bugs in an identifier mapping. We define

\[
\textit{UnchangedRatio} = \frac{\text{\texttt{NumUnchanged}}}{\text{\texttt{NumTotal}}}
\]

where \texttt{NumUnchanged} means the number of occurrences that a given identifier is unchanged, and \texttt{NumTotal} means the number of total occurrences of this identifier in a given copy-
pasted segment. Therefore, the lower the UnchangedRatio, the more likely it is a bug, unless UnchangedRatio = 0, which means that all of its occurrences have been changed. Note that UnchangedRatio is different from ConflictRatio. The former only measures the ratio of unchanged occurrences, whereas the latter measures the ratio of conflicts. In the example shown on Table 5.1, UnchangedRatio for prom._phys_total is 0.25, whereas all other identifiers have UnchangedRatio = 1.

CP-Miner uses a threshold for UnchangedRatio to detect bugs. If UnchangedRatio for an identifier is not zero and not larger than the threshold, the unchanged places are reported as bugs. When CP-Miner reports a bug, the corresponding identifier mapping information is also provided to programmers to help in debugging. In the example shown on Table 5.1, identifier prom._phys_total on line 117 is reported as a bug.

It is possible to further extend CP-Miner’s bug detection engine. For example, it might be useful to exploit variable correlations. Assume variable A always appears in close range to another variable B, and a always appears very close to b. So if in a pair of copy-pasted segments, A is renamed to a, B then should be renamed to b with high confidence. Any violation of this rule may indicate a bug. But the current version of CP-Miner has not exploited this possibility.

5.4 Methodology

We have evaluated the effectiveness of CP-Miner with large software including Linux, FreeBSD, Apache web server and PostgreSQL. The number of files (only C files) and the number of lines of code (LOC) for the software are shown in Table 5.2.

<table>
<thead>
<tr>
<th>Software</th>
<th>version</th>
<th>#files</th>
<th>#LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>2.6.6</td>
<td>6,497</td>
<td>4,365,124</td>
</tr>
<tr>
<td>FreeBSD</td>
<td>5.2.1</td>
<td>7,114</td>
<td>3,299,622</td>
</tr>
<tr>
<td>Apache</td>
<td>2.0.49</td>
<td>479</td>
<td>223,886</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>7.4.2</td>
<td>553</td>
<td>458,058</td>
</tr>
</tbody>
</table>

Table 5.2: Software evaluated in our experiments.
We set the thresholds used in CP-Miner as following. The minimum copy-pasted segment size \textit{min\_size} is 30 tokens. We also vary the gap constraints: (1) when \textit{max\_gap} = 0, CP-Miner only identifies copy-pasted code with identifier-renaming; (2) when \textit{max\_gap} = 1 and \textit{max\_total\_gap} = 2, it means that CP-Miner allows copy-pasted segments with insertion and deletion of one statement between any two consecutive statements, and a total of two statement insertions and deletions in the whole segment. Without specifying, we use setting (2) by default.

We define \textit{CP\_Coverage} to measure the percentage of copy-paste in given software (or a given module):

\[
\text{CP\_Coverage} = \frac{\#\text{LOC in copy-pasted segments}}{\#\text{LOC in the software or the module}} \times 100\%
\]

In our experiments, we also compare CP-Miner with a recently proposed tool called \textit{CCFinder} [KKI02]. Similar to our tool, CCFinder also tokenizes identifiers, keywords, constant, operators, etc. But different from our tool, it uses a suffix tree algorithm instead of a data mining algorithm. Therefore, it cannot tolerate statement insertions and deletions in copy-pasted code. Our results show that CP-Miner detects 17–52\% more copy-pasted code than CCFinder. In addition, CCFinder does not filter incomplete, tiny copy-pasted segments which are very likely to be false positives. CCFinder does not detect copy-paste related bugs, so we cannot compare this functionality between them.

In our experiments, we run CP-Miner and CCFinder on an Intel Xeon 2.4GHz machine with 2GB memory.
5.5 CP-Miner Basic Results

We first present the basic results of CP-Miner in this section, including the number of copy-pasted segments, the number of detected copy-pasted bugs, CP-Miner overhead, comparison with CCFinder, and effects of threshold setting. The statistical results of copy-paste characteristics in Linux and FreeBSD will be presented in Section 5.6.

5.5.1 Overall Results

Detecting Copy-pasted Code CP-Miner has found a significant number of copy-pasted segments in the evaluated software. The total amount of copy-paste covers 17.7–22.3% of the source code in these software. Table 5.3 shows the numbers of copy-pasted segments and \( CP_{\text{Coverage}} \) for these software. As shown on this table, in Linux and FreeBSD, there are more than 100,000 and 120,000 copy-pasted segments without any statement insertion (\( \text{max} \_\text{gap} = 0 \)), which account for about 15% of the source code. We have manually examined the identified copy-pasted segments in one of the Linux modules (file system), and found very few (only 3) false positives out of 90 identified copy-pasted segments (with \( \text{max} \_\text{gap} = 1 \)). The large number of copy-pasted segments motivates a support in software development environments such as Microsoft Visual Studio to maintain copy-pasted code.

<table>
<thead>
<tr>
<th>Software</th>
<th>( \text{max} _\text{gap} = 0 )</th>
<th>( \text{max} _\text{gap} = 1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#Segments Coverage</td>
<td>#Segments Coverage</td>
</tr>
<tr>
<td>Linux</td>
<td>122,282 15.3%</td>
<td>198,605 22.3%</td>
</tr>
<tr>
<td>FreeBSD</td>
<td>101,699 14.9%</td>
<td>153,230 20.4%</td>
</tr>
<tr>
<td>Apache</td>
<td>4,155   13.1%</td>
<td>6,196   17.7%</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>12,105  16.5%</td>
<td>16,662  22.2%</td>
</tr>
</tbody>
</table>

Table 5.3: The number of copy-pasted segments and \( CP_{\text{Coverage}} \)

Our results also show that a large percentage (30–50%) of copy-pasted segments have statement insertions and modifications. For example, when \( \text{max} \_\text{gap} = 1 \), CP-Miner finds 62.4% more copy-pasted segments in Linux. In FreeBSD, the \( CP_{\text{Coverage}} \) increases from 14.9% to 20.4% when \( \text{max} \_\text{gap} \) is relaxed from 0 to 1. These results show that previous tools
including CCFinder that cannot tolerate statement insertions and modifications would miss a lot of copy-paste.

By increasing $max_{gap}$ from 1 to 2 or higher, we can further relax the gap constraint. Also the number of false positives will increase with $max_{gap}$. Our manual examination results with the Linux file system module indicate that false positives are low with $max_{gap} = 1$, and relatively low with $max_{gap} = 2$.

**Detecting Copy-paste Related Bugs** CP-Miner has also reported many copy-paste related errors in the evaluated software. Since the errors reported by CP-Miner may not be bugs, we verify each reported error manually and then report to the corresponding developer community those errors that we suspect to be bugs with high confidence. The numbers of errors found by CP-Miner and verified bugs are shown on Table 5.4. The results are achieved by setting the $UnchangeRatioThreshold$ to be 0.4.

Both Linux and FreeBSD have many copy-paste related bugs. So far, we have verified 28 and 23 bugs in the latest versions of Linux and FreeBSD. Most of these bugs had never been reported before. We have reported these bugs to the kernel developer communities. *Recently, five Linux bugs have been confirmed and fixed by kernel developers, and the others are still in the process of being confirmed.*

Since Apache and PostgreSQL are much smaller compared to Linux and FreeBSD, CP-Miner found much fewer copy-paste related bugs. We have verified 6 bugs for Apache and 2 bugs for PostgreSQL with high confidence. *One bug in Apache was immediately fixed by the Apache developers after we reported it to them.*

In addition to those bugs verified, we also find many “potential bugs” (21 in Linux) that are not bugs by coincidence but might become bugs in the future. We call these types of errors as “careless programming”. Similar to the bugs verified, these errors also forget to change some identifiers consistently at a few places. Fortunately, by coincidence, the new identifiers and the old identifiers happen to have the same values. However, if such implicit
assumptions are violated in future versions of the software, it would lead to bugs that are hard to detect.

5.5.2 False Alarms

Table 5.4 also shows the number of false alarms reported by CP-Miner. These false alarms are mostly caused by the following two major reasons and can be further pruned:

1. **Incorrectly matched copy-pasted segments**: In some copy-pasted segments that contain multiple “case/if” blocks, there are many possible combinations for these contiguous copy-pasted blocks to compose larger ones. Since CP-Miner simply follows the program order to compose larger copy-pastes, it is likely that a wrong combination might be chosen. As a result, identifiers are compared between two incorrectly matched copy-pasted segments, which results in false alarms.

These false alarms can be pruned if we use more semantic information of the identifiers in these segments. The segments with a number of “case/if” blocks usually contain a lot of constant identifiers, but our current CP-Miner treats them as normal variable names. If we use the information of these constants to match “case/if” blocks when composing larger copy-pasted segments, it can reduce the number of incorrectly matched segments and most of such false alarms can be pruned.

2. **Exchangeable orders**: In a copy-paste pair, the orders of some statements or ex-
pressions can be switched. For example, a segment with several similar statements such as “a1=b1; a2=b2;” is the same as “a2=b2; a1=b1;”. The current version of CP-Miner simply compares the identifiers in a pair of copy-pasted segments in strict order and therefore a false alarm might be reported. In Linux, 41 false alarms are caused by such exchangeable orders.

These false alarms can be pruned if we relax the strict order comparison by further checking whether the corresponding “changed” identifiers are in the neighboring statements/expressions.

5.5.3 Time and Space Overheads

CP-Miner can identify copy-pasted code in large software very efficiently. The execution time of CP-Miner is shown in Table 5.5. CP-Miner takes 11–20 minutes to identify 101,699–198,605 copy-pasted segments in Linux and FreeBSD, each with 3–4 millions of lines of code. It takes less than 1 minute to detect copy-pasted segments in Apache and PostgreSQL with more than 200,000 lines of code.

CP-Miner is also space-efficient. For example, it takes less than 530MB to find copy-pasted code in Linux. For Apache and PostgreSQL, CP-Miner only consumes 27–57 MB of memory.

<table>
<thead>
<tr>
<th>Software</th>
<th>max_gap = 0</th>
<th></th>
<th>max_gap = 1</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time(s)</td>
<td>Space(MB)</td>
<td>Time(s)</td>
<td>Space(MB)</td>
</tr>
<tr>
<td>Linux</td>
<td>770</td>
<td>438</td>
<td>1164</td>
<td>527</td>
</tr>
<tr>
<td>FreeBSD</td>
<td>615</td>
<td>334</td>
<td>1155</td>
<td>459</td>
</tr>
<tr>
<td>Apache</td>
<td>14</td>
<td>27</td>
<td>15</td>
<td>30</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>32</td>
<td>44</td>
<td>38</td>
<td>57</td>
</tr>
</tbody>
</table>

Table 5.5: Execution time and memory space of CP-Miner
5.5.4 Comparison with CCFinder

We have compared CP-Miner with CCFinder [KKI02]. CCFinder has similar execution time as CP-Miner, but CP-Miner discovers many more copy-pasted segments. In addition, CCFinder cannot detect copy-paste related bugs. As we explained in Section 5.4, CCFinder allows identifier-renaming but not statement insertions. In addition, CCFinder does not have rigorous pruning operations as CP-Miner. For example, CCFinder reports many tiny copy-pasted segments fewer than 30 tokens, which are too simple to be worthy to copy-paste. In addition, it also includes incomplete statement in copy-pasted segments, which is very unlike to be the case in practice.

CP-Miner can identify 17–52% more copy-pasted code than CCFinder because CP-Miner can tolerate statement insertions and modifications. Table 5.6 compares the CP_Coverages identified by CP-Miner and CCFinder. The results with CP-Miner are achieved using the default threshold setting (min_size = 30 and max_gap = 1). For fair comparison, we also filter those tiny, incomplete segments from CCFinder’s output. The results show that around 25% of copy-paste is pruned after filtering.

<table>
<thead>
<tr>
<th>Software</th>
<th>CCFinder</th>
<th>CP-Miner</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>14.7%(19.8%)</td>
<td>22.3%</td>
</tr>
<tr>
<td>FreeBSD</td>
<td>14.5%(19.6%)</td>
<td>20.4%</td>
</tr>
<tr>
<td>Apache</td>
<td>11.8%(15.3%)</td>
<td>17.0%</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>18.5%(23.8%)</td>
<td>21.7%</td>
</tr>
</tbody>
</table>

Table 5.6: CP_Coverage comparison between CP-Miner and CCFinder
For CCFinder, the first number is the result after pruning those tiny, incomplete segments, and the second number in () is the result before pruning.

5.5.5 Effects of Threshold Settings

Segment Size Threshold Figure 5.4 shows the effect of segment-size threshold min_size on CP_Coverage. As expected, the CP_Coverage decreases when min_size increases because more copy-pasted segments are pruned. The results also show the decrement slowdowns
when \textit{min\_size} is in the range of 30–100 (tokens), which indicates that not too many copy-segments’ sizes fall in this range. This implies that segments with fewer than 30 tokens are very likely to be false positives, whereas those with more than 40 tokens are very likely to be copy-paste.

Figure 5.4: Effects of \textit{min\_size} on \textit{CP\_Coverage}

\textbf{Unchange Ratio Threshold} Figure 5.5 shows the effect of threshold \textit{UnchangeRatioThreshold} on the number of bugs reported. Since \textit{UnchangeRatio} \geq 0.5 means that at least half of the identifiers are not changed after copy-paste, these unchanged identifiers are unlikely “forget-to-change” and so it cannot indicate a copy-paste related error. Therefore, we only show the errors with \textit{UnchangeRatioThreshold} < 0.5.

As expected, more errors are reported by CP-Miner when \textit{UnchangeRatioThreshold}
increases. Specifically, the number of errors reported increases gradually when the threshold is less than 0.25, and then increases sharply when the threshold \( \in (0.25, 0.35) \). We found that most of the errors with high UnchangeRatio turn out to be false bugs during our verification. For example, CP-Miner reports many errors where only 1 out of 3 identifiers is unchanged \( (UnchangeRatio = 0.33) \). However, it cannot strongly support that it is a copy-paste related bug. In order to prune such false bugs, we can further analyze the identifiers in the context of the copy-pasted segments (e.g., the whole function).

### 5.5.6 Another Bug Example

In introduction, we have shown a copy-paste related bug in Figure 5.1. In order to further demonstrate what kind of bugs can be detected by CP-Miner, we show another typical copy-paste related bug in Figure 5.6. In this example, the two segments in lines 246–257 and lines 258–269 are copy-paste group, each of which initializes the IOP (I/O processor) differentiated by constants \( IOP\_NUM\_SCC (=0) \) and \( IOP\_NUM\_ISM (=1) \), respectively. However, the identifier \( IOP\_NUM\_SCC \) in line 264 is not correctly changed to \( IOP\_NUM\_ISM \) correspondingly, and it results in a wrong initial status of IOPs.

This bug would overwrite the old value (0x87) of \( iopc\_base[IOP\_NUM\_SCC]->status\_ctrl \) by 0. This cannot be detected by previous bug detection tools because it is not a simple buffer overflow bug (since \( IOP\_NUM\_SCC \) equals to 0), incorrect pointer manipulation, or free memory access. This bug, if known by malicious users who plan a security attack, may cause the server crash.

### 5.6 Statistics of Copy-paste in OS code

This section presents the statistical results on copy-paste characteristics in large software. Our results include the distribution of copy-pasted segments across different group sizes, segment sizes, granularity, amount of changes, modules, and versions.

98
244 void __init iop_preinit(void)
245 {
246   if (macintosh_config->scc_type == MAC_SCC_IOP) {
247     if (macintosh_config->ident == MAC_MODEL_IIFX) {
248       iop_base[IOP_NUM_SCC] = (struct mac_iop *) SCC_IOP_BASE_IIFX;
249     } else {
250       iop_base[IOP_NUM_SCC] = (struct mac_iop *)SCC_IOP_BASE_QUADRA;
251     }
252     iop_base[IOP_NUM_SCC]->_status_ctrl = 0x87;
257   }
258   if (macintosh_config->adb_type == MAC_ADB_IOP) {
259     if (macintosh_config->ident == MAC_MODEL_IIFX) {
260       iop_base[IOP_NUM_ISM] = (struct mac_iop *) ISM_IOP_BASE_IIFX;
261     } else {
262       iop_base[IOP_NUM_ISM] = (struct mac_iop *)ISM_IOP_BASE_QUADRA;
263     }
264     iop_base[IOP_NUM_SCC]->_status_ctrl = 0;   // bug
269   }

Figure 5.6: Another example of copy-paste-related error detected by CP-Miner.
This bug appears in linux-2.6.6/arch/m68k/mac/iop.c.

### 5.6.1 Copy-pasted Segment Size and Granularity

Figure 5.7 illustrates the distribution of copy-pasted segments with different sizes (in terms of
the number of statements). The results show that most (60–64%) copy-pasted segments are
not very large, with only 5–16 statements. Only a few (0.2–5.0%) copy-pasted segments have
more than 64 statements. In particular, Figure 5.7(a) shows that most (35–40%) copy-paste
groups contain 5–8 statements in each segment. Figure 5.7(b) shows similar characteristics:
copy-pasted segments with 5–8 statements cover about 7–10% of the source code.

Figure 5.8 shows the distribution of copy-paste group size. About 60% of copy-paste
groups contain only two segments, which indicates that there are only two copies (original
and replicated) for most copy-pasted code. But still, 40% of the copy-pasted code groups
contain at least 3 segments, which indicates that a lot of code is replicated more than once.

Total 4.0–6.7% of copy-pasted segments are copy-pasted more than 7 times. If a bug
is detected in one of the copies, it is very difficult for programmers to remember fixing the
(a) The number of copy-paste groups with various segment sizes (number of statements)

(b) The $CP\_Coverage$ with various segment sizes (number of statements)

Figure 5.7: Size distribution of copy-pasted segments

Due to the overlap of copy-pasted segments that have different segment sizes and also belong to different copy-paste groups, the sum of all $CP\_Coverage$ does not equal to the overall $CP\_Coverage$. 
bug in the other 7 or more copies. This motivates a tool that can automatically fix other copy-pasted segments once a programmer fixes one segment.

![Copy-paste group size distribution](image)

**Figure 5.8: Copy-paste group size distribution**

Group size is the number of segments in each copy-paste group. Each bar represents the percentage of copy-paste groups that contains the corresponding number of segments.

Table 5.7 shows the number of copy-pasted segments at basic-block and function granularity. Our results show that only 3–11.8% of all copy-pasted segments are basic blocks, which indicates that programmers seldom copy-paste basic blocks because most of them are too simple to be worthy copy-pasting.

More interestingly, there are many (11.3–19.2%) copy-pasted segments at function granularity. The reason is that many functions provide similar functionality, such as reading data from different types of I/O devices. Those functions can be copy-pasted with some modifications such as replacing parameters’ data types. This motivates some refactoring tools [JO93] to better maintain these copy-pasted functions.
Table 5.7: Distribution of copy-paste granularity

Numbers and percentages of copy-pasted segments at different granularity. Note here the percentage is not CP Coverage. It is calculated by comparing to the total number of copy-pasted segments.

### 5.6.2 Modifications in Copy-pasted Segments

Figure 5.9 shows how many identifiers are changed in copy-pasted segments. Since in some cases there are more than two segments in each copy-paste group, we only present the distribution in the best case: comparing the most similar pair of segments from each copy-paste group. Each bar includes two parts: one with no statement insertion and the other with one statement insertion.

The results indicate that 59–76% of copy-pasted segments require identifier-renaming. For example, in Linux, 27% copy-pasted segments are identical, and 8% segments are almost identical with only one statement inserted. The rest 65% of the copy-pasted segments in Linux rename at least one identifier. Such results motivate a tool to support consistently renaming identifiers in copy-pasted code.

### 5.6.3 Copy-pasted Code across Modules

Different modules in a software have different characteristics of copy-paste. In this subsection, we analyze the copy-pasted code across different modules in operating system code. We split Linux into 9 categories: arch (platform specific), fs (file system), kernel (main kernel), mm (memory management), net (networking), sound (sound device drivers), drivers (device drivers other than networking and sound device), crypto (cryptography), and others (all other code). For FreeBSD, modules are also split into 9 categories: sys (kernel sources), lib (system libraries), crypto (cryptography), usr.sbin (system administration commands),
usr.bin (user commands), sbin (system commands), bin (system/user commands), gnu, and others.

**Distribution of copy-pasted code in modules**  Figure 5.10 shows the number and \( CP\_{Coverage}\) of copy-pasted segments in different modules of Linux and FreeBSD. The \( CP\_{Coverage}\)s are computed based on the size of each corresponding module, instead of the entire software.

Figure 5.10 (a) shows that most copy-pasted code in Linux and FreeBSD is located in one or two main modules. For example, modules “drivers” and “arch” account for 71% of all copy-pasted code in Linux, and module “sys” accounts for 60% in FreeBSD. This is because many drivers are similar, and it is much easier to modify a copy-paste of another driver than writing one from scratch.

Figure 5.10 (b) shows that a large percentage (20–28%) of the code in Linux is copy-pasted in the “arch” module, the “crypto” module and and the device driver modules in-
including “net”, “sound”, and “drivers”. The “arch” module has a lot of copy-pasted code because it has many similar sub-modules for different platforms. The device driver modules contain a significant portion of copy-pasted code because many devices share similar functionalities. Additionally, “crypto” is a very small module (less than 10,000 LOC), but the main cryptography algorithms consist of a number of similar computing steps, so it contains a lot of copy-pasted code. Our results indicate that more attention should be paid to these modules because they are more likely to contain copy-paste related bugs.

In contrast, the modules “mm” and “kernel” contain much less copy-pasted code than others, which indicates that it is rare to reuse code in kernels and memory management modules.
### Table 5.8: Copy-paste code within a module and across modules

<table>
<thead>
<tr>
<th>Module (LOC)</th>
<th>arch</th>
<th>fs</th>
<th>kernel</th>
<th>mm</th>
<th>net</th>
<th>sound</th>
<th>drivers</th>
<th>crypto</th>
<th>others</th>
</tr>
</thead>
<tbody>
<tr>
<td>arch (724858)</td>
<td><strong>25.1</strong></td>
<td>1.4</td>
<td>0.5</td>
<td>0.3</td>
<td>1.1</td>
<td>1.3</td>
<td>3.2</td>
<td>0.1</td>
<td>0.8</td>
</tr>
<tr>
<td>fs (475946)</td>
<td>1.4</td>
<td><strong>16.5</strong></td>
<td>0.6</td>
<td>0.5</td>
<td>1.7</td>
<td>1.2</td>
<td>2.2</td>
<td>0.0</td>
<td>0.7</td>
</tr>
<tr>
<td>kernel (30629)</td>
<td>3.0</td>
<td>1.8</td>
<td><strong>7.9</strong></td>
<td>0.6</td>
<td>2.3</td>
<td>1.6</td>
<td>2.8</td>
<td>0.1</td>
<td>0.8</td>
</tr>
<tr>
<td>mm (23490)</td>
<td>2.6</td>
<td>2.2</td>
<td>0.8</td>
<td><strong>6.2</strong></td>
<td>1.7</td>
<td>1.1</td>
<td>2.0</td>
<td>0.0</td>
<td>0.7</td>
</tr>
<tr>
<td>net (334325)</td>
<td>1.8</td>
<td>2.5</td>
<td>1.1</td>
<td>0.7</td>
<td><strong>20.7</strong></td>
<td>2.1</td>
<td>3.7</td>
<td>0.1</td>
<td>1.0</td>
</tr>
<tr>
<td>sound (373109)</td>
<td>2.3</td>
<td>2.0</td>
<td>1.0</td>
<td>0.6</td>
<td>2.2</td>
<td><strong>27.4</strong></td>
<td>4.6</td>
<td>0.2</td>
<td>1.1</td>
</tr>
<tr>
<td>drivers (2344594)</td>
<td>2.3</td>
<td>1.7</td>
<td>0.6</td>
<td>0.4</td>
<td>1.8</td>
<td>2.0</td>
<td><strong>21.4</strong></td>
<td>0.1</td>
<td>0.6</td>
</tr>
<tr>
<td>crypto (9157)</td>
<td>2.3</td>
<td>2.2</td>
<td>0.3</td>
<td>0.1</td>
<td>1.1</td>
<td>1.5</td>
<td>2.5</td>
<td><strong>26.1</strong></td>
<td>2.2</td>
</tr>
<tr>
<td>others (49016)</td>
<td>3.8</td>
<td>1.9</td>
<td>0.8</td>
<td>0.4</td>
<td>1.7</td>
<td>1.5</td>
<td>2.6</td>
<td>0.3</td>
<td><strong>15.2</strong></td>
</tr>
</tbody>
</table>

(a) Linux 2.6.6

<table>
<thead>
<tr>
<th>Module (LOC)</th>
<th>sys</th>
<th>lib</th>
<th>crypto</th>
<th>usu.sbin</th>
<th>usu.bin</th>
<th>sbin</th>
<th>bin</th>
<th>gnu</th>
<th>others</th>
</tr>
</thead>
<tbody>
<tr>
<td>sys (1767368)</td>
<td><strong>22.5</strong></td>
<td>1.5</td>
<td>1.1</td>
<td>1.5</td>
<td>1.2</td>
<td>1.0</td>
<td>0.3</td>
<td>0.6</td>
<td>0.8</td>
</tr>
<tr>
<td>lib (291132)</td>
<td>3.3</td>
<td><strong>18.1</strong></td>
<td>1.8</td>
<td>1.4</td>
<td>1.2</td>
<td>0.7</td>
<td>0.3</td>
<td>0.5</td>
<td>0.9</td>
</tr>
<tr>
<td>crypto (392020)</td>
<td>2.1</td>
<td>1.6</td>
<td><strong>16.7</strong></td>
<td>1.5</td>
<td>1.2</td>
<td>1.0</td>
<td>0.4</td>
<td>0.9</td>
<td>0.9</td>
</tr>
<tr>
<td>usu.sbin (310949)</td>
<td>3.5</td>
<td>2.5</td>
<td>2.2</td>
<td><strong>17.7</strong></td>
<td>3.8</td>
<td>2.8</td>
<td>1.0</td>
<td>1.6</td>
<td>2.5</td>
</tr>
<tr>
<td>usu.bin (236952)</td>
<td>2.6</td>
<td>1.8</td>
<td>1.7</td>
<td>3.4</td>
<td><strong>11.9</strong></td>
<td>2.2</td>
<td>1.2</td>
<td>1.1</td>
<td>1.9</td>
</tr>
<tr>
<td>sbin (112284)</td>
<td>3.1</td>
<td>2.0</td>
<td>1.7</td>
<td>3.5</td>
<td>3.1</td>
<td><strong>16.9</strong></td>
<td>1.2</td>
<td>1.1</td>
<td>2.2</td>
</tr>
<tr>
<td>bin (47008)</td>
<td>1.3</td>
<td>1.0</td>
<td>2.0</td>
<td>2.0</td>
<td>2.3</td>
<td>1.8</td>
<td><strong>10.9</strong></td>
<td>0.8</td>
<td>1.6</td>
</tr>
<tr>
<td>gnu (64996)</td>
<td>2.8</td>
<td>1.7</td>
<td>2.0</td>
<td>1.7</td>
<td>1.6</td>
<td>0.9</td>
<td>0.7</td>
<td><strong>14.5</strong></td>
<td>1.1</td>
</tr>
<tr>
<td>others (76913)</td>
<td>2.9</td>
<td>2.4</td>
<td>2.0</td>
<td>3.3</td>
<td>3.1</td>
<td>2.6</td>
<td>1.0</td>
<td>1.2</td>
<td><strong>12.7</strong></td>
</tr>
</tbody>
</table>

(b) FreeBSD 5.2.1

Each number in the table represents the \( CP_{Coverage} \) of code copy-pasted from another module. For example, in (a), the number at row “arch” and column “arch” represents that 25.1% of code in module “arch” is copy-pasted within the module itself; the number at row “arch” and column “drivers” represents that 3.2% of the code in module “arch” is copy-pasted from/to another module “drivers”.

**Copy-pasted code within/across modules**  The code in a module can be copy-pasted within the module itself or from other modules. Table 5.8 shows how much code is copy-pasted within the module itself or across different modules.

Most of copy-pasted code is within the same module as indicated by the bold numbers in Table 5.8. The \( CP_{Coverage} \) of copy-pasted code within a module is usually more than 15%, and some are even more than 20%. On the other hand, the \( CP_{Coverage} \) of the code that is copy-pasted across different modules is much lower than those within a module, which is usually lower than 4%.

The exceptional case is that 4.6% of the code in the “sound” module is copy-pasted...
Figure 5.11: Copy-pasted code in Linux and FreeBSD through various versions.

x-axis (version number) is drawn in time scale with the corresponding release time. The versions of Linux we analyze are through 1.0 to the current version 2.6.6. The versions of FreeBSD include the main branch through 2.0 to 4.10.

from “drivers”. The reason is that “sound” was originally one of the sub-modules in module “drivers” before it was separated since version 2.5.5. Therefore, “sound” still shares a lot of code with “drivers”.

5.6.4 Copy-paste Evolution

Figure 5.11 shows that the amount of copy-pasted code increases as the operating system code evolves. For example, Figure 5.11(a) shows the amount of copy-pasted code in Linux through version 1.0 to 2.6.6 over time. As Linux’s code size increases from 141,000 to 4.4 million lines, copy-pasted code also keeps increasing from 23,000 to 975,000 lines.
In terms of $CP\_Coverage$, the percentage of copy-pasted code also steadily increases along software evolution. For example, Figure 5.11(a) shows that $CP\_Coverage$ in Linux increases from 16.2% to 22.3% in Linux from version 1.0 to 2.6.6, and Figure 5.11(c) shows that $CP\_Coverage$ in FreeBSD increases from 17.5% to 21.7% from version 2.0 to 4.10. However, the $CP\_Coverage$ remains relatively stable over the recent several versions for both Linux and FreeBSD. For example, the $CP\_Coverage$ for FreeBSD has been staying at around 21–22% since version 4.0.

Most of the growth of copy-paste coverage comes from a few modules including “drivers” and “arch” in Linux and “sys” in FreeBSD. Figure 5.11(b) shows copy-pasted code in the module “drivers” individually through multiple versions. The percentage of copy-pasted code increases more rapidly in this module than in the entire software. For example, in version 1.0, the $CP\_Coverage$ is only 11.9% in this module, but it increases to 20.4% in version 2.2.0. This is probably because Linux supports more and more similar device drivers during this period.

5.7 Summary

This chapter presents a tool called CP-Miner that uses data mining techniques to efficiently identify copy-pasted code in large software including operating systems, and also detects copy-paste related bugs. Specifically, it takes less than 20 minutes for CP-Miner to identify 190,000 and 150,000 copy-pasted segments that account for 20–22% of the source code in Linux and FreeBSD. Moreover, CP-Miner has detected 28 and 23 copy-paste related bugs in the latest versions of Linux and FreeBSD, respectively. Compared to CCFinder [KKI02], CP-Miner finds 17–52% more copy-pasted segments because it can tolerate statement insertions and modifications in copy-paste. In addition, we have shown some interesting characteristics of copy-pasted codes in Linux and FreeBSD, including distribution of copy-paste across

CP-Miner has already been released to the research community.
different segment sizes, group sizes, granularity, modules, amount of modifications, and software evolution.

Our results indicate that maintaining copy-pasted code would be very useful for programmers because it is commonly used in large software such as operating system code, and it can easily introduce hard-to-detect bugs. We hope our study motivates software development environments such as Microsoft Visual Studio to provide functionality to maintain copy-pasted code and automatically detect copy-paste related bugs.

Even though CP-Miner focuses only on “forget-to-change” bugs caused by copy-paste, copy-paste can introduce many other types of bugs. For example, after copy-paste operation, the programmer forgets to add some statements that are specific to the new copy-pasted segment. However, such bugs are hard to detect because it relies on semantic information. It is impossible to guess what the programmer would want to insert or modify. Another type of copy-paste related bugs is caused by programmers forgetting to fix a known bug in all copy-pasted segments. They only fix one or two segments but forget to change it in the others. Our tool CP-Miner can detect simple cases of this type of errors. But if the fix is too complicated, CP-Miner would miss the bug because the modified code segment becomes too different from the others to be identified as copy-paste. To solve this problem more thoroughly, it would require support from software development environments such as Microsoft Visual Studio.
Chapter 6

Discussion

This chapter will discuss the issues and limitations of the approaches proposed in this dissertation and some possible solutions to address these issues in the future. It first presents some existing issues in the general research area of applying data mining techniques for software reliability, and then discusses the current limitations of PR-Miner and CP-Miner, respectively.

6.1 Issues in Data Mining for Software Reliability

- **Domain-specific knowledge.** Although data mining can be applied to automation of data analysis, it requires special domain-specific knowledge to solve problems. For example, CP-Miner is based on how the programmers do copy-pasting in programming, and therefore CP-Miner analyzes source code in lexical level and so it converts the source code to the data fed to data mining algorithms. In contrast, since PR-Miner analyzes the correlations among function elements (i.e. function calls, variables, data types, etc.) which is syntax level information in source code, and therefore it converts the source code to the data fed to data mining algorithms based on AST (abstract syntax tree). In other words, in order to apply data mining techniques to solve software reliability problem, some domain-specific knowledge about the problem is necessary.

An interesting question is whether we can further combine more domain-specific knowledge to improve this approach. For example, some of the bugs reported by CP-Miner and PR-Miner are false positives, and the programmers who wrote the programs may
know the exact reasons why the detection tools report as false positives. One of the solution is to collect and analyze the domain-specific knowledge provided by programmers. For example, programmers may annotate the source code where violates some programming rules but it is correct, and when PR-Miner would know the exceptions and would not report them as potential bugs. Further, programmers can verify the bug reports and provide the feedbacks on true/false positives. Such feedbacks can be used for future detection. For example, if a bug reported by CP-Miner is false positive, bug signature can be calculated and applied to other bugs or future bug detection so that CP-Miner would not report the potential bugs that match the bug signature.

- **Incremental mining.** Although it is efficient to using data mining to analyze source code, it may not meet the requirement for practical usage if it requires to re-scan all the source code in large applications every time. For example, some development environment may check in source code every 30 minutes and apply automatic bug detection before checking in. It is difficult to apply all the detection methods on the whole millions of lines of code. However, the new or modified source code in each check-in version usually contains less than thousands of lines of code. It there would be no problem if we only need to analyze the new or modified source code each time. To address this problem, we may apply incremental mining [CYH04, MPT03] to analyze only the new data and update the existing patterns. For example, PR-Miner can build the programming rules based on the existing source code, and update the rules when some new source code is checked in. New programming rule may be inserted into the rule database, and some obsolete rules may be also removed from the database.

### 6.2 Current Limitations in PR-Miner

While PR-Miner is very effectively in automatically extracting implicit programming rules and detecting violations, our current version of PR-Miner has the following limitations, which
could be future research.

- **False negatives in rule extraction.** PR-Miner may miss some programming patterns due to the low supports of them, and so PR-Miner cannot find the corresponding programming rules with low support. This problem is caused by the characteristics of frequent pattern mining. In order to address this problem, we need to increase the appearance of the real programming patterns. A possible solution is to combine with dynamic analysis. We may collect the execution traces for correct runs, and then mine the programming rules from the traces. Although the programming patterns appear in source code for very few times, they may be on the frequent execution paths and so the execution traces may contain many of the patterns.

Further, PR-Miner would miss some programming patterns if most of them cross functions in source code. Because PR-Miner converts each function definition into one itemset, if the programming elements in a pattern appear in different functions, the corresponding tokens will appear in different itemsets and so the support would be much lower than the actual support in source code. To address this problem, we may also apply interprocedural analysis when we parse the source code. For example, we may expand the the function call by inlining some function into the call sites.

- **False negatives in violation detection due to copy-pasting.** A violation to a programming rule may be propagated to multiple modules due to copy-pasting [CYC+01, LLMZ04], which would result in a lot of violations to the rule. As a result, PR-Miner would probably miss to report this error. In order to eliminate the propagation effect, we can combine with our previous work on copy-paste detection called CP-Miner [LLMZ04], which also works with large software. Using CP-Miner, we can first identify copy-pasted code, and then each group of copy-pasted code accounts as only 1 support in PR-Miner when extracting programming rules and detecting violations.

- **Noisy effects of macros.** Macro definitions in C can result in false programming
rules as well as false negatives in detecting violations. Since GCC first preprocesses the source code by expanding macros before creating the intermediate representation, the information in a single macro can be duplicated for many times like copy-pasting. Therefore, PR-Miner may report the elements in such a macro as a rule, and may also fail reporting some violations in these macros since they are duplicated for many times. In order to eliminate such noisy effects of macros, we can consider each macro as one element using the technique in some other studies on refactoring [GJ03].

- **Function name collisions.** Since occasionally some functions use the same name and PR-Miner only uses the compiling information from GCC front-end, PR-Miner cannot differentiate them, which can result in false rules. Fortunately, in most software, there are few identical function names especially in a single module, so it does not cause too much trouble to PR-Miner. To eliminate this effect, we can use the linking information when PR-Miner converts the source code to an itemset database so that it can differentiate functions with identical names.

- **False negatives in violation detection in some control paths.** Since PR-Miner uses function as the basic granularity for violation detection, it can miss violations in some control paths. For example, if a rule appears in one of function $F$’s control path, PR-Miner would consider that the entire function $F$ does not violate this rule, even though some of its other control paths may violate this rule. To address this problem, we can borrow techniques from model checking to check different control paths.

- **Detecting specific types of programming rules and violations.** The current PR-Miner can extract the general types of programming rules from source code. As we discussed in Section 6.1, it could improve performance of the approach if we could combine domain-specific information. If we can provide some knowledge about some specific types of programming rules and indicate PR-Miner to extract only of them from source code, it can improve the accuracy and efficiency of PR-Miner. For example,
instead of mining the correlations between any types of function calls and variables, PR-Miner can identify the correlations among only the shared variables and lock/unlock function calls, which may indicate the specific rules about which variables should be protected by locks. Such a specific type of programming rules can be also applied to violation detection and therefore it can detect a specific type of software bugs such as data races.

The results in this dissertation indicate that PR-Miner is an efficient and practical tool to extract implicit, undocumented programming rules and to detect violations in large software. Furthermore, by replacing the GCC front-end parser, PR-Miner can be easily applied to programs in other programming languages such as Java. In addition, we envisage extending PR-Miner in several directions to address the limitations above in the current prototype.

6.3 Current Limitations in CP-Miner

- **False negatives in copy-pasted code detection.** Because CP-Miner tokenizes the source code such as function names and variable names into numbers, some pieces of code may appear similar but they are actually not copy-pasted. This issue can be dealt with by keeping track of programmer’s editing actions, but not much development environment provides such functionality and it cannot be applied to existing code. Since the intention of programmer using copy-pasting is to reduce the effort in writing code, it is not worthy to copy-pasting a piece of code if it involves a lot of editing effort such modifying most of the code. To this end, we could reduce the false positive rate by computing the editing distance between a pair of code segments. If the editing distance is very large, it is probably a false positive.

- **Other types of copy-paste related bugs.** The current version of CP-Miner can only detect one simple but common copy-paste related bugs, forget-to-change bugs. As
the results presented in Chapter 5, copy-pasting can result in a lot of types of errors. Another common error is that when one bug is fixed in one copy of the copy-pasted code, programmers may forget to fix the other copies. To detect such kind of bugs, we can compare each copy of copy-pasted code, if there are some inconsistent changes between two copies, it may indicate an error.
Chapter 7

Related Work

This chapter discusses some previous work related to the dissertation project.

7.1 Detecting Copy-Pasted Code

Several studies have been conducted on detection of copy-pasted code. The techniques used include: line-by-line [Bak95], token-by-token [KKI02, PMP02], fingerprinting [Joh93], visualization [CH93, DRD99], abstract syntax tree [BYM+98, KGD95], and dependence graph [KH01, Kri01].

Dup [Bak95] finds all pairs of matching parameterized code fragments. A code fragment matches another if both fragments are contiguous sequences of source lines with some consistent identifier mapping scheme. Because this approach is line-based, it is sensitive to lexical aspects like the presence or absence of new lines. In addition, it does not find non-contiguous copy-pastes. CP-Miner does not have these shortcomings.

Johnson [Joh93] proposed using a fingerprinting algorithm on a substring of the source code. In this algorithm, calculated signatures per line are compared in order to identify matched substrings. As with line-based techniques, this approach is sensitive to minor modifications made in copy-pasted code.

Some graphical tools were proposed to understand code similarities in different programs (or in the same program) visually. Dotplots [CH93] of source code can be constructed by tokenizing the code into lines and placing a dot in coordinates \((i, j)\) on a 2-D graph, if the \(i^{th}\) input token matches \(j^{th}\) input token. Similarly, Duploc [DRD99] provides a scatter
plot visualization of copy-pastes (detected by string matching of lines) and also textual reports that summarize all discovered sequences. Both Dotplots and Duploc only support line granularity. In addition, they can only detect identical duplicates and do not tolerate renaming, insertions, and deletions.

Baxter et al. [BYM+98] proposed a tool that transforms source code into abstract-syntact trees (AST), and detects copy-paste by finding identical subtrees. Similar to other tools, it is not tolerant to modifications in copy-pasted segments. In addition, it may introduce many false positives because two code segments with the same syntax subtrees are not necessarily copy-pastes.

Komondoor et al. [KH01] proposed using program dependence graph (PDG) and program slicing to find isomorphic subgraphs and code duplication. Although this approach is successful at identifying copies with reordered statements, its running time is very long. For example, it takes 1.5 hours to analyze only 11,540 lines of source code from bison, much slower than CP-Miner. Another slow PDG-based approach is found in [Kri01].

Mayrand et al. [MLM96] used an Intermediate Representation Language to characterize each function in the source code and detect copy-pasted function bodies that have similar metric values. This tool does not detect copy-paste at other granularity such as segment-based copy-paste, which occurs more frequently than function-based copy-paste as shown in our results.

Some copy-paste detection techniques are too coarse-grained to be useful for our purpose. JPlag [PMP02], Moss [SWA03], and sif [Man94] are tools to find similar programs among a given set. They have been commonly used to detect plagiarism. Most of them are not suitable for detecting copy-pasted code in a single large program.

Kontogiannis et al. [KGD95] built an abstract pattern matching tool to identify probable matches using Markov models. This approach does not find copy-pasted code. Instead, it only measures similarity between two programs.
7.2 Detecting Software Bugs

Many tools have been proposed for detecting software bugs. One approach is dynamic checking that detects bugs during execution. Examples of dynamic tools include Purify [HJ92], Valgrind [SNF+], DIDUCE [HL02], Eraser [SBN+97], and CCured [NMW02]. Dynamic tools have more accurate information but may introduce overheads during execution. Moreover, they can only find bugs on the execution paths. Most dynamic tools cannot detect bugs in operating systems.

Another approach is to perform checks statically. Examples of this approach include explicit model checking [ECC01, MPC+02, SD95, KTB+06b] and program analysis [CLL+02, EA03, HCXE02]. Most static tools require significant involvement of programmers to write specifications or annotate programs. But the advantage of static tools is that they add no overhead during execution, and it can find bugs that may not occur in the common execution paths. A few tools do not require annotations, but they focus on detecting different types of bugs, instead of copy-paste related bugs.

Our tool, CP-Miner, is a static tool that can detect copy-paste related bugs, without any annotation requirement from programmers. CP-Miner complements other bug detection tools because it is based on a different observation: finding bugs caused by copy-paste. Some copy-paste related bugs can be found by previous tools if they lead to buffer overflow or some obvious memory corruption, but many of them, especially those semantic ones, cannot be found by previous tools.

Our work is motivated by and related to Engler et al’s empirical analysis of operating systems errors [CYC+01]. Their study gave an overall error distribution and evolution analysis in operating systems, and found that copy-paste is one of the major causes for bugs. Our work presents a tool to detect copy-pasted code and related bugs in large software including operating system code. Many of these bugs such as the one in Figure 5.1 cannot be detected by their tools.
7.3 Specification Generation

Automatically generating specifications has been studied for decades \cite{Cap75, JHMW77, Weg74}. Recently, Bensalem et al describe techniques for automatically generating auxiliary predicates, including the general reaffirmed invariants, invariant propagation, refined strengthening, and invariant combination \cite{BLS96}. Björner et al present the method to generate the auxiliary assertions by extending the traditional methods \cite{BBM97}. Xie and Notkin propose the approach of using inferred program semantic properties for test generation and selection \cite{XN03}. Ammons et al propose a machine learning approach to discovering specifications of the protocols that code must obey when interacting with an API or abstract data type \cite{ABL02}. These studies have different goals from PR-Miner.

In order to reduce programmers’ effort in manually writing specifications for the extended static type checker (ESC) \cite{FLL+02}, a tool called Houdini has been developed \cite{FL01}. Houdini first derives the candidate annotations from the code using annotation templates, and then removes the false annotations by combining ESC. Similar to Engler et al’s work, the annotations derived by Houdini are limited by the templates and also require efforts from programmers.

Dynamic invariant detection can extract specifications from programs’ dynamic executions \cite{ECGN01, PE04}. Nimmer and Ernst investigated the relationship between dynamic and static information, and showed that dynamical specification generation can capture some non-trivial and useful semantic information \cite{NE02a, NE02b}. They also justified that even the unsound techniques can generate useful specifications, which validates our technique for generating specifications by extracting implicit programming rules from source code.

7.4 Specification-based Checking

LCLint \cite{EGHT94} is a light-weight static checker. Provided with the source code and the specification written in the LCL language by programmers, LCLint reports inconsistencies
between the code and specification. Taghdiri proposed a static analysis method to refine specification for error detection \cite{Tag04}. The user first provides a partial specification of a procedure, and the specification is then iteratively refined via counterexample analysis. Compared with these work, our PR-Miner extracts programming rules automatically.

7.5 Empirical Studies on Software Defects

Many efforts have been made to studying fault related characteristics of software systems \cite{BP84, CC00, CYC+01, FO00, Li92, Pan93, OWB05, SC92, VI84}. They show some important results and have thrown up some counter-intuitive findings. For example, Chandra et al. \cite{CC00} show that most bugs in release software are non-transient bugs, which is against the intuition that transient bugs are more difficult to reproduce and hence to fix so that most bugs left in release software should be transient. Another example in case is that Ostrand et al. \cite{OW02} found that majority post-release faults occurred in files that had no pre-release faults. This observation contradicts the conventional wisdom that the best places to find bugs in release software are places that are faulty in pre-release. Therefore, most testing efforts for post-release software should be put on previous fault-free or less-faulty parts instead of most faulty parts.

Our work is different from previous work in two aspects. First, software system characteristics have changed a lot due to various reasons as we discussed in Chapter \ref{chap:background}. In addition, most previous work focus on bug density related characteristics. On the other hand, we did a more comprehensive study on bug characteristics in modern software. We not only classify bugs according to their root causes, impacts, and other characteristics, but also analyze the correlation between them.

Some recent studies have shown the promising benefits of machine learning techniques for bug triage and diagnosis. Anvik et al. have applied machine learning techniques to suggest a small number of developers suitable to resolve a bug report \cite{AHM06, CM04}.
Podgurski et al. proposed to cluster function call profiles from automated automated failure reports [PLF+03]. Different from our study which is based on a large number of real-world bug reports, they applied the classification techniques on profiles of failed executions, and their clusters were used to prioritize software faults and to help diagnosis the causes.

**Fault Prediction**  Bug characteristics and modification history learned from empirical studies can be used to predict what modules are more fault-prone. Several models have been developed on this [GKMS00, GMCS04, OWB05]. By identifying relevant source code from change history, Ying et al. tried to predict the potential source code changes in a modification task [YMNCC04]. Our study analyzes the correlations between root causes and impacts so that our results can also provide developers useful clues to locate the possible root causes.

### 7.6 Open Source Software Systems

With the increasing deployment of OSS systems, the OSS has received more attention recently. Due to the difference between the development of open source and closed source software systems [CLM05, FF00, GT00], they also differ in various aspects of bug characteristics.

Code quality is one of the major concerns for OSS. Mockus et al. analyzed defect density and problem resolution intervals in two OSS projects [MFH00, MFH02]. Similarly, Paulson et al. [PSE04] analyzed the evolutionary and static characteristics by comparing OSS systems with closed source software systems. Both of them drew the same conclusion that OSS systems have fewer defects since defects are found and fixed more rapidly. However, Stamelos et al. assessed code quality in OSS using a measurement tool, and they found that the quality is lower than that which is expected in industrial standard [SAOB02]. Our study is going to further analyze the fixing time for different types of bugs so that the deeper insight can be beneficial to software development and testing.
Security issue is another big concern in OSS systems [Cow03, Pay02], since OSS gives both attackers and defenders a good opportunity to exploit software vulnerability, which means defenders should put more effort and response the the security holes more quickly. In our study, we are going to show the major causes for security issues in OSS so that the results can guide development considering security. To improve security of OSS systems, several security-enhancing techniques, including software auditing, vulnerability mitigation and behavior management, are addressed by Cowan [Cow03].

7.7 Bug Benchmark

Bug benchmark is a standard to evaluate bug detection tools, but there is no widely-accepted benchmark suite yet. Most of the existing benchmarks are not representative for the current situation in software development. Some related benchmark suites such as Siemens benchmark suite [HR] and PEST benchmark suite [JMN] contain only a few types of bugs, most of which are semantic bugs, only few memory-related bugs and no multi-threading bugs. Some benchmarks such as the multi-threading program benchmarks proposed in IBM Haifa [HSU03] are based on artificial bugs instead of real ones. Recently, we have proposed a preliminary bug benchmark, BugBench [LLQ+05]. BugBench summarizes the general guidelines on the criteria for selecting representative bug benchmarks, and it contains around twenty various types of software bugs from real applications. Our study of bug characteristics in modern software can further give the guidance on how to choose representative bugs as the suite candidates.
Chapter 8
Conclusions and Future Work

In order to improve software reliability, this dissertation proposes a novel approach that applies data mining techniques to extract useful information from large software including source code and documents, and illustrates how to apply such information to bug characteristic study and bug detection. Specifically, this dissertation studies bug characteristics by investigating a large number of bug reports using text classification and information retrieval techniques, and proposes two bug detection methods, including CP-Miner that detects copy-pasted code and related bugs, and PR-Miner that extracts application-specific programming rules and detects violations that indicate potential bugs.

This dissertation demonstrates that data mining techniques are efficient and effective to analyze large software, which also indicates that data mining techniques are promising and practical in analyzing large system datasets such as source code and development documents.

To understand the characteristics of software bugs, this dissertation investigates the impacts of new factors on software bugs and studies the bug characteristics in two large modern OSS projects. Using text classification and information retrieval techniques, this studies automatically classifies tens of thousands of bugs and verifies the analysis results from sampled datasets so that the results can be more representative on bug trend and complexity of fixing. This study classifies bugs from three different dimensions, namely, root causes, impacts and software components, and further analyzes the correlation among categories in different dimensions. Furthermore, this study also analyzes the trends of different types of bugs and the difficulty of fixing them based on the large dataset from automatic classification. This study found several new interesting bug characteristics in modern OSS that can provide
useful guidelines for related research.

This empirical study shows that the major cause of software bugs are application-specific and they can also have severe impact on system reliability as memory related bugs. Application-specific knowledge such as specification is needed in order to detect such semantic bugs. However, such information is usually not well documented by programmers manually especially in large software.

To extract such information automatically, this dissertation proposes PR-Miner that uses frequent itemset mining to efficiently and automatically extract implicit programming rules and detect violations with little efforts from programmers. The rules extracted by PR-Miner are in general forms, including both simple pair-wise rules and complex ones with multiple elements of different types. The evaluation results show that PR-Miner can extract thousands of closed programming rules within seconds. In addition, PR-Miner has detected many violations to the extracted rules, and many of them have been confirmed as real bugs. Most of these bugs violate complex application-specific rules that contain more than 2 elements and are thereby difficult to be detected by previous tools. The results indicate that PR-Miner is an efficient and practical tool to extract implicit, undocumented programming rules and to detect violations in large software code.

Furthermore, this dissertation proposes an approach called CP-Miner that uses frequent sequence mining to efficiently identify copy-pasted code in large software including operating systems, and also detects copy-paste related bugs. It takes less than 20 minutes for CP-Miner to identify 190,000 and 150,000 copy-pasted segments that account for 20–22% of the source code in Linux and FreeBSD. Moreover, CP-Miner has detected 28 and 23 copy-paste related bugs in the latest versions of Linux and FreeBSD, respectively. In addition, the results show some interesting characteristics of copy-pasted codes in Linux and FreeBSD. The results indicate that maintaining copy-pasted code would be very useful for programmers because it is commonly used in large software, and it is prone to introducing hard-to-detect bugs.

This dissertation has shown that data mining techniques are promising and effective for
improving software reliability by analyzing a large amount of data including source code and development documents. However, many problems are still open as the future work.

One of the problems is what other types of information can be also analyzed by data mining techniques to improve software reliability. Very recently some studies has shown that patches to software bugs also contain useful information for bug detection [KPW06]. Besides patches, can data mining techniques be applied to some other data sources such as execution traces to improve software reliability?

Another problem is how to apply incremental mining techniques for software reliability. Although this study has shown that data mining techniques is efficient for static analysis, it still requires further efficiency improvement if the approaches are deployed as online services (e.g. scanning the new checked-in source code). Therefore, incremental mining techniques could be a desirable solution.
References


[YHA03] Xifeng Yan, Jiawei Han, and Ramin Afshar. CloSpan: Mining closed sequential patterns in large datasets. In *Proceedings of 2003 SIAM International Conference on Data Mining (SDM’03)*, San Francisco, CA, May 2003.


Vita

RESEARCH INTERESTS


EDUCATION

• University of Illinois at Urbana-Champaign
  2006 Ph.D. Computer Science, Advisor: Professor Yuanyuan Zhou
• Tsinghua University, Beijing, China 2001 M.E. Computer Science & Technology, Advisor: Professor Jinrong Zhao
• Tsinghua University, Beijing, China 1999 B.E. Computer Science & Technology

RESEARCH EXPERIENCE

• Research Assistant, 2002–2006, University of Illinois, Urbana-Champaign, Champaign, Illinois, USA
  Worked on data mining techniques for improving reliability, performance and manageability of computer systems.

• Summer Intern, Summer 2004, IBM almaden Research Lab, San Jose, California
  Worked on automated management of storage systems.
TEACHING EXPERIENCE

- Teaching Assistant. Course: Computer System Organization, University of Illinois at Urbana-Champaign, Spring 2002

- Teaching Assistant. Course: Operating Systems Design, University of Illinois at Urbana-Champaign, Fall 2001

PUBLICATIONS

Journal Articles


Conference Papers


- **Have Things Changed Now? – An Empirical Study of Bug Characteristics in Modern Open Source Software.**

- **PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code.**

- **BugBench: A Benchmark for Evaluating Bug Detection Tools.**

- **CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code.**

- **Performance-directed Energy Management for Main Memory and Disks.**
• UCLog: A Unified, Correlated Logging Architecture for Intrusion Detection.
  Zhenmin Li, Jed Taylor, Elizabeth Partridge, Yuanyuan Zhou, William Yurcik, Cristina
  Abad, James J. Barlow, and Jeff Rosendale. In Proceedings of 12th International
  Conference on Telecommunication Systems - Modeling and Analysis (ICTSM), July
  2004.
• C-Miner: Mining Block Correlations in Storage Systems.
  Zhenmin Li, Zhifeng Chen, Sudarshan M. Srinivasan and Yuanyuan Zhou. Proceedings
• Reducing Energy Consumption of Disk Storage Using Power-Aware Cache Manage-
  ment.
  Qingbo Zhu, Francis M. David, Christo Devaraj, Zhenmin Li, Yuanyuan Zhou and
  Pei Cao. In Proceedings of the 10th International Symposium on High-Performance
• Using Data Mining to Discover Patterns in Autonomic Storage Systems.
  Zhenmin Li, Sudarshan M. Srinivasan, Zhifeng Chen, Yuanyuan Zhou, Peter Tzvetkov,
  Xifeng Yan, and Jiawei Han. 1st Workshop on Algorithms and Architectures for Self-
  Managing Systems in conjunction with ISCA and SIGMETRICS, June 2003.
• Fragmentation-based D-MAC Protocol for Wireless Ad Hoc Networks. Zhenmin Li,
  Pin Zhou, and Chao-Ju Hou. In Proceedings of IEEE International Conference on