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

Zhenmin Li and Yuanyuan Zhou
Department of Computer Science
University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA
{zli4, yyzhou}@cs.uiuc.edu

ABSTRACT

Programs usually follow many *implicit* programming rules, most of which are too tedious to be documented by programmers. When these rules are violated by programmers who are unaware of or forget about them, defects can be easily introduced. Therefore, it is highly desirable to have tools 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.

This paper proposes a *general* method called PR-Miner that uses a data mining technique called frequent itemset mining to efficiently extract implicit programming rules from large software code written in an industrial programming language such as C, *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, we also propose an efficient algorithm to automatically detect violations to the extracted programming rules, which are strong indications of bugs.

Our evaluation with large software code, including Linux, PostgreSQL Server and the Apache HTTP Server, with 84K–3M lines of code each, shows that PR-Miner can efficiently extract thousands of general programming rules and detect violations within 2 minutes. Moreover, PR-Miner has detected many violations to the extracted rules. Among the top 60 violations reported by PR-Miner, 16 have been confirmed as bugs in the *latest version* of Linux, 6 in PostgreSQL and 1 in Apache. Most of them violate complex programming rules that contain more than 2 elements and are thereby difficult for previous tools to detect. We reported these bugs and they are currently being fixed by developers.

Categories and Subject Descriptors: D.2.4 [SOFTWARE ENGINEERING]: Software/Program Verification — *Statistical methods*; D.2.5 [SOFTWARE ENGINEERING]: Testing and Debugging — *Debugging aids*; D.2.7 [SOFTWARE ENGINEERING]: Distribution, Maintenance, and Enhancement — *Documentation*

General Terms: Algorithms, Management, Documentation

ESEC-FSE'05, September 5–9, 2005, Lisbon, Portugal.

Keywords: automated specification generation, automated violation detection, data mining for software engineering, programming rules, pattern recognition, static analysis

1. INTRODUCTION

1.1 Motivation

Programs usually follow many implicit programming rules. A simple example of a programming rule is the function call pair of `lock` and `unlock`: a call to `lock` should be followed by a call to `unlock` later. Besides such a well-known programming rule, there are many other implicit rules in large software. For example, as shown in Figure 1, PostgreSQL, a well-known open-source database server, contains an implicit programming rule that a call to `SearchSysCache` must be followed by `ReleaseSysCache`. The reason is that the function `SearchSysCache` returns a cache copy of a specified tuple; so after the caller finishes using the tuple, it must call `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.

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 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 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.

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

Figure 1: A function-pair rule in PostgreSQL (extracted by PR-Miner). This rule appears 209 times in PostgreSQL code.

```
postgresql-8.0.1/src/backend/commands/tablecmds.c:
5686 AlterTableCreateToastTable(Oid relOid, bool silent)
5687 {
        ......
5692    Relation  class_rel;
        ......
5853    class_rel = heap_openr (...);
        ......
5863    simple_heap_update (class_rel, ...);
        ......
5866    CatalogUpdateIndexes(class_rel, ...);
        ......
5870    heap_close (class_rel, ...);
        ......
5891 }
```

Figure 2: A complex programming rule containing four functions and one variable in PostgreSQL (extracted by PR-Miner). This rule appears 68 times in PostgreSQL code.

```
linux-2.6.11/drivers/isdn/hisax/config.c:
771 void ll_stop(struct IsdnCardState *cs)
772 {
773     isdn_ctrl ic;
775     ic.command= ISDN_STAT_STOP;
776     ic.driver= cs->myid;
777     cs->iif.statcallb(&ic);
779 }
```

Figure 3: A programming rule of variable correlation in Linux. It appears 98 times in Linux code.

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 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(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(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 [8] by Engler, Chen and Chou has conducted a preliminary investigation in this direction. They proposed a method to extract programming rules using programmer-specified rule templates such as "function a must be paired with function 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".

```
linux-2.6.11/drivers/scsi/3w-9xxx.c:
1964 int __devinit twa_probe(struct pci_dev *pdev, ...)
1965 {
1966    struct Scsi_Host *host = NULL;
        ......
1985    host = scsi_host_alloc(...);
        ......
2036    scsi_add_host(host, &pdev->dev);
        ......
2069    scsi_scan_host(host);
        ......
2088 }
```

(a) Programming rule in `twa_probe`

```
linux-2.6.11/drivers/ieee1394/sbp2.c:
688 struct scsi_id_instance_data *sbp2_alloc_device
        (struct unit_directory *ud)
689 {
        ......
692     struct scsi_id_instance_data *scsi_id = NULL;
        ......
745     scsi_host = scsi_host_alloc(...);
        ......
753     if (!scsi_add_host(scsi_host, &ud->device)) {
        ......
        // scsi_scan_host(scsi_host) is missing!
764 }
```

(b) Violation in `sbp2_alloc_device`

Figure 4: An example of a programming rule involving multiple functions in Linux 2.6.11. The rule {`scsi_host_alloc`, `scsi_add_host`}⇒{`scsi_scan_host`} appears 27 times in different functions, 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).

While the work above is inspiring and proposes a promising direction, it extracts only *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.

1.2 Our Contributions

In this paper, we propose 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 de-

[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.

tect violations with *little effort* from programmers. More specifically, our paper 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 [8] 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.

The rest of this paper is organized as follows. Section 2 briefly describes the background of the data mining technique used in PR-Miner. Section 3 presents how PR-Miner automatically extracts implicit programming rules and detects violations. Section 4 presents the evaluation results, followed by discussion of PR-Miner's limitations in Section 5. Section 6 presents related work and Section 7 concludes the paper.

2. BACKGROUND OF DATA MINING

PR-Miner is based on a data mining technique called *frequent itemset mining* [1, 14], which has broad applications, including mining motifs in DNA sequences, analysis of customer shopping behavior, etc. 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 $\mathcal{D}$:

$$\mathcal{D} = \{\{a, b, c, d, e\}, \{a, b, d, e, f\}, \{a, b, d, g\}, \{a, c, h, i\}\}$$

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 $\mathcal{D}$ 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.

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* [14], 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 [1, 15]. 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 [14].

3. 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 5. This section first gives an overview 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.

3.1 Overview

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 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, *FPclose* can find that {`spin_lock_irqsave`, `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:

```
{spin_lock_irqsave}⇒{spin_unlock_irqrestore}
{spin_unlock_irqrestore}⇒{spin_lock_irqsave}
```

The first rule says that whenever the program calls `spin_lock_irqsave`, it should also call `spin_unlock_irqrestore`, while the second says that whenever the program calls `spin_unlock_irqrestore`, it should also call `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 `spin_lock_irqsave` appears but `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 3.3 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 3.4 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 $\mathcal{D} = \{\{a,b,c,d,e\},\{a,b,d,e,f\},\{a,b,d,g\},\{a,c,h,i\}\}$ described in Section 2 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:

$\{a\} \Rightarrow \{b,d\}$ with confidence 3/4=75%
$\{b\} \Rightarrow \{a,d\}$ with confidence 100%
$\{d\} \Rightarrow \{a,b\}$ with confidence 100%
$\{a,b\} \Rightarrow \{d\}$ with confidence 100%
$\{a,d\} \Rightarrow \{b\}$ with confidence 100%
$\{b,d\} \Rightarrow \{a\}$ 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 3.4).

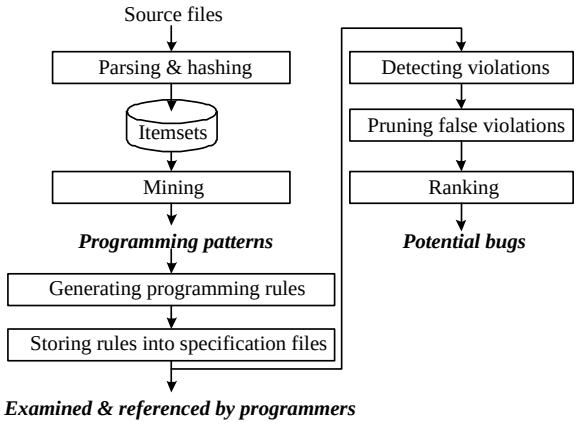


Figure 5: Flowchart of PR-Miner

3.2 Extracting Programming Patterns

3.2.1 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 [25] 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 function 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 *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, so extracting such rules remains as our immediate future work.

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 `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, the return value of calls to the same function `scsi_host_alloc` can be assigned to different local variables such as `host` and `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 `class_ref` in the code segment in Figure 2 is represented by the hash value of its data type `Relation` instead of its name `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" [2], 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 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, ...}.

Source code	Preprocessed identifiers	Hash values
linux-2.6.11/drivers/scsi/3w-9xxx.c: L1964 - 2088	T-Scsi_Host	92
	.....	.....
int __devinit twa_probe(struct pci_dev *pdev,...)	T-Scsi_Host	92
{	F-scsi_host_alloc	39
struct Scsi_Host ***host*** = NULL;	T-scsi_host_template	41
.....	.....	.....
host = ***scsi_host_alloc***(&***driver_template***, ...);	F-scsi_add_host	68
.....	T-Scsi_Host	92
retval = ***scsi_add_host***(***host***, &***pdev->dev***);	T-pci_dev.R-dev	56
.....	.....	.....
scsi_scan_host(***host***);	F-scsi_scan_host	36
.....	T-Scsi_Host	92
}	.....	.....

Table 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.

3.2.2 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, 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 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 *min_support* is set as 15. *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 `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 *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 (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.

3.3 Generating Programming Rules

As we explained briefly in Section 3.1, 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.

3.3.1 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 [1, 15]. 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. $support(I)/support(X)$, where $support(X)$ 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.

3.3.2 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 3.1, 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 (The proof is omitted due to space limitation). 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 6.

Algorithm: CLOSEDRULES($\mathcal{I}$)
Input: $\mathcal{I} = \{I_k | 1 \le k \le n\}$,
I_k has 3 fields $\langle F_k, s_k, E_k \rangle$;
Output: The closed rules $\mathcal{R}$ in condensed format.
1: Sort $\mathcal{I}$ by supports in descending order such that
$s_1 \ge s_2 \ge ... \ge s_n$
2: Mine common closed frequent sub-itemsets from $\mathcal{I}$:
$\mathcal{C} \leftarrow$ FPCLOSE($\{F_i | i = 1, 2, ..., n\}, 2$),
where $\mathcal{C} = \{C_i | 1 \le i \le m\}$ and
C_i has 3 fields $\langle F_i', s_i', E_i' \rangle$
3: **for** $i = 1, 2, ..., m$
4: Denote $E_i' = \{i_j | 1 \le j \le s_i'\}$
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_{i_j} in $\mathcal{R}$

Figure 6: Generating closed rules $\mathcal{R}$ in condensed format from closed frequent itemsets $\mathcal{I}$ mined from the first step explained in Section 3.2. 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 $\langle F_i, s_i, E_i \rangle$, 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, $\langle F_i', s_i', E_i' \rangle$ 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_{i_j} (line 5), if its support s_{i_j} is smaller than s_{i_1} (line 6), C_i is inserted into the subset of the rule for the closed frequent itemset I_{i_j}. 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 [8] can also be applied.

3.4 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 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 `scan` is missing.

As shown in Figure 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.

3.4.1 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 3.3, there would be an exponential number of rules that need to be checked.

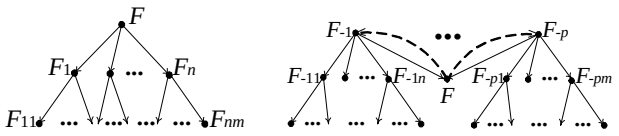
Fortunately, it is unnecessary to check all programming rules. First, if the rule has a low confidence, it is already pruned in the rule-generation step. In other words, if the confidence threshold is t, any rules with confidence smaller than t are discarded. Second, if the rule has 100% confidence, 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 CLOSEDRULES as shown in Figure 6. To do that, PR-Miner computes the confidence for the rule $F_i' \Rightarrow (F_{i_j} - F_i')$ in the loop of line 5 as $c = s_{i_j}/s_{i_1}$. 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_1} and I_{i_j} as follows. F_{i_1} contains the common sub-itemset F_i', but it does not contain $(F_{i_j} - F_i')$. It means that some supporting itemsets in E_{i_1} violate the rule $F_i' \Rightarrow (F_{i_j} - F_i')$. On the other hand, this rule is supported by the supporting itemsets E_{i_j} for F_{i_j}. Therefore, the itemsets in E_{i_1} but not in E_{i_j} violate this rule, and so the corresponding functions of the itemsets violate the programming rule.

3.4.2 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 3.2, each itemset in the database corresponds to a function definition. Suppose in an example with a function-pair (`lock` and `unlock`) rule, `unlock` is called inside a function `F` but `lock` is not. Instead, `F` calls another function `try_lock` that calls `lock`. Without inter-procedural analysis, PR-Miner would report that `F` contains a violation of missing `lock`, even though `F` contains `lock` in its callee.



(a) Checking in callees (b) Checking in all callers

Figure 7: Checking the call paths for pruning false violations

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 $F_1, ..., F_n$ called by F. As shown in Figure 7(a), we can follow the calling path more deeply by checking the functions called by callees $F_1, ..., F_n$ 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 3.2.1, 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 `try_lock` because `lock` is in `try_lock` but `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 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.

3.4.3 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 [17] can be used here to further improve the accuracy of our ranking function, which remains as our future work.

4. EVALUATION

4.1 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 2.

PR-Miner takes three parameters: *min_support*, the confidence threshold, and maximal checking path depth. By default, we set the *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 [25] 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.

Software	version	#C files	LOC	#functions
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

Table 2: Software evaluated in our experiments.

4.2 Extracting Implicit Programming Rules

Table 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.

Software	Total	F-F	V-V	F-V
Linux	32,283	1,075	8,883	22,325
PostgreSQL	6,128	379	687	5,062
Apache	283	33	92	158

Table 3: The number of closed rules extracted by PR-Miner. Notice that all F-V rules that contain more than 2 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 [8] that do not consider rules about variable correlations, PR-Miner can extract substantially more programming rules.

4.2.1 Supports of Programming Rules

Figure 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.

4.2.2 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 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.

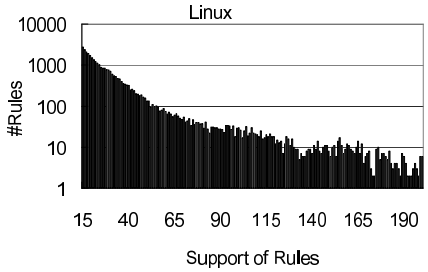


Figure 8: Distribution of rule support in the Linux code. (Y-axis is in logarithmic scale)

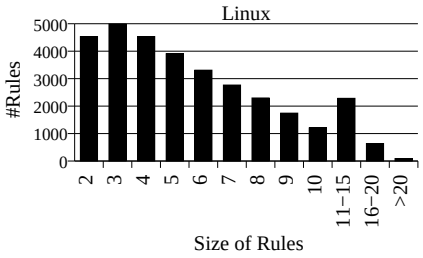


Figure 9: Distribution of rule size in the Linux code.

The above results, along with the results shown on Table 3, indicate the generality of PR-Miner over the previous work [8] because PR-Miner does not constrain the rule format or limit the number of elements in the rules to only 2.

4.3 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. 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 [8] 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.

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, which remains as our future work.

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 $480/481 = 99.8\%$. 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`.

Software	Inspected (top 60)			Uninspected
	Bugs	Specification	False Positives	
Linux	16	20	24	1387
PostgreSQL	6	9	45	87
Apache	1	0	6	0

Table 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.

4.4 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.

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.

Software	Extracting rules		Detecting violations	
	Time(s)	Space(MB)	Time(s)	Space(MB)
Linux	42	441	46	303
PostgreSQL	5	25	4	14
Apache	1	7.3	1	6.2

Table 5: Execution time and memory space of PR-Miner

5. DISCUSSION: CURRENT LIMITATIONS

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 we plan to address in our future work.

False Negatives in Detecting Violations Due to Copy-Pasting. A violation to a programming rule may be propagated to multiple modules due to copy-pasting [7, 18], 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 [18], 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 [13].

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.

Rules Spanning across Multiple Function Definitions. Since PR-Miner converts an entire functional definition to an itemset, it cannot detect programming rules spanning across multiple function definitions. To address this problem, PR-Miner needs to combine with inter-procedural analysis in a way similar to the false positive pruning described in Section 3.4.

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 will need to borrow techniques from model checking to check different control paths.

6. RELATED WORK

Due to space limitation, this section briefly describes those closely related work that are not discussed in previous sections.

Specification Generation. Automatically generating specifications has been studied for decades [6, 16, 27]. Recently, Bensalem et al describe techniques for automatically generating auxiliary predicates, including the general reaffirmed invariants, invariant propagation, refined strengthening, and invariant combination [4]. Bjørner et al present the method to generate the auxiliary assertions by extending the traditional methods [5]. Xie and Notkin propose the approach of using inferred program semantic properties for test generation and selection [28]. 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 [3]. 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) [12], a tool called Houdini has been developed [11]. 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 [9, 23]. 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 [19, 20]. 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.

Specification-based Checking. LCLint [10] 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 [26]. 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.

Applying Data Mining in Software Engineering. With the remarkably increasing scale of software, data mining techniques have demonstrated useful on dealing with a huge amount data in software analysis. Besides PR-Miner and our previous work, CP-Miner [18], mining techniques can be used in various aspects of software engineering, including development management [21], software evolution [29, 31], system understanding[24], fault-prone file identification [22] and software reuse [30], just to name a few.

7. CONCLUSIONS

This paper 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 code 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.

Our 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, by replacing the GCC front-end parser, PR-Miner can be easily applied to programs in other programming languages such as Java. In addition, as we discussed in Section 5, we envisage extending PR-Miner in several directions to address the limitations in the current prototype.

8. ACKNOWLEDGEMENTS

The authors would like to thank the anonymous reviewers for their invaluable feedback. This research is supported by IBM Faculty Award, NSF CNS-0347854 (career award), NSF CCR-0305854 grant and NSF CCR-0325603 grant. Our experiments were conducted on equipment provided through the IBM SUR grant.

9. REFERENCES

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. 20th Int. Conf. Very Large Data Bases*, 1994.
[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools*. 1986.
[3] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *Proc. of the 29th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 4–16, 2002.
[4] S. Bensalem, Y. Lakhnech, and H. Saidi. Powerful techniques for the automatic generation of invariants. In *Proc. of the 8th Int. Conf. on Computer Aided Verification*, 1996.
[5] N. Bjørner, A. Browne, and Z. Manna. Automatic generation of invariants and intermediate assertions. *Theoretical Computer Science*, 173(1), 1997.
[6] M. Caplain. Finding invariant assertions for proving programs. In *Proc. of the Int. Conf. on Reliable software*, 1975.
[7] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. R. Engler. An empirical study of operating system errors. In *Proc. of the 18th ACM Symp. on Operating Systems Principles*, 2001.
[8] D. Engler, D. Y. Chen, and A. Chou. Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In *Proc. of the 18th ACM Symp. on Operating Systems Principles*, 2001.
[9] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. Software Engineering*, 27(2), 2001.
[10] D. Evans, J. Guttag, J. Horning, and Y. M. Tan. LCLint: A tool for using specifications to check code. In *Proc. of the ACM SIGSOFT '94 Symp. on the Foundations of Software Engineering*, 1994.
[11] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In *Proc. of the Int. Symp. of Formal Methods Europe on Formal Methods for Increasing Software Productivity*, 2001.
[12] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proc. of the ACM SIGPLAN 2002 Conf. on Programming Language Design and Implementation*, 2002.
[13] A. Garrido and R. Johnson. Refactoring C with conditional compilation. In *18th IEEE Int. Conf. on Automated Software Engineering*, 2003.
[14] G. Grahne and J. Zhu. Efficiently using prefix-trees in mining frequent itemsets. In *Proc. of the 1st IEEE ICDM Workshop on Frequent Itemset Mining Implementations*, 2003.
[15] J. Hipp, U. Güntzer, and G. Nakhaeizadeh. Algorithms for association rule mining – a general survey and comparison. *SIGKDD Explor. Newsl.*, 2(1), 2000.
[16] J. James H. Morris and B. Wegbreit. Subgoal induction. *Communications of the ACM*, 20(4), 1977.
[17] T. Kremenek, K. Ashcraft, J. Yang, and D. Engler. Correlation exploitation in error ranking. In *Proc. of the 12th ACM SIGSOFT Symp. on the Foundations of Software Engineering*, 2004.
[18] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. In *Sixth Symp. on Operating Systems Design and Implementation*, 2004.
[19] J. W. Nimmer and M. D. Ernst. Automatic generation of program specifications. In *Proc. of the 2002 Int. Symp. on Software Testing and Analysis*, 2002.
[20] J. W. Nimmer and M. D. Ernst. Invariant inference for static checking: An empirical evaluation. In *Proc. of the ACM SIGSOFT 10th Int. Symp. on the Foundations of Software Engineering*, 2002.
[21] M. Ohira, R. Yokomoriz, M. Sakai, K. ichi Matsumotoy, K. Inouez, and K. Torii. Empirical project monitor: A tool for mining multiple project data. In *Proc. of Int. Workshop on Mining Software Repositories*, 2004.
[22] T. J. Ostrand and E. J. Weyuker. A tool for mining defect-tracking systems to predict fault-prone files. In *Proc. of Int. Workshop on Mining Software Repositories*, 2004.
[23] J. H. Perkins and M. D. Ernst. Efficient incremental algorithms for dynamic detection of likely invariants. In *Proc. of the 12th ACM SIGSOFT Int. Symp. on Foundations of software engineering*, 2004.
[24] F. V. Rysselberghe and S. Demeyer. Mining version control systems for FACs (frequently applied changes). In *Proc. of Int. Workshop on Mining Software Repositories*, 2004.
[25] R. M. Stallman and the GCC Developer Community. GNU compiler collection internals (GCC). available at http://gcc.gnu.org/onlinedocs/gccint.ps.gz, 2005.
[26] M. Taghdiri. Inferring specifications to detect errors in code. In *19th IEEE Int. Conf. on Automated Software Engineering*, 2004.
[27] B. Wegbreit. The synthesis of loop predicates. *Communications of the ACM*, 17(2), 1974.
[28] T. Xie and D. Notkin. Tool-assisted unit test selection based on operational violations. In *18th IEEE Int. Conf. on Automated Software Engineering*, 2003.
[29] A. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll. Predicting source code changes by mining revision history. In *Proc. of Int. Workshop on Mining Software Repositories*, 2004.
[30] Y. Yusof and O. F. Rana. Template mining in source-code digital libraries. In *Proc. of Int. Workshop on Mining Software Repositories*, 2004.
[31] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *Proc. of the 26th Int. Conf. on Software Engineering*, 2004.