AntMiner: Mining More Bugs by Reducing Noise Interference

Bin Liang[1,2], Pan Bian[1,2], Yan Zhang[1,2], Wenchang Shi[1,2], Wei You[1,2] Yan Cai[3]

[1] Key Laboratory of Data Engineering and Knowledge Engineering (Renmin University of China), MOE, Beijing, China
[2] School of Information, Renmin University of China, Beijing, China
{liangb, bianpan, annazhang, wenchang, youwei}@ruc.edu.cn

[3] State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
ycai.mail@gmail.com

ABSTRACT

Detecting bugs with code mining has proven to be an effective approach. However, the existing methods suffer from reporting serious false positives and false negatives. In this paper, we developed an approach called *AntMiner* to improve the precision of code mining by carefully preprocessing the source code. Specifically, we employ the program slicing technique to decompose the original source repository into independent sub-repositories, taking critical operations (automatically extracted from source code) as slicing criteria. In this way, the statements irrelevant to a critical operation are excluded from the corresponding sub-repository. Besides, various semantics-equivalent representations are normalized into a canonical form. Eventually, the mining process can be performed on a refined code database, and false positives and false negatives can be significantly pruned. We have implemented *AntMiner* and applied it to detect bugs in the Linux kernel. It reported 52 violations that have been either confirmed as real bugs by the kernel development community or fixed in new kernel versions. Among them, 41 cannot be detected by a widely used representative analysis tool *Coverity*. Besides, the result of a comparative analysis shows that our approach can effectively improve the precision of code mining and detect subtle bugs that have previously been missed.

CCS Concepts

• **Software and its engineering →Automated static analysis.**

Keywords

Bug detection; Code mining; Program slicing

1. INTRODUCTION

In recent years, various code mining approaches have been proposed to automatically extract implicit programming rules from source code repositories [5, 8, 11, 24-28, 30, 31, 34-38, 41, 42, 48, 49]. In particular, such approaches on bug detection have been proven to be very effective [25, 36, 38, 42, 48, 49]. For example, *PR-Miner* [25] has detected many real bugs in large-scale systems, including Linux kernels, Apache HTTP Server, and PostgreSQL database. Most of these bugs violate complex implicit programming rules, which are hard to be detected by traditional approaches as they need well documented rules [13, 20]. Currently, some commercial bug detection systems have also employed the idea of programming rules extraction. For example, *Coverity* [3], one of the most widely used bug detection tools, leverages the statistical approach to automatically extract implicit programming rules and detects related bugs in some of its checkers (e.g., the *NULL_RETURNS* checker).

ICSE '16, May 14-22, 2016, Austin, TX, USA
© 2016 ACM. ISBN 978-1-4503-3900-1/16/05...$15.00
DOI: http://dx.doi.org/10.1145/2884781.2884870

Generally, detecting bugs with code mining involves three steps:

(1) Preprocessing the source code repository to generate a database (called code database in this paper) suitable for mining, in which each record is mapped from a program unit (e.g., a function definition).

(2) Applying data mining algorithms to extract frequent patterns in the code database as programming rules.

(3) Detecting any violations to the extracted programming rules (often on the code database) as potential bugs.

Among three steps, the first one is the key step. It, to a large extent, determines whether a set of precise programming rules could be mined in the second step and how many false positives and false negatives are reported in the third step.

Like traditional bug detection approaches [13, 20, 46], detecting bugs with code mining also suffers from reporting false positives and false negatives, and may produce worse results than traditional ones. Although some works could be adapted to reduce these false positives and false negatives [25, 30, 34, 38], they mainly focus on reducing the imprecisions of the second and third steps. However, few works focus on the first step. According to our empirical investigation (see Section 2 for some examples), a large number of false positives and false negatives are introduced in the first step due to the following two reasons.

First, in a program that involves a certain (implicit) programming rule, there often exist statements that are irrelevant to the rule except those implementing it. If such statements are not excluded in the first step, they may confuse the data mining algorithms adopted in the second step. As a result, rules that contain irrelevant statements may be mined; but such rules are usually useless in practical coding. More seriously, if the irrelevant statements have similar forms with those in the rule, the detection algorithm in the third step may be misled. Therefore, real violations to the rule may be missed (see §2.1), resulting in false negatives.

Second, programmers may adopt different ways to implement a same logic. If we do not transform them into a same form when generating the code database, the mining and detecting algorithms may mistake them as different programming patterns. Therefore, the supports and confidences of the mined rules may be improperly calculated. Some interested rules may even be ignored if their supports and confidences are much lower than that they are deemed to have. Previous works [11, 25] have noticed this problem. But their solutions are insufficient, and may also result in some other problems (see Section 2).

To ease our presentation, we refer to the statements that are either irrelevant to interested programming rules or semantics-equivalent but are implemented in different forms as *noise*.

In this paper, we propose an approach called *AntMiner* to address the above issues. *AntMiner* carefully preprocesses the source code to eliminate the two kinds of noises in the code database as much as possible. First, it employs a divide-and-conquer code mining approach to reduce noise introduced by irrelevant statements. Specifically, the whole program is decomposed into different sub-repositories according to a set of critical operations (misusing them tends to cause bugs). As a result, program statements irrelevant to the critical operations are excluded; that is, all remaining statements in each sub-repository are highly relevant to certain rules. On mining programming rules, *AntMiner* only works on each sub-repository rather than on the whole repository. However, how to effectively identify critical operations is not a trivial task. In this paper, we propose a method to automatically extract them from the source code (see §3.3).

Second, when converting a sub-repository to a code database, *AntMiner* carefully normalizes program statements such that different implementation forms of the same logic are normalized. Currently, it focuses on the variants of basic elements of a program, including variable names, expressions, and control structures. Hence, most common semantics-equivalent representation forms that might interfere with the mining algorithms are normalized into a same canonical representation form, and the precision of code mining could be significantly improved accordingly.

We have implemented *AntMiner* as a prototype tool to evaluate its bug detection ability. We applied it to the Linux kernel (v2.6.39) and compared it with a widely used bug detection tool *Coverity*. Although the Linux kernel has been heavily analyzed previously for bug detections [24, 25, 28, 36], *AntMiner* is still able to detect a set of 52 real bugs (violations). Among these bugs, 24 of them were directly confirmed as unknown bugs by kernel developers [1] (see Table 3 and Table 5), and 28 of them have been fixed in new kernel versions before we submitted these 52 bugs to the Bugzilla. In addition to the above confirmed bugs, there are also 9 suspects waiting for confirmation by Linux kennel developers. As a comparison, there are 41 out of the 52 confirmed bugs cannot be detected by *Coverity*, whose checkers also adopt the implicit rules extraction technique to detect bugs, in addition to doing this based on well-defined bug detection rules. Besides, it should be noted that, among those 24 bugs confirmed by kernel developers, there are 2 bugs having existed in Linux kernel for more than 8 years (one was introduced in v2.6.5 and the other was introduced in v2.6.14); and other 17 bugs have existed since Linux kernel 2.6.34 released in May 2010. We further conducted a comparative evaluation; and the result shows that, without our method, about 73% (22 out of 30) bugs were not detected, and the imprecision of the mined rules increased from 24.5% to 87.9%.

linux-2.6.39/sound/pci/lx6464es/lx6464es.c:

```
839 static int __devinit lx_pcm_create (struct lx6464es *chip)
840 {
         ......
853      err = snd_pcm_new(chip->card, (char *)card_name, 0,
854                1, 1, &pcm);
         // if (err < 0) {… return err;} is neglected there!
855
856      pcm->private_data = chip;
857
858      snd_pcm_set_ops(pcm, …);
         ……
864      err = snd_pcm_lib_preallocate_pages_for_all(pcm, …,
865                                   snd_dma_pci_data(chip->pci),
866                                   size, size);
867      if (err < 0)
868            return err;
         ......
873      return 0;
874 }
```

Figure 1. A violation to the implicit rule {*err* = *snd_pcm_new*(), if (*err* < 0), *snd_pcm_set_ops* ()}, lacking a necessary checking for the return value of the call to *snd_pcm_new*().

The evaluation result shows that our approach can effectively improve the precision of code mining and detect subtle bugs that have previously been missed.

This paper makes the following main contributions.

- A divide-and-conquer code mining method. The program slicing technique is employed to decompose the source code repository into a series of independent sub-repositories. Rule mining and violation detecting on a sub-repository can survive from the noise items caused by irrelevant statements.
- A simple but effective statement normalization method. With this method, the most common semantics-equivalent representations that might interfere with the mining algorithms can be transformed to a canonical representation form.
- A bug detection prototype system. It can be applied to real world large systems. The evaluation result shows that the system can effectively detect a number of subtle bugs that have previously been missed.

2. MOTIVATING EXAMPLES

In this section, we demonstrate the necessity on eliminating noise items from the code database by some program samples collected from the Linux kernel 2.6.39.

2.1 Noise Introduced by Irrelevant Statements

In Linux kernel programs, there is an implicit programming rule that the return value of the function *snd*_pcm_*new*() should be checked to make sure that a new *snd_pcm* instance is created successfully before passing it to the function *snd_pcm_set_ops*(). Figure 1 shows a program that violate the above rule. That is, right after line 854, there is no checking on the returned value.

Using a mining algorithm (e.g., *frequent itemset mining*), the frequent pattern {*err* = *snd_pcm_new*(), if (*err* < 0), *snd_pcm_set_ops*()} can be extracted from the kernel code. The pattern are taken as a programming rule to detect related bugs, as done in [25]. Unfortunately, the program in Figure 1 actually contains all elements of the rule. As a result, the program will be mistaken as a support rather than a violation to the rule. This false negative is actually caused by the conditional statement at line

```
linux-2.6.39/kernel/taskstats.c:
403 static int cgroupstats_user_cmd (…, struct genl_info *info)
404 {
        ……
408     struct nlattr *na;
        ……
414     na = info->attrs[CGROUPSTATS_CMD_ATTR_FD];
415     if (!na)
416         return -EINVAL;
        ……
425     rc = prepare_reply(info, CGROUPSTATS_CMD_NEW, &rep_skb,
426                        size);
427     if (rc < 0)
428         goto err;
429
430     na = nla_reserve(rep_skb, CGROUPSTATS_TYPE_CGROUP_STATS,
431                      sizeof(struct cgroupstats));
        // if (!na) {… return -EMSGSIZE;} is neglected there!
432     stats = nla_data(na);
433     memset(stats, 0, sizeof(*stats));
        ……
446 }
```

Figure 2. A violation to the rule {*na*=*nla_reserve*(), if (!*na*), *nla_data*()}, lacking a necessary checking for the return value of the call to *nla_reserve*().

867 as the statement is irrelevant to both function *snd_pcm_new*() (at line 853) and *snd_pcm_set_ops*() (called at line 858).

Methods based on order-sensitive data mining techniques, e.g., *frequent subsequence mining* [41, 42], may detect such a bug. However, if the irrelevant statements "err = *f*(); if (*err* < 0)" appear between line 853 and 858 (which is quite possible in practice), the methods would fail to detect the bug. Besides, such order-sensitive methods have higher time complexity than itemset mining methods on extracting programming rules. To make them scalable to mine rules from large-scale software (e.g., the Linux kernel, which has tens of millions lines of code), a bigger *minimum support threshold* should be set. However, this would miss rules with relatively small supports.

The root cause of the false negative is that, the statement (at line 867) irrelevant to the rule confuses the mining algorithm. Therefore, we can detect the bug by actively removing the irrelevant statements before mapping the program into a code database. To achieve this goal, we have to identify which statements we do care about and which we do not care about. Note that, a bug often occurs because of incorrectly performing some critical operations, such as calling a function (e.g., *strcpy*()) without satisfying its preconditions, or returning an improper value under certain conditions. From this observation, we only care about statements that either directly perform a critical operation or impact the execution of a critical operation; and the other statements are regarded as irrelevant ones and are removed. Given a critical operation, the program slicing techniques [43] can help achieve this goal. On the example in Figure 1, assuming that the call to function *snd_pcm_set_ops*() is a critical operation (act as a slicing criterion), the conditional statement (at line 867) can be excluded from the corresponding slice; hence, the detection algorithm is able to catch this violation.

2.2 Noise Introduced by Relevant Statements

In some cases, the interference can also be introduced by statements that are relevant to the critical operations. In practice, a statement may have no effect on the logic of a programming rule even when there is a control or data dependence relationship between it and the critical operation. For example, in Linux kernel code, there is another implicit programming rule that the return

```
linux-2.6.39/net/bluetooth/rfcomm/tty.c:
        if (dev->tty && !C_CLOCAL(dev->tty))
                tty_hangup(dev->tty);
```

(a) Effective validation

```
linux-2.6.39/drivers/tty/moxa.c:
        tty = tty_port_tty_get(&p->port);
        if (tty && C_CLOCAL(tty) && !dcd)
                tty_hangup(tty);
```

(b) Ineffective validation

Figure 3. Two validation samples to the actual parameter of function *tty_hangup*()

value of the function *nla_reserve*() should be checked against NULL before it is passed to the function *nla_data*(). Figure 2 shows a violation to this rule, where a necessary validation on the return value right after line 431 is missing. However, the call to the critical operation *nla_data*() at line 432 is directly control dependent on the statement at line 415. That is, the statement at line 415 is reserved even after program slicing. This also confuses the mining algorithms in a way similar to the example in Figure 1, resulting in a false negative.

In essence, this false negative can be reduced if the mining algorithm could distinguish the variable *na* used at line 415 from the one in the frequent pattern {*na* = *nla_reserve*(), if(!*na*), *nla_data*()}. In a previous work [25], variables are represented with their data types. However, to address the above problem, more semantic information needs to be introduced when renaming variables. In fact, the variable *na* in the above pattern keeps the return value from *nla_reserve*(), whereas the variable *na* used at line 415 keeps the value of an array element. If a variable is renamed with a new canonical name which can reflect where its value comes from (see §3.5), the confusion will be avoided to a large extent. Therefore, the detection algorithm would not be interfered, and can then detect the bug in Figure 2.

2.3 Noise Introduced by Inconsistence

The validation to the sensitive data is usually implemented with conditional statements. In general, missing effective validations may result in programming bugs. For example, the program shown in Figure 3(a) illustrates an effective validation to the actual parameter of function *tty_hangup*(). However, in the program shown in Figure 3(b), an incorrect conditional expression (i.e., "*C_CLOCAL*(*tty*)" rather than "!*C_CLOCAL*(*tty*)") is used to enforce the validation. This will result in a bug. On the other hand, programmers can use different or even opposite conditional expressions to enforce the same validation. For example, there are two validations for the return value of function *dev_alloc_skb*() shown in Figure 4(a) and (b) respectively. Although the two validations employ completely opposite conditional expressions, considering the contexts, both of them effectively guarantee that the return value is not NULL before passing to *skb_reserve*().

For these programs, if the mining algorithm is directly applied to them, we may miss the bug in Figure 3(b) or receive a false alarm on one of the two programs in Figure 4. A previous work [11] also noticed this problem, and proposed a method to simply make the similar expressions identical, e.g., replacing "!=" with "==" in the control points without considering the semantics of related control structures. However, this method is not suitable for processing the above samples.

A better solution is to normalize the control structure to a canonical form, ensuring that the predicate, which must be satisfied when executing a critical operation, is explicitly specified in its

```
linux-2.6.39/drivers/isdn/i4l/isdn_v110.c:
422     if ((skb = dev_alloc_skb(v->framelen + v->skbres))) {
423             skb_reserve(skb, v->skbres);
424             memcpy(skb_put(skb, v->framelen), ...);
425     }
```

(a) Effective validation

```
linux+v2.6.39/drivers/media/dvb/dvb-core/dvb_net.c:
856     if (!(skb = dev_alloc_skb(pkt_len - 4 - 12 + 14 + 2 - snap))) {
                // do something
859             return;
860     }
861     skb_reserve(skb, 2);    /* longword align L3 header */
```

(b) Another effective validation with different condition expression

Figure 4. Two effective validations for the return value of function *skb_reserve*().

conditional expression (see §3.5). For example, for the program shown in Figure 4(b), the conditional expression (at line 856) can be standardized to "*skb = dev_alloc_skb* (...)" to explicitly specify that the return value has been checked against null before passing it to the function *skb_reserve*(). On the other hand, the control structures in Figure 3 remain their original forms. As a result, the bug in Figure 3(b) can be detected, and no false positives are produced for the programs in Figure 4.

3. AntMiner APPROACH

3.1 Overview

Compared with most traditional code mining approaches, *AntMiner* does not directly handle the whole source code of the target system. Instead, it decomposes the source repository into a set of independent sub-repositories on preprocessing source code. The code mining is then independently performed on these sub-repositories one by one.

Figure 5 shows an overview of *AntMiner*. First, the source code is parsed into parse trees, and a *program dependence graph* (PDG) is generated for each function definition. Second, it extracts critical operations from the source code itself without human involvement. Third, according to the critical operations, the program slicing technique is employed to generate a series of sub-repositories. A sub-repository consists of the program slices associated with a specific type of critical operations. Fourth, the program slices are normalized, and then every sub-repository is converted to an *itemset database*. Fifth, a *frequent sub-itemset mining* algorithm is applied to the databases one by one to extract frequent patterns and generate programming rules. Finally, violations to these rules are detected and reported as potential bugs.

3.2 Parsing Source Code

AntMiner uses a modified GCC compiler [33] frontend to parse the source code. The source code is parsed and represented in GIMPLE, which is a language-independent, tree based representation. It should be noted that complex expressions are split into a three-address code in GIMPLE. The rest of this subsection reviews the preliminary knowledge, mainly about the PDG. Readers who are familiar with it may skip the rest of this subsection.

A PDG is computed for each function definition by using an improved algorithm proposed in [16]. In a PDG, a node represents a GIMPLE statement, and an edge represents the dependency information between two nodes. A PDG consists of a *control dependence subgraph* (CDS) and a *data dependence subgraph* (DDS):

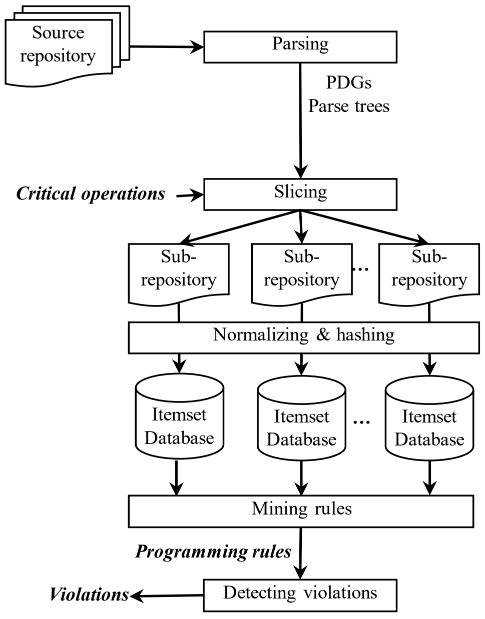


Figure 5. An overview of *AntMiner*.

- The CDS describes the control dependencies among statements. In CDS, if a statement s_2 is control dependent on a condition statement s_1, there is a control dependence edge from s_1 to s_2 labeled with either *T* or *F*, indicating that s_2 is executed on the *True* or *False* branch of s_1, respectively. They are denoted as $\langle s_1, s_2, T\rangle$ or $\langle s_1, s_2, F\rangle$, respectively.
- The DDS describes the data dependencies among statements. A statement s_2 is *data dependent* on a statement s_1 if there is a variable *x* defined in s_1, used at s_2, and an executable path from s_1 to s_2 along which there is no intervening definitions of *x*. In the DDS, there is a data dependence edge from s_1 to s_2 labeled with *x* to indicate the dependence relationship, denoted as $\langle s_1, s_2, x\rangle$. In our implementation, a variable is regarded to be defined at a statement if it is explicitly assigned to a value or it is passed to a function by reference. For example, in Figure 1, both variables *err* and *pcm* are defined at the statement at line 853. Variable *pcm* is used at line 858. Thus, the data dependence edge $\langle 853, 858, pcm\rangle$ is added in the DDS.

3.3 Extracting Critical Operations

Bugs or vulnerabilities often stem from incorrectly performing some critical operations. Currently, without loss of generality, *AntMiner* mainly concerns two types of critical operations.

Bug-prone function calls. A function can be regarded as bug-prone if an inappropriate invocation to it tends to cause a program bug. In practice, the call to a bug-prone function often acts as the key element of a programming rule. In fact, in the Common Weakness Enumeration (CWE, a list of software weaknesses) [2], the sinks of many weaknesses are calls to some security-sensitive functions. For example, the call to function *strcpy*() is a bug-prone operation. Note that many bug-prone functions are not well-known like *strcpy*(). In many cases, they may even be undocumented.

It is unreasonable to take all function calls as critical operations because some functions may hardly cause a bug. For example, the call to function *isdigit*() (a function in C language to check whether a character is a decimal digit) should not be treated as a bug-prone operation.

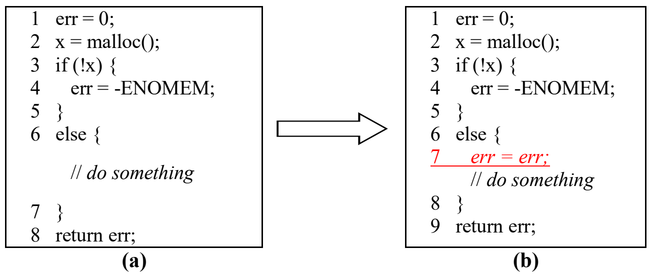
One way to collect bug-prone functions is to identify them by manually analyzing the system documents or even the source code. However, it is very difficult and tedious, if not impossible, to manually identify the undocumented application-specific bug-prone functions from a large-scale system (e.g., the Linux kernel). To address this issue, we design a heuristic method to automatically extract potential bug-prone operations from source code.

In practice, a bug-prone function call usually produces an error when one or more of its parameters hold illegal values. In a practical system, to make sure that the system works correctly, these sensitive parameters are often validated before passing them to the bug-prone function. In programming, a validation to sensitive data is generally implemented as a conditional comparison. To this end, our approach to identifying bug-prone functions is based on the intuition: before a bug-prone function is called, one or more of its parameters should be directly or indirectly checked by a conditional statement; and the function should not be executed if the check fails.

Specifically, we perform a dependence flow analysis on the PDGs (see §3.2) to identify potential bug-prone functions. First, a *set* of *validated variables* (*VVS*) is computed for each conditional statement. A *VVS* contains all variables that are directly or indirectly checked by a certain conditional statement. To compute *VVS*, the DDS of the PDG is backward traversed starting from the conditional statement, and labels (i.e., variables) of edges visited during the traversal are added into the *VVS*. Second, every function call is examined to see whether it is control dependent on a conditional statement by backward traversing the CDS of the PDG. If there exists such a conditional statement, we further examine whether there are parameters protected by the conditional statements. A variable v is protected by a conditional statement if either it belongs to the *VVS* of the conditional statement or there is another variable v' used in the definition statement of v and v' is protected by the conditional statement. If there exists such a parameter p, the function is identified as a bug-prone function candidate, and a protected-counter for p (each parameter with a protected-counter) is increased by one. Finally, for every candidate, a simple statistical method is applied to determine whether it is bug-prone. Assuming that the function $f()$ is called for T times and the protected-counter of one of its parameter p is t. If the ratio t/T is larger than a predefined threshold λ (e.g., 70% in this paper), function $f()$ is then considered as a bug-prone function on p.

For example, in Figure 4(a), the *VVS* of the conditional statement at line 422 is {*skb*, *v->framelen*, *v->skbres*, *v*}. The function call to *skb_reserve*() at line 423 is control dependent on the conditional statement at line 422, and its two actual parameters *skb* and *v->skbres* belong to the *VVS* of the conditional statement. Therefore, function *skb_reserve*() is taken as a bug-prone function candidate and the protected-counters of its two parameters are increased by one respectively. After scanning the whole kernel code, we find that *skb_reserve*() is called 503 times in total, among which 491 times its first parameter is checked by conditional statements, and the corresponding t/T is about 97.61%. Consequently, the function *skb_reserve*() is identified as a bug-prone function with respect to its first parameter.

Based on the above method, *AntMiner* automatically collects potential bug-prone functions, without requiring any prior knowledge on the target system. In our empirical study, it finds thousands of bug-prone functions from the Linux kernel in about 30 minutes, which saves a great deal of human efforts.



(a)

```
1  err = 0;
2  x = malloc();
3  if (!x) {
4     err = -ENOMEM;
5  }
6  else {

      // do something

7  }
8  return err;
```

(b)

```
1  err = 0;
2  x = malloc();
3  if (!x) {
4     err = -ENOMEM;
5  }
6  else {
7     err = err;
      // do something
8  }
9  return err;
```

Figure 6. An example of inserting dummy statements.

Function returns. Function return statements are very common in programming, but very subtle bugs may be caused if they return improper values. When the return value cannot correctly reflect the execution result of a function on a certain path, its callers is no way to know what actually happens in the callee. For example, in Linux kernel, programmers may incorrectly set the error code to *zero* rather than *-ENOMEM* when a function fails to allocate a requested memory. The callers may believe an allocated memory space is ready for subsequent operations. This may cause memory access errors and even crash the system. In practice, how to correctly set the return values is often undocumented. As a result, it is very difficult to detect improper return bugs with traditional tools.

The return value may be improperly set at every return point of a program. Hence, all return statements are directly selected as potential critical operations.

3.4 Slicing Source Code

The original definition of program slicing was proposed by Weiser [43]. By introducing the notion of PDG, Ottenstein et al. [29] converted the slicing problem into a reachability problem in a dependence graph representation of the program. Based on their study, several algorithms are proposed for effective slices computing [16, 22]. Based on these algorithms, a program slice consists of all statements which may affect the values at some points of interest (i.e., slicing criterion) or determine whether it should be executed.

Identifying Slicing Criteria. To compute program slices for a critical operation, the corresponding slicing criteria should be identified firstly. For a bug-prone function, nodes that call this function in the PDG can be directly taken as slicing criteria. For example, in Figure 4(b), the function *skb_reserve*() is bug-prone on its first parameter and is called at line 861. Therefore, ⟨861, {*skb*}⟩ is used as a slicing criterion, where *skb* is the interested parameter at line 856.

For the return statements (i.e., the second type of our critical operations), they are not directly taken as the slicing criteria. It is because a return statement is usually a merging point of multiple execution paths of a function. And the return value may also represent multiple execution results of the function. In practice, different execution results often represent different runtime logics. To reduce the noise introduced by such runtime logics that are irrelevant to a certain return value as much as possible, the program points where the return values are actually determined are taken as the slicing criteria.

However, the slicing result may not be what we desire when the return value is implicitly defined. For example, in Figure 6(a), variable *err* keeps the return value and is initialized with *zero* (at line 1). When the call to *malloc*() (at line 2) fails (checked at line 3), *err* is set to *-ENOMEM* (at line 4). Otherwise, the value of *err* remains *zero*. The logic of this program is "*-ENOMEM* should be returned when the call to *malloc*() fails; otherwise, *zero* should be returned". When directly taking statements that explicitly set the value of *err* (i.e., line 1 and line 4) as the slicing criteria, the slicing result is no way to capture the logic "*zero* should be returned when the call to *malloc*() succeeds".

To address the above issue, we insert dummy statements that reset return values into the original program. For every conditional statement, if the return value (e.g., kept in variable *err*) is only explicitly defined on one of its two branches, a dummy statement (i.e., "*err* = *err*;") will be added to the beginning of the other branch. Then, on the PDG of the modified program, a backward dataflow analysis can be performed to identify desirable slicing criteria. A statement (may be a dummy one) is regarded as a slicing criterion if there exists a data dependence edge from it to a return statement.

For example, in Figure 6(a), the dummy statement "*err* = *err*;" is inserted into the "else" branch of the conditional statement "if (!*x*)" (at line 3), because the return value is only explicitly defined on the other branch. The modified program is shown in Figure 6(b), and in the PDG of it, there exists two data dependence edges to the return statement (at line 9): one from the statement at line 4, and the other from the statement at line 7. Therefore, the statements at line 4 and line 7 are identified as slicing criteria.

Slicing for Every Criterion. For each slicing criterion of return statements, the PDG is traversed backward from it, and the encountered nodes are marked. All the marked nodes make up the program slice of the slicing criterion.

The above slicing algorithm is suitable for slicing criteria of return statements. However, when a slicing criterion is a call to a bug-prone function, the traversing strategy should be slightly adapted. Otherwise, some statements causing noise may not be thoroughly excluded from the slice. For example, in Figure 7, taking ⟨7, {*x*}⟩ (bug-prone function *sensitive_op*1() is called at line 7 with *x*) as a slicing criterion, the conditional statement at line 5 (i.e., "if (*len* > MAX_LEN)") remains in the obtained slice. This is because the function call is control dependent on it. However, this conditional statement does check the input to the call to *sensitive_op*2(*y*) (at line 8) rather than that to *sensitive_op*1(). If the statement remains in the slice, it may be incorrectly taken as a checking for *sensitive_op*1(*x*) by the mining algorithm.

In essence, this issue is caused by the fact that the semantic relationship between two statements may still be weak even if there is a control dependence relationship between them. To address this issue, we design a more aggressive slicing algorithm for slicing criteria that call bug-prone functions. Our algorithm also backward traverses the PDG paths starting from the statement invoking the bug-prone function (e.g. *sensitive_op*1()), and marks the encountered statements to compute the program slice. The difference is that a conditional statement is not marked if it is not homologous to the statement of the slicing criterion. Two statements *s*1 and *s*2 are homologous if either (1) *s*1 and *s*2 are data dependent on the same statement *s*3, or (2) *s*1 (or *s*2) is control dependent on statement *s*3, and *s*2 (or *s*1) and *s*3 are homologous. In this way, the conditional statement that has only control dependence relationship with the slicing criterion is not taken as a potential validation to the function and, hence, is not added into the slice. For example, in Figure 7, the conditional statement at line 5 (i.e. "if (*len* > MAX_LEN)") is homologous to the statement at line 8, but

```
1  x = get_input();
2  y = get_input();
3  len = get_length (x);
4  len = get_length (y);
5  if (len > MAX_LEN)
6     return;
7  sensitive_op1 (x);
8  sensitive_op2 (y);
```

Figure 7. A noise example that may remain in the slice.

not the statement at line 7. As a result, statement 5 is marked for the slicing criterion ⟨8, {*y*}⟩, but is not marked for ⟨7, {*x*}⟩.

Constructing Sub-repositories. The program slices for a bug-prone function make up an independent sub-repository for it. For return statements, program slices with the same return type are clustered into a sub-repository.

3.5 Normalizing and Hashing Statements

Every sub-repository is converted to an itemset database suitable for the adopted data mining algorithm [18]. Every statement is converted to a string and hashed to a number using an existing hash function *hashpjw* [7]. The hash numbers of the statements in a program slice will constitute an itemset (a bag of numbers). Before that, statements are normalized by the following three methods:

Renaming Variables. In practice, names of variables in similar contexts may vary greatly. To reduce the differences in naming, variables in every statement are given new canonical names. Specifically, (1) for each variable that either accepts a return value of a function or is taken as a reference parameter of a function is renamed as the function name plus a suffix. The string "*ret*" is used as a suffix for the former case, and an integer *i* is used for the latter case where the integer *i* indicates that the variable is taken as the *i*-th parameter of the function. (2) In other cases, each variable is renamed as its data type. For example, in Figure 8, in the statement at line 3 (i.e., "if (*a* < *b*)"), variable *a* keeps the return value of *foo*() (called at line 1), and variable *b* is a reference parameter of *foo*(). Thus, variable *a* is renamed as "*foo-ret*", while variable *b* is renamed as "*foo*-1" (*b* is the first parameter of *foo*()). Because the value of variable *c* in "*d* = *c* + *a*;" is not assigned by a function, it is renamed as its data type, i.e. "*int*".

```
1  c = 10;
2  a = foo (&b);
3  if (a < b)
4      return;
5  d = c + a;
6  bugprone_op (d);
```

Figure 8. An example for illustrating statements normalizing

Rewriting Expressions. Expressions in different forms may represent the same semantics. For example, "*a* + *b*" is equivalent to "*b* + *a*" in semantics. In theory, it is impossible to recognize and normalize all kinds of semantics-equivalent representations. In this paper, considering the significance of conditional statements and assignment statements for identifying programming rules, we mainly concern with the normalization of them. Thanks to the GIMPLE representation, this work can be focused on how to normalize binary expressions. For a binary expression "v_1 *op* v_2":

- If the operator *op* has a commutative property (i.e., "+", "*", "&", "|", "==", "!=") and the data type name of operand v_1 is lexicographically after that of v_2, the expression is transformed into "v_2 *op* v_1". For example, for an expression "*int* + *char*", because "*int*" is lexicographically after "*char*", the resulting expression is "*char* + *int*".

- If the operator *op* is a non-commutative relational operator (i.e., ">", "<", ">=", and "<=") and the data type name of operand v_1 is lexicographically after that of v_2, the positions of the two operands are exchanged, and the operator *op* is synchronously changed to *op'* (i.e., the complement operation of *op*) to preserve the semantic. For example, for a given expression "*int* >= *char*", the resulting expression is "*char* <= *int*".

Rearranging Control Structures. The same program logic may be implemented in different control structures. For example, programs in Figure 4(a) and Figure 4(b) are different in form, but they both follow the constraint that "the first parameter of *skb_reserve*() should not be NULL". To reduce the differences in form, the control structures are rearranged as follows: if a critical operation is called only when a predicate ***p*** evaluates to *false*, it is negated to ***p'*** (e.g., the negation of "*a* > *b*" is "*a* <= *b*"); accordingly, the two branches of the control structure are exchanged such that the critical operation is called only when predicate ***p'*** evaluates to *true*. In this way, the validation modes about critical operations are unified without alerting the original validation logic. For example, in Figure 4(b), the critical operation *skb_reserve*() is executed only when the predicate (at line 856) evaluates to *false*. The related control structure is rearranged, as shown in Figure 9.

By doing so, all conditional predicates, which determine whether the bug-prone operation is executed or not, will be normalized to a standard form as far as possible, making the mining algorithm more likely to be able to extract potential frequent programming patterns.

3.6 Mining Rules and Detecting Violations

AntMiner adopts the data mining algorithm *FPclose* [18] to discover *closed frequent sub-itemsets* from the itemset database. For a given sub-itemset, the number of itemsets that contain all its items is called the *support* of it. A sub-itemset is considered to be frequent if its support is bigger than or equal to a specified threshold (*min_support*). A frequent sub-itemset *A* is closed if there is no frequent sub-itemset *B* where *B* is a proper subset of *A* and *support*(*A*) = *support*(*B*), where the function *support*(*P*) computes the support of *P*.

We then mine association rules as programming rules from the extracted closed frequent sub-itemsets. An association rule has the form *A* => *B*, where *A* and *B* are closed frequent sub-itemsets, and *support*(*B*) ÷ *support*(*A*) × 100% (i.e., confidence of the rule) is larger than or equal to a given threshold *min_confidence*. The association rule *A* => *B* indicates: if an itemset in the database contains all statements in *A*, it should also contain all statements in *B*. And a violation to the rule is an itemset that contains all the items in *A* but not all the items in *B*.

Detecting violations is straightforward. A trivial method to detect the violations is to inspect all itemsets in the database and examine which is a superset of *A* but not of *B*. However, given a database with a large number of itemsets, this method might be time-consuming. To speed up violation detecting, we slightly modified *FPclose* such that when it discovers a closed frequent sub-itemset *X*, the itemsets that support *X* are also recorded, denoted as *supporter*(*X*). By doing so, any violations to an association rule *A* => *B* can be easily computed via *supporter*(*A*) − *supporter*(*B*).

Before reporting any violations to programmers, they are ranked by an empirical method. In our experience, the violation that miss

```
 if ((skb = dev_alloc_skb(pkt_len - 4 - 12 + 14 + 2 - snap))) {
           skb_reserve(skb, 2);    /* longword align L3 header */
}
else {
           // do something
           return;
}
```

Figure 9. Rearranged program of the one in Figure 4(b).

conditional statements is more likely to be a bug than those that miss function call statements. Besides, the fewer statements a violation misses, the more likely it is a bug. For that, all violations are firstly categorized into three categories: missing conditional statements, missing function call statements, and the others. Among these three categories, violations in the first category are ranked with a highest priority, followed by violations from the second category, and violations from the third category have a lowest priority. Within each category, a violation that misses fewer statements is ranked with a higher priority; and if any two violations miss the same number of statements, they are ranked by the confidences of their violated rules (i.e., violations with higher confidences are ranked with higher priority).

4. EVALUATION

4.1 Experiment Setup

We implemented *AntMiner* based on GCC compiler [33] (V4.5.0) and evaluated it on the Linux kernel 2.6.39. The kernel includes about 16,300 C files, and 110,000 functions. The Linux kernel has been scanned by dozens of bug detection tools [3, 15, 23, 24, 25, 28, 36]. The main reason for choosing the Linux kernel as the evaluation target is that we want to demonstrate the effectiveness of our approach by revealing some new bugs that are difficult to detect previously on real-world large-scale systems.

In our experiment, *AntMiner* runs on a machine with a Core i5-2520M, 2.5GHZ Intel processor and 4GB memory. Three parameters need to be specified: λ, *min_support*, and *min_confidence*. In our evaluation, we empirically determine reasonable parameters by performing a sampling analysis to the results of several experiments with different parameters settings. Specifically, we set λ to 70%, *min_support* to 10, and *min_confidence* to 85% respectively.

4.2 Experiments

In the evaluation, we firstly performed two independent experiments to automatically extract programming rules and detect related bugs for the two types of critical operations (i.e., bug-prone function calls, and function returns). The results of the two experiments are shown in §4.2.1 and §4.2.2, respectively. As a comparison, we also applied *Coverity* to detect bugs in the Linux kernel 2.6.39, to determine whether *AntMiner* can effectively discover the bugs missed by *Coverity*. For highlighting the effectiveness of *AntMiner* on reducing noise introduced by irrelevant statements and inconsistent implementations, we further evaluated *AntMiner* by disabling its program slicing and normalizing to perform a comparative analysis. The result is illustrated in §4.2.3.

Table 1. Classification of violations detected by *AntMiner*

# of total violations: 38	12 (~32%) Already fixed.	Real bugs
	18 (~47%) Confirmed as unknown bugs.	Real bugs
	5 (~13%) Regarded as false positives.	False positives
	3 (~8%) Waiting for confirmation.	Unknown

Table 2. This table describes the profile of the found bugs that have been fixed in the new versions. The first column shows the function that contains the bug; the second shows the bug-prone function; and the last labels whether the bug is detected by *Coverity*.

Function	Bug-prone Function	*Coverity*
btrfs_real_readdir()	btrfs_next_leaf()	✓
btrfs_insert_dir_item()	btrfs_release_path()	✓
picolcd_init_framebuffer()	framebuffer_release()	✓
pstore_mkfile()	d_add()	✓
nl80211_remain_on_channe l()	genlmsg_end()	×
nl80211_tx_mgmt()	genlmsg_end()	×
nl80211_get_key()	genlmsg_end()	×
l2tp_nl_session_send()	genlmsg_end()	×
l2tp_nl_tunnel_send()	genlmsg_end()	×
l2tp_nl_cmd_noop()	genlmsg_end()	×
efs_iget()	unlock_new_inode()	✓
bfs_iget()	unlock_new_inode()	✓

4.2.1 Detecting Misusages of Bug-Prone Functions

This experiment ran about 145 minutes. In total, 1,984 bug-prone functions were automatically extracted from the source code. For all these bug-prone functions, 3,524 programming rules are generated. Violations to these rules were detected.

Similar to all other static analysis tools [e.g. 3, 4], violations detected by *AntMiner* also need to be identified manually. In this study, for each bug-prone function, the top ranked rules (at most 10) and the violations to these rules were manually audited. It spent one of us about 16 hours to audit the results. The cost of manual work is acceptable on large-scale systems like the Linux kernel. Eventually, we found 38 violations that were most likely to be real bugs. Table 1 summarizes these 38 violations. To verify these violations, we firstly checked the kernel archive and found that 12 of them have already been fixed in the new kernel versions (e.g., v3.17). This means that these 12 violations (shown in Table 2) are real bugs. We then reported the other 26 violations to Linux kernel Bugzilla (the kernel development community) [1]. And so far, 18 of them have been confirmed as previously unknown bugs (i.e., real bugs) as shown in Table 3. Among the rest 8 violations, 5 are regarded as false positives by kernel developers, and the other 3 are still waiting for confirmation.

We further surveyed when the 18 confirmed bugs were introduced in the Linux kernel. We found 15 of them were introduced before kernel 2.6.34 (released in May 2010). To our surprise, the bug 44491 was introduced in kernel 2.6.5 (released in April 2004), it has been latent for 8 years until it is detected by *AntMiner*. This bug has been fixed now after we reported it. It should be noted that *AntMiner* can successfully detect some deeply hidden bugs with the help of program slicing and statement normalization, such as the three bugs presented in §2.1, §2.2 and §2.3 respectively (BugzillaID: 44541, 44621, and 49911).

We also applied *Coverity* on the same kernel source code. *Coverity* can only reported 11 of above 30 real bugs (i.e., 12 + 18) found by *AntMiner*. In other words, *Coverity* neglected 19 (63%) real bugs. Among the 11 bugs hit by *Coverity*, 10 of them were detected by the *NULL_RETURNS* checker, and the rest one was detected by the *CHECKED_RETURN* checker. Both of the two checkers can automatically infer the implicit program rules for the unmodeled function (e.g., *alloc_skb*()) to detect related bugs. For example, the *NULL_RETURNS* checker can infer the rule "*alloc_skb*() may return NULL and its return should be checked against NULL before dereferencing" by scanning the code and computing how frequently the function return is checked against NULL. According to the rule, *Coverity* can detect a real bug (BugzillaID: 44431) in function *st_int_recv*().

Table 3. This table describes the profile of the confirmed bugs. The first column lists the bug's ID in the Linux kernel Bugzilla; the second shows name of the function that contains the bug; the third shows the bug-prone function that the bug violates a rule about; and the last labels whether the bug is detected by *Coverity*.

Bugzilla ID	Function	Bug-prone Function	*Coverity*
44431	st_int_recv()	skb_reserve()	✓
44441	ldisc_open()	register_netdevice()	×
44461	sfb_dump()	nla_nest_end()	✓
44471	tmiofb_probe()	ioremap()	×
44491	setup_isurf()	pnp_port_start()	×
44541	lx_pcm_create()	snd_pcm_set_ops()	×
44551	poseidon_audio_init()	snd_pcm_set_ops()	×
44561	pcf50633_probe()	platform_device_add()	✓
44571	dcbnl_ieee_set()	nla_parse_nested()	×
44621	cgroupstats_user_cmd()	nla_data()	×
44671	ocfs2_create_refcount_tree ()	ocfs2_set_new_buffer_upt odate()	×
44681	ocfs2_create_xattr_block()	ocfs2_set_new_buffer_upt odate()	×
44691	lkdtm_debugfs_read()	free_pages()	×
49851	ipw_packet_received_skb( )	skb_reserve()	✓
49861	wl1271_debugfs_update_st ats()	wl1271_ps_elp_sleep()	×
49871	omninet_read_bulk_callba ck()	tty_flip_buffer_push()	×
49911	moxa_new_dcdstate()	tty_hangup()	×
49921	btree_write_block()	logfs_put_write_page()	✓

Because *Coverity* directly infers the implicit programming rules from the original source code, its precision is heavily interfered by the noise statements (as discussed in Section 2). As a result, some implicit programming rules and related bugs may be neglected. For example, it can't discover all the three subtle bugs presented in Section 2, which should be covered by its corresponding checkers (e.g., *NULL_RETURNS* checker).

4.2.2 Detecting Improper Return Values

As mentioned in §3.4, program slices with the same return type are clustered into a sub-repository. In kernel, when an unexpected event occurs in a function, an error code should be returned. In the kernel development, a number of subtle bugs caused by incorrect error code assigning [19, 32]. In practice, an error code is often represented by an integer. Therefore, we are especially interested in the sub-repository consists of program slices involving error code returns.

Table 4. Classification of violations detected by *AntMiner*

# of total violations: 28	16 (~57%) Already fixed.	Real bugs
	6 (~21%) Confirmed as unknown bugs.	Real bugs
	6 (~21%) Waiting for confirmation.	Unknown

The experiment ran about 120 minutes, and 6,366 programming rules were mined. Considering the sub-repository about error codes consists of much more slices than those sub-repositories about bug-prone functions, we manually inspected the top 200 rules and their violations. Eventually, we found 28 violations were most likely to be real bugs. Table 4 summarizes these 28 violations. By checking the kernel archive, we found 16 of them have been fixed in the new kernel versions (e.g., v3.17). This means

Table 5. This table describes the profile of the confirmed bugs that return improper values. The first column lists the bug's ID in the Linux kernel Bugzilla; the second shows name of the function that contains the bug; the last labels whether the bug is detected by *Coverity*.

Bugzilla ID	Function	*Coverity*
96741	atl2_probe()	✗
98551	mptfc_probe()	✗
98561	mkiss_open()	✗
98611	r592_probe()	✗
98671	mantis_dma_init()	✗
99011	myri10ge_probe()	✗

that these 16 violations are real bugs. We submitted the rest 12 suspected bugs to the kernel development community [1], and so far, 6 of them have been confirmed to be real bugs and will be fixed in later versions. These 6 bugs are shown in Table 5. The other 6 suspects are still waiting for confirmation (however, none has been confirmed as a false positive).

For example, in Figure 10, when the call to *register_netdev*() at line 766 fails, the returned error code should be further propagated upward to the callers of function *mkiss_open*(). However, the programmer forgot to set the value of *err* when *register_netdev*() fails, and *zero* is returned. This misleads the callers into believing *mkiss_open*() runs normally, even some unexpected events have occurred. *AntMiner* successfully extracted the rule that "when a call to *register_netdev*() fails, its return value rather than *zero* should be propagated upward". According to the rule, *AntMiner* successfully detected this bug that has been confirmed by kernel developers (BugzillaID: 98561). Again, compared to the results of *Coverity*, none of these 22 real bugs were reported by *Coverity*. In fact, inferring this kind of rules is not supported by *Coverity*.

Note that, 5 of the 6 confirmed bugs were introduced before kernel 2.6.34. In particular, the bug 98561 was introduced in the kernel 2.6.14 (released in October 2005), it has been latent for almost 10 years until it is detected it by *AntMiner*.

4.2.3 *Comparative Analysis*

To highlight the effectiveness of *AntMiner* in reducing false negatives and false positives, we conducted another experiment to directly mine rules for bug-prone functions from the original source repository. We refer to this experiment as *AntMiner--*. As suggested by its name, *AntMiner--* is based on *AntMiner* but without program slicing and statement normalizing.

This experiment ran about 264 minutes and 458,905 programming rules were mined. To evaluate *AntMiner--*, we manually inspected the reported violations, and found that 22 of the 30 real bugs were not reported. That is, 73% bugs were missed. For example, the bug (ID: 44621) shown in Figure 1 is detected by *AntMiner* but missed by *AntMiner--* (where the reason was explained in §2.1). Note that, theoretically, *AntMiner* may fail to report some bugs detected by *AntMiner--*; however, we have not found such an instance in the experiment. That is, all (confirmed) real bugs detected by *AntMiner--* were detected by *AntMiner*.

To evaluate the ability of *AntMiner* on reducing false positives, we collected and further analyzed the mined rules for the 21 bug-prone functions listed in Table 2 and Table 3 respectively. As shown in Table 6, *AntMiner--* mined 2,159 rules related to these bug-prone functions. For each function, its top ranked rules (at most 10) were manually verified to see whether they are correct. A rule is correct if it should be followed, and if violated, bugs may occur. For example, *AntMiner--* mined 30 rules related to the

```
linux-2.6.39/drivers/net/hamradio/mkiss.c:
728 static int mkiss_open(struct tty_struct *tty)
729 {
            ......
762         if ((err = ax_open(ax->dev))) {
763             goto out_free_netdev;
764         }
765
766         if (register_netdev(dev))
                // forgot to set err to the return of register_netdev() !
767             goto out_free_buffers;
            ......
799         return 0;
800
801 out_free_buffers:
802         kfree(ax->rbuff);
803         kfree(ax->xbuff);
804
805 out_free_netdev:
806         free_netdev(dev);
807
808 out:
809         return err;
810 }
```

Figure 10. An example that returns improper values.

bug-prone function *nla_nest_end*(). We inspected the top 10 of these 30 rules manually, and found only one of them was a correct rule. In total, 149 rules were inspected, and only 18 of them were confirmed to be correct ones. The false positive rate is up to 87.9%. In most cases, we found that elements of incorrect rules are irrelevant or weakly relevant to each other. Violations to such rules are always false positives. Similar analysis was performed on the result of the experiment in §4.2.2. From Table 6, it can be seen that the false positives were greatly reduced by applying our method, i.e., 24.5%. At the same time, more correct rules were extracted by *AntMiner*(i.e., 80 vs. 18 by *AntMiner--*).

Table 6. Statistics of extracted rules related to the bug-prone functions listed in Table 1 and 2.

Approach	Related Rules	Analyzed Rules	Correct Rules	False Positive Rate
AntMiner	200	106	80	24.5%
AntMiner--	2,159	149	18	87.9%

4.3 Summary

From the first two experiments, *AntMiner* successfully finds 52 bugs from the Kernel 2.6.39. It is well demonstrated that *AntMiner* can detect a number of subtle bugs that are difficult to be found by other detection tools (e.g., *Coverity*). The third experiment further illustrates that introducing program slicing and statement normalizing is significant for reducing both false negatives and false positives.

5. DISCUSSION

While *AntMiner* is effective in revealing bugs that may be missed previously, there are still some limitations that we need to consider in our future works.

Critical Operations. Currently, *AntMiner* mainly concerns two types of critical operations. However, other operations may also be critical to detecting bug, such as reading or overwriting some fields of a specific type structure. How to cover such operations is an important problem that we need to address in the future. Intuitively, a direct solution is to transform such operations into a special type of function call. To this end, it may be helpful to introduce a little prior knowledge to identify which (types of) operations are critical ones, as done in [17, 36].

Data Mining Algorithms. In this study, we adopt the frequent itemset mining algorithm to extract programming rules considering its scalability. For some types of programming patterns, others mining algorithms may be more suitable. Programming logics can be represented in forms of sequences [15, 41, 42], or even graphs [11, 23, 48, 49]. Note that our approach is compatible with other mining algorithms. Applying them on a refined code database will produce better results. This will be one of our future works.

Normalization. In theory, even for a simple expression, completely recognizing all semantics-equivalent forms of it is not a trivial task. In this study, *AntMiner* can handle some most common semantics-equivalent representations. In fact, more bugs can be found if more semantics-equivalent representations are covered. In the future, we plan to employ deeper semantics analysis [40] to normalize complicated semantics-equivalent representations that cannot be handled in the current version of *AntMiner*.

Concurrency Bugs. Detecting concurrency bugs in multithreaded programs is both significant and challenging [9, 10, 28]. There are two main obstacles to finding concurrency bugs statically. First, it is difficult to statically determine concurrent codes. Second, prior knowledge about locks are not always available. We will try to address above issues from the perspective of code mining.

6. RELATED WORK

Engler et al. [15] proposed a method to detect programming bugs by employing statistical analysis to infer temporal rules from rule templates such as "<*a*> must be paired with <*b*>". They have developed six checkers and detected hundreds of bugs in real systems. The study proposes a promising direction to detect bugs without specifying concrete rules. Kremenek et al. [23] proposed a more general method that uses factor graphs to infer specification from programs by incorporating disparate sources of information. While these two approaches are inspiring, the types of inferred rules are restricted to predetermined templates. This requires users to specify some specific knowledge about the target.

Data mining techniques are introduced to extract more general rules from real large systems [5, 11, 24-28, 30, 34, 36, 38, 41, 42, 48, 49]. All mining based methods along with those statistical-based methods [15, 23] accept the reasonable assumption: in a practical system, the coding is correct in most cases, and a small number of anomalies are likely to be bugs. These methods firstly infer frequently appeared patterns in the source code, such patterns specify the (implicit) programming rules that should be followed in coding. Then, programs that violate these rules are detected and regarded as potential bugs.

Code mining methods can be categorized into four groups. (1) *Frequent sub-itemset* based methods represent a rule as an itemset [25, 28, 38], indicating that when a program executes some operations (e.g. call one or more functions), it should simultaneously execute the other operations in the same itemset (e.g. verify the function parameters). (2) *Frequent sub-sequence* based methods represent a rule as a sequence of events [5, 27, 28, 31, 41, 42], indicating that these events should be executed in order. (3) *Frequent sub-graph* based methods represent a rule as a graph [11, 48, 49], indicating that the control flow and data flow should also be correctly implemented. And (4) *template-based* methods [30, 34] adapt the mined rules to templates provided by traditional static analysis tools (e.g. *Klocwork* [4]), and then utilize these tools to detect bugs. Among various mining methods, the frequent itemset mining is most practical due to its scalability. Currently, *AntMiner* mines frequent sub-itemset as programming rules, but it can be easily extended to mine rules represented in sequences or graphs.

In theory, mining based methods can detect many types of bugs. However, in practical, two types of bugs are often detected: (1) one or more necessary function calls [5, 25, 27, 30, 41, 42] are missed, and (2) some prerequisite conditions are neglected [11, 31, 36, 38, 48, 49]. *AntMiner* can detect both of them. In the studies of Gunawi et al. [19] and Rubio-González et al. [32], they found that error codes are often incorrectly propagated in file systems, and such bugs are very hard to detect both statically and dynamically. *AntMiner* provides an effective way to detect this kind of bugs by mining the function return patterns.

If some domain knowledge can be introduced into mining rules, better results may be produced. Some approaches have been specially designed to infer rules for critical APIs [5, 30, 38, 41, 42] or security-sensitive functions [36, 49], and have gained great results. *AntMiner* also mines rules for specific operations. However, it does not require users to specify the interested operations, which are automatically extracted from the source code.

It is noticed that code mining can be applied to not only the source code but also other forms of software engineering data. Rules can be mined from revision histories [26], execution paths [27], program comments [35, 37], or even documentations written in natural language [44, 50]. The *natural language processing* (NLP) technique is employed when extracting rules from comments and documentations. NLP is also helpful for methods mining rules from source code [15, 49]. We will leverage NLP to discover the semantic information behind the names of program elements (e.g. variables, functions). The information can be used to further improve the statements normalization.

Program slicing was originally proposed by Weiser [43], and Chen and Cheung [12] extended it to make the slicing process effective in some circumstances, known as dynamic program slicing. Program slicing is mainly used to help debugging or simplifying testing [6, 21]. Agrawal et al. [6] applied program slicing to locate known faults, while we employ program slicing to help mining unknown bug in this study.

7. CONCLUSION

Many efforts have been paid to use various code mining methods to extract programming rules and detect bugs. However, less attention has been given to exclude the noise items from the code database. This paper presents a novel approach *AntMiner* to improve the precision of code mining. It reduces noises in code database by (1) excluding statements that are irrelevant to certain critical operations and (2) transforming statements with the same logic into a same canonical representation form. We have implemented *AntMiner* and applied it to some large-scale systems. The evaluation results show that *AntMiner* effectively improved the precision of code mining and detected a number of subtle bugs that have been missed previously.

8. ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their insightful comments. The work is supported by National Natural Science Foundation of China (NSFC) under grants 61170240, 91418206, 61472429, and 61502465, National 973 program of China under grant 2014CB340702, and National Science and Technology Major Project of China under grant 2012ZX01039-004.

9. REFERENCES

[1] Bugzilla for kernel, https://bugzilla.kernel.org.

[2] Common Weakness Enumeration, http://cwe.mitre.org.

[3] Coverity, http://www.coverity.com, v7.5.

[4] Klocwork, www.klocwork.com.

[5] M. Acharya, T. Xie, J. Pei, and J. Xu. Mining API patterns as partial orders from source code: from usage scenarios to specifications. In *Proceedings of the 11th European Software Engineering Conference Held Jointly with 15th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 163-173, 2007.

[6] H. Agrawal, J. R. Horgan, S. London, and W. E. Wong. Fault localization using execution slices and dataflow tests. In *Proceedings of International Symposium on Software Reliability Engineering*, pages 143-151, 1995.

[7] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers*: *principles*, *techniques*, *and tools*. 1986.

[8] G. Ammons, R. Bodik, J. R. Larus. Mining specifications. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 4-16, 2002.

[9] Y. Cai, and L. Cao. Effective and precise dynamic detection of hidden races for Java programs. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, pages 450-461, 2015.

[10] Y. Cai, and W.K. Chan. Magiclock: Scalable Detection of Potential Deadlocks in Large-Scale Multithreaded Programs [J]. *IEEE Transactions on Software Engineering*, 40(3), pages 266-281, 2014.

[11] R-Y. Chang, A. Podgurski, and j. Yang. Finding what's not there: a new approach to revealing neglected conditions in software. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, pages 163-173, 2007.

[12] T. Y. Chen and Y. Y. Cheung. Dynamic program dicing. In *Proceedings of the Conference on Software Maintenance*, pages 378-385, 1993.

[13] B. Chess and G. McGraw. Static analysis for security [J]. *IEEE Security & Privacy*, 2(6), pages 76-79, 2004.

[14] S. Christey. 2011 CWE/SANS Top 25 Most Dangerous Software Errors. http://cwe.mitre.org/top25.

[15] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *Proceedings of the 18th ACM Symposium on Operating System Principles*, pages 57-72, 2001.

[16] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. In *Proceedings of ACM Transactions on Programming Languages and Systems*, vol. 9, pages 319-349, 1987.

[17] V. Ganapathy, D. King, T. Jaeger, and S. Jha. Mining security-sensitive operations in legacy code using concept analysis. In *Proceedings of the 29th international conference on Software Engineering*, pages 458-467, 2007.

[18] G. Grahne and J. Zhu. Efficiently using prefix-trees in mining frequent itemsets. In *Proceedings of Workshop on Frequent Itemset Mining Implementations*, 2003.

[19] H. S. Gunawi, C. Rubio-González, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and B. Liblit. EIO: Error handling is occasionally correct. In *6th USENIX Conference on File and Storage Technologies*, pages 1-16, 2008.

[20] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analysis. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 69-82, 2002.

[21] M. Harman, S. Danicic. Using program slicing to simplify testing [J]. *Software Testing, Verification and Reliability*, 5(3), pages 143-162, 1995.

[22] S. Horwitz, H. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of ACM Transactions on Programming Languages and Systems*, vol. 12, pages 26-60, 1990.

[23] T. Kremenek, P. Twohey, G. Back, A. Y. Ng, and D. Engler. From uncertainty to belief: inferring the specification within. In *Proceedings of 7th Symposium on Operating Systems Design and Implementation*, pages 161-176, 2006.

[24] 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 *Proceedings of 21st ACM SIGOPS symposium on Operating systems principles*, pages 145-158, 2007.

[25] Z. Li and Y. Zhou. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 306-315, 2005.

[26] B. Livshits and T. Zimmermann. DynaMine: finding common error patterns by mining software revision histories. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 296-305, 2005.

[27] D. Lo, S-C. Khoo, C. Liu. Mining past-time temporal rules from execution traces. In *Proceedings of the 2008 international workshop on dynamic analysis*, pages 50-56, 2008.

[28] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou. MUVI: Automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *Proceedings of 21st ACM SIGOPS symposium on Operating systems principles*, pages 103-116, 2007.

[29] K. J. Ottenstein, and L. M. Ottenstein. The program dependence graph in a software develop environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 177-184, 1984.

[30] M. Pradel, C. Jaspan, J. Aldrich, and T. R. Gross. Statically checking API protocol conformance with mined multi-object specifications. In *Proceedings of the 34th International Conference on Software Engineering*, pages 521-530, 2012.

[31] M. K. Ramanathan, A. Grama, and S. Jagannathan. Static specification inference using predicate mining. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 123-134, 2007.

[32] C. Rubio-González, B. Liblit. Defective error/pointer interactions in the linux kernel. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 111-121, 2011.

[33] R. M. Stallman and the GCC Developer Community. GNU compiler collection internals (for GCC version 4.5.0), http://gcc.gnu.org/onlinedocs/gcc-4.5.0/gccint.ps.gz, 2010.

[34] B. Sun, G. Shu, A. Podgurski, and B. Robinson. Extending static analysis by mining project-specific rules. In *Proceedings of the 34th International Conference on Software Engineering*, pages 1054-1063, 2012.

[35] L. Tan, D. Yuan, G. Krishna, and Y. Zhou. /* iComment: Bugs or bad comments?*/. In In Proceedings of the 21st ACM Symposium on Operating Systems Principles, pages 145-158, 2007.

[36] L. Tan, X. Zhang, X. Ma, W. Xiong, and Y. Zhou. AutoISES: automatically inferring security specifications and detecting violations. In *Proceedings of USENIX Security Symposium '08*, pages 379-394, 2008.

[37] L. Tan, Y. Zhou, and Y. Padioleau. aComment: mining annotations from comments and code to detect interrupt related concurrency bugs. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 11-20, 2011.

[38] S. Thummalapenta and T. Xie. Alattin: mining alternative patterns for detecting neglected conditions. In *Proceedings of 24th IEEE/ACM International Conference on Automated Software Engineering*, pages 283-294, 2009.

[39] W. Visser, J. Geldenhuys, and M. B. Dwyer. Green: reducing, reusing and recycling constraints in program analysis. In *Proceedings of FSE*, 2012.

[40] T. Wang, K. Wang, X. Su, and P. Ma. Detection of semantically similar code [J]. *Frontiers of Computer Science*, 8(6), pages 996-1011, 2014.

[41] A. Wasylkowski and A. Zeller. Mining temporal specifications from object usage. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*, pages 263-292, 2009.

[42] W. Weimer and G. C. Necula. Mining temporal specifications for error detection. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 461-476, 2005.

[43] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439-449, 1981.

[44] X. Xiao, A. Paradkar, S. Thummalapenta, and T. Xie. Automated extraction of security policies from natural-language software documents. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, 2012.

[45] S. Xu, Y.S. Chee, Transformation-based diagnosis of student programs for programming tutoring systems, In *IEEE Trans. Software Eng. 29 (4)*, pages 360–384, 2003.

[46] Z. Xu, J. Zhang, and Z. Xu. Melton: a practical and precise memory leak detection tool for C programs [J]. *Frontiers of Computer Science*, 9(1), pages 34-54, 2015.

[47] F. Yamaguchi, N. Golde, D. Arp, K. Rieck. Modeling and discovering vulnerabilities with code property graphs. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 590-604, 2014.

[48] F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck. Automatic inference of search patterns for taint-style vulnerabilities. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 797-812, 2015.

[49] F. Yamaguchi, C. Wressnegger, H. Gascon, and K. Rieck. Chucky: Exposing missing checks in source code for vulnerability discovery. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 499-510, 2013.

[50] H. Zhong, L. Zhang, and T. Xie, M. Hong. Inferring resource specifications from natural language API documentation. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 307-318, 2009.