Detecting Bugs by Discovering Expectations and Their Violations

Pan Bian, Bin Liang, Yan Zhang, Chaoqun Yang, Wenchang Shi, and Yan Cai

Abstract—Code mining has been proven to be a promising approach to inferring implicit programming rules for finding software bugs. However, existing methods may report large numbers of false positives and false negatives. In this paper, we propose a novel approach called *EAntMiner* to improve the effectiveness of code mining. *EAntMiner* elaborately reduces noises from statements irrelevant to interesting rules and different implementation forms of the same logic. During preprocessing, we employ program slicing to decompose the original source repository into independent sub-repositories. In each sub-repository, statements irrelevant to critical operations (automatically extracted from source code) are excluded and various semantics-equivalent implementations are normalized into a canonical form as far as possible. Moreover, to tackle the challenge that some bugs are difficult to be detected by mining frequent patterns as rules, we further developed a *k*NN-based method to identify them. We have implemented *EAntMiner* and evaluated it on four large-scale C systems. *EAntMiner* successfully detected 105 previously unknown bugs that have been confirmed by corresponding development communities. A set of comparative evaluations also demonstrate that *EAntMiner* can effectively improve the precision of code mining.

Index Terms—Bug detection, code mining, program slicing, instance-based learning

1 Introduction

In recent years various code mining approaches have been proposed to automatically extract implicit programming rules from source code repositories [1], [5], [10], [18], [27], [31], [36], [37] [39], [44], [48], [54], [55], [62]. In particular, such approaches on bug detection have proven to be effective [10], [27], [48], [55], [60], [62], and have detected numbers of real bugs in large-scale systems, such as in the Linux kernel. This further motivates commercial bug detection systems to incorporate the mining idea. For example, *Coverity*, one of the most widely used bug detection tools, leverages the statistical method to automatically extract programming rules and detect bugs in some of its checkers (e.g., the *NULL_RETURNS* checker [43]).

The code mining approach is based on an insightful observation: most of the invocations to a specific operation (e.g., an API) are correct, whereas the defective ones are relatively rare. For example, in the Linux kernel v2.6.39, the return value of the function $alloc\_skb()$ is validated before being passed to the function $skb\_reserve()$ in most cases (127 out of 128). The only one exception or violation (i.e., without the validation) is actually a programming bug. To detect such violations, data mining algorithms are first employed to extract frequently appeared patterns from source code as (implicit) programming rules. Next, any violations to these rules can be treated as potential bugs.

Unfortunately, in practice, detecting bugs with code mining heavily suffers from reporting large numbers of false positives and false negatives. Many of them result from the interference of irrelevant statements and semantics-equivalent implementations (both are called *noise* in this paper). Mining programming rules requires the source code to be fully transformed into a database for mining. However, in practice, a piece of source code may contain irrelevant statements and such statements, if not effectively seperated, will affect the correctness of the mined rules. On the other hand, if the irrelevant statements have the similar forms with those in the rule, some real violations may be missed (see Section 2.1). Moreover, programmers may adopt different ways to implement the same logic. If they are not transformed into the same form, the mining and the detection algorithms may mistake them as different patterns. All these together affect the mining process. That is, both the support values and the confidence values (i.e., the two measures of the interestingness of a mined rule) of the mined rules may be deviated from what they are deemed to be. As a result, lots of uninteresting rules may be inferred but many interesting rules will be neglected.

To solve the above challenges, researchers have proposed several approaches [10], [27]. However, their solutions are insufficient and may further introduce new problems (see Section 2.2).

In the preliminary version [28] of this paper, we proposed *AntMiner*, a divide-and-conquer method to reduce noise by carefully preprocessing source code. It employs the slicing technique to decompose the whole program into independent

• *P. Bian, B. Liang, Y. Zhang, C. Yang, and W. Shi are with School of Information, Renmin University of China, Beijing 100872, China, and with the Key Laboratory of DEKE, Renmin University of China, Beijing 100872, China. E-mail: {bianpan, liangb, annazhang, cqyang, wenchang}@ruc.edu.cn.*
• *Y. Cai is with the State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing 100080, China. E-mail: yancai@ios.ac.cn.*

Manuscript received 29 June 2017; revised 14 Feb. 2018; accepted 26 Feb. 2018. Date of publication 15 Mar. 2018; date of current version 23 Oct. 2019. (Corresponding author: Pan Bian.)
Recommended for acceptance by K. Sen.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2018.2816639

sub-repositories according to a set of *critical operations* that often serve as the indispensable elements of the logic of bugs. *AntMiner* reduces the impact of above noises by aggressively removing program statements irrelevant to the critical operations and carefully normalizing statements into canonical forms as far as possible. *AntMiner* takes *bug-prone functions* and *function return* as critical operations, and can detect bugs caused by inappropriate function calls or incorrect function return values. A function can be regarded as *bug-prone* if an inappropriate invocation to it is likely to cause a program bug (see its formal definition in Section 3.3.1).

An evaluation on the Linux v2.6.39 shows that *AntMiner* can effectively reduce false positives and false negatives in detecting real bugs caused by misusing bug-prone functions and returning incorrect values.

However, *AntMiner* still suffers from missing some return value bugs whose corresponding programming rules are difficult to infer with frequent pattern mining. A major reason is that the semantics-based normalization method adopted in *AntMiner* is not effective enough in normalizing return statements due to their mutable semantics. The semantics of a return statement is often determined by how the return value is handled in the callers apart from preceding statements (i.e., the context for the return statement). In some contexts, different return statements may have an identical semantics; but in some other contexts, the same return statement can have quite different meanings. For example, in the Linux kernel, statements "$return - ENOMEM;$" and "$return - EIO;$" are both acceptable to represent a memory allocation error in many driver functions (e.g., $st\_probe()$). At the same time, the return statement "$return\,0;$" indicates no exception occurs in some functions (e.g., $ioat\_dma\_self\_test()$), but is used as an error flag in some other functions (e.g., $xhci\_align\_td()$). This makes the normalization methods adopted in *AntMiner* fail to transform many return statements with equivalent semantics into the same form or mistakenly transform some inequivalent ones into the same form. As a direct result, mining frequent patterns to get the related programming rules does not work well for return operations, and some interesting return rules may be neglected (an example will be shown in Section 2.3).

In this paper, we present *EAntMiner*, an enhancement of *AntMiner*, to improve the effectiveness of *AntMiner* in three major aspects. (i) It extends the normalization method to map return values to a canonical abstract form. In this way, more semantics-equivalent return statements can be transformed into the same form. (ii) It converts the problem of detecting return value bugs into a non-parametric classification problem. The return value of a function is regarded as its class label. Taking a set of functions that are most similar to the one under consideration as training samples, an instance-based learning method (*k*NN in this paper) is employed to discover the most common label among them as the expected label of the target function. If the function returns an unexpected value, a potential bug will be reported. In addition, (iii) it takes *decisive conditions* under which a function should directly return to more effectively exclude irrelevant statements. We will further illustrate decisive conditions in Section 3.3.2. Note that, the decisive conditions, as well as the bug-prone functions, are automatically extracted with a statistics-based method.

We have implemented *EAntMiner* as a prototype tool by integrating the above improvements into *AntMiner*. We evaluated *EAntMiner* on the Linux kernel of both versions v2.6.39 and v4.9-rc3, OpenSSL v1.1.0g, FFmpeg v3.4, and PostgreSQL v10.1. In the preliminary version [28] of this paper, *AntMiner* was applied on the Linux v2.6.39, and found 24 previously unknown bugs. All of these bugs can also be detected by *EAntMiner*. The Linux v4.9-rc3 was the latest version at our experiment time. The evaluation on it shows that *EAntMiner* detected 69 previously unknown bugs, including 20 misusages of bug-prone functions and 49 incorrect return values. By contrast, *AntMiner* missed about 89.8 percent (44 out of 49) of incorrect return value bugs. The evaluation results show that *EAntMiner* performs significantly better than *AntMiner* in detecting return value bugs. *EAntMiner* also detected 12 previously unknown bugs from OpenSSL, FFmpeg, and PostgreSQL. The result demonstrates that our method is not specially designed for the Linux kernel, but can also be applied to find bugs in other large-scale systems.

This paper makes the following main improvements.

- We presented a normalization method based on value abstraction to transform more semantics-equivalent return statements into the same form.
- We proposed an instance-based learning method to overcome the obstacle of mining programming rules for return operations.
- We implemented a prototype tool *EAntMiner* and evaluated it on four large-scale systems to demonstrate its effectiveness. Dozens of unknown bugs previously missed were successfully detected.

The rest of this paper is organized as follows. In Section 2, we show a high-level overview of our method with some motivating examples. We present the design and implementation of *EAntMiner* in Section 3, and evaluate it in Section 4. After discussing the limitation and perspective of future work in Section 5, we review related work in Section 6. Finally, Section 7 concludes this paper.

2 Motivating Examples

In this section, we show four samples from the Linux kernel to demonstrate the limitations of the general code mining approaches, including (1) how irrelevant statements and inconsistent statements can interfere with the mining and detection processes (Sections 2.1 and 2.2), and (2) how mining association rules as a fixed baseline can miss some return value bugs (Section 2.3).

2.1 Interference from Irrelevant Statements

We illustrate this problem with an example from the well-tested Linux kernel source code. In the Linux kernel programs, there is an implicit programming rule: the return value of the function *snd_pcm_new*() should be checked immediately to make sure that a new *snd_pcm* instance is created successfully before being passed to the function $snd\_pcm\_set\_ops()$. Fig. 1 shows a piece of program that violates the rule. That is, right after line 854, the return value *err* is not checked.

Applying a frequent pattern mining algorithm (e.g., *FPclose* [17]) on the kernel code, the frequent pattern $\{err = snd\_pcm\_new(), \text{if } (err < 0), snd\_pcm\_set\_ops()\}$ can

```
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 }
```

Fig. 1. A violation to the pattern $\{err = snd\_pcm\_new()$, if $(err < 0)$, $snd\_pcm\_set\_ops\,()\}$, lacking a necessary check of the return value of function $snd\_pcm\_new()$.

be extracted. The pattern can be taken as a programming rule to detect related bugs [27]. Unfortunately, the program in Fig. 1 actually contains all three elements of the rule (at lines 853, 867, and 858, respectively). As a result, the program will be mistaken as an obedience rather than a violation of the rule. This false negative is caused by the conditional statement at line 867 as the statement is irrelevant to both functions $snd\_pcm\_new()$ (line 853) and $snd\_pcm\_set\_ops()$ (called at line 858).

Analysis and Our Solution. Approaches based on order-sensitive data mining algorithms, e.g., *frequent subsequence mining* [54], [55], may detect such a bug. However, if the irrelevant statements "err $= f()$; if $(err < 0)$" appear between lines 853 and 858 (which is quite possible in practice), the methods would also fail. Besides, the order-sensitive solutions are not scalable (due to their much higher time complexity than itemset mining methods [1], [10]) on extracting programming rules. In order to make them scalable to mine rules from large-scale software (e.g., the Linux kernel, which has tens of millions of lines of code), a larger *minimum support threshold* should be set. However, this would miss rules with relatively small supports.

For the example in Fig. 1, the false negative is actually caused by the statement (line 867) which is irrelevant to the rule, confusing the detection process. Hence, the bug can be detected by actively removing the irrelevant statements before transforming the program into mining databases. To achieve this goal, we have to distinguish the statements we do care about from those we do not care about. Our observation is that a bug often occurs when some critical operations are called without satisfying some conditions (or preconditions). Based on this observation, we should only focus on statements that either directly call a critical operation or impact the execution of a critical operation. The other statements are considered irrelevant and should be removed.

Given a critical operation, one can adopt the backward program slicing technique [56] to achieve this goal. For example, in Fig. 1, assuming that the function $snd\_pcm\_set\_ops()$ is a critical operation (i.e., acting as a slicing criterion), the conditional statement (line 867) can be then excluded from the corresponding slice. Hence, the detection algorithm is able to catch this violation because the

```
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

Fig. 2. Two validation samples to the actual parameter of function $tty\_hangup()$.

remaining statements do not contain the second element in the identified rule.

2.2 Interference from Inconsistent Implementations

In the preliminary version [28] of this paper, we explained how variable names and control structures interfere with code mining (i.e., Sections 2.2 and 2.3). In this paper, we discuss the two aspects in this section.

The validation to the sensitive data usually depends on conditional statements. In general, missing effective validations may result in programming bugs. For example, the program shown in Fig. 2a illustrates an effective validation to the actual parameter of function $tty\_hangup()$. However, in the program shown in Fig. 2b, an incorrect conditional expression (i.e., "$C\_CLOCAL(tty)$" rather than "$!C\_CLOCAL(tty)$") is used in the validation. This introduces a bug. On the other hand, programmers may use different or even opposite conditional expressions to enforce the same validation. For example, there are two validations on the return value of the function $dev\_alloc\_skb()$ shown in Figs. 3a and 3b, respectively. Although the two validations employ different variable names and completely opposite conditional expressions, considering the contexts, both of them effectively guarantee that the return value is not NULL before being passed to the critical operation $skb\_reserve()$.

Analysis and Our Solution. Efforts are proposed to replace variables with their data types [27], and simply make the similar expressions identical, e.g., replacing "$! =$" with "$==$" in the control points without considering the semantics of related control structures [10]. However, program statements with different semantics may be mistakenly transformed into the same form, because these normalization methods ignore

```
linux-2.6.39/drivers/net/pcnet32.c:
    newskb = dev_alloc_skb(PKT_BUF_SKB);
    if (newskb) {
        skb_reserve(newskb, NET_IP_ALIGN);
        // do something
    }
```

(a) Effective validation 1

```
linux-2.6.39/drivers/net/wireless/ray_cs.c:
    skb = dev_alloc_skb(total_len + 5);
    if (!skb) {
        // do something
        return;
    }
    skb_reserve(skb, 2);        /* Align IP on 16 byte (TBD check this) */
```

(b) Effective validation 2

Fig. 3. Two effective validations for the return value of function $dev\_alloc\_skb\,()$.

```
linux-4.9-rc3/drivers/dma/ioat/init.c:
302 static int ioat_dma_self_test(struct ioatdma_device *ioat_dma)
303 {
         …
313      int err = 0;
         …
318      src = kzalloc(sizeof(u8) * IOAT_TEST_SIZE, …);
319      if (!src)
320           return -ENOMEM;
         …
340      dma_src = dma_map_single(dev, src, GFP_KERNEL);
341      if (dma_mapping_error(dev, dma_src)) {
342           dev_err(dev, "mapping src buffer failed\n");
              // forget to set err to -ENOMEM
343           goto free_resources;
344      }
         …
389 unmap_src:
390      dma_unmap_single(dev, dma_src, …);
391 free_resources:
392      dma->device_free_chan_resources(dma_chan);
393 out:
394      kfree(src);
395      kfree(dest);
396      return err;
397 }
     …
1028 static int ioat3_dma_self_test(struct ioatdma_device
                    *ioat_dma)
1029 {
1030      int rc;
1031
1032      rc = ioat_dma_self_test(ioat_dma);
1033      if (rc)
1034           return rc;
          …
1039 }
```

Fig. 4. An example that returns incorrect value.

the semantics of program statements. Therefore, these methods are ineffective to process the above samples; otherwise we may miss the bug in Fig. 2b.

A better solution is to incorporate the semantic information into the canonical form of a program statement. In our approach, a variable is given a new canonical name that can reflect where its value comes from; and the control structure is rearranged to ensure that the predicate, which must be satisfied before executing a critical operation, is explicitly specified in its conditional expression. For example, in Fig. 3, because both variables *newskb* and *skb* keep the return value of the function $dev\_alloc\_skb()$, they are renamed as FN-dev_alloc_skb-0. Additionally, for the program in Fig. 3b, the conditional expression "$!skb$" (line 2175) can be reversed to "skb" to explicitly specify that the variable *skb* has been checked against null before being passed to the function $skb\_reserve()$. On the other hand, the conditional expressions in Figs. 2 and 3a remain their original forms. In this way, the bug in Fig. 2b can then be detected, and no false positives are reported for the programs in Fig. 3.

2.3 Interference from Variant Semantics

We observed that many functions take return values as a payload to pass the execution result to the callers. Subtle bugs can arise if the return value is incorrect. For example, in the Linux kernel, the function $dma\_map\_single()$ maps a piece of processor virtual memory to a DMA address such that it can be accessed by peripheral drivers. The mapping errors are checked by testing the returned address against the function $dma\_mapping\_error()$, which returns a non-zero value if the mapping fails. As shown in Fig. 4, the function $ioat\_dma\_self\_test()$ attempts to map memory (line 340). It returns zero if there are errors during the mapping process (see line 343). However, in this scenario, a return value zero means *no error*. Its callers (e.g., the function $ioat3\_dma\_self\_test()$) cannot realize the mapping error and will perform unexpected and dangerous behaviors, e.g., forcing the $Intel\, I/OAT$ driver to startup even if the self-test fails, which may result in unpredictable bugs on future access to the device.

Analysis and Our Solution. To detect the bug in Fig. 4, traditional static methods [14], [23], [24] require users to provide the return rule to indicate that: *function* $ioat\_dma\_self\_test()$ *should return a non-zero value if the return value of* $dma\_mapping\_error()$ *is not zero*. This becomes very difficult or even impossible as it requires a large amount of human effort. In the preliminary version [28] of this paper, *AntMiner* attempts to detect this kind of bugs by mining return rules. A return rule is represented as $P => \{\text{“return } v\text{”}\}$, which means a function should return v on paths that contain a certain frequent pattern P. Though this method can find some real bugs from large-scale systems, it may miss many bugs due to the inability to mine some kinds of interesting rules. For example, the frequent pattern $P_1 = \{dma = dma\_map\_single(),\ error = dma\_mapping\_\ error(), \text{if}(error)\}$ appears in 188 functions in the Linux v4.9-rc3. The functions that contain the pattern P_1 can return "0", "-EIO", "-EINVAL", "-ENODEV", "-1", "-ENOMEM", "-EAGAIN", etc. Among them, "$-$ENOMEM" is the most frequently used return value, and the number of functions returning it is 77. From these data, the confidence of the return rule $P_1 => \{\text{“return} - \text{ENOMEM”}\}$ is about 41 percent, which is much smaller than the minimum confidence 85 percent [28]. Besides, the number of functions that return "-ENOMEM", "-EINVAL", "$-$ENODEV", "-EIO", "-1", "-EAGAIN" is 147. Even if we can normalize them all to "$!=0$" with the abstracting method presented in Section 3.5, the confidence of the rule $P_1 => \{\text{“return} != 0\text{”}\}$ is about 78 percent, which is not large enough yet. Therefore, *AntMiner* is unable to mine a desired return rule, and thus cannot detect the bug in Fig. 4.

The underlying reason is that the same return value can be interpreted as different or even opposite meanings in different functions. For example, the return value "0" indicates *no error* in function $ioat\_dma\_self\_test()$; but, oppositely, it means *error* in function $xhci\_align\_td()$, which also contains pattern P_1. Using a universal rule may falsely report $xhci\_align\_td()$ as a bug. Or more seriously, as discussed above, the rule would not be mined if the functions that take "0" as *error* and *no error* are fifty-fifty. Fortunately, in practical systems, the more similar two functions are (e.g., performing similar operations and defined in the same file or adjacent files), the more likely they will use the same return value to represent the same semantics. To this end, we can infer the expected return value from a set of functions that are most similar to the one under detection. And the real return value is considered to be incorrect if it mismatches the expected one. For example, 19 out of the top 20 functions that are most similar with the function $ioat\_dma\_self\_test()$ return *non-zero* values when a DMA mapping operation fails. Therefore, we have a high confidence $(19/20)$ that the function $ioat\_dma\_self\_test()$ expects a *non-zero* return value on

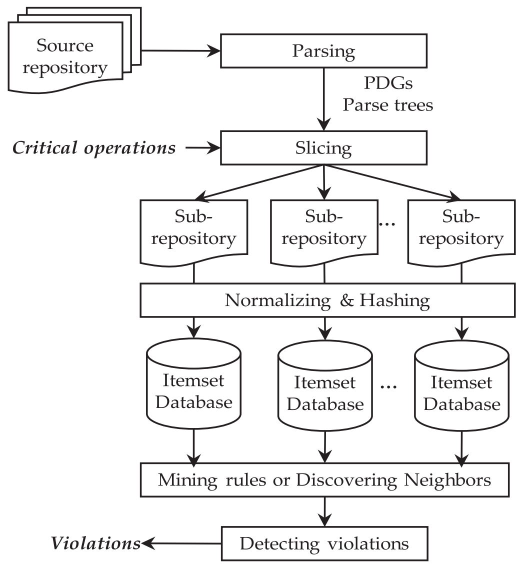

Fig. 5. An overview of *EAntMiner*.

the DMA mapping error path. As the real return value (i.e., *zero*) is different with the expected one, the function $ioat\_dma\_self\_test()$ is taken as a potential bug.

3 EAntMiner Approach

3.1 Overview

Compared with most traditional code mining approaches, *EAntMiner* 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. The code mining is then independently performed on these sub-repositories.

Fig. 5 shows an overview of *EAntMiner*. First, the source code is parsed into parse trees, and a *program dependence graph* (PDG) is generated for each function definition (Section 3.2). Second, it extracts critical operations from the source code itself without any human involvement (Section 3.3). Third, according to the critical operations, the program slicing technique is employed to generate a series of sub-repositories (Section 3.4). A sub-repository consists of the program slices associated with a specific critical operation. Fourth, program statements are normalized, and every sub-repository is converted to an *itemset database* (Section 3.5). Finally, *EAntMiner* iterates over itemset databases to detect abnormal itemsets. On databases of bug-prone functions, a *frequent sub-itemset mining* algorithm is employed to mine frequent patterns and generate programming rules, and violations to the extracted rules are taken as potential bugs (Section 3.6.1). Whereas, on databases of decisive conditions, *EAntMiner* lets the k nearest neighbors of an itemset under consideration to decide whether the corresponding return value is correct or not (Section 3.6.2).

3.2 Parsing Source Code

EAntMiner uses a modified GCC compiler [42] frontend to parse 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 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 by Ferrante et al. [15]. In a PDG, a vertex represents a GIMPLE statement, and an edge represents the dependency information between two vertexes. A PDG consists of a *control dependence subgraph* (CDS) and a *data dependence subgraph* (DDS):

- The CDS describes the control dependencies among statements. In the CDS, if a statement s_2 is control dependent on a conditional 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. A statement s_n is indirectly control dependent on s_1 if there is a path from s_1 to s_n on the CDS. For example, in Fig. 4, the statement at line 342 is directly control dependent on the conditional statement at line 341, and is indirectly control dependent on the conditional statement at line 319.
- 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 exists 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 Fig. 1, both variables *err* and *pcm* are defined at line 853, and 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 invoking some critical operations. An operation can be regarded as critical if misusing it tends to cause a program bug. In practice, either passing an illegal parameter to a *bug-prone function* or returning an incorrect value under a *decisive condition* can result in serious bugs. In the preliminary version [28] of this paper, we proposed a statistics-based method to automatically extract bug-prone functions from the source code itself. Based on this method, *AntMiner* can automatically collect potential bug-prone functions, without requiring any prior knowledge of the target system. In a previous evaluation, it found thousands of bug-prone functions from the Linux kernel in about 30 minutes [28], which saves a great deal of human efforts. In this paper, we generalize this method to further automatically extract decisive conditions.

3.3.1 Extracting Bug-Prone Functions

As explained in the introduction, a function is considered to be bug-prone if an inappropriate invocation to it tends to cause a program bug when executing the function. Formally, we define bug-prone functions as follows:

Definition 1. *Given a function f and a set of constraints c on the inputs of f, if (1) the execution of function f exhibits no program bugs iff all constraints in c are satisfied, and (2) the constraint set c is non-empty, then f is bug-prone.*

That is, a bug-prone function will certainly result in a program bug if any of its inputs (e.g., one of its actual parameters) fails to satisfy the preconditions (i.e., the constraint set c in the above definition). For example, the function $strcpy(dest\_buffer, src\_buffer)$ is a bug-prone one because it requires that the size of $dest\_buffer$ (i.e., the destination buffer) be large enough so as to hold characters in src_buffer (i.e., the source buffer); and a buffer overflow will be caused if a call instance of $strcpy()$ violates the above constraint. 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)[1], the sinks of many weaknesses are calls to some security-sensitive functions.

In real programs, some functions are less likely to get involved in bugs than functions like $strcpy()$. For example, the function $isdigit()$ (a function in C language to check whether a character is a decimal digit) has no special requirements for its parameter. The mined rules for it will be often insignificant for detecting bugs. Compared with functions like $isdigit()$, we are more interested in bug-prone functions such as $strcpy()$. However, many bug-prone functions are not well-known like $strcpy()$. In many cases, they may even be undocumented. Therefore, it is necessary to develop a method that can automatically identify bug-prone functions.

In practice, a bug-prone function call usually produces an error when one or more of its parameters hold illegal values. We refer to such parameters as sensitive parameters. In a practical system, to make sure that the system works correctly, these sensitive parameters are often validated to ensure that constraints on them are satisfied before being passed to the bug-prone function. In real-world programming, a validation to a sensitive parameter 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. In other words, the conditional statement is in control of the execution of the function call statement.

A parameter p of function $f()$ is protected by a conditional statement cs if (1) the function call statement is directly or indirectly control dependent on cs and (2) p is checked by cs. A protected-counter for p (each parameter with a protected-counter) counts how many times p is protected before calling function $f()$. Assuming that function $f()$ is called for T times and the protected-counter of p is t. If the *protected-ratio* t/T is larger than a predefined threshold λ (e.g., 70 percent in this paper), function $f()$ is then considered as a bug-prone function on parameter p.

For example, in Fig. 3b, the call to $skb\_reserve()$ at line 2184 is control dependent on the conditional statement at line 2175, which checks the first actual parameter of $skb\_reserve()$ (i.e., *skb*). Therefore, $skb\_reserve()$ is taken as a bug-prone function candidate and the protected-counter of its first parameter is increased by one. After scanning the whole kernel code, we find that the function $skb\_reserve()$ is called 503 times in total, among which 491 times its first parameter is checked by some conditional statements, and thus the corresponding protected-ratio is around 97.61 percent. Consequently, with a high belief, the function $skb\_reserve()$ is identified as a bug-prone function with respect to its first parameter.

1. Common Weakness Enumeration, http://cwe.mitre.org

3.3.2 Extracting Decisive Conditions

In practical programs, a function will end the execution flow and probably return some values to its callers when certain conditions occur. For example, when $kzalloc()$ returns a NULL pointer, a function will return a specific value (e.g., "-ENOMEM") to inform its callers of the memory allocation failure. We refer to such conditions as decisive conditions. The return value is often determined (e.g., equal to *zero*, not equal to *zero*, or equal to "$-$ENOMEM") on the branch that a decisive condition occurs. In practice, not all conditions are decisive ones. For example, in Fig. 4, on the *true* branch of the conditional statement at line 319, the execution flow jumps out of the function $ioat\_dma\_self\_test()$ and a specific value (i.e., "$-$ENOMEM") is returned. Therefore, the corresponding condition "$src == 0$" is a decisive one. Whereas, there are multiple paths on the *false* branch of the same conditional statement, and the return values on these paths are not always the same. As a result, the corresponding condition "$src\,! = 0$" is not a decisive one.

Every condition can be normalized by renaming variables with canonical names (similar to Section 3.5). If a variable keeps the return value of a function $f()$, "f" is used to build the canonical name, prefixed with "$FN-$" and suffixed with "$\_0$". In other cases, a variable will be renamed with its data type. For example, the variable *src* keeps the return value of function $kzalloc()$ and its canonical name is "*FN-kzalloc_0*". As a result, the conditions "$src == 0$" and "$src\,! = 0$" are normalized to "$FN\text{-}kzalloc\_0 == 0$" and "$FN\text{-}kzalloc\_0\,! = 0$", respectively.

In practice, if the return values on the two branches of a conditional statement are the same, the conditional test will have little impact on the execution result, and thus the conditional statement is unlikely to contain decisive conditions. Moreover, according to the concept of decisive conditions, on the branch that a decisive condition evaluates to *true* (that is, the decisive condition occurs), a program should return as early as possible. Generally, there should be no other decisive conditions on that branch. Based on these insights, we adopt a three-stage method to identify decisive conditions. First, we compute a *return constraint* (short for constraint on return value) for each conditional branch to indicate possible return values on the branch. Second, we mark conditional statements whose both branches have the same return constraint. Finally, a condition *cond* is taken as a decisive condition candidate if there are no unmarked conditional statements on the branch where *cond* evaluates to *true*.

For example, in Fig. 6, on branch #1, variable *err* is negative (see the assertion at line 6), the return constraint is "<0". Similarly, the return constraints of branches #2 and #3 are "$== 0\,||== 1$" and "$== 1$", respectively. And the return constraints of branches #4$\sim$6 are the same, i.e.,

```
1  int start_device(char *dev_name, bool hint)
2  {
       /* Startup a device in two steps: register the device and then launch it
        *  Generate a success message when parameter hint is true
        *  On success, return 0; otherwise, return a non-zero value
        */
3      int err;
4      int id = get_device_id(dev_name);
5      err = register_dev(id);
6      if (err < 0) {
           // branch #1: condition "FN-register_dev_0 < 0"
7          err_log("failed to register device");
8          return err;
9      }
           // branch #2: condition "FN-register_dev_0 >= 0"
10     err = launch_dev (id);
11     if(err) {
           // branch #3: condition "FN-launch_dev _0 != 0"
12         err_log("failed to launch device");
13         return 1;
14     }
           // branch #4: condition "FN-launch_dev _0 == 0"
15     if (hint) {
           // branch #5: condition "PARM-bool != 0"
16         dev_log("succeeded to launch device");
17     }
           // branch #6: condition "PARM-bool == 0"
18     return 0;
19 }
20
21 int dev_entry(char *dev_name)
22 {
23     int err = start_device(dev_name);
24     if(err) {
25         err_log("failed to start the device");
26         return err;
27     }
       ...
25     return 0;
26 }
```

Fig. 6. An example for identifying decisive conditions and normalizing return statements.

```
1  x = get_input();
2  y = get_input();
3  len = get_length (x);
4  printf("length: %d", len);
5  len = get_length (y);
6  if (len > MAX_LEN)
7     return;
8  sensitive_op1 (x);
9  sensitive_op2 (y);
```

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

"$==0$". Because the return constraints of branches #5 and #6 are the same, the conditional statement at line 15 is unlikely to contain a decisive condition and is marked. On branches #1, #3 and #4, there is no unmarked conditional statements. Therefore, the corresponding conditions are taken as decisive condition candidates. Whereas, because there is an unmarked conditional statement (at line 11) on branch #2, the condition "$FN\text{-}register\_dev\_0\ >=0$" cannot be taken as a decisive one.

Assuming that a condition *cond* appears in T functions, and in t functions *cond* is identified as a decisive condition candidate. If the *decisive-ratio* t/T is larger than a given threshold λ (e.g., 70 percent in this paper), condition *cond* is then regarded as a decisive one. For example, in the Linux v4.9-rc3, in 234 out of 279 functions, the condition "$FN\text{-}dma\_mapping\_error\_0 != 0$" is identified as a decisive condition candidate. Its decisive-ratio is about 83.9 percent, which is larger than 70 percent. Therefore, the condition is taken as a decisive one.

3.4 Slicing Source Code

The original definition of program slicing was proposed by Weiser [56]. By introducing the notion of PDG, Ottenstein et al. [33] 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 [15], [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 at first. For a bug-prone function, vertexes that call this function in the PDG can be directly taken as slicing criteria. For a decisive condition, vertexes containing it are taken as slicing criteria, and the return constraint for each of them is also computed. For example, in Fig. 3b, the function $skb\_reserve()$ is bug-prone on its first parameter and is called at line 2184. Therefore, $\langle 2184,\ \{skb\}\rangle$ is taken as a slicing criterion, where variable *skb* is the interesting parameter. For another example, in Fig. 4, the conditional statement at line 341 contains the decisive condition "$FN\text{-}dma\_mapping\_error\_0 != 0$", and the return value on the *true* branch of the conditional statement is 0. Therefore, $\langle 341,\ \{dev,\ dma\_src\}\rangle$ is taken as a slicing criterion, and the return constraint of it is "$==0$".

Slicing for Every Criterion. In classical program slicing algorithm [33], the PDG is traversed backward from a slicing criterion, and the encountered vertexes are marked. All the marked vertexes constitute the program slice of the slicing criterion.

However, the above classical program slicing cannot be directly used by *EAntMiner*. The reason is that, the classical program slicing cannot excludes some noise statements from the obtained slice. For example, in Fig. 7, taking $\langle 8,\ \{x\}\rangle$ (the bug-prone function $sensitive\_op1()$ is called at line 8 with actual parameter x) as a slicing *criterion*, the conditional statement at line 6 (i.e., "if $(len\ >\ \mathrm{MAX\_LEN})$") remains in the obtained slice. This is because the function call is control dependent on it. But this conditional statement does validate the input to the call to $sensitive\_op2(y)$ (line 9) rather than that to $sensitive\_op1()$. If the statement is reserved in the slice, it may be incorrectly taken as a checking for $sensitive\_op1(x)$ 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 edge between them. To address this issue, we design a more aggressive slicing algorithm. Our algorithm also backward traverses the PDG paths starting from the statement invoking the bug-prone function (e.g., $sensitive\_op1()$), 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 $s1$ and $s2$ are homologous if either (1) $s1$ and $s2$ are data dependent on the same statement $s3$, or (2) $s1$ (or $s2$) is data dependent on statement $s3$, and $s2$ (or $s1$) and $s3$ are homologous. In this

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

Fig. 8. An example for illustrating statements normalizing.

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 Fig. 7, the conditional statement at line 6 (i.e., "if $(len\ >\ \mathrm{MAX\_LEN})$") is homologous to the statement at line 9, but not the statement at line 8. As a result, statement 6 is marked for the slicing criterion $\langle 9,\ \{y\}\rangle$, but is not marked for $\langle 8,\ \{x\}\rangle$.

Constructing Sub-repositories. There is a program slice for each invocation instance of a critical operation. All the program slices for a specific critical operation make up an independent sub-repository for it.

In the preliminary version [28] of this paper, to exclude statements that are irrelevant to a return statement, the program points where the return values are actually determined are taken as slicing criteria. On slicing for a criterion, all conditional statements that the slicing criterion is control dependent on are marked. But some reserved conditional statements may be irrelevant to each other, and they will interfere with code mining, resulting in false negatives and false positives. In this paper, conditional statements that contain a decisive condition are taken as slicing criteria. A decisive condition is highly related to return statements but has relatively more specific semantics. This improvement enables *EAntMiner* being able to reduce more irrelevant statements.

3.5 Normalizing and Hashing Statements

Every sub-repository is converted to an itemset database. Every statement is converted to a string and hashed to a number using the PJW hash function [3]. The hash numbers of the statements in a program slice will constitute an itemset (a bag of numbers). Before hashing, statements are normalized by the following four methods:

A. 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 re?>turn value of a function or is taken as a reference parameter of a function is renamed as the function name plus a suffix. '0' 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 Fig. 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*-0", 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".

```
skb = dev_alloc_skb(total_len + 5);
if (skb) {
       skb_reserve(skb, 2);            /* Align IP on 16 byte (TBD check this) */
       …
}
else {
       // do something
       return;
}
```

Fig. 9. Rearranged program of the one in Fig. 3b.

B. Rewriting Expressions. Expressions in different forms may represent the same semantics. For example, "$a + b$" is equivalent to "$b + a$" in semantics. It is not practical to normalize all kinds of semantics-equivalent expressions. 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 normalizing 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 canonical 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, the expression "$int\ \ >= char$" is rewritten as "$char\ \ <= int$".

C. Rearranging Control Structures. The same program logic may be implemented in different control structures. For example, programs in Figs. 3a and 3b 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 $\boldsymbol{p}$ evaluates to *false*, it is negated to $\boldsymbol{p}'$ (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 $\boldsymbol{p}'$ evaluates to *true*. In this way, the validation modes about critical operations are unified without alerting the original validation logic. For example, in Fig. 3b, the critical operation $skb\_reserve()$ is executed only when the predicate (line 2175) evaluates to *false*. The related control structure is rearranged, as shown in Fig. 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.

D. Abstracting return values. In practice, some different return values may have the same implications from a high-level point of view. For example, in Fig. 6, the function $start\_device()$ returns a negative value on registration error and returns "1" when the launch operation fails. However, both the negative value and "1" are interpreted as errors and are handled in the same way (see lines $24 \sim 27$) in the function $dev\_entry()$, a caller of $start\_device()$. Therefore, when the register operation fails, it is also acceptable to return "1", without leading to any bugs.

We abstract return values according to the manner they are checked in the callers to transform semantics-equivalent ones into the same form as far as possible. For example, in Fig. 6, the conditional statement "if (err)" (at line 24) checks the return value of function $start\_device()$. The return values in function $start\_device()$ can thus be abstracted as "$! = 0$" or "$== 0$". The return constraint of branch #1 is "< 0", under which "$! = 0$" is always satisfied. Therefore, "< 0" can be abstracted as "$! = 0$". Similarly, the return value on branches $4 \sim 6$ equals to "0", and they can be normalized to "$== 0$". Whereas, the return constraint of branch #3 is "$== 1 \,||\, == 0$", under which neither "$! = 0$" nor "$== 0$" can always be satisfied. In this case, the return constraints cannot be further abstracted and will keep their original forms.

Note that a function may be called multiple times, and its return values may be checked in different ways, e.g., "if $(rv\,! = 0)$" or "if $(rv < 0)$". In this case, a negative return value will be normalized to a couple of different forms, such as "$! = 0$" and "< 0".

As discussed in Section 3.4, a return constraint is computed for each slicing criterion of decisive conditions. The return constraint is abstracted with the above method. The program slice of a slicing criterion is transformed into an itemset, say X; the corresponding abstracted return values are also stored in the database along with the itemset X, denoted as $rvs(X)$.

Note that, *AntMiner* [28] normalizes program statements by properly renaming variables, rewriting expressions, and rearranging control structures. However, these strategies cannot effectively normalize return statements. In this paper, by incorporating inter-procedural information to abstract return values, *EAntMiner* normalizes return statements into canonical forms as far as possible.

3.6 Detecting Violations

EAntMiner employs different detection methods to identify inappropriate invocations of bug-prone functions and incorrect return values. On the database of a bug-prone function, as done in *AntMiner*, *EAntMiner* first extracts association rules from it, and then reports itemsets that violate the extracted rules as potential bugs. On detecting return value bugs, *EAntMiner* adopts a method different to that in *AntMiner*. *EAntMiner* iterates over each itemset in the database of a decisive condition. It selects a set of itemsets (called *neighbors*) that are highly similar to the one under consideration. An itemset is taken as an outlier if it has different (abstracted) return values with its neighbors, and the corresponding function potentially contains a return value bug.

3.6.1 Detecting Misusages of Bug-Prone Functions

EAntMiner adopts the data mining algorithm *FPclose* [17] to discover *closed frequent sub-itemsets* from the itemset database of a bug-prone function. We first present the background of association rules to ease our presentation in the next two paragraphs.

The *support* of a sub-itemset P is the number of itemsets in a database that contain all items in P, denoted as *support* (P). A sub-itemset is frequent if its support is larger than or equal to a specified threshold (known as *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 $support(A) = support(B)$.

Given a threshold *min_confidence*, an association rule is defined to be the form $A => B$, where A and B are two closed frequent sub-itemsets, and $support(B) \div support(A) \times 100$ percent (i.e., the *confidence* of the rule, denoted as $confidence(A => B)$) is larger than or equal to *min_confidence*. The association rule $A => B$ indicates: if an itemset in the database contains all items in A, it should also contain all items in B with a probability of $confidence(A => B)$. And a violation to the rule is an itemset that contains all the items in A but not all the items in B.

A trivial method to detect the violations is to enumerate 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, the set of violations to an association rule $A => B$ can be obtained via $supporter(A) - \mathrm{supporter(B)}$.

Ranking potential bugs. Violations are ranked before reporting them to the users. In our experience, a violation that misses 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. Hence, all violations are first arranged 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).

3.6.2 Detecting Incorrect Return Values

EAntMiner leverages the k Nearest Neighbors algorithm (kNN) to identify abnormal return values on the itemset database for a decisive condition. kNN is an instance-based learning method for classification. For an instance, the expected label is the most frequently appeared one among the k neighbors that are most similar to it. The k instances are selected according to their similarities with the one under consideration from high to low.

For an itemset X in the database of a decisive condition, the corresponding abstracted return values $rvs(X)$ can be viewed as class labels of it. An itemset is taken as a potential bug if it is mislabeled; that is, its class labels are different with the expected ones. The expected labels are obtained by kNN. In this way, the incorrect return value identification problem is converted to a classification problem. The k nearest neighbors of an itemset X are itemsets in the database that has highest similarities with X, denoted as $k\_neigs(X)$. The similarity between two itemsets A and $B\,(A \neq B)$ is computed by

$$sim\ (A, B) = 0.9 * J(A,\ B) + 0.1 * sim_{\text{file}}, \tag{1}$$

where $J(A,\ B)$ is the Jaccard similarity coefficient between the two itemsets A and B, and sim_{file} is the similarity between the file paths in which the functions corresponding to A and B are defined. $J(A,\ B)$ actually reflects the degree of similarities between the corresponding program slices, and contributes most to the final similarity. Whereas, sim_{file} is used for fine-tuning the similarity such that functions defined adjacently will have slightly higher similarities and can be preferentially selected.

The average similarity between X and its k\text{-}nearest neighbors is computed by

$$av\ g_{sim(X,k)} = \frac{1}{k}\sum_{j}^{k} sim\big(X,\ X_j\big) \tag{2}$$

where X_j is one of the k neighbors of X.

An itemset X is said to be labeled with rv if rv belongs to $rvs(X)$. Let $support(rv)$ indicate the number of itemsets among the k nearest neighbors of the itemset X that are labeled with rv. And $confidence(rv) = support(rv)/k$ will indicate how much we believe the itemset X should also be labeled with rv. The itemset X is probably mislabeled if $confidence(rv)$ is not less than a predefined threshold $min\_confidence$ but rv does not belong to $rvs(X)$. And X is taken as a potential bug with confidence $confidence(X) = confidence(rv)$.

In kNN, if the parameter k is 1, only the nearest neighbor will be considered. However, a common practice in information retrieval [7] suggests to use a few nearest neighbors. A larger k can guarantee that the classification result is more convincible. However, more neighbors do not always lead to better results. Along with the increasing number of neighbors, the average similarity between X and its neighbors decreases. As a result, the classification result may become less reliable. Because the sizes of different databases may vary greatly, a fixed k for all databases is less than ideal. For this reason, we compute an appropriate k according to the database size. We introduce three parameters that need to be specified by users: w, min_k, and max_k. The value of parameter w is from 0 to 1 (e.g., 0.2), and min_k and max_k are two integers. Let N be the number of itemsets in a database D. If $w * N$ is less than min_k, we set k to min_k; if $w * N$ is larger than max_k, we set $k = max\_k$; otherwise, $k = w * N$.

Ranking potential bugs. The $confidence(X)$ implies to what extent we believe X is a real bug, while $avg\_sim(X, k)$ implies how reliable the detecting result is. *EAntMiner* assigns a score to every potential bug. The score for X can be computed by

$$score(X) = confidence(X) * avg\_sim(X, k). \tag{3}$$

The detected potential bugs are ranked by their scores in a descend order.

In the preliminary version [28] of this paper, to detect return value bugs, *AntMiner* first mines closed frequent patterns as done in Section 3.6.1. And it then infers the association rules like $P => \{\text{"return}\,v;\text{"}\}$, where P is a closed frequent pattern, and "$\text{return}\,v;$" is the most frequently appeared return statement among itemsets that contain the pattern P. An itemset that contains P but the corresponding return value is not v is regarded as a potential bug. However, as discussed in Section 2.3, this method is susceptible to semantics inequivalent but form identical return statements, and thus may miss some interesting rules, resulting in false negatives. In essence, the more similar two itemsets are, the more likely they will obey the same return rule. However, itemsets that contain the same pattern may be very different from each other. In this paper, *EAntMiner* uses the instance-based method to address the above issues as explained in this subsection.

4 EVALUATION

4.1 Experiment Setup

We implemented *EAntMiner* on the top of the GCC compiler [42] (V4.5.0) and mainly evaluated it on two versions of Linux kernel: v2.6.39 and v4.9-rc3. The preliminary version [28] of this method, *AntMiner*, is evaluated on the Linux v2.6.39. The Linux v4.9-rc3 is the latest version on evaluating *EAntMiner*. In our experiments, both *EAntMiner* and *AntMiner* ran on a machine with a Core i5-2520M, 2.5 GHZ Intel processor and 4 GB memory. The source code for X86 were scanned: 8,042 C files in the Linux v2.6.39, and 15,027 C files in the Linux v4.9-rc3.

The Linux kernel is one of large-scale systems that have been well analyzed by dozens of bug detection tools including [14], [20], [25], [26], [27], [31], [46], [47], [65]. Such a large-scale and well-analyzed system may still contain both kinds of bugs that can be easily detected and bugs that are extremely difficult to be detected. Hence, on such a system, the effectiveness of a bug analysis approach can be shown.

To illustrate the fact that our apporach is not special for the Linux kernel, we also evaluated *EAntMiner* on three other large-scale popular C systems in different fields: OpenSSL v1.1.0g, FFmpeg v3.4, and PostgreSQL v10.1. OpenSSL is an implementation of TLS and SSL protocols. FFmpeg is a framework to play and convert videos and audios and PostgreSQL is one of the most widely used open source databases. Many bug detection oriented methods select them as the target of evaluations [23], [24], [27], [35], [62].

On detecting bugs caused by misusages of bug-prone functions, three parameters need to be specified: λ, *min_support*, and $min\_confidence$. The parameter λ is used to identify bug-prone functions. The precision (i.e., the likelihood of an extracted function to be bug-prone) is higher if we set a larger value for λ. However, with a larger value of λ, some real bug-prone functions may be missed, resulting in false negatives. Similarly, with larger values of *min_support* and $min\_confidence$, we gain lower false positive rate but higher false negative rate. By default, we set λ to 70 percent, $min\_support$ to 10, and *min_confidence* to 85 percent, respectively.

On detecting bugs caused by incorrect return values, five parameters need to be specified: λ, $min\_confidece$, w, min_k and max_k. The former two parameters have the same meaning with those in detecting bugs for bug-prone functions, and we set them to 70 percent and 85 percent,

TABLE 1
Statistics of *EAntMiner* on Extracting Critical Operations

kernel version	Functions		Conditions		Total	Time (m)
	#All	#Bug-prone	#All	#Decisive		
v2.6.39	6,314	1,984	4,616	1,381	3,365	80
v4.9-rc3	9,758	3,325	8,112	2,557	5,882	140

respectively. Higher values for w, min_k and max_k will produce fewer false positives but more false negatives. By default, we set w to 0.2, min_k to 5, and max_k to 20.

In essence, the above parameters allow users to control false positive rate and false negative rate. Users can tune them during reviewing the reports. In practice, users can empirically determine reasonable parameter settings by performing a sampling analysis to the results.

4.2 Experiments on the Linux Kernel

We first evaluated the effectiveness of the statistics-based method to automatically extract critical operations (Section 4.2.1). Then, we evaluated the ability of *EAntMiner* on detecting real bugs (Section 4.2.2). Finally, we conducted a comparative analysis to demonstrate the improvements compared to the preliminary work [28] and highlight the benefits of reducing noise interferences (Section 4.2.3).

4.2.1 Extracting Critical Operations

Overall. Table 1 shows the statistics of *EAntMiner* on extracting critical operations on both versions of Linux Kernel, including the number of functions called more than 10 times and the number of bug-prone ones, the number of conditions that appear more than 10 times and the number of decisive ones, the total number of critical operations, and the taken time. From Table 1, *EAntMiner* extracted 5,885 critical operations from the Linux v4.9-rc3 in about 140 minutes including 3,328 bug-prone functions and 2,557 decisive conditions; and it extracted 3,365 critical operations from the Linux v2.6.39 in about 80 minutes.

In the Linux v4.9-rc3, there are 9,758 functions and 8,112 conditions appear more than 10 times. We performed a systematic sampling analysis to estimate the recall and precision of our method. The sampling starts from a random point in the sample space, and selects one at the interval of every i samples, where $i = 25$ is known as the sampling interval.

Recall. We systematically selected 390 functions from all the 9,758 functions, and 325 conditions from all the 8,112 conditions. We manually reviewed these functions and conditions. A function is marked with "critical" if its parameters should be protected to avoid bugs, and a condition is marked with "critical" if the program should return once it occurs. To avoid human bias, the protected-ratios and decisive-ratios were invisible during reviewing. Finally, 98 functions and 101 conditions were marked with "critical". Among them, 84 functions and 87 conditions were identified by *EAntMiner*. As a result, the recall of our method to identify bug-prone functions and decisive conditions is about 85.9 percent.

Precision. We also systematically selected 133 samples from the extracted 3,328 bug-prone functions, and 103 samples from the extracted 2,557 decisive conditions. After a manual analysis, 86 functions and 85 conditions were confirmed to be real critical operations. The average precision of our method is 72.5 percent. Particularly, it achieves a precision of 82.5 percent in identifying decisive conditions. We further studied the reason why bug-free functions are misclassified as bug-prone ones by our method. We found that the most common reason is that the parameters of a function are validated at the beginning of the function itself, but are checked again in many callers for the sake of safety. For example, function $brelse()$ decrements the reference count of a *buffer_header* structure. Before calling it, its parameter is checked against NULL in most cases (87.5 percent: 706 out of 807). Naturally, *EAntMiner* marked it as "bug-prone". However, at the beginning of the function $brelse()$, the parameter is validated and the decrement operation will not be performed if it is illegal. Therefore, function $brelse()$ will not cause any bug even if it takes a null parameter.

Threat to Validity. We note that the manual classification of functions and conditions is somewhat subjective. Human mistakes may affect the accuracy of the obtained recall and precision. Despite that, the result of the sampling analysis shows that our method can reduce a large amount of functions and conditions that we are not interested in at the cost of missing a small number of interesting ones (about 14.1 percent). For example, the number of functions that act as slicing criteria in our evaluation on the Linux v4.9-rc3 decreases from 9,758 to 3,328. Non-critical operations are often weakly related to bugs as discussed in Section 3.3. Many false positives can thus be reduced by distinguishing critical operations from non-critical ones.

4.2.2 Detecting Bugs

On the Linux v4.9-rc3, *EAntMiner* ran about 245 minutes to generate mining databases for the 5,885 critical operations and detect violations from them. It detected 7,532 violations. On the Linux v2.6.39, *EAntMiner* ran about 120 minutes in total, and detected 4,246 violations.

Similar to all other static analysis tools, e.g., [14], [27], violations reported by *EAntMiner* also need to be inspected manually. As presented in Section 3.6, violations are sorted in a descending order by the likelihood of being real bugs. Users can inspect them in order, and stop the inspection task when the false positive rate is too high, e.g., no new real bugs are identified within 20 violations.

The Linux v2.6.39 was released in May 2011. The preliminary tool *AntMiner* was mainly evaluated on it [28]. In this paper, we took it as a baseline to verify whether *EAntMiner* can cover the ability of *AntMiner* in detecting bugs. *AntMiner* detected 24 previously unknown bugs, including 18 misusages of bug-prone functions and 6 incorrect return values. These bugs are listed in Table 2, marked with IDs 1$\sim$24. We examined the detecting result of *EAntMiner*, and found that all of these 24 bugs are detected. It indicates that the enhanced approach, *EAntMiner*, incorporates the bug detection ability of *AntMiner*.

One of us spent about 20 hours to inspect the detection results on the Linux v4.9-rc3. The cost of manual work is acceptable on large-scale systems like the Linux kernel, which have more than 10 million lines of code. Eventually, we found that 96 violations are suspected bugs, including 33 potential misusages of bug-prone functions and 63 potential

TABLE 2
Previously Unknown Bugs in the Linux kernel Detected by Our Tools

ID	Bugzilla ID	Function	Critical Operation	Kernel Version	*EAntMiner*	*AntMiner*	*AntMiner—*	*Cov*
1	44431	st_int_recv()	skb_reserve()	v2.6.39	✓	✓	×	✓
2	44441	ldisc_open()	register_netdevice()	v2.6.39	✓	✓	✓	×
3	44461	sfb_dump()	nla_nest_end()	v2.6.39	✓	✓	✓	✓
4	44471	tmiofb_probe()	ioremap()	v2.6.39	✓	✓	✓	×
5	44491	setup_isurf()	pnp_port_start()	v2.6.39	✓	✓	×	×
6	44541	lx_pcm_create()	snd_pcm_set_ops()	v2.6.39	✓	✓	×	×
7	44551	poseidon_audio_init()	snd_pcm_set_ops()	v2.6.39	✓	✓	✓	×
8	44561	pcf50633_probe()	platform_device_add()	v2.6.39	✓	✓	×	✓
9	44571	dcbnl_ieee_set()	nla_parse_nested()	v2.6.39	✓	✓	×	×
10	44621	cgroupstats_user_cmd()	nla_data()	v2.6.39	✓	✓	×	×
11	44671	ocfs2_create_refcount_tree()	ocfs2_set_new_buffer_uptodate()	v2.6.39	✓	✓	×	×
12	44681	ocfs2_create_xattr_block()	ocfs2_set_new_buffer_uptodate()	v2.6.39	✓	✓	×	×
13	44691	lkdtm_debugfs_read()	free_pages()	v2.6.39	✓	✓	×	×
14	49851	ipw_packet_received_skb()	skb_reserve()	v2.6.39	✓	✓	×	✓
15	49861	wl1271_debugfs_update_stats()	wl1271_ps_elp_sleep()	v2.6.39	✓	✓	×	×
16	49871	omninet_read_bulk_callback()	tty_flip_buffer_push()	v2.6.39	✓	✓	×	×
17	49911	moxa_new_dcdstate()	tty_hangup()	v2.6.39	✓	✓	×	×
18	49921	btree_write_block()	logfs_put_write_page()	v2.6.39	✓	✓	✓	✓
19	96741	atl2_probe()	FN-pci_set_consistent_dma_mask_0 ! = 0	v2.6.39	✓	✓	*NA*	*NA*
20	98551	mptfc_probe()	FN-__alloc_workqueue_key_0 = = 0	v2.6.39	✓	✓	*NA*	*NA*
21	98561	mkiss_open()	FN-register_netdev_0 ! = 0	v2.6.39	✓	✓	*NA*	*NA*
22	98611	r592_probe()	FN-request_irq_0 ! = 0	v2.6.39	✓	✓	*NA*	*NA*
23	98621	sd_probe()	FN-device_add_0 ! = 0	v2.6.39	✓	✓	*NA*	*NA*
24	99011	myri10ge_probe()	FN-dma_alloc_coherent_0 = = 0	v2.6.39	✓	✓	*NA*	*NA*
25	195491	snvs_rtc_probe()	devm_clk_get()	v4.9-rc3	✓	✓	×	×
26	195495	lwtunnel_fill_encap()	nla_nest_end()	v4.9-rc3	✓	✓	✓	✓
27	195501	mt7601u_mcu_msg_alloc()	skb_reserve()	v4.9-rc3	✓	✓	✓	✓
28	195503	tipc_nl_node_get_monitor()	nlmsg_free()	v4.9-rc3	✓	✓	✓	✓
29	195505	team_nl_send_port_list_get()	nlmsg_free()	v4.9-rc3	✓	✓	×	×
30	195507	team_nl_send_options_get()	nlmsg_free()	v4.9-rc3	✓	✓	×	×
31	195509	isp1704_charger_probe()	devm_gpio_request_one()	v4.9-rc3	✓	✓	×	×
32	195511	send_fw_pass_open_req()	__skb_put()	v4.9-rc3	✓	✓	✓	✓
33	195513	s5k83a_start()	wake_up_process()	v4.9-rc3	✓	✓	×	×
34	195515	pc300_pci_init_one()	request_irq()	v4.9-rc3	✓	✓	×	✓
35	195517	apci3xxx_auto_attach()	request_irq()	v4.9-rc3	✓	✓	×	×
36	195527	WILC_WFI_mon_xmit()	netif_rx()	v4.9-rc3	✓	✓	✓	×
37	195529	qlcnic_sriov_virtid_fn()	pci_read_config_word()	v4.9-rc3	✓	✓	✓	×
38	195533	intel_soc_pmic_i2c_probe()	regmap_add_irq_chip()	v4.9-rc3	✓	✓	×	×
39	195535	gemini_rtc_probe()	devm_request_irq()	v4.9-rc3	✓	✓	×	×
40	195543	if_spi_probe()	destroy_workqueue()	v4.9-rc3	✓	✓	✓	×
41	195545	rndis_wlan_bind()	destroy_workqueue()	v4.9-rc3	✓	✓	✓	×
42	195547	r420_cp_errata_init()	radeon_ring_unlock_commit()	v4.9-rc3	✓	✓	✓	✓
43	195549	r420_cp_errata_fini()	radeon_ring_unlock_commit()	v4.9-rc3	✓	✓	✓	✓
44	195551	radeon_test_create_and_emit_fence()	radeon_ring_unlock_commit()	v4.9-rc3	✓	✓	×	✓
45	188441	nbd_init()	FN-alloc_disk_0 = = 0	v4.9-rc3	✓	×	*NA*	*NA*
46	188451	lbs_cmd_802_11_sleep_params()	FN-__lbs_cmd_0 ! = 0	v4.9-rc3	✓	×	*NA*	*NA*
47	188521	skcipher_recvmsg_async()	FN-kcalloc_0 = = 0	v4.9-rc3	✓	×	*NA*	*NA*
48	188531	mtip_block_initialize()	FN-ida_pre_get_0 = = 0	v4.9-rc3	✓	×	*NA*	*NA*
49	188561	wm831x_clkout_is_prepared()	FN-wm831x_reg_read_0 < 0	v4.9-rc3	✓	✓	*NA*	*NA*
50	188591	ioat_dma_self_test()	FN-dma_mapping_error_0 ! = 0	v4.9-rc3	✓	×	*NA*	*NA*
51	188601	ioat_xor_val_self_test()	FN-dma_mapping_error_0 ! = 0	v4.9-rc3	✓	×	*NA*	*NA*
52	188611	extcon_sync()	FN-get_zeroed_page_0 = = 0	v4.9-rc3	✓	×	*NA*	*NA*
53	188621	kfd_wait_on_events()	FN-copy_from_user_0 ! = 0	v4.9-rc3	✓	✓	*NA*	*NA*
54	188631	vc4_cl_lookup_bos()	FN-drm_malloc_ab_0 = = 0	v4.9-rc3	✓	×	*NA*	*NA*
55	188641	cm3232_reg_init()	FN-i2c_smbus_write_byte_data_0 < 0	v4.9-rc3	✓	×	*NA*	*NA*
56	188651	sram_reserve_regions()	FN-devm_kstrdup_0 = = 0	v4.9-rc3	✓	×	*NA*	*NA*
57	188661	bnxt_hwrm_stat_ctx_alloc()	FN-_hwrm_send_message_0 ! = 0	v4.9-rc3	✓	×	*NA*	*NA*
58	188691	sci_request_irq()	FN-kasprintf_0 = = 0	v4.9-rc3	✓	×	*NA*	*NA*
59	188701	xhci_mtk_probe()	FN-platform_get_irq_0 < 0	v4.9-rc3	✓	×	*NA*	*NA*
60	188711	wusb_dev_sec_add()	FN-krealloc_0 = = 0	v4.9-rc3	✓	✓	*NA*	*NA*
61	188721	xenstored_local_init()	FN-get_zeroed_page_0 = = 0	v4.9-rc3	✓	×	*NA*	*NA*
62	188731	btrfs_uuid_tree_iterate()	FN-btrfs_alloc_path_0 = = 0	v4.9-rc3	✓	×	*NA*	*NA*
63	188751	caif_sktinit_module()	FN-sock_register_0 ! = 0	v4.9-rc3	✓	×	*NA*	*NA*
64	188761	load_asic()	FN-load_asic_generic_0 < 0	v4.9-rc3	✓	×	*NA*	*NA*
65	188771	lan78xx_probe()	FN-usb_alloc_urb_0 = = 0	v4.9-rc3	✓	×	*NA*	*NA*
66	188781	br_sysfs_addbr()	FN-kobject_create_and_add_0 = = 0	v4.9-rc3	✓	×	*NA*	*NA*
67	188791	lanai_dev_open()	FN-ioremap_0 = = 0	v4.9-rc3	✓	×	*NA*	*NA*
68	188801	bdisp_debugfs_create()	FN-debugfs_create_dir_0 = = 0	v4.9-rc3	✓	×	*NA*	*NA*
69	188811	lstcon_group_info()	FN-copy_to_user_0 ! = 0	v4.9-rc3	✓	×	*NA*	*NA*
70	188821	c4iw_rdev_open()	FN-__get_free_pages_0 = = 0	v4.9-rc3	✓	×	*NA*	*NA*
71	188831	ocrdma_mbx_create_ah_tbl()	FN-dma_alloc_coherent_0 = = 0	v4.9-rc3	✓	×	*NA*	*NA*
72	188841	typhoon_init_one()	FN-register_netdev_0 < 0	v4.9-rc3	✓	×	*NA*	*NA*
73	188871	ad7150_write_event_config()	FN-i2c_smbus_read_byte_data_0 < 0	v4.9-rc3	✓	×	*NA*	*NA*
74	188881	dcbnl_cee_fill()	FN-nla_nest_start_0 = = 0	v4.9-rc3	✓	×	*NA*	*NA*
75	188891	public_key_verify_signature()	FN-kmalloc_0 = = 0	v4.9-rc3	✓	×	*NA*	*NA*
76	188901	cpuidle_add_state_sysfs()	FN-kzalloc_0 = = 0	v4.9-rc3	✓	×	*NA*	*NA*
77	188911	qxl_release_alloc()	FN-kmalloc_0 = = 0	v4.9-rc3	✓	×	*NA*	*NA*

TABLE 2
(*Continued*)

ID	Bugzilla ID	Function	Critical Operation	Kernel Version	*EAntMiner*	*AntMiner*	*AntMiner—*	*Cov*
78	188921	hid_post_reset()	FN-kmalloc_0 = = 0	v4.9-rc3	✓	×	*NA*	*NA*
79	188931	hfc4s8s_probe()	FN-__request_region_0 = = 0	v4.9-rc3	✓	×	*NA*	*NA*
80	188941	beiscsi_create_cqs()	FN-pci_alloc_consistent_0 = = 0	v4.9-rc3	✓	×	*NA*	*NA*
81	188951	beiscsi_create_eqs()	FN-pci_alloc_consistent_0 = = 0	v4.9-rc3	✓	×	*NA*	*NA*
82	188961	mvs_task_prep()	FN-dma_pool_alloc_0 = = 0	v4.9-rc3	✓	×	*NA*	*NA*
83	188971	irda_usb_probe()	FN-kzalloc_0 = = 0	v4.9-rc3	✓	×	*NA*	*NA*
84	188981	klsi_105_open()	FN-usb_control_msg_0 < 0	v4.9-rc3	✓	×	*NA*	*NA*
85	189011	meye_probe()	FN-__request_region_0 = = 0	v4.9-rc3	✓	×	*NA*	*NA*
86	189021	eni_do_init()	FN-ioremap_nocache_0 = = 0	v4.9-rc3	✓	✓	*NA*	*NA*
87	189031	mlx4_ib_query_device()	FN-kzalloc_0 = = 0	v4.9-rc3	✓	×	*NA*	*NA*
88	189041	qed_ll2_start_xmit()	FN-dma_mapping_error_0 ! = 0	v4.9-rc3	✓	×	*NA*	*NA*
89	189071	toggle_ecc_err_reporting()	~FN-zalloc_cpumask_var_0 ! = 0	v4.9-rc3	✓	×	*NA*	*NA*
90	189091	__wa_xfer_setup_segs()	FN-kmalloc_0 = = 0	v4.9-rc3	✓	×	*NA*	*NA*
91	189111	add_grefs()	FN-alloc_pages_0 = = 0	v4.9-rc3	✓	×	*NA*	*NA*
92	189121	wa_nep_queue()	FN-kzalloc_0 = = 0	v4.9-rc3	✓	×	*NA*	*NA*
93	189141	bnx2x_init_firmware()	FN-kmalloc_0 = = 0	v4.9-rc3	✓	✓	*NA*	*NA*

incorrect return values. We reported them to the kernel Bugzilla (the kernel development community).[2] Because both association rules and k\text{-}nearest neighbors can suggest the best way to fix a bug, we also wrote patches for the suspected bugs, and committed them to the Linux kernel maintainers.[3] Up to now, 69 of the patches, including 20 for misusages of bug-prone functions and 49 for incorrect return values, have been accepted and merged into the Linux kernel mainline (see bugs 25~93 in Table 2). Among the rest 27 violations, only six are directly identified as false positives by kernel developers, and the others have received no response from the Linux kernel maintainers and thus their statuses are unknown. Table 3 summarizes the 96 bugs.

4.2.3 Comparative Analysis

We further applied other four approaches *AntMiner*, *AntMiner—*, *Cov* and *Coccinelle*[4] on the Linux kernel v2.6.39 and v4.9-rc3 to see whether they can detect the previously unknown bugs found by *EAntMiner*. *AntMiner* is the preliminary version of our method. *AntMiner—*, as suggested by its name, is based on AntMiner but without program slicing and statement normalization. We also applied a widely used static analysis tool on the Linux kernel. But according to the user agreement, we cannot name it in the comparison. We refer to it as *Cov*. *AntMiner—* and *Cov* do not support the detection of return value bugs. As shown in Table 2, we use the symbol ✓ and × to indicate whether a bug is detected or not by the corresponding approach. Besides, we use "*NA*" to indicate the case that an approach is not applicable to detect a bug. Table 4 counts the number of real bugs detected by each tool. Finally, we compared *EAntMiner* with *Coccinelle*, a static analysis tool that detects bugs according to user-specified matching patterns, to show whether rules mined by *EAntMiner* can be used to improve the bug-detection ability of pattern-based static analysis tools.

EAntMiner versus *AntMiner*. *EAntMiner* detected 69 previously unknown bugs from the Linux v4.9-rc3. *AntMiner* also detected the 20 misusages of bug-prone functions. However, it missed 44 return value bugs. For example, *AntMiner* failed to detect the bug (ID: 50) shown in Fig. 4. The reason has been explained in Section 2.3. The comparison shows that about 89.8 percent of return value bugs were missed without abstracting return values and applying the kNN algorithm to identify anomalies. Therefore, *EAntMiner* performs significantly better than *AntMiner* in detecting bugs that return incorrect values under certain decisive conditions.

*EAntMiner versus AntMiner—*In total, *EAntMiner* detected 38 misusages of bug-prone functions. *AntMiner—*detected only 15 of them. That is, about 60.5 percent of bugs were missed. For example, the bugs (IDs: 6 and 17) shown in Figs. 1 and 2b were detected by *EAntMiner* and *AntMiner* but were missed by *AntMiner–* (the reason was explained in Section 2.1 and Section 2.2). Note that, theoretically, EAntMiner may fail to report some bugs detected by *AntMiner—*if no proper critical operations can be extracted; however, we have not encountered such a case in the experiments. That means, all (confirmed) bugs detected by *AntMiner–* were detected by *EAntMiner*.

Without loss of generality, we selected the Linux v2.6.39 as the target to evaluate the effectiveness of *EAntMiner* on

TABLE 3
Classification of Violations Detected in Linux-4.9-rc3

# of total violations: 96	69 (~72%) Confirmed as unknown bugs.	Real bugs
	6 (~6%) Regarded as false positives.	False positives
	21 (~22%) Waiting for confirmation.	Unknown

TABLE 4
Number of Bugs Detected by Each Tool

Tool	Linux-v2.6.39		Linux-v4.9-rc3		Total
	fun	*ret*	*fun*	*ret*	
EAntMiner	18	6	20	49	93
AntMiner	18	6	20	5	49
AntMiner—	5	*NA*	10	*NA*	15
Cov	5	*NA*	8	*NA*	13

"fun" means misusages of bug-prone functions, and "ret" means return value bugs. "NA" means the tool does not support this kind of bugs.

2. Bugzilla for the Linux kernel, https://bugzilla.kernel.org
3. The Linux kernel Patchwork, https://patchwork.kernel.org
4. Coccinelle, http://coccinelle.lip6.fr

TABLE 5
Accuracy of Extracted Rules

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

reducing false alarms caused by irrelevant statements and inconsistent implementations. We further analyzed the mined rules for the 21 bug-prone functions for which there are real bugs detected by *EAntMiner*. As shown in Table 5, *AntMiner–* mined 2,159 rules related to these bug-prone functions. For each bug-prone function, its top ranked rules (at most 10) were manually verified to see whether they are interesting. A rule is interesting if it should be followed, and if violated, bugs may occur. For example, AntMiner– mined 30 rules related to the bug-prone function $nla\_nest\_end()$. We inspected the top 10 of these 30 rules manually, and found only one of them was an interesting rule. In total, 149 rules were inspected, and only 18 of them were confirmed to be interesting ones. The false positive rate is up to 87.9 percent. In most cases, we found that elements of an uninteresting rule are irrelevant or weakly relevant to each other. Violations to such rules are always false alarms. From Table 5, we see that the false positive rate is heavily reduced by applying our method, i.e., 24.5 percent. At the same time, more correct rules were extracted by EAntMiner (i.e., 80 vs. 18 by AntMiner–).

The comparison between EAntMiner and AntMiner– shows that, by elaborately removing irrelevant statements and normalizing semantics-equivalent statements as far as possible, a large number of false positives and false negatives can be reduced.

EAntMiner versus *Cov*. *Cov* can also automatically infer implicit programming rules for unmodeled functions (e.g., $alloc\_skb()$) to detect related bugs. For example, the *NULL_ RETURNS* checker can infer the rule "*the return value of function alloc_skb() should be checked against NULL before dereferencing*" by scanning the code and computing how frequently the return value of $alloc\_skb()$ is checked against NULL. According to the rule, Cov can detect a real bug (Bugzilla ID: 44431) in function $st\_int\_recv()$. However, because Cov directly infers programming rules from the original source code, its precision is heavily interfered by the noise statements (as discussed in Sections 2.1 and 2.2). As a result, some interesting rules and related bugs may be neglected. For example, it missed about 65.8 percent (25 out of 38) real bugs caused by misusing bug-prone functions in Linux kernel, which should have been detected by its corresponding checkers (e.g., the $NULL\_RETURNS$ checker).

EAntMiner versus Coccinelle. *Coccinelle* is a tool that matches and transforms source code according to patterns written in SmPL (Semantic Patch Language) [34]. It is widely used by kernel developers and maintainers to detect potential bugs and to generate patches for the confirmed bugs. We ran *Coccinelle* on the Linux kernel with the latest public available rules for Linux (defined in the directory linux/scripts/coccinelle). We manually reviewed the reports of *Coccinelle* and found that none of the bugs in Table 2 is detected by *Coccinelle* due to lack of rules. We then manually converted rules mined by *EAntMiner* that the bugs 1~18 (i.e., bugs in the Linux v2.6.39 that misuse bug-prone functions) in Table 2 violate to the form that *Coccinelle* can accept and ran *Coccinelle* again. This time, *Coccinelle* successfully detected the above 18 bugs. This experiment demonstrates that the bug detection ability of pattern-based static analysis tools (e.g., *Coccinelle*) can be improved by integrating rules extracted by EAntMiner.

TABLE 6
Experiments on Three Large-scale C Systems

System	LoC	Time	Violations			
			#All	#Audit	#Suspected	#True
OpenSSL	269,355	46 m	457	20	3	3
FFmpeg	851,191	30 m	439	40	8	7
PostgreSQL	864,530	41 m	1,331	20	5	2

4.3 Experiments on Other Targets

We applied EAntMiner to OpenSSL v1.1.0g, FFmpeg v3.4, and PostgreSQL v10.1. Table 6 lists lines of C source code in a system (LoC), the time overhead (*Time*) to scan it, and the number of violations detected by EAntMiner (*#All*).

We manually audited the top ranked violations (see column *#Audit*) in each project as done in experiments on the Linux kernel and found 16 suspected bugs in total. We reported the suspected bugs to corresponding communities for bug reports. Finally, 12 of them are confirmed to be real bugs by developers or maintainers and have been fixed in the latest versions (see column #*True* for more details). The rest four suspected bugs are false positives when considering their contexts.

The 12 confirmed bugs are listed in Table 7, with the *reference ID* to retrieve the report of the bug, the name of the *function* that contains the bug, and the corresponding *critical operation* (a bug-prone function or a decisive condition). Six out of the 12 bugs are caused by misusing some bug-prone functions, and the other six result from returning incorrect values under certain conditions.

4.4 Summary

The evaluation shows that the enhanced method *EAntMiner* successfully found more than one hundred bugs from four large-scale systems. The comparative analysis well demonstrated that *EAntMiner* could detect a number of subtle bugs that are difficult to be found by other tools, e.g., *Cov*. In particular, the comparison with *AntMiner* shows that the improvements significantly improves the effectiveness of the approach presented in the preliminary version [28] of this paper.

5 Discussion

While the method proposed in this paper is effective in revealing bugs that may be missed previously, there are still some limitations that we need to consider in our future studies.

Critical Operations. Currently, *EAntMiner* mainly concerns two types of critical operations. However, other operations may also be critical to detecting bugs, such as reading or overwriting some fields of a specific type structure. How to cover such operations is an important problem that we

TABLE 7
Bugs in Three Systems Detected by *EAntMiner*

System	Reference ID	Function	Critical Operation
OpenSSL	4800	CMS_SignerInfo_verify()	EVP_DigestVerifyInit()
	4807	dtls_construct_change_cipher_spec()	FN-WPACKET_put_bytes__0 = = 0
	4808	file_lshift()	FN-test_ptr_0 = = 0
FFmpeg	6381	read_ffserver_streams()	avformat_open_input()
	6382	rtp_mpegts_write_header()	avformat_write_header()
	6383	process_output_surface()	FN-av_mallocz_0 = = 0
	6386	mov_read_cmov()	FN-uncompress_0 ! = 0
	6387	mov_read_custom()	FN-av_malloc_0 = = 0
	6400	sami_paragraph_to_ass()	av_strtok()
	6405	ff_mpeg_ref_picture()	FN-av_buffer_ref_0 = = 0
PostgreSQL	14927	heap_drop_with_catalog()	ReleaseSysCache()
	14928	ATExecDetachPartition()	heap_freetuple()

*A **Reference ID** can be used to retrieve the corresponding bug. An OpenSSL bug report, say 4800, can be accessed at site https://github.com/openssl/openssl/issues/4800. A FFmpeg bug report, say 6381, can be accessed at site https://patchwork.ffmpeg.org/patch/6381. A bug report for PostgreSQL can be accessed via searching the corresponding reference ID at the site https:// www.postgresql.org/list/pgsql-bugs.*

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 (kinds of) operations are critical ones, as done in [16], [46].

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, other mining algorithms may be more suitable. Programming logics can be represented in forms of sequences [14], [54], [55], or even graphs [10], [25], [62], [63]. Note that our approach is compatible with other mining algorithms. Applying them on a refined mining 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, *EAntMiner* 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 [53] to normalize complicated semantics-equivalent representations that cannot be handled in the current version of *EAntMiner*.

Types of Bugs. *EAntMiner* mainly focus on detecting bugs that misuse some bug-prone functions or return incorrect values in certain contexts. We will extend our method to cover more types of interesting bugs, such as concurrency bugs that miss locks when performing critical operations in multithreaded programs [8], [9], [31]. 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 in our future work.

6 Related Work

6.1 Mining Rules for Bug Detection

Engler et al. [14] 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. [25] 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.

Various data mining algorithms are introduced to extract more general rules from real large systems [1], [5], [10], [26], [27], [29], [30], [31], [38], [44], [48], [54], [55], [63]. All mining based methods along with those statistical-based methods [14], [25], [51], [65] accept the reasonable assumption: in a practical system, programs are correct in most cases, while on the opposite, a small number of anomalies are likely to be bugs. These methods first infer frequently appeared patterns from the source code. Such patterns specify the (implicit) rules that should be followed in coding. Then, programs that violate the rules are detected and regarded as potential bugs.

Code mining methods can be categorized into three groups. (1) *Frequent pattern* based methods represent a rule as a frequent pattern (e.g., frequent sub-itemset [27], [31], [48], frequent sub-sequence [1], [30], [31], [39], [54], [55], or frequent sub-graph [10], [62], [63]). The relatively small number of violations are considered as potential bugs. (2) Template-based methods [38], [44] adapt the mined rules to templates provided by traditional static analysis tools (e.g., *Klocwork*), and then utilize these tools to detect bugs. And (3) instance-based methods [61], [63] employ the neighbors of a function under consideration to identify anomalies. A function behaves different with its neighbors may be defective [63], and neighbors of a known vulnerability are likely to contain similar vulnerabilities [61]. In this study, *EAntMiner* employs the frequent sub-itemset mining methods to mine rules for bug-prone functions due to its scalability, and utilizes the kNN technique to identify functions that return incorrect values under certain decisive conditions.

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 [1], [27], [30], [38], [54], [55] are missed, and (2) some prerequisite conditions are neglected [10], [39], [46], [48], [62], [63]. *EAntMiner* can detect both of them.

If some domain knowledge can be introduced when mining rules, better results may be produced. Some approaches have been specially designed to infer rules for critical APIs [1], [38], [48], [54], [55] or security-sensitive functions [46], [63], and have gained great results. *EAntMiner* also mines implicit programming 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 [29], execution paths [30], [36], [37], program comments [45], [47], or even documentations written in natural language [57], [66]. The *natural language processing* (NLP) technique is employed to extract rules from comments and documentations. NLP is also helpful for methods mining rules from source code [14], [63]. 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 improve the statements normalization.

6.2 Detecting Return Value Bugs

In the studies of Gunawi et al. [19] and Rubio-González et al. [40], they found that error codes are often incorrectly propagated in file systems, and such bugs are very hard to detect both statically and dynamically. Jana et al. [23] proposed a method $EPEx$ to detect error-handling bugs by introducing some error specifications. $EPEx$ explores error paths and uses under-constrained symbolic execution to decide whether the error is correctly handled (e.g., logging error message or propagating the error value upstream). Kang et al. [24] developed a method $APEx$ to automatically infer error specifications of API functions, which can then be used in $EPEx$. Different with $EPEx$, *EAntMiner* detects such bugs by checking inconsistencies between the real return values and the expected ones. Without any prior knowledge, the expected return values can be inferred from contexts that are most similar to the one under identification.

6.3 Supervised Learning in Bug Detection

Supervised learning techniques such as classification can be used to predict whether a software component is vulnerable or not [32], [35], [41], [49]. Arzt et al. [6] developed a tool SuSi to identify sources and sinks in Android applications. Two major challenges for supervised learning methods are features selection and training data acquisition. Deep learning provides a good solution to automatically extract features [18], [52], [64]. Perl et al. [35] proposed a smart method to label which commit is vulnerable. Some unsupervised idea can also be used to help find bugs. Yamaguchi et al. [61] identified functions that have very similar implementations with that contains a known vulnerability. These functions probably contain similar vulnerabilities.

6.4 Program Slicing and Its Applications

Program slicing was originally proposed by Weiser [56], and Chen and Cheung [11] 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 [2], [21]. Agrawal et al. [2] applied program slicing to locate known faults. Pradel and Gross [37] developed a method to automatically infer specifications from runtime traces. They also adopted the slicing idea to divide large traces into small ones that consist of related objects and method calls. Pradel et al. [36] proposed a framework to compare effectiveness of different mining approaches, and found that a mining algorithm can achieve a higher precision on traces that contain less irrelevant classes. In this paper, we employ the static program slicing to identify statements that are relevant to a specific critical operation.

7 Conclusion

This paper presents *EAntMiner*, a novel solution to improve the effectiveness of code mining by elaborately reducing noise introduced by irrelevant statements and semantics-equivalent but form-different representations. *EAntMiner* applies a divide-and-conquer approach to excluding statements that are irrelevant to certain critical operations, transforms various representations of the same logic into a canonical form as far as possible, and utilizes an instance-based learning method to reduce the interferences of return statements that are form identical but semantics different. We implement *EAntMiner* and evaluate it on four large-scale systems, especially the Linux kernel. The evaluation results show that our approach greatly improves the effectiveness of code mining, and can detect a large number of subtle bugs that have been missed previously.

Acknowledgments

The authors would like to thank the anonymous reviewers for their insightful comments on the preliminary version of this paper. The work is supported by National Natural Science Foundation of China (NSFC) under grants 91418206, 61170240, 61472429, and 61502465, National 973 program of China under grant 2014CB340702, National Science and Technology Major Project of China under grant 2012ZX01039-004, and Youth Innovation Promotion Association of the Chinese Academy of Sciences (YICAS) under grant 2017151.

References

[1] M. Acharya, T. Xie, J. Pei, and J. Xu, "Mining API patterns as partial orders from source code: From usage scenarios to specifications," in *Proc. 11th Eur. Softw. Eng. Conf. Held Jointly 15th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2007, pp. 163–173.

[2] H. Agrawal, J. R. Horgan, S. London, and W. E. Wong, "Fault localization using execution slices and dataflow tests," in *Proc. Int. Symp. Softw. Rel. Eng.*, 1995, pp. 143–151.

[3] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Boston, MA, USA: Addison Wesley, pp. 434–438, 1986.

[4] N. S. Altman, "An introduction to kernel and nearest neighbor nonparametric regression," *J. Amer. Statistician*, vol. 46, no. 3, pp. 175–185, 1992.

[5] G. Ammons, R. Bodik, and J. R. Larus, "Mining specifications," in *Proc. 29th ACM SIGPLAN-SIGACT Symp. Principles Program. Languages*, 2002, pp. 4–16.

[6] S. Arzt, S. Rasthofer, and E. Bodden, "Susi: A tool for the fully automated classification and categorization of android sources and sinks," University of Darmstadt, Karolinenpl, Darmstadt, Tech. Rep. TUDCS-2013-0114, 2013.

[7] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*, New York, NY, USA: ACM Press, 1999.

[8] Y. Cai and L. Cao, "Effective and precise dynamic detection of hidden races for java programs," in *Proc. 10th Joint Meeting Found. Softw. Eng.*, 2015, pp. 450–461.

[9] Y. Cai and W. K. Chan, "Magiclock: Scalable detection of potential deadlocks in large-scale multithreaded programs," *J. IEEE Trans. Softw. Eng.*, vol. 40, no. 3, pp. 266–281, 2014.

[10] R-Y. Chang, A. Podgurski, and J. Yang, "Finding what's not there: A new approach to revealing neglected conditions in software," in *Proc. Int. Symp. Softw. Testing Anal.*, 2007, pp. 163–173.

[11] T. Y. Chen and Y. Y. Cheung, "Dynamic program dicing," in *Proc. Softw. Maintenance*, 1993, pp. 378–385.

[12] B. Chess and G. McGraw, "Static analysis for security," *J. IEEE Security Privacy*, vol. 2 no. 6, pp. 76–79, Nov./Dec. 2004.

[13] B. Martin, M. Brown, A. Paller, D. Kirby, and S. Christey, "2011 CWE/SANS top 25 most dangerous software errors," [Online]. Available: http://cwe.mitre.org/top25, 2011.

[14] 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 *Proc. 18th ACM Symp. Operating Syst. Principles*, 2001, pp. 57–72.

[15] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *J. ACM Trans. Program. Languages Syst.*, vol. 9, no. 3, pp 319–349, 1987.

[16] V. Ganapathy, D. King, T. Jaeger, and S. Jha, "Mining security-sensitive operations in legacy code using concept analysis," in *Proc. 29th Int. Conf. Softw. Eng.*, 2007, pp. 458–467.

[17] G. Grahne and J. Zhu, "Efficiently using prefix-trees in mining frequent itemsets," in *Proc. Workshop Frequent Itemset Mining Implementations*, 2003, pp. 123–132.

[18] X. Gu, H. Zhang, D. Zhang, and S. Kim. "Deep API learning," in *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2016, pp. 631–642.

[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 *Proc. 6th USENIX Conf. File Storage Technol.*, pp. 1–16, 2008.

[20] S. Hallem, B. Chelf, Y. Xie, and D. Engler, "A system and language for building system-specific, static analysis," in *Proc. ACM SIGPLAN Conf. Program. Language Des. Implementation*, 2002, pp. 69–82.

[21] M. Harman and S. Danicic, "Using program slicing to simplify testing," *J. Softw. Testing Verification Rel.*, vol. 5 no. 3, pp. 143–162, 1995.

[22] S. Horwitz, H. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *J. ACM Trans. Program. Languages Syst.*, vol. 12, no. 1, pp 26–60, 1990.

[23] S. Jana, Y. Kang, S. Roth, and B. Ray, "Automatically detecting error handling bugs using error specifications," in *Proc. USENIX Security Symp.*, 2016, pp. 345–362.

[24] Y. Kang, B. Ray, and S. Jana, "APEx: Automated inference of error specifications for C APIs," in *Proc. 31st IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2016, pp. 472–482.

[25] T. Kremenek, P. Twohey, G. Back, A. Y. Ng, and D. Engler, "From uncertainty to belief: Inferring the specification within," in *Proc. 7th Symp. Operating Syst. Des. Implementation*, 2006, pp. 161–176.

[26] 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 *Proc. 21st ACM SIGOPS Symp. Operating Syst. Principles*, 2007, pp. 145–158.

[27] Z. Li and Y. Zhou, "PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code," in *Proc. 10th Eur. Softw. Eng. Conf. Held Jointly 13th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2005, pp. 306–315.

[28] B. Liang, P. Bian, Y. Zhang, W. Shi, W. You, and Y. Cai, "AntMiner: Mining more bugs by reducing noise interference," in *Proc. 38th Int. Conf. Softw. Eng.*, 2016, pp. 333–344.

[29] B. Livshits and T. Zimmermann, "DynaMine: Finding common error patterns by mining software revision histories," *Proc. 10th Eur. Softw. Eng. Conf. Held Jointly 13th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2005, pp. 296–305.

[30] D. Lo, S-C. Khoo, and C. Liu, "Mining past-time temporal rules from execution traces," in *Proc. Int. Workshop Dynamic Anal.*, 2008, pp. 50–56.

[31] 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 *Proc. 21st ACM SIGOPS Symp. Operating Syst. Principles*, 2007, pp. 103–116.

[32] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller, "Predicting vulnerable software components," in *Proc. 14th ACM Conf. Comput. Commun. Security*, 2007, pp. 529–540.

[33] K. J. Ottenstein and L. M. Ottenstein, "The program dependence graph in a software develop environment," in *Proc. ACM SIGSOFT/SIGPLAN Softw. Eng. Symp. Practical Softw. Develop. Environments*, 1984, pp. 177–184.

[34] Y. Padioleau, J. L. Lawall, and G. Muller, "SmPL: A domain-specific language for specifying collateral evolutions in linux device drivers," *Electron. Notes Theoretical Comput. Sci.*, vol. 166, pp. 47–62, 2007.

[35] H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl, and Y. Acar, "Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Security*, 2015, pp 426–437.

[36] M. Pradel, P. Bichsel, and T. R. Gross, "A framework for the evaluation of specification miners based on finite state machines," *Proc. IEEE Int. Conf. Softw. Maintenance*, 2010, pp. 1–10.

[37] M. Pradel and T. R. Gross, "Automatic generation of object usage specifications from large method traces," *Proc. IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2009, pp. 317–382.

[38] M. Pradel, C. Jaspan, J. Aldrich, and T. R. Gross, "Statically checking API protocol conformance with mined multi-object specifications," in *Proc. 34th Int. Conf. Softw. Eng.*, 2012, pp. 521–530.

[39] M. K. Ramanathan, A. Grama, and S. Jagannathan, "Static specification inference using predicate mining," *Proc. ACM SIGPLAN Conf. Program. Language Des. Implementation*, 2007, pp. 123–134.

[40] C. Rubio-González and B. Liblit, "Defective error/pointer interactions in the linux kernel," *Proc. Int.Symp. Softw. Testing Anal.*, 2011, pp. 111–121.

[41] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen, "Predicting vulnerable software components via text mining," *J. IEEE Trans. Softw. Eng.*, vol. 40, no. 10, pp. 993–1006, Oct. 2014.

[42] R. M. Stallman and the GCC Developer Community, "GNU compiler collection internals (for GCC version 4.5.0)" [Online]. Available: http://gcc.gnu.org/onlinedocs/gcc-4.5.0/gccint.ps.gz, 2010, pp. 217–250.

[43] Synopsys Inc., "Synopsys static analysis (Coverity) coverage for Common Weakness Enumeration (CWE)" [Online]. Available: https://www.synopsys.com/content/dam/synopsys/sig-assets/datasheets/coverity-cwe-coverage.pdf, p. 8, Mar. 2017.

[44] B. Sun, G. Shu, A. Podgurski, and B. Robinson, "Extending static analysis by mining project-specific rules," in *Proc. 34th Int. Conf. Softw. Eng.*, 2012, pp. 1054–1063.

[45] L. Tan, D. Yuan, G. Krishna, and Y. Zhou, "/* iComment: Bugs or bad comments? */," in *Proc. 21st ACM Symp. Operating Syst. Principles*, 2007, pp. 145–158.

[46] L. Tan, X. Zhang, X. Ma, W. Xiong, and Y. Zhou, "AutoISES: Automatically inferring security specifications and detecting violations," in *Proc. USENIX Security Symp.*, 2008, pp. 379–394.

[47] L. Tan, Y. Zhou, and Y. Padioleau, "aComment: Mining annotations from comments and code to detect interrupt related concurrency bugs," in *Proc. 33rd Int. Conf. Softw. Eng.*, 2011, pp. 11–20.

[48] S. Thummalapenta and T. Xie, "Alattin: Mining alternative patterns for detecting neglected conditions," in *Proc. 24th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2009, pp. 283–294.

[49] O. Vandecruys, D. Martens, B. Baesens, C. Muesb, M. D. Backera, and R. Haesena, "Mining software repositories for comprehensible software fault prediction models," *J. Syst. Softw.*, vol. 81, no. 5, pp. 823–839, 2008.

[50] W. Visser, J. Geldenhuys, and M. B. Dwyer, "Green: Reducing, reusing and recycling constraints in program analysis," *Proc. ACM SIGSOFT 20th Int. Symp. Foun. Softw. Eng.*, 2012, Art. no. 58.

[51] S. Wang, D. Chollak, D. Movshovitz-Attias, and L. Tan, "Bugram: Bug detection with n-gram language models," in *Proc. 31st IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2016, pp. 708–719.

[52] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Proc. 38th Int. Conf. Softw. Eng.*, 2016, pp. 297–308.

[53] T. Wang, K. Wang, X. Su, and P. Ma, "Detection of semantically similar code," *J. Frontiers Comput. Sci.*, vol. 8, no. 6, pp. 996–1011, 2014.

[54] A. Wasylkowski and A. Zeller, "Mining temporal specifications from object usage," in *Proc. 24th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2009, pp. 263–292.

[55] W. Weimer and G. C. Necula, "Mining temporal specifications for error detection," in *Proc. 11th Int. Conf. Tools Algorithms Construction Anal. Syst.*, 2005, pp. 461–476.

[56] M. Weiser. "Program slicing," in *Proc. 5th Int. Conf. Softw. Eng.*, 1981, pp. 439–449.
[57] X. Xiao, A. Paradkar, S. Thummalapenta, and T. Xie, "Automated extraction of security policies from natural-language software documents," *Proc. ACM SIGSOFT 20th Int. Symp. Found. Softw. Eng.*, 2012, Art. no. 12.
[58] S. Xu and Y.S. Chee, "Transformation-based diagnosis of student programs for programming tutoring systems," *J. IEEE Trans. Softw. Eng.*, vol. 29, no. 4, pp. 360–384, 2003.
[59] Z. Xu, J. Zhang, and Z. Xu, "Melton: A practical and precise memory leak detection tool for C programs," *J. Frontiers Comput. Sci.*, vol. 9, no. 1, pp. 34–54, 2015.
[60] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *Proc. IEEE Symp. Security Privacy*, 2014, pp. 590–604.
[61] F. Yamaguchi, M. Lottmann, and K. Rieck, "Generalized vulnerability extrapolation using abstract syntax trees," in *Proc. 28th Annu. Comput. Security Appl. Conf.*, 2012, pp. 359–268.
[62] F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck, "Automatic inference of search patterns for taint-style vulnerabilities," in *Proc. IEEE Symp. Security Privacy*, 2015, pp. 797–812.
[63] F. Yamaguchi, C. Wressnegger, H. Gascon, and K. Rieck, "Chucky: Exposing missing checks in source code for vulnerability discovery," in *Proc. ACM SIGSAC Conf. Comput. Commun. Security*, 2013, pp. 499–510.
[64] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun, "Deep learning for just-in-time defect prediction," in *Proc. Softw. Quality Rel. Security*, 2015, pp. 17–26.
[65] I. Yun, C. Min, X. Si, Y. Jang, T. Kim, and M. Naik, "APISan: Sanitizing API usages through semantic cross-checking," in *Proc. 25th USENIX Security Symp.*, 2016, pp. 363–378.
[66] H. Zhong, L. Zhang, T. Xie, and M. Hong, "Inferring resource specifications from natural language API documentation," in *Proc. IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2009, pp. 307–318.

Pan Bian received the MS degree in computer science from the School of Information, Renmin University of China, where he is currently working toward the PhD degree in information security. His research interest focuses on program static analysis.

Bin Liang received the PhD degree in computer science from the Institute of Software, Chinese Academy of Sciences. He is currently a Professor with the School of Information, Renmin University of China. His research interests include program analysis, vulnerability detection, and web security.

Yan Zhang received the MS degree in computer science from the School of Information, Renmin University of China. She is currently a Junior Researcher with the Pangu Team. Her research interests include program static analysis and malware detection in Android applications.

Chaoqun Yang received the BS degree in network engineering from the School of Information, Beijing Forestry University. She is currently working toward the MS degree in software engineering at the School of Information, Renmin University of China. Her research interest focuses on data mining.

Wenchang Shi received the PhD degree in computer science from the Institute of Software, Chinese Academy of Sciences. He is currently a Professor with the School of Information, Renmin University of China. His research interests include trusted computing, cloud computing, and computer forensics.

Yan Cai received the PhD degree from the City University of Hong Kong, Hong Kong, in 2014. He is currently an Associate Research Professor with the State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences. His current research interests include concurrency bugs, such as detection, reproduction, and fixing, especially in large-scale multithreaded programs.

Query to the Author

When you submit your corrections, please either annotate the IEEE Proof PDF or send a list of corrections. Do not send new source files as we do not reconvert them at this production stage.

Corrections to "Detecting Bugs by Discovering Expectations and Their Violations"

Pan Bian, Bin Liang, Yan Zhang, Chaoqun Yang, Wenchang Shi, and Yan Cai

IN [1], the corresponding author should have been listed as Bin Liang. The footnote information is corrected below.

REFERENCES

[1] P. Bian, B. Liang, Y. Zhang, C. Yang, W. Shi, and Y. Cai, "Detecting bugs by discovering expectations and their violations," *IEEE Trans. Softw. Eng* ., vol. 45, no. 10, pp. 984–1001, Oct. 2019.

- *P. Bian, B. Liang, Y. Zhang, C. Yang, and W. Shi are with the School of Information, Renmin University of China, Beijing 100872, China, and also with the Key Laboratory of DEKE, Renmin University of China, Beijing 100872, China. E-mail: {bianpan, liangb, annazhang, cqyang, wenchang}@ruc.edu.cn.*
- *Y. Cai is with the State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing 100080, China. E-mail: yancai@ios.ac.cn.*

Manuscript Date of publication 0 . 0000; date of current version 0 . 0000.
(Corresponding author: Bin. Liang.)
Digital Object Identifier no. 10.1109/TSE.2019.2958750