NAR-Miner: Discovering Negative Association Rules from Code for Bug Detection

Pan Bian, Bin Liang, Wenchang Shi, Jianjun Huang
{bianpan,liangb,wenchang,hjj}@ruc.edu.cn
School of Information, Renmin University of China
Key Laboratory of DEKE, Renmin University of China
Beijing, China

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

ABSTRACT

Inferring programming rules from source code based on data mining techniques has been proven to be effective to detect software bugs. Existing studies focus on discovering positive rules in the form of $A \Rightarrow B$, indicating that when operation A appears, operation B should also be here. Unfortunately, the negative rules ($A \Rightarrow \neg B$), indicating the mutual suppression or conflict relationships among program elements, have not gotten the attention they deserve. In fact, violating such negative rules can also result in serious bugs.

In this paper, we propose a novel method called NAR-Miner to automatically extract negative association programming rules from large-scale systems, and detect their violations to find bugs. However, mining negative rules faces a more serious rule explosion problem than mining positive ones. Most of the obtained negative rules are uninteresting and can lead to unacceptable false alarms. To address the issue, we design a semantics-constrained mining algorithm to focus rule mining on the elements with strong semantic relationships. Furthermore, we introduce information entropy to rank candidate negative rules and highlight the interesting ones. Consequently, we effectively mitigate the rule explosion problem. We implement NAR-Miner and apply it to a Linux kernel (v4.12-rc6). The experiments show that the uninteresting rules are dramatically reduced and 17 detected violations have been confirmed as real bugs and patched by kernel community. We also apply NAR-Miner to PostgreSQL, OpenSSL and FFmpeg and discover six real bugs.

CCS CONCEPTS

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

KEYWORDS

Bug Detection, Code Mining, Negative Rule, Rule Explosion

ACM Reference Format:
Pan Bian, Bin Liang, Wenchang Shi, Jianjun Huang, and Yan Cai. 2018. NAR-Miner: Discovering Negative Association Rules from Code for Bug Detection. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3236024.3236032

*Bin Liang is the corresponding author

ESEC/FSE '18, November 4–9, 2018, Lake Buena Vista, FL, USA

ACM ISBN 978-1-4503-5573-5/18/11…$15.00
https://doi.org/10.1145/3236024.3236032

1 INTRODUCTION

Static bug/vulnerability detection techniques usually require some prior knowledge (i.e., detection rules or vulnerability signatures) [9, 13, 17, 19]. In recent years, it has been widely proven that code mining approaches on bug/vulnerability detection are very effective [1, 4, 6, 7, 14, 20, 25–27, 29, 30, 36, 38, 44, 47, 49, 51–53, 57, 60]. Such approaches automatically extract implicit programming rules from program source code and further detect violations of these rules as bugs or vulnerabilities. Generally, during code mining, the program source code is first transformed into itemsets [6, 25, 26, 49], graphs [7, 34, 57], or other forms. Next, data mining algorithms are applied on the transformed forms to extract patterns (e.g., frequent itemsets or sub-graphs) and infer programming rules. The last step is to detect violations against the inferred rules. For example, PR-Miner [25] and AntMiner [26] mine frequent itemsets from the Linux kernel to extract association rules (as detection rules) and have detected dozens of unknown bugs.

The basic idea of these existing studies is to utilize statistics to find program elements with accompanying relationship from source code. This relationship exhibits as a ***positive*** programming pattern. That is, in the target project, some program elements appear together frequently (up to a given threshold) or there is a certain connection among them. For example, PR-Miner [25] and AntMiner [26] both extract the positive association rules in the form of $A \Rightarrow B$, indicating that, within a function, when the program element A appears, the element B should also appear. Accordingly, if a function implementation violates the rule (i.e., containing A without B), a potential bug is expected. Similar techniques have been proposed to detect potential object misuse bugs [53] and missing program elements following control structures [7], and to infer the correct usage of the APIs [60]. All these approaches target on a group of ad-joint program elements that have a mutually supportive relationship, i.e., positive associations.

However, we observed that, in practice, some implicit programming patterns appear as ***negative*** associations in the form of $A \Rightarrow \neg B$. That is, when A appears, B should not appear, and vice versa. In this sense, a negative rule reflects the mutually suppressing or conflicting relationship between A and B. Violating negative rules may also result in serious bugs. It is usually impossible to manually identify all negative rules from a project, especially a large-scale

one like the Linux kernel. But to the best of our knowledge, no existing approaches can automatically extract negative rules from source code for bug detection. The state-of-the-art mining-based solutions only extract positive programming rules. Compared with the positive rules that indicate frequent patterns, a negative rule is typically more implicit and the corresponding bugs are more insidious. For example, the bug in Figure 1 has existed in Linux kernel for more than 10 years (i.e., presented in Linux-2.6.4 or earlier). Hence, it is both challenging and urgent to develop an effective approach to automatically extracting implicit negative programming rules to detect related bugs and vulnerabilities.

In this paper, we propose `NAR-Miner` to address the above issue. The overarching idea is to infer interesting negative association rules through infrequent pattern mining and to further detect the corresponding violations as potential program bugs. Essentially, due to the nature of negative rules [56], directly mining infrequent patterns to extract negative rules will produce a huge number of rules. We call this the ***rule explosion problem***. Most of these rules are *uninteresting* for bug detection, i.e., they do not embody any real application logic and violations of them do not lead to bugs or program quality issues. Hence, the detection results based on these uninteresting rules will produce unacceptable false positives. For example, directly applying an existing negative rule mining algorithm [43, 56, 61] to Linux kernel will extract up to hundreds of thousands of rules and millions of violations. That became impossible for manual audit under limited human resource. To address the rule explosion problem, we propose a semantics-constrained negative association programming rule mining algorithm to avoid generating excessive uninteresting rules as far as possible. Moreover, we utilize the information entropy to identify the general functions that are prone to result in uninteresting rules. This step helps to further filter potentially uninteresting rules. As a result, `NAR-Miner` can effectively mitigate the rule explosion problem and gain desirable interesting rules to detect potential bugs.

We implement a prototype of `NAR-Miner` and first evaluate it on a Linux kernel (v4.12-rc6). The experiments show that, semantics-constrained negative rule mining and information entropy based rule filtering perform well on reducing the number of uninteresting rules. That is, it reduces 46% uninteresting rules (i.e., from 198 to 107 among the 200 top ranked negative rules). Especially, it achieves a true positive rate of 62% among the top ranked 50 negative rules. `NAR-Miner` reports 356 violations of the top ranked 200 rules. We manually inspect the results and find 23 suspicious bugs and dozens of quality problems. We report the suspicious bugs to Linux kernel maintainers. 17 of them have been confirmed as real bugs and the corresponding patches have been merged into the latest version (e.g., v4.16). We further apply `NAR-Miner` to PostgreSQL v10.3, OpenSSL v1.1.1, and FFmpeg v3.4.2. From the top ranked rules and violations, we manually identify six suspicious bugs, all of which have been confirmed and fixed by the corresponding maintainers.

This paper makes the following main contributions:

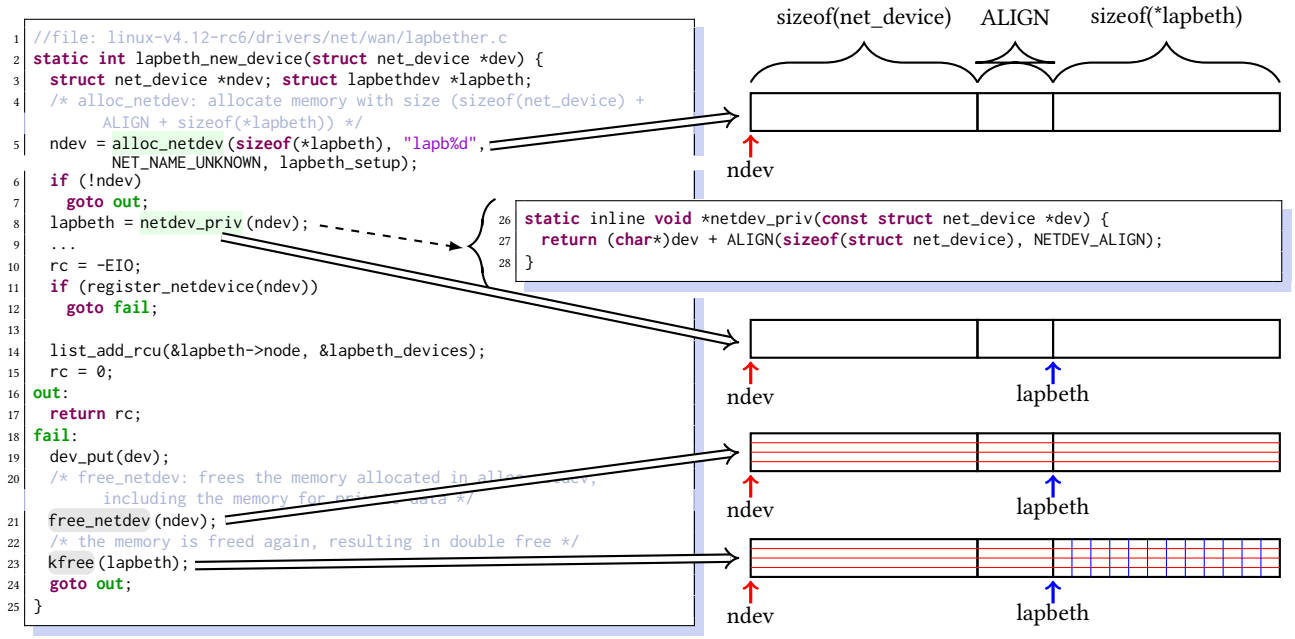
- To the best of our knowledge, our work is the first one that focuses on extracting negative programming rules from source code to detect bugs. It extends the capability of the mining-based bug detection technique.
- We propose an approach to mitigating the rule explosion problem by introducing program semantics in rule mining and using information entropy to identify the general functions, which can effectively extract desirable interesting negative programming rules for bug detection.
- We implement a prototype of `NAR-Miner` targeting on real-world large-scale software projects. We apply the tool to four large-scale systems (i.e., the Linux kernel, PostgreSQL, OpenSSL and FFmpeg) and find a considerable number of bugs, among which 23 have been confirmed.

2 MOTIVATING EXAMPLE

We use a simplified code snippet from Linux kernel (v4.12-rc6) to motivate our method. In Figure 1, the function *lapbeth_new_device* calls *alloc_netdev* at line 5 to allocate a chunk of memory for a network device. Then at line 8, in inlined function *netdev_priv* is invoked to get the starting address of the private data and store it to variable *lapbeth*. As shown in Figure 1, *lapbeth* points to a location inside the previously allocated memory. If the device registration fails at line 11, the memory chunk for the device will be released. The allocated memory pointed by variable *ndev* is first freed at line 21 and then the private data is freed at line 23. The critical operations in the execution sequence from memory allocation to free are highlighted in the code snippet and the corresponding memory states are depicted accordingly. We use red horizontal lines to describe freeing memory via *free_netdev* and blue vertical lines for *kfree*. With the illustration, it is easy to tell there is a *double free* bug in the code.

Traditional static detection methods are difficult to discover this bug and other similar bugs in a large-scale system (e.g., Linux kernel), as the required bug patterns or rules are application-specific and hard to collect. Existing mining based approaches cannot report the bug either. For example, the state-of-the-art approach `AntMiner` [26] extracts the positive association rules and checks the violations. From the Linux kernel, we discover 77 appearances of the pair {*alloc_netdev*, *free_netdev*} among 90 invocations of *alloc_netdev* (85.6%). And only one invocation of *free_netdev* is followed by *kfree* among its 533 call instances (0.2%). `AntMiner` treats $\{alloc\_netdev\} \Rightarrow \{free\_netdev\}$ as a positive rule with a minimum confidence threshold as 85% [26]. However, the code in Figure 1 calls both *alloc_netdev* and *free_netdev*, and thus is a support, instead of a violation, of the rule. Hence, the aforementioned bug is undiscovered. Besides, due to the low confidence ($0.2\% \ll 85\%$), `AntMiner` does not treat $\{free\_netdev\} \Rightarrow \{kfree\}$ as a valid rule (i.e., filtered out by the minimum confidence threshold). As a result, `AntMiner` cannot report "*kfree* follows *free_netdev*" as a bug.

From the above analysis, it is quite difficult or even impossible to detect the bug in Figure 1 through analyzing the ad-joint relationships among elements. In essence, the two program elements (i.e., *free_netdev* and *kfree*) closely related to the bug are negatively correlated. This knowledge can be discovered by a statistic analysis. Specifically, we find that in most cases (about 99.8%) within the Linux kernel, *free_netdev* is not followed by *kfree*, we learn that the developers are mostly aware of the case in which calling *kfree* after *free_netdev* may be unnecessary or cause serious problems.




```
1  //file: linux-v4.12-rc6/drivers/net/wan/lapbether.c
2  static int lapbeth_new_device(struct net_device *dev) {
3    struct net_device *ndev; struct lapbethdev *lapbeth;
4    /* alloc_netdev: allocate memory with size (sizeof(net_device) +
          ALIGN + sizeof(*lapbeth)) */
5    ndev = alloc_netdev(sizeof(*lapbeth), "lapb%d",
           NET_NAME_UNKNOWN, lapbeth_setup);
6    if (!ndev)
7      goto out;
8    lapbeth = netdev_priv(ndev);
9    ...
10   rc = -EIO;
11   if (register_netdevice(ndev))
12     goto fail;
13
14   list_add_rcu(&lapbeth->node, &lapbeth_devices);
15   rc = 0;
16 out:
17   return rc;
18 fail:
19   dev_put(dev);
20   /* free_netdev: frees the memory allocated in alloc_netdev,
          including the memory for priv data */
21   free_netdev(ndev);
22   /* the memory is freed again, resulting in double free */
23   kfree(lapbeth);
24   goto out;
25 }
```

```
26 static inline void *netdev_priv(const struct net_device *dev) {
27   return (char*)dev + ALIGN(sizeof(struct net_device), NETDEV_ALIGN);
28 }
```

Figure 1: A bug in Linux kernel, violating the rule: $\{free\_netdev\} \Rightarrow \neg\{kfree\}$.

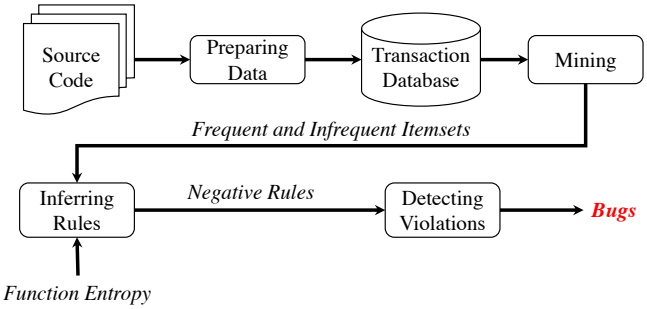


Figure 2: A high-level overview of NAR-Miner.

Inspired by that, relatively small amount of paired occurrences can be considered as anomalies.

We leverage infrequent itemset mining algorithms to infer the negative association rules and apply the rules to detect their violations (e.g., the one in Figure 1). During the mining and detection, we also take the program semantics (e.g., data flow information) into consideration. In our example, *kfree* shares data with *free_netdev* and their appearance together is treated as an infrequent pattern. Therefore, we infer a negative rule $\{free\_netdev\} \Rightarrow \neg\{kfree\}$. Applying the rule to our example discovers the corresponding bug (bug 12# in Table 2)

3 OUR APPROACH

3.1 Overview

We present NAR-Miner, targeting on detecting bugs where a program contains some operations (e.g. two function calls) that are deemed not to appear together without requiring any prior knowledge. The high-level idea of NAR-Miner is to *employ the data mining technique to infer negative association rules from source code and detect their violations.*

Figure 2 presents the overview of NAR-Miner. It first prepares data for the mining phase. Similar to most of mining-based methods [6, 25, 26, 49, 60], we only extract programming rules from each individual function, i.e., intra-procedural, to avoid overcomplicated analysis. Program elements as well as their semantic relationships within each function are identified and transformed into a transaction, which is then stored in a database (called transaction database). Next, it mines frequent and infrequent itemsets, which denote sets of program elements, from the database. Then it infers negative association rules from the mined itemsets and leverages the confidence and the entropy of functions to rank the rules automatically. Finally, it detects the violations of the inferred rules and reports the top ranked ones as potential bugs for auditing.

3.2 Challenge

Previous studies [43, 56, 61] have shown that only a small number of patterns that exhibit negative association relationship are interesting. Their techniques of identifying interesting rules via computing their support and confidence cannot be directly adopted to code mining. The program elements constructing an interesting negative association rules should suppress each other in their semantics, not just infrequently appearing together. Our empirical study shows that directly applying the algorithm proposed by Wu et al. [56] on the transaction database extracted from the Linux kernel generates 183,712 negative association rules (see §4.2.2) while 99% of them are uninteresting in a sampling analysis. Up to 309,689 violations are reported, which makes human auditing impossible. We call this the **rule explosion problem** and recognize it as the main challenge of extracting negative association rules. We attribute the following two aspects as the root cause of rule explosion.

(1) Existing negative association rule mining approaches [43, 45, 56, 61] mainly target on *market basket*, *medical diagnosis*, *protein sequences*, etc.. For such kinds of data, any two elements from a mining unit (e.g., a shopping receipt) do not have any particular relationship except belonging to the same unit. In other words, the elements of a mining unit are homogeneous. However, between program elements, there are often various semantic relationships, e.g., data dependence. Ignoring such relationships may result in many uninteresting rules consisting of semantically independent elements, which actually do not suppress each other but just infrequently appear together.
(2) A large scale project usually contains a certain number of general-purpose APIs that can be used in almost all programming contexts, such as *printk* and *isalpha* in Linux. They can be coincidentally paired with other operations to form association rules. Even considering strong semantic relationships (e.g., data dependence) during mining, these APIs can still result in many negative uninteresting rules. Actually, the more general an API is , the less mutual suppression it has with other operations.

Based on above insights, we mitigate the rule explosion problem as follows. (1) Considering the essence of program elements as discussed above, we focus rule mining on the program elements that have strong semantic relationships (e.g., data dependence) to reduce uninteresting rules as far as possible. (2) We employ the information entropy to measure the generality of APIs and use it to rank mined candidate negative rules. The uninteresting rules involving high-generality APIs will be excluded from the final audit.

3.3 Preparing Data

`NAR-Miner` transforms the program elements into transactions and store them to a database. In this paper, we focus on two kinds of program elements: function calls and condition checks, because many bugs result from function or condition misuses [1, 6, 21, 26, 33, 38, 47, 57]. As discussed earlier, we aim to extract negative association rules from transactions, whose elements have a strong semantic relationship. We define two elements to have a **strong semantic relationship** if there is a data correlation between them, including data dependence and data share [7]. In details, given two statements s_1 and s_2, if either s_2 uses a value defined in s_1 (i.e., data dependence) or they both appear on the same execution path and use the same non-constant value (i.e., data share), we say they are strongly semantically related. The semantic relationships of program elements is identified via data flow analysis [3, 15]. `NAR-Miner` associates the semantic relationships with program elements and stores them into the database for mining.

The preprocessor of `NAR-Miner` is built on top of the GCC (v4.8.2) frontend, which provides control flow graphs and intermediate representations in SSA form [12] for data flow analysis. Figure 3(a) presents a piece of code and Figure 3(b) shows the corresponding intermediate representation in SSA form. `NAR-Miner` can tell that *is_valid*, *foo* and *bar* are data dependent on *read2* at line 2. As *is_valid* and *foo* are within the same execution path, they have a data share relationship. Because there is no path between *foo* and *bar*, they are not considered to be semantically related.

To ease the mining phase, the intermediate representation is then converted into a transaction database. Every function definition is mapped to a transaction. A transaction consists of two parts: a bag of program elements and a set of semantic relations among these elements. Each program element is normalized before dumped to the database. A function call is represented with the function name without the arguments; and variables in a conditional statement are renamed with "RET" if they keep return values of some functions or with their data types in other cases as done in many mining methods [6, 7, 25, 26, 57, 58]. For example, the condition expression in Figure 3 is normalized to "$RET == 0$". The semantic relationship between two program elements is represented as a tuple in the transaction. For example, the tuple (*foo*, *is_valid*) says that functions *foo* and *is_valid* have some semantic relation. Figure 3(c) presents the semantic relationships for the code snippet, where a node denotes a program element and an edge indicates the relationship ($\longrightarrow$ for data dependence and $\leftarrow\!\!-\!\!\rightarrow$ for data share).

We map each program element to a unique integer and thus mining is applied on the integer set to improve performance, as a large number of string equivalence comparison is time-consuming.

3.4 Extracting Frequent & Infrequent Itemsets

A rarely invoked program element always infrequently appears together with other program elements and will result in a large number of negative patterns. However, such patterns are meaningless in statistics [56]. Hence, we focus on mining negative patterns whose elements are frequent alone but infrequent together. For a negative rule $A \Rightarrow \neg B$, its *antecedent* (i.e., A) and *consequent* (i.e., B) are frequent, but the combination of them (i.e., $(A \cup B)$) is infrequent. In this section, we present our algorithm on extracting interesting frequent and infrequent itemsets, and will explain how to generate negative rules in the next section.

Existing algorithms extracting frequent and infrequent itemsets only rely on the occurrences of itemsets in the transaction database [43, 56, 61]. None of them considers the semantic relations among elements. Directly applying them on the database generated in §3.3 will result in a large number of uninteresting rules. As far as we know, there is no infrequent itemset mining algorithm that can be directly applied in our work. To address this issue, we design a semantics-constrained mining algorithm, which focuses on extracting strong semantics-related itemsets. Elements in a strong semantics-related itemset are all related to each other in semantics, e.g, have data dependence or data share relationships.

We design our algorithm based on the well-known Apriori algorithm [2], which applies a bottom-up approach to generate relatively large candidate itemsets by joining smaller frequent ones together. The principle behind the bottom-up approach is the Apriori property: any subset of a frequent itemset is also frequent. In our cases, any subset of a strong semantics-related itemset is also strong semantics-related, because any two elements in the subset must be related in semantics. Hence, strong semantics-related itemset also complies with the Apriori property and can be mined in a bottom-up manner.

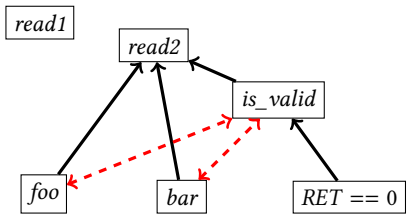
Our algorithm of mining frequent and infrequent strong semantics-related itemset is shown in Algorithm 1. Besides the transaction database, it requires users to specify two parameters: the minimum frequency support mfs and the maximum infrequency support mis. An itemset is considered to be frequent if its support is larger than

```
1.   str = read1();
2.   str = read2();
3.   valid = is_valid(str);
4.   if (valid) {
5.       foo(str);
6.   } else {
7.       bar(str);
8.   }
```

(a) Source Code.

```
1.   str_1 = read1();
2.   str_2 = read2();
3.   valid = is_valid(str_2);
4.   if (valid)) {
5.       foo(str_2);
6.   } else if {
7.       bar(str_2);
8.   }
```

(b) Intermediate representation in SSA.



(c) Transaction.

Figure 3: Source code transformation.

Algorithm 1 Extract Frequent and Infrequent Itemsets

```
1:  procedure mine_itemsets(DB, mfs, mis)
2:      FIs = ∅, IIs = ∅;
3:      L_1 = {frequent 1−itemsets};
4:      for (k = 2;  L_{k−1} ≠ ∅;  k++) do
5:          S_k = L_{k−1} ⋈ L_{k−1};
6:          S_k = prune(S_k);
7:          for (each itemset I in S_k) do
8:              support(I) = count_support(DB, I);
9:              if (support(I) ≥ mfs) then
10:                 L_k = L_k ∪ {I};
11:             else if (0 < support(I) ≤ mis) then
12:                 N_k = N_k ∪ {I};
13:             end if
14:         end for
15:         FIs = FIs ∪ L_k,  IIs = IIs ∪ N_k;
16:     end for
17:     output FIs and IIs;
18: end procedure
19: procedure count_support(DB, I)
20:     counter = 0;
21:     for (each transaction t in DB) do
22:         if (I ⊆ t.elements ∧ relations(I) ⊆ t.relations) then
23:             counter = counter + 1;
24:         end if
25:     end for
26:     return counter;
27: end procedure
```

Algorithm 2 Generate Negative Association Rules

```
1:  procedure generate_rules(FIs, IIs, min_conf)
2:      for (each itemset I in IIs) do
3:          A = min_support_subset(I);
4:          B = I − A;
5:          R = A ⇒ ¬B;
6:          confidence(R) = 1 − support(I)/support(A);
7:          if (confidence(R) < min_conf) then
8:              continue;
9:          end if
10:         interestingness(R) = confidence(R) / entropy(I);
11:         NARs = NARs ∪ R;
12:     end for
13:     sort NARs by their interestingness;
14:     return NARs;
15: end procedure
```

or equal to mfs, and an itemset is infrequent if its support is less than or equal to mis. The output of the algorithm is all frequent itemsets (FIs) and infrequent itemsets (IIs) of interest.

At the beginning, the algorithm scans the transaction database to find all frequent 1-itemsets (line 3). Then it attempts to discover frequent and infrequent k-itemsets from frequent $(k-1)$-itemsets (lines 4 ~ 14). A k-itemset contains k items. First, it generates candidate k-itemsets of interest by joining frequent $(k-1)$-itemsets (see line 5). Two $(k-1)$-itemsets are joinable if they have k-2 common items. Suppose that two joinable (k-1)-itemsets are $\{i_1, ..., i_{k-2}, i_{k-1}\}$ and $\{i_1, ..., i_{k-2}, i_k\}$, the join result is a k-itemset $\{i_1, ..., i_{k-2}, i_{k-1}, i_k\}$. Second, the algorithm leverages the Apriori property to prune k-itemsets that have infrequent sub-itemsets (line 6). After that, function *count_support* is called to compute the support of every k-itemset (line 8). The itemset is inserted into L_k (line 10), a set of frequent k-itemsets, if its support is not less than mfs; otherwise, if its support is not larger than mis, it is inserted into the set of infrequent k-itemsets N_k (line 12). It should be noted that, itemsets whose supports are 0 are naturally ignored. L_k and N_k are finally merged to FIs and IIs (line 15), respectively. The frequent itemset L_k is then used to generate larger itemsets L_{k+1} and N_{k+1}. The algorithm terminates when L_k is empty for certain k, and outputs the collected frequent/infrequent itemsets (line 17).

The function *count_support* scans the database to compute the support of an itemset I (lines 19 ~ 27). The counter will increase by 1 if a transaction supports I (line 23). A transaction supports I if and only if it contains all items in I as well as all possible relations among the included items (denoted as $relations(I)$). For example, the transaction in Figure 3 supports the itemset {*foo*, *is_valid*}, because it includes not only both the two items *foo* and *is_valid*, but also the semantic relation between them, i.e., the tuple (*foo*, *is_valid*). However, the transaction is not a support of itemset {*foo*, *bar*} because it does not contains the tuple (*foo*, *bar*).

In order to mine rules like $\{kfree\} \Rightarrow \neg\{kfree\}$, we also extract 2-itemsets like $\{g, g\}$, where g is frequently called but two call instances of it in the same function are rarely related in semantics. This helps NAR-Miner find the bug 14# in Table 2 (see §4.2.3).

3.5 Generating Negative Association Rules

A negative association rule $A \Rightarrow \neg B$ implies that two frequent itemsets A and B rarely appear in the same transaction. That is, $(A \cup B)$ is infrequent. In fact, the antecedent and consequent of a

rule are actually a disjoint partition of an infrequent itemset. One straightforward negative association rule generation method is to find all pairs like $< A, B >$ from an infrequent itemset I, where $A \cup B = I$ and $A \cap B = \emptyset$. It uses the confidence of rule $A \Rightarrow \neg B$ to determine its infrequency. The confidence is defined of as follows:

$$confidence(A \Rightarrow \neg B) = \frac{support(A \cup \neg B)}{support(A)} \quad (1)$$

where, $support(A \cup \neg B)$ is the number of transactions that support $A \cup \neg B$, supporting A but not $A \cup B$. Consequently, we have $support(A \cup \neg B) = support(A) - support(A \cup B) = support(A) - support(I)$, where $I = A \cup B$. Hence, equation 1 can be rewritten to:

$$confidence(A \Rightarrow \neg B) = 1 - \frac{support(I)}{support(A)} \quad (2)$$

From an infrequent itemset I with n elements ($n \geq 2$), at most $2^{(n-1)}$ negative association rules can be generated by directly applying the above method. However, violations to them are all the same, i.e., the transactions that support the itemset I. Hence, it is enough to keep track of only one of them in the bug detection oriented application. Note that, from the perspective of programming, the rule $A \Rightarrow \neg B$ means that elements in B should not appear in contexts that include A. Its violations are always false positives if the reversed rule $B \Rightarrow \neg A$ is not interesting, because the existence of B is not meant to reject A. Inspired by that, in almost all cases, if we expect transactions that support the infrequent itemset I to be real bugs, all the negative association rules derived from I should be interesting. Hence, we can select the rule with the lowest confidence to represent these rules. Taking confidence as the metric, the other rules will be interesting if it is interesting.

The logic in real-world programming is often very complex. Some mined rules may not be applicable in programming practice. Violations detected according to them are usually false alarms. A general methodology is to rank the mined rules such that the interesting rules are top ranked and uninteresting rules are bottom ranked. Existing works mainly rank rules according to their confidence (high confidence rules are top ranked). However, the confidence only reflects the (negative/positive) correlations among several limited elements. In fact, the interestingness of a programming rule is also related to whether its elements are concentrated in certain contexts. That is, if the calling contexts of an element trend to be more homogeneous, the rules composed by it is more likely to be an interesting one. Otherwise, if the element is used in quite different contexts, it is more general and is more likely to appear together with various elements. In this paper, we use the **generality** to indicate how different the contexts for an element are. In general, rules consist of elements with high generalities are more likely to be uninteresting ones.

We introduce the information entropy to quantitatively measure the generality of elements. For a call instance of a program element g, we describe its context by both elements g depends on and elements that depend on g. We extract such elements in all call instances of g and put them into a bag. The information entropy of the bag of g reflects how different the call instances are and can be used to measure the generality of g. The information entropy for the bag of g (denoted as $H(g)$) can be computed by Equation 3:

$$H(g) = \frac{1}{log_{10}(N)} \sum p_i \times log_2(p_i) \quad (3)$$

where p_i is the frequency of the i-th element in the bag; N is the number of call instances of g. The entropy for an itemset is a sum of the entropy for each element.

With the generality of each program element, the interestingness of a negative association rule R can be measured as:

$$interestingness(R) = \frac{confidence(R)}{\sum H(g_i)} \quad (4)$$

where $H(g_i)$ is the information entropy of element g_i.

Our method to generate negative association rules is formalized in Algorithm 2. The algorithm takes the frequent itemset FIs and infrequent itemsets IIs extracted in §3.4 as well as a user specified threshold min_conf as inputs. It returns the set of negative association rules $NARs$. It first generates the representative rule for each infrequent itemset I that has the lowest confidence (lines 3 ~ 5). From Equation 2, the smaller the support of A is, the lower the confidence of $A \Rightarrow \neg B$ becomes. Hence, we select the subset of I whose support is smallest to generate the representative rule. Then, the confidence of the rule is computed (line 6) and checked against the threshold min_conf (line 7). The *information entropy* is employed to measure the interestingness of a potentially interesting rule (line 10). Finally, the negative association rules are sorted in the descending order of their interestingness.

3.6 Detecting Violations

The violations of a negative association rule $R : A \Rightarrow \neg B$ are those transactions that support itemset $A \cup B$. A straightforward approach is to directly scan the database to find all transactions that contain both itemsets A and B. However, such an enumerating approach is time-consuming, especially for databases with hundred of thousands of transactions. To speed up the detection process, we adopt the trick used in PR-Miner [25]. When generating frequent and infrequent itemsets, `NAR-Miner` also collects transactions that support them. We use $supporters(I)$ to indicate all transactions that support an itemset I. Then, the set of violations of the negative association rule R is exactly $supporters(A \cup B)$.

4 EVALUATION

4.1 Experiment Setup

We implement `NAR-Miner` as a prototype system to detect bugs in large-scale C programs. We evaluate `NAR-Miner` on the well known Linux kernel (v4.12-rc6). The Linux kernel has been widely used as the target of evaluation (TOE) in mining based bug detection methods [6, 14, 20, 20, 22, 24–26, 30, 42, 46–48, 57, 60]. The major reason for choosing the Linux kernel as our target is that we want to examine the effectiveness of our method by detecting some real bugs that were not found in previous work. Linux-v4.12-rc6 was the latest version at the experiment time. It contains 24,919 C files and 19,295 header files, including 376,680 functions and 15,501,651 lines of code (LoC).

To Verify whether `NAR-Miner` can be applied for bug detection in other systems, we also select three popular large-scale C systems from different domains: PostgreSQL v10.3, OpenSSL v1.1.1 and

FFmpeg v3.4.2. PostgreSQL is an open source database, OpenSSL is a library for secure communications, and FFmpeg is a framework for encoding/decoding multimedia files. Many bug detection methods select them as the targets of evaluations [19, 20, 25, 35, 57].

NAR-Miner requires to specify three parameters: (1) the minimum support threshold of frequent itemsets (i.e., mfs), (2) the maximum support threshold of infrequent itemsets (i.e., *mis*), and (3) the minimum confidence threshold of interesting negative rules (i.e., min_conf). Generally, an itemset will be more interesting if it is either a frequent itemset with a higher support or an infrequent one with a lower support. Besides, a higher minimum confidence can further filter out uninteresting negative rules. In practice, different parameter settings may result in either failing to report some real bugs or producing too many false alarms. Users can tune these parameters according to the detection strategies, conservatively or aggressively. To determine reasonable parameters, we perform an empirical study as done in [6, 25, 26, 49, 60]. Specifically, with a sampling analysis, a parameter setting is considered acceptable when more than half of the top 10 ranked negative rules are interesting ones. In this study, we set mfs to be 15, *mis* to be 5 and min_conf to be 85%. In our experiments, the default parameter setting works well against the four different TOEs (see §4.2 and §4.3).

4.2 Detecting Bugs in the Linux Kernel

4.2.1 Preprocessing the Source Code. NAR-Miner took about 77 minutes to parse the kernel source code and to transform it into a transaction database. Among all the function definitions, there are 333,248 functions contain some program elements (i.e., function calls or condition checks). After the transformation, each function definition is mapped to a transaction in the database. The database includes 227,246 different elements, where each element corresponds to a function call or a condition check. Among them, 6,203 are frequent elements that appear in more than mfs transactions.

4.2.2 Effectiveness on Mining Negative Rules. To evaluate the effectiveness of our method that incorporates both the semantics-constrained rule mining and the information entropy based rule ranking, we conduct three experiments: **NAR-Miner−−**, **NAR-Miner−**, **NAR-Miner**. The methodology adopted in each experiment is explained below:

a) **NAR-Miner−−**: The mining algorithm does not consider semantic relations among items and the mined rules are ranked according to their confidence;
b) **NAR-Miner−**: Based on NAR-Miner−−, the semantic relations among items are used as constraints to filter out weak semantics-related itemsets;
c) **NAR-Miner**: Based on NAR-Miner−, we introduce the information entropy to measure the interestingness of negative association programming rules. This experiment evaluates the full capability of NAR-Miner.

We show the experiments results in Table 1, with the number of frequent itemsets (*#FIs*), infrequent itemsets (*#IIs*), number of inferred negative association rules, number of detected violations and the time cost for mining, ranking and detection (*Time*) in seconds.

Comparing the results for NAR-Miner−− and NAR-Miner− or NAR-Miner, we observe that adopting the semantics-constrained mining reduces the total number of rules and violations in an order of magnitude (about 88% reduction for *#All* columns). The rule explosion problem is mitigated to a large extent.

Due to the limited time, we manually examine the 200 top ranked negative association rules in each experiment. The rules are ranked by their confidence in NAR-Miner−− and NAR-Miner−, whereas by their interestingness in NAR-Miner. A negative association rule is marked as "True" if it is really interesting and violating it will result in bugs or quality problems. For example, $\{free\_netdev\} \Rightarrow \neg \{kfree\}$ is an interesting ("True") rule because a violation to it will result in a potential double free bug such as the one discussed in §2. In NAR-Miner−−, only 2 among the top 200 rules are considered as interesting rules. In other words, 99% of them leads to false alarms for violation detection. The main reason for this low rate of interesting rules is that the program elements consisting of such rules are usually independent of each other in semantics. For below example, though ranked number one with a confidence of 99.96% in NAR-Miner−−, the rule $\{static\_key\_false\} \Rightarrow \neg \{atomic\_read\}$ is uninteresting, because the two functions take entirely independent variables as actual arguments in the program that contains both of them, and they do not really suppress each other.

```
1 static inline void load_mm_cr4(struct mm_struct *mm) {
2   if (static_key_false(&rdpmc_always_available) ||
3       atomic_read(&mm->context.perf_rdpmc_allowed))
```

Introducing semantics-constrained mining reduces the false positive rate to 90.5% in NAR-Miner−, which, however, is still too high to be acceptable in practice. While the inferred rules have all involved program elements semantically related, some elements are very generic and can be used in various contexts where violations do not lead to bugs. For example, function *iowrite32* is data dependent on *readl* when they appear together, only one occurrence in Linux kernel, and a rule $\{iowrite32\} \Rightarrow \neg \{readl\}$ is inferred. The rule is ranked top 8th with a confidence of 99.94% in NAR-Miner− but still uninteresting because both functions are used in various manners and a combination of them will not result in any bugs.

NAR-Miner employs the information entropy to measure generalities of functions. It is 5.9 and 4.6 for *iowrite32* and *readl*, respectively. The interestingness of the rule $\{iowrite32\} \Rightarrow \neg \{readl\}$ is 9.5%, small enough to be lowly ranked. In this way, most uninteresting rules are assigned low interestingness values and thus ranked at bottom, meanwhile potentially interesting ones are assigned with relatively high interestingness values and ranked at the top. In NAR-Miner, 93 out of the top 200 negative rules are marked with "True", nearly 5 times of the number in NAR-Miner−. In particular, there are 31 "True" negative rules among the first 50 ones. The true positive rate is 62%. That's, we can find an interesting rule within less than two manual audits, which is acceptable in practical bug detection against real-world large-scale systems such as the Linux kernel.

We also inspect the violations of the top 200 inferred rules. From the columns *Violations* and *#Bugs* in Table 1, we observe more reported violations and confirmed bugs due to the application of semantics-constrained mining and information entropy based ranking, which eventually enhances the ability of NAR-Miner, enabling it to infer much more interesting rules (columns *#True* and *TP Rate*).

4.2.3 Detecting Violations. Against the 21,166 negative association rules extracted by NAR-Miner, 37,453 violations are detected. We manually inspect the reported negative association rules and their

Table 1: Results of the three experiments.

Experiment	#FIs	#IIs	Negative Association Rules				Violations			Time
			#All	#Reviewed	#True	TP Rate	#All	#Reviewed	#Bugs	
NAR-Miner−−	266,449	201,381	183,712	200	2	1%	309,689	231	0	24s
NAR-Miner−	16,323	24,040	21,166	200	19	9.5%	37,453	262	1	13s
NAR-Miner	16,323	24,040	21,166	200	93	46.5%	37,453	356	17	16s

Table 2: Previously unknown bugs in Linux-v4.12-rc6 detected by `NAR-Miner`.

ID	Function	Violated Rule	Rule Ranking			PatchID
			NAR-Miner−−	NAR-Miner−	NAR-Miner	
1	hisi_sas_shost_alloc	$\{scsi\_host\_alloc\} \Rightarrow \neg\{kfree\}$	*NA*	12,058	81	9887619
2	pm8001_pci_probe					9887647
3	mvs_pci_init					9887693
4	xfs_test_remount_options	$\{kmem\_zalloc\} \Rightarrow \neg\{kfree\}$	67,667	5,924	87	9887959
5	mxs_lradc_ts_probe	$\{devm\_ioremap\} \Rightarrow \neg\{IS\_ERR\}$	*NA*	1,879	199	9888123
6	ccp_init_dm_workarea	$\{dma\_map\_single\} \Rightarrow \neg\{RET == 0\}$	*NA*	1,303	39	9888311
7	qla26xx_dport_diagnostics					9888341
8	flctl_dma_fifo0_transfer					9888435
9	kexec_calculate_store_digests	$\{crypto\_alloc\_shash\} => \neg\{kfree\}$	*NA*	2,870	111	9890383
10	vpd_sections_init	$\{memremap\} \Rightarrow \neg\{iounmap\}$	180525	17,539	155	9894697
11	vpd_section_init					
12	lapbeth_new_device	$\{free\_netdev\} \Rightarrow \neg\{kfree\}$	*NA*	118	79	10031525
13	ubi_scan_fastmap	$\{kmem\_cache\_alloc\} \Rightarrow \neg\{kfree\}$	*NA*	2,207	42	10031421
14	psb_mmu_pt_unmap_unlock	$\{kunmap\_atomic\} \Rightarrow \neg\{kunmap\_atomic\}$	*NA*	1,651	2	10031217
15	lan9303_probe_reset_gpio	$\{devm\_gpiod\_get\_optional\} \Rightarrow \neg\{RET == 0\}$	*NA*	7,382	127	10054823
16	cpcap_adc_probe	$\{platform\_get\_irq\_byname\} \Rightarrow \neg\{RET == 0\}$	*NA*	5,705	159	10054831
17	esrt_sysfs_init	$\{memremap\} \Rightarrow \neg\{kfree\}$	*NA*	9835	100	10031539

violations according to the ranking of the rules. To gain maximum benefit from the detection, we select the top ranked rules for review as violations of them are more likely to be real bugs. We examine the 200 top ranked negative association rules and corresponding 356 violations (see the last row in Table 1) within one person day. We find 23 suspicious bugs and dozens of program quality problems such as redundant condition checks and computations. As Linux kernel maintainers often ignore the quality problems, we only submit patches of the 23 suspicious bugs to the Linux kernel maintainers. Up to now, 17 of these patches have been confirmed and accepted by the kernel maintainers.

The confirmed bugs are listed in Table 2, with the functions that contain the bugs (*Function*), the rules they violate (*Violated Rule*) and the rule ranking in the three experiments. The last column shows the patch IDs to the bugs and our patches at the PatchWork site. Among these found bugs, six (2#, 3#, 8#, 12#, 13#, and 14#) have presented in kernel 2.6 and two (3# and 12#) even have been latent for more than 10 years.

These bugs violate 12 negative association rules in total. If ranked by confidence, only one of them is within the top 200 rules (12# in column *NAR-Miner−*). But ranking the rules with the information entropy makes all of them in top 200 (*NAR-Miner*). This observation illustrates that introducing information entropy into ranking is significantly useful in highlighting interesting rules. We also observe that only 2 of these rules are extracted in NAR-Miner−− ("NA" for no hit) and the other rules are all missing. For example, the rule $\{free\_netdev\} \Rightarrow \neg\{kfree\}$ (12#) is missing because there are 106 functions calling both *free_netdev* and *kfree*. Without considering semantic relations, the support of the itemset {*free_netdev*, *kfree*} is 106, which is much higher than the predefined threshold $mis = 5$. As a result, it will not be taken as an infrequent itemset and thus the negative association rule cannot be inferred. However, with semantics-constrained mining and information entropy, `NAR-Miner` successfully infers the rule and discovers corresponding bugs (Figure 1). Therefore, we claim that *semantics-constrained mining can help reduce not only false positives but also false negatives.*

4.2.4 Comparison with Positive Rule Mining Based Methods. In practice, certain bugs violating a negative rule may also violate a corresponding positive rule. Therefore, such bugs are supposed to be detected by both negative and positive rule mining based methods. We investigate to see if such cases occur commonly. We choose the 17 bugs in Table 2 as a base line, conduct another experiment that infers positive association rules from the 266,449 frequent itemsets mined in §4.2.2 with the same settings for mfs and min_conf with `NAR-Miner`, and then detects violations of the rules, as done in [25] and [26]. A manual inspection shows three out of the 17 bugs are detected (2#, 3# and 15# in Table 2), whereas the other 14 bugs (about 82.4%) are missing. We then augment the positive rule mining based approach with the semantics-constrained mining, i.e., taking the data relations among program elements into account. Two more bugs (5# and 14#) are discovered, but there are still 12 bugs (about 70.6%) are undetected. Consequently, we claim that

while the semantics-constrained mining is able to help the positive rule mining based approaches detect more bugs, the negative rule mining based approach can exclusively discover a lot of bugs that positive rule mining based approaches cannot.

4.3 Detecting Bugs in Other Systems

NAR-Miner is further applied to PostgreSQL v10.3, OpenSSL v1.1.1 and FFmpeg v3.4.2. NAR-Miner extracted 690, 382 and 335 negative rules from PostgreSQL, OpenSSL and FFmpeg, respectively. We manually inspect the top ranked negative rules (no more than 50) and their violations in each system as done in §4.2. As a result, we identify six violations (two per target), and have reported them to the corresponding communities. Up to now, all of the six suspected bugs have been confirmed and fixed by the corresponding system maintainers. The details can be found in the bug reports for PostgreSQL from the mailing list [41] with IDs #15104 and #15105, for OpenSSL from the issue list [40] with IDs #5567 and #5568, and for FFmpeg from the mailing list [39] with IDs #7074 and #7075. The experiments demonstrate that NAR-Miner is not limited to a specific target system (e.g., Linux kernel), but can be used to find real bugs in various large-scale C systems.

4.4 Case Study

In this section, we illustrate the capability of NAR-Miner with comparison with positive association rule (PAR) mining based methods on the confirmed bug #15105 In PostgreSQL.

In PostgreSQL, the function *OpenTransientFile* allocates a file descriptor and stores it to a globally maintained list of allocated files. The return descriptor must be released with *CloseTransientFile*, which removes the descriptor from the list before truly closing it with *close*. Directly using *close* will make the list keep the released descriptors and may result in use-after-free bugs. Statistically, in PostgreSQL v10.3, *OpenTransientFile* is invoked within 28 functions. In 27 of the functions its return value is passed to *CloseTransientFile*, but in 1 function its return value is directly passed to *close*, resulting in a negative association rule $\{OpenTransientFile\} \Rightarrow \neg\{close\}$ and a positive association rule $\{OpenTransientFile\} \Rightarrow \{CloseTransientFile\}$ with the same confidence of 96.4%.

Figure 4 shows a but in function *dsm_impl_mmap* that incorrectly passes the file descriptor *fd* allocated with *OpenTransientFile* at line 4 to *close* at line 12 along a path. It violates the above negative rule and is thus reported by NAR-Miner. However, from the view of accompanying analysis, because on certain paths *fd* is correctly passed to *CloseTransientFile* at lines 8 and 16, which complies with the requirement of the above positive rule. Hence, the buggy code is indeed a support rather than a violation of the rule.

We fix the bug by replacing line 12 with *CloseTransientFile*(*fd*), as shown in Figure 4. The patch has been accepted by the maintainers.

5 DISCUSSION AND LIMITATIONS

Negative Rules vs. Positive Rules. In this paper, we detect bugs mainly based on negative association rules rather than positive ones. However, these two kinds of approaches have no essential conflicts. As they concentrating on different types of programming rules, they can be complementary to each other. From the point of view of bug detection, our approach is able to extract negative

```
1  //postgresql-10.3/src/backend/storage/ipc/dsm_impl.c
2  static bool dsm_impl_mmap(dsm_op op, dsm_handle h, Size s,
       void **p, void **ma, Size *ms, int l) {
3    int fd, flags = ...; struct stat st; ... // omitted
4    if ((fd = OpenTransientFile(name, flags, 0600)) == -1)
5      return false;
6    if (op == DSM_OP_ATTACH) {
7      if (fstat(fd, &st) != 0) {
8        CloseTransientFile(fd);
9        return false;
10     }
11   } else {
12     close(fd);
13     if (op == DSM_OP_CREATE) unlink(name);
14     return false;
15   }
16   CloseTransientFile(fd);
17   return true;
18 }
```

NAR
PAR
+ Patch
CloseTransientFile(fd);

Figure 4: A bug in PostgreSQL, violating the rule: $\{OpenTransientFile\} \Rightarrow \neg\{close\}$.

programming rules and to detect bugs that cannot be revealed by approaches based on mining positive association rules, and vice verse. Theoretically, a combination of the two kinds of approaches may exhibit better detection performance (less false negatives).

In addition, compared with mining positive rules, mining negative rules is often accompanied by generating more uninteresting rules leading to a large amount of false positives. In this case, the positive rules of the same program can be helpful to reduce them. For example, if a piece of code violates a negative rule but satisfies a positive one, the corresponding violation of the negative rule is more unlikely to be a real bug. We can lower its ranking to filter such a violation. Similarly, bug detection based on positive rules may also face the same challenge (i.e., reporting false positives). So, a straight-forward question is that, in such cases, whether the two approaches can be helpful to reduce the false positives each other? We will further research on this in the future.

Rule Explosion. In essence, the rule explosion problem in negative association rule mining cannot be completely resolved. In this paper, we adopted a relatively straight-forward approach. Specifically, we utilize semantic relationships between elements to eliminate a vast majority of uninteresting rules during mining, and then employed the information entropy to measure interestingness of rules such that potential interesting rules are further highlighted. However, there may exist multiple solutions. For example, we can further quantify the strength of semantic relations among program elements to refine the mining results. Besides the data dependence and data share relations, other relations can also be utilized, such as control flow relation. These potential improvements can further mitigate the rule explosion problem to lower manual audit effort. This is also one of our future works.

Mining Algorithms. In this paper, we adopt the itemset mining algorithm to extract negative programming rules. In practice, for some kinds of programming rules, other forms of representation and mining algorithms may be more appropriate. For example, using sequences to represent order-sensitive programming logics [1, 29, 53, 55] is more suitable than using itemsets. However, the sequence-based algorithm has poor robustness in discovering order-insensitive programming logics. If we can effectively determine whether a programming pattern is order-sensitive or not, a

targeted algorithm can be adopted to mine related rules. This will be one of our future works.

6 RELATED WORK

Program analysis has been widely and successfully used for bug finding. For example, model checking can automatically verify the correctness properties of finite-state systems with the model of the target system and the specification [10]. Due to the high cost of writing a model for the target system, implementation-level model checkers are then developed and find real bugs in system code [32, 59]. Researchers also leverage program analysis to detect the violations of specific rules. Typically, a set of programming rules are provided to the tools which either statically or dynamically check whether the target system violates the given rules. Pasareanu and Rungta developed SPF to generate test cases for Java programs by introducing symbolic execution into model checking [37]. Engler et al. proposed techniques to statically check system rules using system-specific compiler extensions [13] while FindBugs runs as a standalone tool to inspect occurrences of bug patterns in Java bytecode [11]. Livshits and Lams [28] translated user-provided specifications of vulnerabilities into static analyzers, and use them to detect vulnerabilities, such as SQL injections and cross-site scripting, in Web applications written in Java. In addition, Molnar et al. utilized dynamic test generation to find integer bugs in binary programs by checking the violations of particular assertions [31]. Despite their success in finding bugs, these approaches largely depend on the models of the system or the patterns of the bugs, e.g., high-level API semantics [50], which we call prior knowledge. Without that kind of knowledge, they are unable to find bugs. Our work, by contrast, discovers the knowledge automatically and then detects bugs based on the collected knowledge.

Techniques that can automatically extract knowledge from the target system are also presented. The pioneer work proposed by Engler et al. employed statistical analysis to infer temporal rules from given rule templates, detecting bugs without specifying concrete rules [14]. Kremenek et al. used factor graphs to infer specification from programs by incorporating disparate sources of information [22]. These two approaches are limited to infer rules with predetermined templates and specific knowledge that must be provided by the users. Some approaches rely on certain domain knowledge within the mining rules and are specially designed to infer rules for critical APIs [1, 16, 36, 49, 53, 55] or security-sensitive functions [47, 58]. They also require the users to provide the domain knowledge to facilitate the mining process. However, `NAR-Miner` requires nothing from the users while extracting rules based on the association rules (implicitly) included in the programs.

Recently, researchers leverage data mining algorithms to extract more general rules from real large systems [4, 7, 8, 21, 23–25, 27, 29, 30, 33, 34, 44, 54, 58]. The overarching idea behind these mining based techniques is that: *in most cases, programs are correct and thus any anomalies are likely to be bugs*. In general, these approaches first infer frequently appeared patterns from the target system and consider such patterns as the (implicit) rules that developers should follow in coding. Then, they detect any violations of those rules as potential bugs. The inferred patterns can be either positive or negative. For example, PR-Miner [25] and `AntMiner` [26] extract positive association rules that enforce paired appearances of program behaviors. Chang et al. detect missing code structures by mining frequently associated sub-graphs from program control flows and inspecting occasional violations [7]. Yun et al. infer the correct usage of APIs based on the mined positive association rules of the semantics among different APIs [60]. Different with these approaches, `NAR-Miner` focuses on mining negative association rules from source code and detects bugs that violate those rules. Similar rules can also be extracted from dynamic execution traces. Beschastnikh et al. developed `Synoptic` to generate temporal system invariants from system execution logs [5].

Wang et al. developed `Bugram` that employs the n-gram language model to measure the probability of token sequences and treats low probability sequences as anomalies, i.e., potential bugs [52]. `Bugram` can also detect certain bugs caused by the co-occurrence of mutually suppressed program elements. However, due to the limited size of the sequence window, `Bugram` is difficult to capture bugs involving long distance program elements.

Mining negative association rules has been applied to data like market basket, protein sequences, and financial data [18]. For such data, the relationship between two elements is much simpler than that for program elements which contribute different intensities to the relationship. Wu et al. presented algorithms to effectively and efficiently mine negative association rules in large databases [56]. Zhou and Yau proposed a combined algorithm to mine interesting association rules, reducing large number of negative rules [61]. These algorithms can also be adopted by `NAR-Miner` as the basic mining algorithm but need to handle program semantics in order to reduce uninteresting rules.

7 CONCLUSION

Data mining techniques have been widely used to infer programming rules and then detect software bugs based on the rules. Existing approaches have proven that positive association rules, indicating that associated program elements must appear together, are useful to detect bugs via checking the violations. However, the negative programming rules, which disallow the co-occurrences of involved program elements, are mostly neglected. We present `NAR-Miner` to mine negative association rules from source code. We introduce program semantics to guide the mining phase. We also leverage function entropy to rank candidate rules and highlight the interesting ones. By this means, `NAR-Miner` dramatically reduces the number of uninteresting rules and mitigates the rule explosion problem to a certain degree. We evaluate the prototype on four popular large-scale systems and find a considerable number of bugs, some of which have been confirmed by the maintainers.

ACKNOWLEDGEMENTS

The work is supported in part by National Natural Science Foundation of China (NSFC) under grants 91418206, 61802413, 61170240, 61472429, and 61502465, National 973 program of China under grant 2014CB34-0702, National Science and Technology Major Project of China under grant 2012ZX01039-004, Youth Innovation Promotion Association of the Chinese Academy of Sciences (YICAS) under grant 2017151, and Young Elite Scientists Sponsorship Program by CAST (2017QNRC001).

REFERENCES

[1] Mithun Acharya, Tao Xie, Jian Pei, and Jun Xu. 2007. Mining API patterns as partial orders from source code: from usage scenarios to specifications. In *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007*. 25–34.

[2] Rakesh Agrawal and Ramakrishnan Srikant. 1994. Fast algorithms for mining association rules in large databases. In *Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*. 487–499.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers: principles, techniques, and tools*. Addison-Wesley.

[4] Glenn Ammons, Rastislav Bodík, and James R Larus. 2002. Mining specifications. *ACM Sigplan Notices* 37, 1 (2002), 4–16.

[5] Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D. Ernst. 2011. Leveraging Existing Instrumentation to Automatically Infer Invariant-Constrained Models. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE '11)*. 267–277.

[6] Pan Bian, Bin Liang, Yan Zhang, Chaoqun Yang, Wenchang Shi, and Yan Cai. 2018. Detecting Bugs by Discovering Expectations and Their Violations. *IEEE Transactions on Software Engineering* (2018). https://doi.org/10.1109/TSE.2018.2816639

[7] Ray-Yaung Chang, Andy Podgurski, and Jiong Yang. 2007. Finding what's not there: a new approach to revealing neglected conditions in software. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2007, London, UK, July 9-12, 2007*. 163–173.

[8] Xiao Cheng, Zhiming Peng, Lingxiao Jiang, Hao Zhong, Haibo Yu, and Jianjun Zhao. 2016. Mining revision histories to detect cross-language clones without intermediates. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*. 696–701.

[9] Brian Chess and Gary McGraw. 2004. Static analysis for security. *IEEE Security & Privacy* 2, 6 (2004), 76–79.

[10] E. M. Clarke, E. A. Emerson, and A. P. Sistla. 1986. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.* 8, 2 (April 1986), 244–263.

[11] Brian Cole, Daniel Hakim, David Hovemeyer, Reuven Lazarus, William Pugh, and Kristin Stephens. 2006. Improving your software using static analysis to find bugs. In *Companion to the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*. 673–674.

[12] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems* 13, 4 (1991), 451–490.

[13] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. 2000. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of 4th Symposium on Operating System Design and Implementation (OSDI 2000), San Diego, California, USA, October 23-25, 2000*. 1–16.

[14] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. 2001. Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In *Proceedings of the Proceedings of the 18th ACM Symposium on Operating System Principles, SOSP 2001, Chateau Lake Louise, Banff, Alberta, Canada, October 21-24, 2001*. 57–72.

[15] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.* 9, 3 (1987), 319–349.

[16] Mark Gabel and Zhendong Su. 2008. Javert: Fully Automatic Mining of General Temporal Properties from Dynamic Traces. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '08/FSE-16)*. 339–349.

[17] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler. 2002. A system and language for building system-specific, static analyses. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*. 69–82.

[18] Jiawei Han, Micheline Kamber, and Jian Pei. 2011. *Data mining: concepts and techniques, 3rd edition*. Morgan Kaufmann, Chapter 13.3, 607–615. http://hanj.cs.illinois.edu/bk3/

[19] Suman Jana, Yuan Jochen Kang, Samuel Roth, and Baishakhi Ray. 2016. Automatically detecting error handling bugs using error specifications. In *Proceedings of the 25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*. 345–362.

[20] Yuan Jochen Kang, Baishakhi Ray, and Suman Jana. 2016. APEx: automated inference of error specifications for C APIs. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*. 472–482.

[21] Samantha Syeda Khairunnesa, Hoan Anh Nguyen, Tien N. Nguyen, and Hridesh Rajan. 2017. Exploiting Implicit Beliefs to Resolve Sparse Usage Problem in Usage-based Specification Mining. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 83 (2017), 83:1–83:29 pages.

[22] Ted Kremenek, Paul Twohey, Godmar Back, Andrew Ng, and Dawson Engler. 2006. From uncertainty to belief: Inferring the specification within. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06), Seattle, WA, USA, November 6-8*. 161–176.

[23] Ivo Krka, Yuriy Brun, and Nenad Medvidovic. 2014. Automatic Mining of Specifications from Invocation Traces and Method Invariants. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. 178–189.

[24] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. 2004. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. In *Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*. 289–302.

[25] Zhenmin Li and Yuanyuan Zhou. 2005. 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*. 306–315.

[26] Bin Liang, Pan Bian, Yan Zhang, Wenchang Shi, Wei You, and Yan Cai. 2016. AntMiner: mining more bugs by reducing noise interference. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. 333–344.

[27] Benjamin Livshits and Thomas Zimmermann. 2005. 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, 2005, Lisbon, Portugal, September 5-9, 2005 (ESEC/FSE-13)*. 296–305.

[28] V. Benjamin Livshits and Monica S. Lam. 2005. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *Proceedings of the 14th USENIX Security Symposium, Baltimore, MD, USA, July 31 - August 5, 2005*.

[29] David Lo, Siau-Cheng Khoo, and Chao Liu. 2008. Mining past-time temporal rules from execution traces. In *Proceedings of the 2008 International Workshop on Dynamic Analysis: held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008), WODA 2008, Seattle, Washington, USA, July 21, 2008*. 50–56.

[30] Shan Lu, Soyeon Park, Chongfeng Hu, Xiao Ma, Weihang Jiang, Zhenmin Li, Raluca A Popa, and Yuanyuan Zhou. 2007. MUVI: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*. 103–116.

[31] David Molnar, Xue Cong Li, and David A. Wagner. 2009. Dynamic test generation to find integer bugs in x86 binary Linux programs. In *Proceedings of the 18th USENIX Security Symposium, Montreal, Canada, August 10-14, 2009*.

[32] Madanlal Musuvathi, David Y. W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. 2002. CMC: A pragmatic approach to model checking real code. *SIGOPS Oper. Syst. Rev.* 36, SI (Dec. 2002), 75–88.

[33] Hoan Anh Nguyen, Robert Dyer, Tien N. Nguyen, and Hridesh Rajan. 2014. Mining Preconditions of APIs in Large-scale Code Corpus. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. 166–177.

[34] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. 2009. Graph-based Mining of Multiple Object Usage Patterns. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE '09)*. 383–392.

[35] Henning Perl, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Acar. 2015. Vccfinder: finding potential vulnerabilities in open-source projects to assist code audits. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*. 426–437.

[36] Michael Pradel, Ciera Jaspan, Jonathan Aldrich, and Thomas R Gross. 2012. Statically checking API protocol conformance with mined multi-object specifications. In *Proceedings of the 34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*. 925–935.

[37] Corina S. Păsăreanu and Neha Rungta. 2010. Symbolic PathFinder: Symbolic Execution of Java Bytecode. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE '10)*. 179–180.

[38] Murali Krishna Ramanathan, Ananth Grama, and Suresh Jagannathan. 2007. Static specification inference using predicate mining. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*. 123–134.

[39] Bug report list for FFmpeg. March 2018. https://trac.ffmpeg.org.

[40] Bug report list for OpenSSL. March 2018. https://github.com/openssl/openssl/issues.

[41] Bug report mailing list for PostgreSQL. March 2018. https://www.postgresql.org/list/pgsql-bugs.
[42] Cindy Rubio-González and Ben Liblit. 2011. Defective error/pointer interactions in the linux kernel. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ACM, 111–121.
[43] Ashok Savasere, Edward Omiecinski, and Shamkant B. Navathe. 1998. Mining for strong negative associations in a large database of customer transactions. In *Proceedings of the Fourteenth International Conference on Data Engineering, Orlando, Florida, USA, February 23-27, 1998*. 494–502.
[44] Boya Sun, Gang Shu, Andy Podgurski, and Brian Robinson. 2012. Extending static analysis by mining project-specific rules. In *Proceedings of the 34th International Conference on Software Engineering, ICSE 2012, Zurich, Switzerland, June 2-9, 2012*. 1054–1063.
[45] Laszlo Szathmary, Amedeo Napoli, and Petko Valtchev. 2007. Towards rare itemset mining. In *Proceedings of the 19th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2007), Patras, Greece, October 29-31, 2007*. 305–312.
[46] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. 2007. /* iComment: bugs or bad comments?*. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*. 145–158.
[47] Lin Tan, Xiaolan Zhang, Xiao Ma, Weiwei Xiong, and Yuanyuan Zhou. 2008. AutoISES: automatically inferring security specification and detecting violations. In *Proceedings of the 17th USENIX Security Symposium, July 28-August 1, 2008, San Jose, CA, USA*. 379–394.
[48] Lin Tan, Yuanyuan Zhou, and Yoann Padioleau. 2011. aComment: mining annotations from comments and code to detect interrupt related concurrency bugs. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*. 11–20.
[49] Suresh Thummalapenta and Tao Xie. 2009. Alattin: Mining alternative patterns for detecting neglected conditions. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering, Auckland, New Zealand, November 16-20, 2009*. 283–294.
[50] John Toman and Dan Grossman. 2017. Taming the Static Analysis Beast. In *Proceedings of the 2nd Summit on Advances in Programming Languages, SNAPL 2017, Asilomar, CA, USA, 7-10, May, 2017*. 18:1–18:14.
[51] Olivier Vandecruys, David Martens, Bart Baesens, Christophe Mues, Manu De Backer, and Raf Haesen. 2008. Mining software repositories for comprehensible software fault prediction models. *Journal of Systems and software* 81, 5 (2008), 823–839.
[52] Song Wang, Devin Chollak, Dana Movshovitz-Attias, and Lin Tan. 2016. Bugram: bug detection with n-gram language models. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*. 708–719.
[53] Andrzej Wasylkowski and Andreas Zeller. 2011. Mining temporal specifications from object usage. *Automated Software Engineering* 18, 3-4 (2011), 263–292.
[54] Andrzej Wasylkowski, Andreas Zeller, and Christian Lindig. 2007. Detecting Object Usage Anomalies. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE '07)*. 35–44.
[55] Westley Weimer and George Necula. 2005. Mining temporal specifications for error detection. *Proceedings of the 11th Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2005, , Edinburgh, UK, April 4-8, 2005,*, 461–476.
[56] Xindong Wu, Chengqi Zhang, and Shichao Zhang. 2004. Efficient mining of both positive and negative association rules. *ACM Trans. Inf. Syst.* 22, 3 (July 2004), 381–405.
[57] Fabian Yamaguchi, Alwin Maier, Hugo Gascon, and Konrad Rieck. 2015. Automatic inference of search patterns for taint-style vulnerabilities. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. 797–812.
[58] Fabian Yamaguchi, Christian Wressnegger, Hugo Gascon, and Konrad Rieck. 2013. Chucky: Exposing missing checks in source code for vulnerability discovery. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*. 499–510.
[59] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. 2006. Using model checking to find serious file system errors. *ACM Trans. Comput. Syst.* 24, 4 (Nov. 2006), 393–423.
[60] Insu Yun, Changwoo Min, Xujie Si, Yeongjin Jang, Taesoo Kim, and Mayur Naik. 2016. APISan: sanitizing API usages through semantic cross-checking. In *Proceedings of the 25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*. 363–378.
[61] Ling Zhou and Stephen S.-T. Yau. 2007. Efficient association rule mining among both frequent and infrequent items. *Computers & Mathematics with Applications* 54, 6 (2007), 737–749.