SinkFinder: Harvesting Hundreds of Unknown Interesting Function Pairs with Just One Seed

Pan Bian
bianpan@ruc.edu.cn
School of Information, Renmin University of China
Key Laboratory of DEKE, Renmin University of China
Beijing, China

Bin Liang*
liangb@ruc.edu.cn
School of Information, Renmin University of China
Key Laboratory of DEKE, Renmin University of China
Beijing, China

Jianjun Huang
hjj@ruc.edu.cn
School of Information, Renmin University of China
Key Laboratory of DEKE, Renmin University of China
Beijing, China

Wenchang Shi
wenchang@ruc.edu.cn
School of Information, Renmin University of China
Key Laboratory of DEKE, Renmin University of China
Beijing, China

Xidong Wang
wangxidong@ruc.edu.cn
School of Information, Renmin University of China
Key Laboratory of DEKE, Renmin University of China
Beijing, China

Jian Zhang
zj@ios.ac.cn
State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences
Beijing, China

ABSTRACT

Mastering the knowledge about security-sensitive functions that can potentially result in bugs is valuable to detect them. However, identifying this kind of functions is not a trivial task. Introducing machine learning-based techniques to do the task is a natural choice. Unfortunately, the approach also requires considerable prior knowledge, e.g., sufficient labelled training samples. In practice, the requirement is often hard to meet.

In this paper, to solve the problem, we propose a novel and practical method called `SinkFinder` to automatically discover function pairs that we are interested in, which only requires very limited prior knowledge. `SinkFinder` first takes just one pair of well-known interesting functions as the initial seed to infer enough positive and negative training samples by means of sub-word word embedding. By using these samples, a support vector machine classifier is trained to identify more interesting function pairs. Finally, checkers equipped with the obtained knowledge can be easily developed to detect bugs in target systems. The experiments demonstrate that `SinkFinder` can successfully discover hundreds of interesting functions and detect dozens of previously unknown bugs from large-scale systems, such as Linux, OpenSSL and PostgreSQL.

CCS CONCEPTS

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

*Corresponding author.

ESEC/FSE '20, November 8–13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7043-1/20/11…$15.00
https://doi.org/10.1145/3368089.3409678

KEYWORDS

Interesting Function Pairs, Bug Detection, Machine Learning, Word Embedding

ACM Reference Format:
Pan Bian, Bin Liang, Jianjun Huang, Wenchang Shi, Xidong Wang, and Jian Zhang. 2020. SinkFinder: Harvesting Hundreds of Unknown Interesting Function Pairs with Just One Seed. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), November 8–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3368089.3409678

1 INTRODUCTION

Software bug detection is critical to protect system security and reliability. To effectively detect certain bugs, current methods, especially the static ones, need to master related bug knowledge in advance [11, 16, 18, 52]. For many types of bugs, the essence of this prior knowledge is ***which functions*** can potentially result in the target bugs. For example, only if a static analysis tool knows *brelse*() is a memory deallocation function, can it successfully discover the use-after-free bug shown in §2.

In practice, the security-sensitive functions can be manually collected from known secure programming knowledge or dataset, such as common weakness enumerations (CWE), common vulnerability enumerations (CVE) or even usage specifications of public functions. Obviously, this requires a great deal of human effort, which is tedious and error-prone. Worse, it is very difficult to manually identify application-specific security-sensitive functions in a large-scale system. In fact, many of them are often less well documented and less well understood by analysts, e.g., the function *brelse*() in the Linux kernel.

Naturally, the machine learning technique was employed to automatically infer security-sensitive functions. The basic idea is to use a training set to train a classifier, and then use it to identify whether a given function is security-sensitive or not. For example, Rasthofer

et al. [45] developed SuSi to train a support vector machine (SVM) classifier to flag unknown taint sources and sinks in the Android framework. This kind of method has been proven to be effective in inferring unknown security-sensitive functions. SuSi successfully identified thousands of taint sources or sinks from Android framework (v4.2), which have been used to support practical taint analysis on Android applications [3].

Unfortunately, this straightforward supervised learning mode requires considerable prior knowledge, i.e., enough labelled training samples. In fact, the training set of SuSi consists of 779 labeled Android Framework APIs. This is indeed a great limitation when applying to other large-scale systems. In practice, so much prior knowledge is not always available. For example, an analyst, who wants to find memory corruption bugs in the Linux kernel, may only be aware of a very limited number of memory allocation and deallocation functions, e.g., *kmalloc*() and *kfree*(). It is very difficult, if not impossible, to train a usable classifier only with a few known positive functions. The problem may become more serious when facing a less popular target system.

In this paper, we propose a method, called SinkFinder, to solve the above problem. The basic idea is to employ the unsupervised learning technique and only rely on very limited prior knowledge to automatically discover enough positive and negative samples, and then use them to train an effective classifier of interesting function pairs. We refer to functions that we desire to extract as *interesting* functions, which are often security-sensitive ones in bug detection. Considering that many security-sensitive functions come in pairs (e.g., *kmalloc*() and *kfree*()), we choose to use a pair of known interesting functions as the seed to infer unknown interesting ones. Those functions (e.g. *memset*()) that do not pair with certain functions are out of our consideration.

SinkFinder takes just one pair of functions (called the seed pair) of the target system and its source code as inputs, and employs the *word embedding* technique to infer function pairs that are *analogous* to the seed pair. All functions are first embedded in vectors, just like vectorizing words in NLP [6, 39]. As the learned vectors embody semantics of functions, the desired unknown interesting function pairs are identified by measuring the semantic distance between their vectors and the given seed vector. Subsequently, these identified function pairs are used to train a SVM classifier to gain more positive ones that are difficult to be found by analogical reasoning. Finally, all discovered positive function pairs are integrated into some static checkers to detect bugs in the target system.

However, there are two issues need to be addressed. First, traditional word embedding methods, e.g., word2vec [39], only consider contexts that the word appears but ignore the name of the word. But for a function, its name is also important as well as its calling contexts when encoding its semantics [1, 23, 30]. Second, blindly combining all functions into candidate pairs would introduce unacceptable overhead. There would be n^2 pairs need to be handled for a target system with n functions. In other words, the pairing space will explode with the increase of the number of functions. In practice, a real-world large-scale system may have hundreds of thousands of functions, which would result in a huge pairing space.

We employ sub-word word embedding and data mining techniques to address above two issues, respectively. First, the state-of-the-art word embedding method fastText [6] is used to vectorize functions, which can encode sub-word level semantics besides inter-word relationships. Second, to reduce the pairing space, a specially designed frequent sequence mining algorithm is used to mine candidate pairs of functions that frequently co-occur.

We implemented SinkFinder and have evaluated it on the Linux kernel. We select six different well-known function pairs as seeds to identify different categories of unknown interesting functions. With these seed pairs, SinkFinder successfully discovers a large number of unknown interesting pairs with an average precision of 91.24%. For example, given security-sensitive pairs <*kmalloc*, *kfree*> and <*mutex_lock_nested*, *mutex_unlock*>, SinkFinder automatically discovers 237 pairs of "Alloc / Free" functions and 101 pairs of "Lock / Unlock" functions, respectively. In the knowledge of such security-sensitive functions, we can easily design checkers to detect bugs. Without loss of generality, we developed two checkers to detect use-after-free and mismatched-alloc-free bugs according to the obtained "Alloc / Free" functions. Using the two checkers, 55 suspects are detected in the Linux kernel v4.19. We have written patches for these suspects and submitted them to the kernel community. Up to now, 37 out of them have been confirmed to be real bugs and fixed in latest kernel versions. The others are waiting for further confirmation. We believe that the evaluation results are amazing and encouraging, considering that all findings are derived from just two well-known security-sensitive functions.

We also evaluated it on two other large systems (OpenSSL and PostgreSQL) with different initial seed pairs, i.e., <*CRYPTO_alloc*, *CRYPTO_free*> for OpenSSL and <*fopen*, *fclose*> for PostgreSQL. According to the identified functions, four suspects are detected and have been confirmed to be real bugs by the project developers.

This paper makes the following contributions.

- **Novel methodology.** By combining supervised and unsupervised learning techniques, SinkFinder can successfully discover unknown application-specific security-sensitive functions to effectively support bug detection. As far as we know, SinkFinder is the first practical learning-based interesting functions extraction method only requiring very limited prior knowledge.
- **Practicable solution.** By introducing sub-word word embedding and data mining techniques, sufficient training samples are produced and the pairing space is dramatically reduced, making SinkFinder applicable to large systems.
- **Encouraging results.** Hundreds of security-sensitive functions are identified and dozens of confirmed bugs are detected from three real-world large-scale systems [1].

The rest of this paper is organized as follows. §2 provides the motivating example. §3 presents the framework of SinkFinder and §4 describes the two checkers. §5 presents our evaluation result. After discussing the limitation and possible future works in §6, we review related work in §7. Finally, §8 concludes the paper.

2 MOTIVATING EXAMPLE

In this section, we use a real use-after-free (UAF) bug we found in the Linux kernel (v4.19) to motivate our method. Figure 1 illustrates the simplified code snippets related to the bug. The function *ext2_xattr_set*() calls *sb_bread*() to read a specified block from a block device (line 10). The function *sb_bread*() returns the buffer

[1] https://github.com/SinkFinder/data

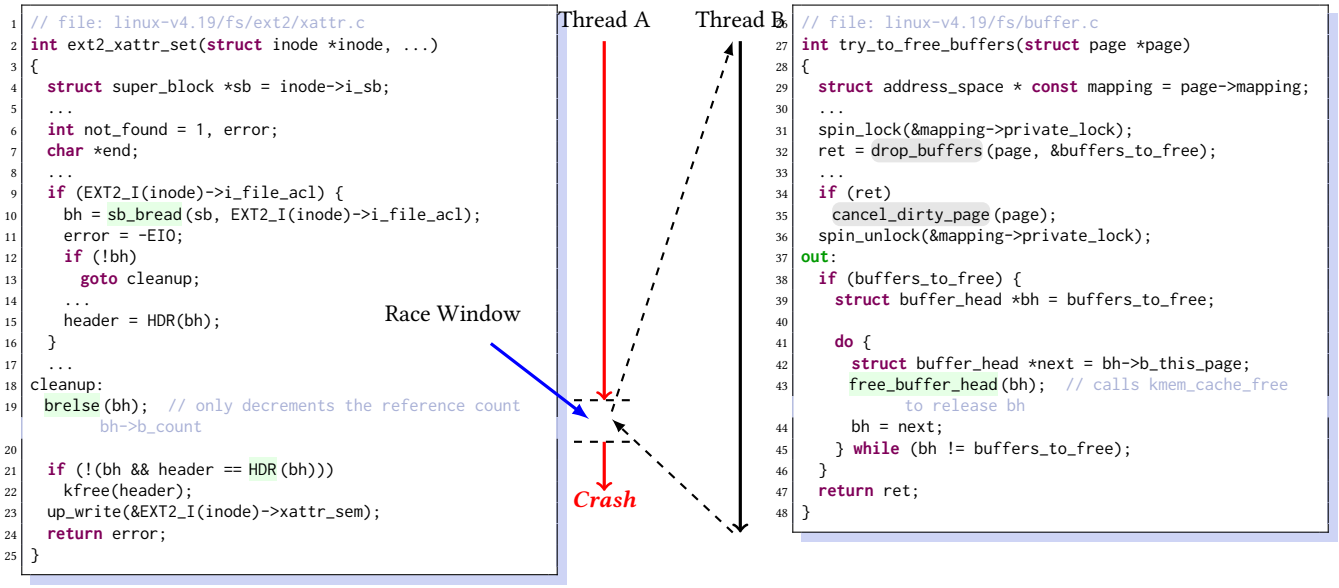

Figure 1: A use-after-free bug in Linux kernel.

head of the block on success. A buffer head is a shared memory with a reference count which records how many objects are using it. In *sb_bread*(), the reference count is incremented by 1. When it is no longer needed, the function *brelse*() should be called to decrement the reference count of the buffer head (line 19). If the reference count of the buffer head reaches zero, it will be released within the function *try_to_free_buffers*() (line 43).

The problem in Figure 1 is that the buffer head *bh* is used again (line 21) after its reference count is decremented (line 19). If another kernel thread is executed to release the buffer head during the window between calling *brelse*(*bh*) and dereferencing *bh*, a use-after-free bug may occur, resulting in a system crash or other security impacts.

Challenges in previous works. To detect the bug, static analysis methods need to know that the function *brelse*() behaves like a *free*() function [11, 18, 52]. However, *brelse*() is not as well known as other kernel deallocation functions such as *kfree*(). In fact, as far as we know, *brelse*() is not configured as a *free*() function in static analysis tools including both commercial ones (e.g., `Coverity` and `Fortify`) and open source ones (e.g., `Coccinelle` and `K-Miner`). Besides, a precise inter-procedural analysis does not work because the function *brelse*() just decrements the reference count of the buffer head and there is no explicit execution path to the real point where the buffer head is actually released.

Dynamic methods may also find it difficult to detect the bug. Dynamic symbolic execution methods are confronted with the same problem in static methods [8]. They also need to know the semantics of *brelse*() to simulate the behavior of *brelse*(). Fuzzing methods have to run a very long time before observing the crash due to the small race window and the unknown interactive threads that can result in the bug [13, 54].

Our solution. An intuitive but effective approach is to automatically identify the semantics of the functions closely related to the bug, e.g., *sb_bread*() and *brelse*(). Despite that the two functions may have other purposes, they can be viewed as allocation and deallocation functions when detecting memory-corruption bugs. With the knowledge that *brelse*() is a free-like function, static and dynamic methods can be applied to detect the bug. As an example, we replaced *kfree*() with *brelse*() in a `Coccinelle`'s UAF detection rule and the bug was successfully uncovered.

However, semantics inference is non-trivial. As discussed in Section 1, manual identification for a large-scale system is nearly impossible due to lack of documentations. For example, there is no entry describing the semantics of the function *brelse*() in the Linux kernel documentations. Besides, a typically supervised machine learning method always demands sufficient labeled functions for training an effective classifier, while such prior knowledge is usually extremely difficult, if not impossible, to obtain.

In this paper, we propose a two-stage method, `SinkFinder`, which requires merely a couple of known functions to infer the semantics of the other unknown ones. The high-level idea is to utilize the analogical reasoning ability of modern word embedding techniques [39, 42], with which the semantic relations between functions can be captured by subtraction of function vectors [21]. We can learn a vector for each function pair, and use it to discover semantically similar function pairs. We will show below how `SinkFinder` infers the fact that the relationship between *sb_bread*() and *brelse*() is similar to that between *kmalloc*() and *kfree*().

`SinkFinder` first discovers pairs that are quite similar or dissimilar to a given interesting function pair in both naming conventions and usage contexts. For example, given <*kmalloc*, *kfree*>, the pair <*vmalloc*, *vfree*> is identified to be similar, while <*lp_gpio_reg*, *outl*> is dissimilar. Then, `SinkFinder` discards the preference on function names to discover interesting pairs that "look" different to the given pair. Specifically, the pairs found in the first stage are taken as samples to train a SVM classifier, with which `SinkFinder` labels

the rest function pairs. As a result, <*sb_bread*, *brelse*> is reported analogous to <*kmalloc*, *kfree*>. With this knowledge, existing static analyzers could easily report the use-after-free bug in Figure 1.

3 OUR APPROACH

3.1 Overview

In this section, we present `SinkFinder`, an approach to discovering analogous function pairs of a given function pair, which is expected but not limited to consist of security-sensitive functions. The given function pair is referred to as the ***seed pair***, the function pairs analogous to the seed pair (i.e., pairs that we desire to identify) are viewed as ***positive pairs***, and all the other pairs that we are not interested in are treated as ***negative pairs***. As shown in Figure 2, `SinkFinder` works in a two-stage schema.

The first stage tries to infer a set of reliable positive pairs as well as a set of reliable negative pairs. To begin with, `SinkFinder` mines frequent function pairs from the source code (§3.2) to enhance the effectiveness, as blindly paired functions contain massive but mostly meaningless pairs. Then, it extracts function call sequences from the source code and learns vector representations of all functions (§3.3). According to the findings that the function names also provide an important source to understand the functions' semantics, we accomplish the function embedding with `fastText` [6] to take both the function names and the invocation contexts into account. At the end, `SinkFinder` performs analogical reasoning to output the positive, negative and unlabelled pairs (§3.4).

In the second stage, taking the positive and negative function pairs as training samples, `SinkFinder` trains a SVM classifier to further identify more positive pairs from the unlabelled ones (§3.5). To avoid being limited to only acquiring positive function pairs that have similar names as the seed, `word2vec` is used to transform functions to vectors, which discards the naming information of functions (also done in §3.3). As a result, `SinkFinder` can discover the especially semantically similar pairs with the seed, e.g., <*sb_bread*, *brelse*> for <*kmalloc*, *kfree*>.

Eventually, the interesting functions identified in the second stage can be used for further analysis, e.g., crafting rules for static bug detection targeting on the specific system.

3.2 Mining Frequent Function Pairs

As mentioned in §3.1, we mine frequent function pairs from source code of the target system. The desired pairs identified in §3.4 and §3.5 come from the frequent ones mined in this section. We are interested in pairs of functions that have strong data flow relationships. We concentrate on two types of data flow relationships, data dependence (DataDep) and data sharing (DataShare) [5]. Within a function definition, function g is data dependent on function f if g takes the output value of f as one of its arguments. Functions g_1 and g_2 have data sharing relationship if they both take the same variable value as their arguments and there is at least one feasible path between them in the control flow graph. A path is feasible if all conditions on it are satisfiable [27, 56]. A pair of two functions g_1 and g_2 is denoted as the tuple $<g_1, g_2>$, where g_2 is data dependent on or is data sharing with g_1.

Taking the code snippets in Figure 1 as an example. In the function *ext2_xattr_set*(), the first argument of *brelse*() at line 19 comes from the return value of *sb_bread*() called at line 10. Obviously, there is a data dependence relationship between the two functions. Whereas there is a data sharing relationship between *drop_buffers*() and *cancel_dirty_page*() because both of them take the variable *page* as their first argument in the function *try_to_free_buffers*().

A function definition is considered to contain a function pair p: $<g_1, g_2>$ if it calls both g_1 and g_2, and there is either DataDep or DataShare relationship between g_1 and g_2. The number of function definitions that contain a pair p is referred to as the *support* of p. We consider p is frequent if its support is larger than a given threshold *min_support*. For example, throughout Linux-v4.19, there are 107 functions containing the pair p_1: <*sb_bread*, *brelse*>. Assuming the threshold *min_support* of 10, p_1 is a frequent pair.

3.3 Embedding Functions

Mapping functions to vectors can benefit the semantic similarity computation for discovering analogous function pairs (see §3.4). To capture the semantics of functions, we leverage the advanced word embedding techniques in natural language processing to convert functions into continuous vectors.

State-of-the-art word embedding methods attempt to learn a convolutional neural network model that connects a word with its contexts. The most widely adopted method is `word2vec`, proposed by Mikolov et al. [39]. The basic idea behind `word2vec` is that words appearing in similar contexts are similar or closely related in semantics. Thus, `word2vec` attempts to map words with similar contexts to similar vectors. `word2vec` ignores the name of words and treats them as distinct ones even if the names are merely slightly different (e.g., *kfree* v.s. *kzfree*). Bojanowski et al. proposed `fastText` to encode the sub-word information of a word into the resultant vector as well as its contexts [6]. `fastText` works similarly to `word2vec` except that the sub-word information is also taken into account during learning vectors. Specifically, it first learns a vector for each constituent part of a word (called character n-grams). Then, the word is represented as a sum of the vectors of its n-grams.

`SinkFinder` uses both `fastText` and `word2vec` to learn vector representations for functions. We refer to the function vectors learned with `fastText` as *fastText-vectors*, and those learned with `word2vec` as *word2vec-vectors*.

A function definition acts as a context of the enclosed function invocations. `SinkFinder` performs a random walk on the control flow graph of each function definition intra-procedurally. During the random walk, loops are expanded only once and at most 100 random paths are extracted to mitigate the path explosion. Every feasible random path is transformed into a sequence of functions invoked on the path. Then, all the sequences generated from the target system are fed to `fastText` and `word2vec`, respectively. Each tool returns a vector for each function appearing in the sequences.

We represent the vector of a function f as $v(f)$. The vector of a pair p: $<f, g>$ is denoted as $v(p)$ and computed via the vector subtraction:

$$v(p) = v(f) - v(g) \tag{1}$$

Vector $v(p)$ embodies possible semantic relationships between f and g, such as co-occurrences and invocation orders [21, 42].

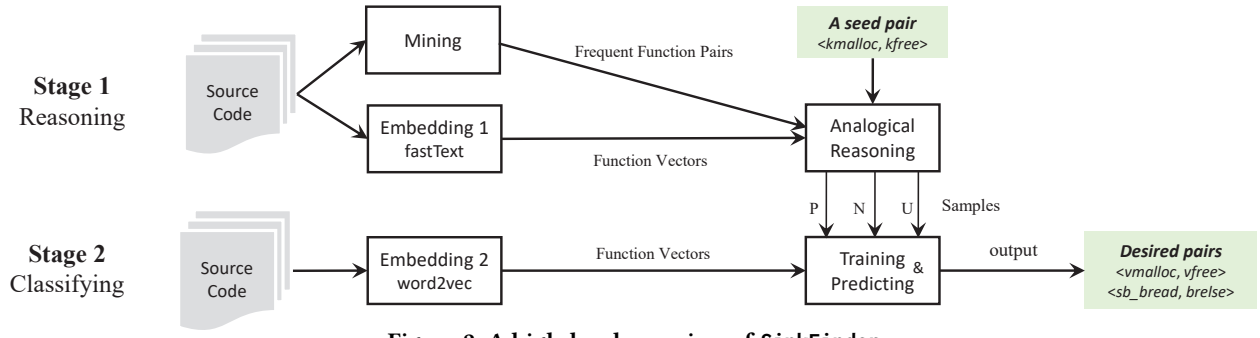

Figure 2: A high-level overview of `SinkFinder`.

3.4 Inferring Reliable Function Pairs

Given one pair of functions, `SinkFinder` performs an analogical reasoning to infer a set of *reliable* positive and negative function pairs. The core idea of analogical reasoning is that two pairs have similar semantics if they are close to each other in the vector space [6, 21, 39, 42]. The essence of reasoning is to find pairs that are close to the seed pair (i.e., positive pairs) as well as those that are far away from the seed pair (i.e., negative pairs). To achieve a high precision, `SinkFinder` works on the fastText-vectors in this phase such that the names of functions also act as a criterion for identifying semantically similar or dissimilar functions.

To quantitatively evaluate how much a pair is interesting, each pair p is associated with a positive confidence (denoted by $PConf(p)$) and a negative confidence (denoted by $NConf(p)$), reflecting how much we believe the pair is in our interest or not, respectively. Initially, the seed pair is assigned a positive confidence of 1.0 and a negative confidence of 0.0.

Ideally, we can take the function pairs with high positive confidence as positive ones, and those with low positive confidence as negative ones. However, we confront a challenge of how to determine the boundary of "high" and "low" confidence values. Specifying an absolute confidence threshold is not a good idea as it may not work well for all target systems. We overcome the problem by first screening out a set of reliable negative pairs that are highly dissimilar to the seed pair and then using them as references to pick out the reliable positive pairs iteratively.

Algorithm 1 shows the details of pair inference in this stage. Besides the seed pair p_0, the function fastText-vectors $\mathbb{V}$ and the set of frequent pairs $\mathbb{F}$, the algorithm requires a *ratio of negative pairs* $\mathcal{R}$ and a *positive coefficient* C. The ratio $\mathcal{R}$ indicates how many pairs can be directly recognized as negative pairs and the coefficient C determines whether the positive confidence of a pair is large enough. We denote the set of positive pairs and the set of negative pairs as $\mathbb{P}$ and $\mathbb{N}$, respectively. And the set of pairs waiting for identification is called *pending pairs*, represented as $\mathbb{L}$.

The algorithm walks three steps to infer the reliable negative and positive pairs.

Step 1: inferring reliable negative pairs. The positive confidence and negative confidence of each pending pair are computed according to their similarity to the seed pair (lines 5 ∼ 8). $cosine(p, p_0)$ is the *cosine* similarity between p and p_0. A higher cosine value means the two pairs are more similar. Ranking the pending pairs (line 9) in a descending order of their negative confidence values allows us easily extract the top $\mathcal{R}$ as the set of reliable negative pairs $\mathbb{N}$ (line 10). These pairs are highly dissimilar to the seed pair in both the function names and the calling contexts. In other words, they are unlikely to be our interesting pairs.

Step 2: computing negative confidence of pending pairs. The set of reliable negative pairs, after computed, will keep unchanged. `SinkFinder` is then expected to extract reliable positive

Algorithm 1 Identifying Reliable Positive and Negative Pairs

```
1:  procedure EXTRACT_PAIRS(𝕍, 𝔽, p0, ℛ, C)
2:      ℙ = {p0};
3:      𝕃 = 𝕃 − ℙ;
4:      // STEP 1: inferring reliable negative pairs
5:      for (each pair p in 𝕃) do
6:          PConf(p) = cosine(p0, p)};
7:          NConf(p) = 1 − PConf(p);
8:      end for
9:      quick_sort(𝕃, NConf);
10:     ℕ = {top ℛ pairs of 𝕃};
11:     // STEP 2: updating NConf of pending pairs
12:     𝕃 = 𝕃 − ℕ;
13:     for (each pair p in 𝕃) do
14:         NConf(p) = max_{p_i∈ℕ} {NConf(p_i) × cosine(p, p_i)}
15:     end for
16:     // STEP 3: iteratively inferring reliable positive pairs
17:     while (𝕃 ≠ ∅) do
18:         p_m = arg max{PConf(p) | p ∈ 𝕃};
19:         𝕃 = 𝕃 − {p_m};
20:         if (PConf(p_m) < C × NConf(p_m)) then
21:             continue;
22:         end if
23:         ℙ = ℙ ∪ {p_m};
24:         // Updating positive confidence
25:         for (each pair p_i in 𝕃) do
26:             confTmp = PConf(p_m) × cosine(p_i, p_m);
27:             if (PConf(p_i) < confTmp) then
28:                 PConf(p_i) = confTmp;
29:             end if
30:         end for
31:     end while
32:     quick_sort(ℙ, PConf);
33:     return (ℙ, ℕ);
34: end procedure
```

pairs from the rest pending pairs (line 12). A potentially reliable positive pair, intuitively, should be close enough to the seed pair (i.e., high positive confidence) and far away from the reliable negative pairs (i.e., low similarity to them). To this end, we use the reliable negative pairs to recalculate the pending pairs' negative confidence (line 13 $\sim$ 15).

Step 3: Iteratively inferring reliable positive pairs. This step examines the pending pairs one by one until all pairs have been tested. During each iteration, the pair $p_m \in \mathbb{L}$ with the highest positive confidence is selected as the target for identification (line 18). If p_m's positive confidence, which is initially computed at line 6, is not C times larger than its negative confidence (lines 20 $\sim$ 22), we resort to labeling it in the classification stage (§3.5). Otherwise, we mark p_m as a reliable positive pair (line 23). Considering that a pair can be a potentially positive pair when its cosine similarity to a reliable positive pair is high enough, we update the positive confidence of each pair in $\mathbb{L}$ if it is more semantically similar to p_m with a coefficient than to the other existing reliable positive pairs (line 25 $\sim$ 30).

Finally, the algorithm outputs $\mathbb{P}$ and $\mathbb{N}$, which contain all the identified reliable positive pairs and reliable negative pairs, respectively (line 33).

3.5 Classifying Function Pairs

The reliable positive pairs that `SinkFinder` obtains are very similar to the seed pair in both the function names and the calling contexts. For example, taking <*kmalloc*, *kfree*> as the seed, <*vmalloc*, *vfree*> is identified in the above stage.

Even though the first stage produces a precise set of positive pairs, we think it is more interesting and valuable to discover pairs that look very different with the seed pair but possess quite similar semantics, such as <*sb_bread*, *brelse*> versus <*kmalloc*, *kfree*>. We will show in §5.4 that, most of the Linux kernel memory-corruption bugs detected by our checkers are caused by deallocation functions whose names do not contain the keyword "free" (see Table 2).

To achieve the goal, `SinkFinder` trains a linear *Support Vector Machine* (SVM) to flag the undetermined function pairs. In order to exclude the impact of the function names, we use the word2vec-vectors to construct the feature vectors. As `word2vec` may encode various semantics (interesting and uninteresting) of a function into the function's vector and all dimensions are regarded to be equally important, the analogical reasoning technique is not suitable to infer interesting functions. Whereas, by learning the weight of each dimension from a number of training samples, an SVM addresses the issue by emphasizing the importance of the interesting semantics while lowering the uninteresting ones.

Training. The key to training a classifier is the preparation of the training samples, e.g., $(\mathbf{X}_1, y_1)$, $(\mathbf{X}_2, y_2)$, ..., $(\mathbf{X}_n, y_n)$, where $\mathbf{X}_i$ is a feature vector and y_i is the class label. We take the positive and negative pairs identified in the first stage as the training samples. More specifically, we utilize the corresponding word2vec-vectors of the functions within the identified pairs to make up the feature vectors. The class labels for positive pairs and negative pairs are 1 and 0, respectively. Because each function as well as the pair itself reflect certain semantic information, we build the feature vector by concatenating the function vectors and the pair vector. That is, the resulting feature vector $\mathbf{V}(p)$ of a pair p: <f, g> is computed via

$$\mathbf{V}(p) = \left[v(f),\ v(g),\ v(f) - v(g)\right] \tag{2}$$

`SinkFinder` builds feature vectors for both positive and negative pairs and feeds them to a linear SVM. The SVM then attempts to learn a hyperplane to separate the positive samples from the negative ones.

Predicting. After trained, the classifier can then be used to classify unlabeled samples (denoted as $\mathbb{U}$). Given the feature vector of a function pair, the classifier predicts its label, 1 or 0, which directly tells whether the function pair is positive or negative. However, we observed in practice that such a naive method will misclassify a large number of negative pairs as positive ones (i.e., false positives), especially when the number of positive samples is far smaller than that of the negative samples, known as the imbalanced training data issue[7, 10, 20].

To address the imbalance issue, we adopt the Spy idea adopted in PULearning [33]. When training a classifier, PULearning takes a small number of positive examples as unlabelled ones (called spies) to automatically determine a reasonable threshold. In this study, we take the reliable positive pairs as spies to determine a minimum probability threshold *min_proba* for identifying positive pairs and compute a positive probability for each target pair p, say *pos_prob*(p). An unlabeled pair is considered to be positive only when its positive probability exceeds the threshold. We notice that the SVM classifier can compute the probability of a sample belonging to a certain class via cross validation. We leverage this feature to compute the positive probabilities of the unlabeled pairs. Moreover, we generate *min_proba* by averaging the probabilities of the bottom s% reliable positive pairs with the lowest probabilities. To gain as many interesting positive pairs as possible, we set the percentage s% to 10%, which is a bit smaller than that in conventional PULearning [33] (i.e., 15%). Therefore, `SinkFinder` updates the set of positive pairs as follows:

$$\mathbb{P} = \mathbb{P} \cup \{p \mid p \in \mathbb{U},\ \textsc{Svm}(p) = 1,\ pos\_prob(p) > min\_proba\}$$

4 BUG DETECTION

We can use `SinkFinder` to identify more security-sensitive functions according to one or several known security-sensitive functions, and then integrate the identification result to checkers to detect bugs. Without loss of generality, we developed two checkers that utilize the identified "Alloc / Free" functions to detect memory corruption bugs. In C programs, correctly invoking allocation and deallocation functions is significant to guarantee the system security and such functions can be directly transformed to checkers to verify the usefulness of `SinkFinder` in bug detection.

1. use-after-free checker. A use-after-free bug is caused by accessing a memory chunk or resource that has already been released or closed via a deallocation function. The key to detecting use-after-free bugs is knowing which functions act as deallocators [11, 18, 19, 52]. Such functions are referred to as *free-like* functions. The free-like functions are extracted from "Alloc / Free" pairs (see §5.2) automatically identified by `SinkFinder`. With the knowledge of free-like functions, the use-after-free checker (UAF in short)

Table 1: Identification Result of Each Category.

Category	Seed Pair	Stage 1		Stage 2	
		#True / #All	Precision	#True / #All	Precision
Alloc / Free	<kmalloc, kfree>	104 / 105	99.05%	237 / 272	87.13%
Lock / Unlock	<mutex_lock_nested, mutex_unlock>	95 / 95	100%	101 / 101	100%
Start / End	<nla_nest_start, nla_nest_end>	9 / 9	100%	9 / 9	100%
Enable / Disable	<pci_enable_device, pci_disable_device>	11 / 11	100%	11 / 11	100%
Register / Unregister	<device_register, device_unregister>	18 / 20	90%	26 / 28	92.86%
Map / Unmap	<ioremap, iounmap>	12 / 13	92.31%	12 / 13	92.31%
Total		249 / 253	**98.42%**	396 / 434	**91.24%**

performs a data flow analysis to find potential usages of the freed variables [15].

2. mismatched-alloc-free checker. Using an unexpected deallocation function to release memory allocated with a certain allocation function can also result in bugs. For example, in Linux, misusing *kfree*() to release a page memory allocated with *__get_free_pages*() can result in a system crash or other security impacts (see bug #2 in Table 2). We implement a mismatched-alloc-free checker (MAF in short) to detect such bugs. The MAF checker scans the source code and reports a potential bug if there is a data flow from an allocation function f to a deallocation function g but $<f, g>$ is not one of the identified "Alloc / Free" pairs.

Considering that our identification method is unsound, some non-security-sensitive functions may be recognized as security-sensitive ones, leading to incorrect detection rules and then false alarms of bugs. We adopt a widely used ranking mechanism to mitigate the impact of false positives [31, 55]. The mechanism is inspired by the empirical observation that the more violations associated to a rule are uncovered, the more likely the rule is a FALSE rule and the corresponding discoveries are false positives. Therefore, a bug report is ranked at top if the rule it violates has very few violations. Otherwise, it is ranked at bottom.

5 EVALUATION

5.1 Experiment Setup

We have implemented our method as a prototype system and applied it to real-world large-scale systems to evaluate its effectiveness by mainly answering the following three key research questions.

RQ1 How effective is `SinkFinder` in identifying previously unknown interesting function pairs? (§5.2)

RQ2 How do parameter settings affect the identification result? (§5.3)

RQ3 Can the identified interesting functions help to detect previously unknown bugs? (§5.4 and §5.5)

RQ4 Why should we use different word embedding techniques in different stages? (§5.6)

To answer the above questions, we conduct a series of experiments mainly on the Linux kernel, which has been widely used as the target of evaluation (TOE) in both embedding methods [14, 21] and bug detection methods [16, 18, 28, 47, 51, 55]. We want to demonstrate the urgency of identifying system-specific security-sensitive functions for bug detection, especially for static methods, by detecting bugs that are missed in previous works. Linux-v4.19 is the latest version at our experiment time. It includes about 15 million lines of C code.

To show the generality of our method, we also apply `SinkFinder` to two other large-scale open source C systems, OpenSSL v1.1.1 and PostgreSQL v11.1 (§5.4), which have also been taken as TOEs by many bug detection methods [24, 25, 31, 55].

`SinkFinder` requires users to provide three parameters to identify interesting function pairs, the threshold *min_support*, the ratio $\mathcal{R}$ and the coefficient C. Their default values are 10, 0.05 and 1.5, respectively. In practice, however, users can tune them to gain better results as demonstrated in §5.3. Besides, different settings of hyperparameters of `fastText` and `word2vec` may also affect the experiment results. In our experiments, we adopt the well-tuned hyperparameter settings that `fastText` and `word2vec` used in their experiments to train vectors for words in natural language. Our experiment results show that they also work well on learning function vectors. There may be more suitable hyperparameter settings that could result in better results, but how to tune such hyperparameters is out of scope of this paper.

5.2 Identifying Interesting Function Pairs

Input. We conduct a series of experiments on Linux-v4.19 to evaluate the major ability of `SinkFinder` on identifying unknown interesting function pairs. We select six categories of function pairs to show the effectiveness of `SinkFinder`. Table 1 lists these categories and the corresponding seed pairs. Generally, we take the most frequent pair of each category as the seed because it is usually the most well-known one in practice. Note that the name of a category summarizes the common semantics of function pair instances. For example, "Alloc / Free" implies that one of the two functions allocates memory and the other function releases it, while "Lock / Unlock" functions are used to protect a certain resource from being read or written inconsistently in concurrent programs.

Time overhead. For each category, `SinkFinder` takes the well-known seed pair as input to discover pairs that are semantically similar to it. In total, `SinkFinder` took about 96 minutes for all six categories. Most of the time was spent on preparing data for analogical reasoning and classifying, i.e., parsing the source code with GCC, mining frequent function pairs, and learning function vectors, which took about 36 minutes, 25 minutes, 30 minutes, respectively. After embedding, there are 110,125 fastText-vectors and 110,125 word2vec-vectors generated. Through mining, 8,659 frequent function pairs are extracted. When the preparation is done,

SinkFinder can perform the identification of analogous function pairs for every seed pair within one minute.

Raw identification result. Under default parameter settings (i.e., *min_support* = 10, ratio $\mathcal{R}$ = 0.05, and coefficient C = 1.5), the raw identification result of each category is shown in Table 1 (columns *#All*). In total, 434 function pairs are identified for all the six categories. In addition to the statistical result in the first stage (*Stage 1*) and second stage (*Stage 2*), Table 1 also shows the overall result (*Overall*).

Manual inspection details. Because there are no available golden data to evaluate the precision of our method, we spent a great deal of human effort to inspect the identified functions one by one. To gain an accurate evaluation of our method, we inspect the source code implementation of each identified function and randomly examine some of the call instances to determine whether the functions have very close semantics with the given seed. The inspection costs one of us about 40 hours for all the 434 function pairs. However, in practice, users do not have to spend so much effort to examine the discovered function pairs. For example, in the application of detecting memory-corruption bugs, the identified "Alloc / Free" functions are directly integrated into the UAF and MAF checkers, and the ranking mechanism can help us highlight the true ones (see §5.4).

A pair is marked as "True" if both of the functions have similar semantics as those in the seed pair. For example, <*sb_bread*, *brelse*> is marked as a true "Alloc / Free" pair because *sb_bread*() occupies a buffer head (either an old one or a newly created one) by increasing the reference count while *brelse*() releases the occupation by decreasing the reference count. Whereas, <*dev_get_drvdata*, *kfree*> is not a valid pair because *dev_get_drvdata*() directly returns the data held by its argument rather than a newly allocated memory chunk.

Manual inspection result. The inspection result of each category is illustrated in Table 1, with the number of pairs marked as "True" (*#True*) and the precision (*Precision*). The precision is computed via $precision = \frac{\#True}{\#All}$. We did not compute the recall as it is impractical to collect all the function pairs of a certain category without bias.

From Table 1, our method achieves a high precision. Especially, the first stage achieves an average precision of 98.42%, which is high enough to be directly taken as training samples. The high precision of the first stage benefits from incorporating both invocation contexts and sub-word information of function names during embedding functions.

The precision of the second stage is also encouraging. The average precision is 91.24%. More importantly, hundreds of interesting pairs that do not follow common naming conventions are identified, which are difficult to be inferred by analogical reasoning.

5.3 Parameter Sensitivity

We conduct some experiments on Linux v4.19 to study how the three parameters, i.e., the minimum support *min_support*, the ratio $\mathcal{R}$, and the coefficient C, can impact the identification of interesting function pairs.

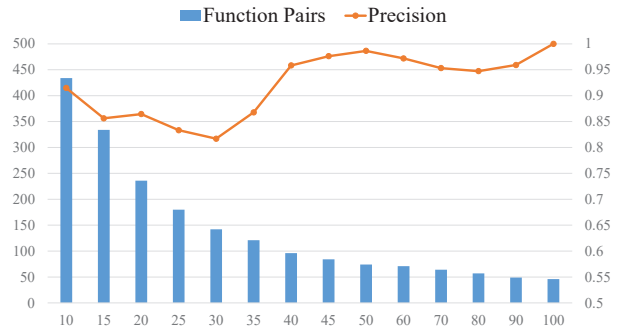

Figure 3: Function pairs identified by SinkFinder under different minimum support *min_support*.

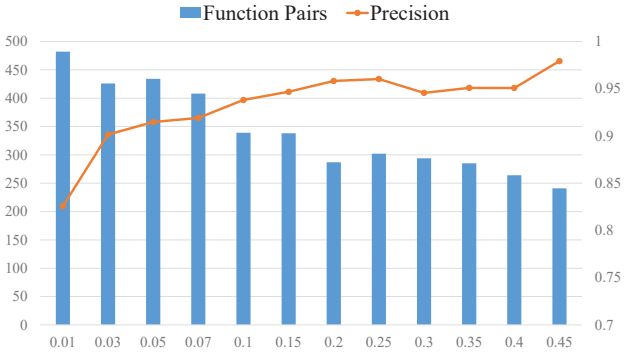

Figure 4: Function pairs identified by SinkFinder under different ratio $\mathcal{R}$.

To evaluate the impact of minimum support *min_support*, we conduct 14 experiments with different *min_support* settings (ranging from 10 to 100) and fixed $\mathcal{R}$ and C. Figure 3 shows the number of identified function pairs and the precision under each minimum support setting. We can see that the number of function pairs decreases along with the increase of *min_support*, but the precision of identification is much less sensitive to *min_support*. In general, SinkFinder can identify more interesting function pairs if a smaller *min_support* threshold is specified. The practice in mining programming patterns suggests that the mining result may be unreliable when the *min_support* is too small [31, 32]. In this study, to identify as many true interesting function pairs as possible, we take 10 as the default minimum support setting.

Similarly, we conducted 12 experiments to evaluate the impact of ratio $\mathcal{R}$. As shown in Figure 4, along with the increase of ratio $\mathcal{R}$ (from 0.01 to 0.45), the number of identified function pairs decreases but the precision increases. Finally, 16 experiments are used to evaluate the impact of coefficient C. From Figure 5, we can see that the number of identified function pairs decreases along with the increase of ratio C (from 1.0 to 2.5). However, the precision first increases but then decreases as insufficient positive training samples can be identified in the analogical reasoning stage with a large C. To balance precision and outcome, the default values of $\mathcal{R}$ and C are set to 0.05 and 1.5, respectively.

5.4 Detecting Bugs

In this subsection, we show the usefulness of the identified security-sensitive functions in detecting previously unknown bugs. In this

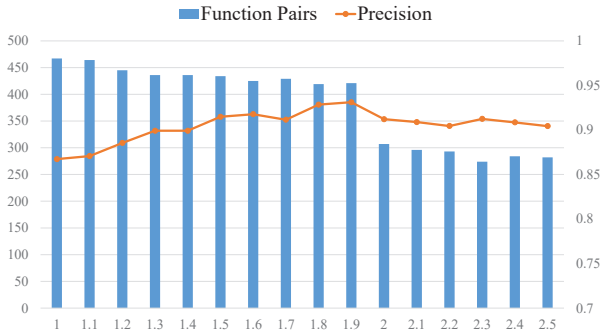

Figure 5: Function pairs identified by `SinkFinder` under different coefficient C.

experiment, we pass the raw identification result of the "Alloc / Free" category to the use-after-free checker and the mismatched-alloc-free checker. We manually inspect the detecting result of the two checkers. As the detected violations are ranked, we can only inspect the top ranked ones.

In the following of this subsection, we first demonstrate the bug detection result in the Linux kernel. And then, we evaluate `SinkFinder` on two other real systems to demonstrate that our method is general and is not limited to specific target systems (e.g., the Linux kernel).

Detecting bugs in the Linux kernel. The two checkers run 45 minutes to detect bugs in the Linux-v.4.19. The UAF checker and MAF checker report 835 and 522 violations, respectively. We manually inspect the top ranked 100 UAF violations and 10 MAF violations. In total, we have found 55 suspicious bugs, including 52 use-after-free bugs and 3 mismatched-alloc-free bugs. We have written patches for them and submitted the patches to the kernel maintainers. Up to now, 37 of the reported bugs have been confirmed to be real bugs and fixed by the kernel maintainers. And the other 18 reported suspicious bugs are waiting for further confirmation.

Table 2 is a list of the 37 already confirmed bugs, with the functions that contain the bug (*Function*), the related security-sensitive function pairs (*Pair*), the checker that detects the bug (*Checker*), and the IDs of our patches at the PatchWork site (*PatchID*). Each patch can be retrieved with its ID. For example, the patch 1016715# for the bug in Figure 1 can be found at https://lore.kernel.org/patchwork/patch/1016715.

Checking other systems. We also apply `SinkFinder` and the two checkers to OpenSSL v1.1.1 and PostgreSQL v11.1. The pair <*CRYPTO_malloc*, *CRYPTO_free*> and <*fopen*, *fclose*> are taken as seed pairs for OpenSSL and PostgreSQL, respectively. Finally, 40 and 5 "Alloc / Free" pairs are identified from OpenSSL and PostgreSQL, respectively. According to these function pairs, the two checkers report 92 and 22 violations in OpenSSL and PostgreSQL, respectively. We only inspect the top 10 reports of each type of bugs. And even so, we have found 1 and 3 suspicious bugs in OpenSSL and PostgreSQL, respectively. We have reported these suspicious bugs to the corresponding communities. All of them have been confirmed to be real bugs and have been fixed in the latest versions. The ID of the OpenSSL bug report[2] is #7845, and the IDs of the PostgreSQL bug reports[3] are #15539, #15540 and #15541.

5.5 Comparative Analysis

`Coverity`. `Coverity` is one of the best static bug detection tools. It has been applied to more than 6,200 open source projects including the Linux kernel. `Coverity` has been equipped with the basic memory deallocation function *kfree*() as a default rule to detect use-after-free bugs. According to the predefined rules, it conducts inter-procedural and path-sensitive analysis to detect potential violations [15]. We apply `Coverity` to Linux-v4.19 and review the detection results. As shown in Table 2, all of the 22 bugs are missed by `Coverity` (column *Cov*). A possible reason is that `Coverity` lacks the knowledge of the corresponding allocation or deallocation functions. The comparison with `Coverity` well demonstrates that identifying security-sensitive functions is helpful for static detection tools to detect real bugs.

Grep-like methods. We notice that some semantically similar functions follow specific naming conventions. For example, the name of a function that allocates memory may contain the sub-word "alloc", while the name of a function that releases memory may contain "free". We can utilize such heuristics to identify interesting functions with tools like `grep`. However, by reviewing the identified pairs, we find that only a small portion of the true functions follow the naming convention (83/237 = 35.02%) and thus can be discovered by `grep` with the naming pattern <.*alloc.*, .*free.*> in the Linux kernel. Besides, the sub-word may not indicate an expected behavior. For example, "free" in *get_free_page*() means "not being used" rather than "to release memory". Befitting from embedding contexts into function vectors, `SinkFinder` can recognize its true meaning. Consequently, our method performs much better than naive grep-like methods.

5.6 Using Different Vectors

To better understand the benefit of incorporating names and contexts of functions in the first stage and only considering contexts of functions in the second stage, we conduct three more experiments with <*kmalloc*, *kfree*> as the seed pair by employing different word embedding tools. We name each experiment in the form of **A→B** where **A** and **B** can be either `fastText` or `word2vec`, denoting the choices in the first stage and the second stage, respectively. Row 0 in Table 3 shows the results of in §5.2 and rows 1 ∼ 3 describe the results of the other three configurations.

In the 1^{st} experiment, we get the same reliable positive pairs as the original experiment in the first stage. However, only 35 new pairs are flagged as positive in the second stage. And 105 out of 133 (about 78.95%) "True" pairs are missed. Most of the 35 pairs have similar names with those identified in the first stage. In other words, the names of functions stop us from identifying interesting function pairs that are similar in semantics but dissimilar in naming conventions.

In the 2^{nd} and 3^{rd} experiments, only 24 pairs are extracted in the first stage, which are insufficient to train an effective classifier. After a manual inspection, only 20 of them belong to the category

[2]https://github.com/openssl/openssl/issues
[3]https://www.postgresql.org/list/pgsql-bugs

Table 2: Previously Unknown Bugs in Linux-v4.19 Detected by the Two Checkers.

ID	Function	Pair	Checker	PatchID	Cov
1	xfrm_add_acquire	<xfrm_state_alloc, xfrm_state_put>	MAF	1015392	×
2	create_active	<__get_free_pages, free_pages>	MAF	1015912	×
3	truncate_node	<f2fs_get_node_page, f2fs_put_page>	UAF	1016104	×
4	dentry_connected	<d_lookup, dput>	UAF	1016402	×
5	hfs_bmap_free	<hfs_bnode_find, hfs_bnode_put>	UAF	1016628	×
6	hfsplus_bmap_free	<hfsplus_bnode_find, hfsplus_bnode_put>	UAF	1016665	×
7	vfs_rename	<d_lookup, dput>	UAF	1016714	×
8	xfs_alloc_get_freelist	<xfs_perag_get, xfs_perag_put>	UAF	1017605	×
9	ext2_xattr_set	<sb_bread, brelse>	UAF	1016715	×
10	ocfs2_get_dentry	<igrab, iput>	UAF	1016725	×
11	autofs_expire_run	<d_lookup, dput>	UAF	1016907	×
12	ext4_quota_enable	<igrab, iput>	UAF	1016937	×
13	gfs2_create_inode	<get_acl, posix_acl_release>	UAF	1016991	×
14	sis_find_family	<pci_get_device, pci_dev_put>	UAF	1017915	×
15	sis_init_one	<pci_get_device, pci_dev_put>	UAF	1017916	×
16	sl82c105_bridge_revision	<pci_get_device, pci_dev_put>	UAF	1017916	×
17	ubi_detach_mtd_dev	<get_mtd_device, put_mtd_device>	UAF	1017919	×
18	rionet_start_xmit	<dev_alloc_skb, dev_kfree_skb_any>	UAF	1018023	×
19	hip04_mac_probe	<alloc_netdev_mqs, free_netdev>	UAF	1018030	×
20	lio_vf_rep_packet_sent_callback	<octeon_alloc_soft_command, octeon_free_soft_command>	UAF	1018512	×
21	mwifiex_parse_single_response_buf	<cfg80211_get_bss, cfg80211_put_bss>	UAF	1018810	×
22	rtl8187_init_urbs	<usb_alloc_urb, usb_free_urb>	UAF	1018816	×
23	amdgpu_benchmark_do_move	<dma_fence_get, dma_fence_put>	UAF	1149388	×
24	amdgpu_do_test_moves	<dma_fence_get, dma_fence_put>	UAF	1149298	×
25	intel_vgpu_get_dmabuf	<i915_gem_object_create, i915_gem_object_put>	UAF	1149266	×
26	rga_probe	<video_device_alloc, video_device_release>	UAF	1151291	×
27	nfc_genl_llc_set_params	<nfc_get_device, nfc_put_device>	UAF	1149816	×
28	fdp_nci_i2c_read_device_properties	<devm_kmalloc, devm_kfree>	UAF	1148731	×
29	st21nfca_hci_complete_target_discovered	<alloc_skb, kfree_skb>	UAF	1149734	×
30	_rtl92e_hard_data_xmit	<alloc_skb, kfree_skb>	UAF	1148926	×
31	bnx2i_free_hba	<pci_get_device, pci_dev_put>	UAF	1149409	×
32	pch_udc_free_dma_chain	<dma_pool_alloc, dma_pool_free>	UAF	1149287	×
33	i40iw_addr_resolve_neigh_ipv6	<sk_dst_get, dst_release>	UAF	1149186	×
34	qedr_addr6_resolve	<sk_dst_get, dst_release>	UAF	1149184	×
35	dwc2_hcd_init	<kmem_cache_create, kmem_cache_destroy>	UAF	1149083	×
36	qla4xxx_initialize_fw_cb	<dma_alloc_coherent, dma_free_coherent>	UAF	1148789	×
37	fnic_fcpio_icmnd_cmpl_handler	<mempool_alloc, mempool_free>	UAF	1147894	×

Table 3: Result of Using Different Vectors.

#	Experiment	Stage 1	Stage 2	Overall
0	**fastText→word2vec**	**104** / 105	**133** / 167	**237** / 272
1	**fastText→fastText**	104 / 105	28 / 38	132 / 143
2	**word2vec→word2vec**	20 / 24	8 / 11	28 / 35
3	**word2vec→fastText**	20 / 24	3 / 5	23 / 29

"Alloc / Free". Compared with that of the original experiment, 84 out of 104 (about 80.77%) "True" pairs are missed due to discarding the naming information in the first stage.

Summary. From the above comparisons, we can conclude that (1) introducing sub-word information of function names in the first stage can help identify more reliable positive pairs as well as improve the precision. And (2) discarding the sub-word information in the second stage can avoid being limited to only find function pairs that have similar names as the seed.

6 DISCUSSION

Though `SinkFinder` has automatically discovered hundreds of security-sensitive functions and detected dozens of bugs according to only several pairs of well known security sensitive functions, there are some points to be improved in our future work.

Using more seeds. In our method, only one seed pair is used to infer unknown interesting ones. However, in practice, more than one seed pair may be available. For example, one may know other "Alloc / Free" instances in the Linux kernel, e.g., <*vmalloc*, *vfree*>. In future, we will explore methods that incorporate multiple seed pairs to improve the efficiency of our method. On one hand, we can gain more reliable positive pairs by merging analogous pairs of

each seed. On the other hand, we can leverage multipass validation to reduce false positives.

Detecting bugs. To show the usefulness of our method in assisting bug detection, we have developed two static checkers to detect bugs associated with "Alloc / Free" functions. However, the other types of security-sensitive functions are also useful for detecting bugs such as deadlocks [37, 38]. We will apply the discovered knowledge to detect more types of bugs in the future.

Introducing inter-procedural analysis. We take the same strategy as most mining-based methods to extract frequent pairs intra-procedurally [16, 31, 32, 48, 55]. Though currently it works well, we will explore to integrate inter-procedural mining in our future work to discover frequent pairs that the functions are called in different but correlated contexts.

7 RELATED WORK

Several types of methods have been proposed to automatically identify security-sensitive functions. Engler et al. [16] and Kremenek et al. [28] developed a method to identify functions that may have certain semantics [16]. For example, observing that pointers passed to a deallocation function should not be used afterward to avoid bugs, a function is a potential deallocation function if its parameter is rarely used after it is called.

Some other methods heuristically identify security-sensitive operations according to domain-specific knowledge. Ganapathy et al. used concept analysis to infer fingerprints of security-sensitive operations according to the given domain knowledge, such as one or more critical data structures [17]. Tan et al. proposed a method called `AutoISES` to infer operations that should be protected by certain security check functions [47]. `SinkFinder` differs from these methods in the types of extracted operations.

Rasthofer et al. developed a method called `SuSi` to identify taint sources and taint sinks in the Android framework [45]. `SuSi` uses a set of labelled APIs to train a SVM classifier, which is then used to predict labels of the other APIs. `SuSi` has successfully identified a large number of taint sources and taint sinks in Android framework to facilitate taint analysis on Android Apps [3]. `SinkFinder` also trains a classifier to identify security-sensitive functions. The biggest difference between `SinkFinder` and `SuSi` is the source of training samples. `SuSi` requires users to provide hundreds of already known APIs. Whereas, `SinkFinder` automatically infers positive and negative training samples with only one well-known seed pair. The required prior knowledge in `SinkFinder` is much less than that in `SuSi`. Another difference is in the manner of generating feature vectors. In `SuSi`, the feature vectors are extracted according to predefined rules, some of which are biased to taint style APIs and should be redesigned when adapting for other categories of APIs. Whereas, in `SinkFinder`, the feature vectors are generated by word to vector tools and are applicable for different targets.

The research of Hindle et al. suggests that most software is also natural, and learning techniques for natural language can be used to process programming languages [22]. Nguyen et al. used word embedding techniques to learn vectors for APIs and found that the learned API vectors did encode useful semantic information [42]. What's more, their experiment shows that the subtraction of two API vectors encodes relationships between APIs. Henkel et al. further uses the word embedding techniques to map the Linux kernel functions to vectors [21]. Their experiment results demonstrate that function vectors can be used to answer analogy questions in the form "*a* is to *b* as *c* is to *what*?" `SinkFinder` also takes advantage of vector representations of functions to identify analogous functions. The biggest difference is that `SinkFinder` only needs a seed pair and does not require users to provide a third function (i.e., *c*), which is often unavailable in practice.

Researchers also propose to mine implicit programming rules from real-world large-scale systems for bug detection [2, 9, 26, 29, 31, 32, 34, 36, 37, 41, 43, 46, 50, 51, 55]. The mining methods are based on the deep insight that *most implementations are correct in real-world systems*. Generally, these approaches extract frequent patterns from target source code and take them as (implicit) programming rules that should be followed in coding. And any violations to them are regarded as potential bugs. `SinkFinder` also adopts the mining idea to extract frequent function pairs to mitigate pairing space explosion rather than to detect bugs directly. Our method to focus on mining correlated pairs is inspired by `NAR-Miner` [5].

Various program analysis techniques have been proposed to detect software bugs, including both static and dynamic ones [4, 12, 15, 18, 35, 40, 44, 53]. Despite their success in finding bugs, these approaches largely depend on the models of the system or the patterns of the bugs, e.g., high-level function semantics [49], which we call prior knowledge. Without that kind of knowledge, they are unable to find bugs. Our work, in contrast, discovers the knowledge of security-sensitive functions automatically. The knowledge can be employed by above detection methods. For example, the "Alloc / Free" and "Lock / Unlock" functions identified by `SinkFinder` can be used by `DCUAF` to detect concurrency use-after-free bugs.

8 CONCLUSION

In this paper, we develop a novel approach called `SinkFinder` to automatically identify hundreds of application-specific interesting functions with only one seed function pair. `SinkFinder` works in a two-stage schema. In the first stage, it performs analogous reasoning to infer a set of reliable negative pairs and reliable positive pairs. In the second stage, the reliable pairs are taken as training samples to train a linear SVM classifier. The trained classifier is then used to predict the labels of the undetermined pairs. Our experiments on the Linux kernel show that given a function pair, we can harvest hundreds of semantically similar function pairs. And the precision is quite high, i.e., 91.24%. We also implemented two checkers to detect bugs related to "Alloc / Free" functions and found a considerable number of memory corruption bugs from real-world large-scale systems.

ACKNOWLEDGMENTS

The authors would like to thank the reviewers for their constructive comments and suggestions. This work was supported in part by National Natural Science Foundation of China (NSFC) under grants U1836209, 61802413 and 61472429, the Fundamental Research Funds for the Central Universities, the Research Funds of Renmin University of China under grant 19XNLG02, and the Key Research Program of Frontier Sciences, Chinese Academy of Sciences under grant QYZDJ-SSW-JSC036.

REFERENCES

[1] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles A. Sutton. 2015. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, August 30 - September 4, 2015*. ACM, Bergamo, Italy, 38–49. https://doi.org/10.1145/2786805.2786849

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

[3] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick D. McDaniel. 2014. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, June 09 - 11, 2014*. ACM, Edinburgh, United Kingdom, 259–269. https://doi.org/10.1145/2594291.2594299

[4] Jia-Ju Bai, Julia Lawall, Qiu-Liang Chen, and Shi-Min Hu. 2019. Effective Static Analysis of Concurrency Use-After-Free Bugs in Linux Device Drivers. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, Dahlia Malkhi and Dan Tsafrir (Eds.). USENIX Association, Renton, WA, 255–268.

[5] Pan Bian, Bin Liang, Wenchang Shi, Jianjun Huang, and Yan Cai. 2018. NAR-miner: discovering negative association rules from code for bug detection. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, , November 04-09, 2018*. ACM, Lake Buena Vista, FL, USA, 411–422. https://doi.org/10.1145/3236024.3236032

[6] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. 2017. Enriching Word Vectors with Subword Information. *TACL* 5 (2017), 135–146. https://transacl.org/ojs/index.php/tacl/article/view/999

[7] Janez Brank, Marko Grobelnik, Nataša Milic-Frayling, and Dunja Mladenic. 2003. *Training text classifiers with SVM on very few positive examples*. Technical Report MSR-TR-2003-34. Microsoft Research. 1–27 pages.

[8] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008*. USENIX Association, San Diego, California, USA, 209–224. http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf

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

[10] Nitesh V. Chawla, Nathalie Japkowicz, and Aleksander Kotcz. 2004. Editorial: special issue on learning from imbalanced data sets. *SIGKDD Explorations* 6, 1 (2004), 1–6.

[11] Benjamin Chelf, Dawson R. Engler, and Seth Hallem. 2002. How to write system-specific, static checkers in metal. In *Proceedings of the 2002 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE'02, November 18-19, 2002*. ACM, Charleston, South Carolina, USA, 51–60.

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

[13] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. 2017. DIFUZE: Interface Aware Fuzzing for Kernel Drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, October 30 - November 03, 2017*. ACM, Dallas, TX, USA, 2123–2138. https://doi.org/10.1145/3133956.3134069

[14] Daniel DeFreez, Aditya V. Thakur, and Cindy Rubio-González. 2018. Path-based function embedding and its application to error-handling specification mining. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, November 04-09, 2018*. ACM, Lake Buena Vista, FL, USA, 423–433. https://doi.org/10.1145/3236024.3236059

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

[16] Dawson R. Engler, David Yu Chen, and Andy Chou. 2001. Bugs as Inconsistent Behavior: A General Approach to Inferring Errors in Systems Code. In *Proceedings of the 18th ACM Symposium on Operating System Principles, SOSP 2001, October 21-24, 2001*. ACM, Chateau Lake Louise, Banff, Alberta, Canada, 57–72.

[17] Vinod Ganapathy, David King, Trent Jaeger, and Somesh Jha. 2007. Mining security-sensitive operations in legacy code using concept analysis. In *Proceedings of 29th International Conference on Software Engineering (ICSE 2007), May 20-26, 2007*. IEEE Computer Society, Minneapolis, MN, USA, 458–467.

[18] David Gens, Simon Schmitt, Lucas Davi, and Ahmad-Reza Sadeghi. 2018. K-Miner: Uncovering Memory Corruption in Linux. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, February 18-21, 2018*. The Internet Society, San Diego, California, USA.

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

[20] Haibo He and Edwardo A. Garcia. 2009. Learning from imbalanced data. *IEEE Transactions on Knowledge & Data Engineering* 21 (2009), 1263–1284.

[21] Jordan Henkel, Shuvendu K. Lahiri, Ben Liblit, and Thomas W. Reps. 2018. Code vectors: understanding programs through embedded abstracted symbolic traces. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, November 04-09, 2018*. ACM, Lake Buena Vista, FL, USA, 163–174. https://doi.org/10.1145/3236024.3236085

[22] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar T. Devanbu. 2012. On the naturalness of software. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012*. IEEE Computer Society, Zurich, Switzerland, 837–847. https://doi.org/10.1109/ICSE.2012.6227135

[23] Einar W. Høst and Bjarte M. Østvold. 2009. Debugging Method Names. In *ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, July 6-10, 2009*. Springer, Genoa, Italy, 294–317. https://doi.org/10.1007/978-3-642-03013-0_14

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

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

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

[27] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (1976), 385–394. https://doi.org/10.1145/360248.360252

[28] Ted Kremenek, Paul Twohey, Godmar Back, Andrew Y. Ng, and Dawson R. Engler. 2006. From Uncertainty to Belief: Inferring the Specification Within. In *7th Symposium on Operating Systems Design and Implementation (OSDI'06), November 6-8, 2006*. USENIX Association, Seattle, WA, USA, 161–176.

[29] Ivo Krka, Yuriy Brun, and Nenad Medvidovic. 2014. Automatic Mining of Specifications from Invocation Traces and Method Invariants. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), November 16 - 22, 2014 (FSE 2014)*. ACM, Hong Kong, China, 178–189.

[30] Dawn J. Lawrie, Christopher Morrell, Henry Feild, and David W. Binkley. 2006. What's in a Name? A Study of Identifiers. In *14th International Conference on Program Comprehension (ICPC 2006), 14-16 June 2006*. IEEE Computer Society, Athens, Greece, 3–12. https://doi.org/10.1109/ICPC.2006.51

[31] Zhenmin Li and Yuanyuan Zhou. 2005. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, September 5-9, 2005*. ACM, Lisbon, Portugal, 306–315.

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

[33] Bing Liu, Yang Dai, Xiaoli Li, Wee Sun Lee, and Philip S. Yu. 2003. Building Text Classifiers Using Positive and Unlabeled Examples. In *Proceedings of the 3rd IEEE International Conference on Data Mining (ICDM 2003), December 19-22, 2003*. IEEE Computer Society, Melbourne, Florida, 179–188.

[34] Benjamin Livshits and Thomas Zimmermann. 2005. DynaMine: finding common error patterns by mining software revision histories. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, September 5-9, 2005 (ESEC/FSE-13)*. ACM, Lisbon, Portugal, 296–305.

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

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

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

[38] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, March 1-5, 2008 (ASPLOS XIII)*. ACM, Seattle, WA, USA, 329–339. http://doi.acm.org/10.1145/1346281.1346323

[39] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. *CoRR* abs/1301.3781 (2013). http://arxiv.org/abs/1301.3781

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

[41] Hoan Anh Nguyen, Robert Dyer, Tien N. Nguyen, and Hridesh Rajan. 2014. Mining Preconditions of APIs in Large-scale Code Corpus. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), November 16 - 22, 2014 (FSE 2014)*. ACM, Hong Kong, China, 166–177.

[42] Trong Duc Nguyen, Anh Tuan Nguyen, Hung Dang Phan, and Tien N. Nguyen. 2017. Exploring API embedding for API usages and applications. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, May 20-28, 2017*. IEEE / ACM, Buenos Aires, Argentina, 438–449. https://doi.org/10.1109/ICSE.2017.47

[43] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. 2009. Graph-based Mining of Multiple Object Usage Patterns. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, August 24-28, 2009 (ESEC/FSE '09)*. ACM, Amsterdam, The Netherlands, 383–392.

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

[45] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. 2014. A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, February 23-26, 2014*. The Internet Society, San Diego, California, USA.

[46] Boya Sun, Gang Shu, Andy Podgurski, and Brian Robinson. 2012. Extending static analysis by mining project-specific rules. In *Proceedings of the 34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012*. IEEE Computer Society, Zurich, Switzerland, 1054–1063.

[47] Lin Tan, Xiaolan Zhang, Xiao Ma, Weiwei Xiong, and Yuanyuan Zhou. 2008. AutoISES: automatically inferring security specification and detecting violations. In *Proceedings of the 17th USENIX Security Symposium, July 28-August 1, 2008*. USENIX Association, San Jose, CA, USA, 379–394.

[48] Suresh Thummalapenta and Tao Xie. 2009. Alattin: Mining Alternative Patterns for Detecting Neglected Conditions. In *ASE 2009, 24th IEEE/ACM International Conference on Automated Software Engineering, November 16-20, 2009*. IEEE Computer Society, Auckland, New Zealand, 283–294. https://doi.org/10.1109/ASE.2009.72

[49] John Toman and Dan Grossman. 2017. Taming the Static Analysis Beast. In *Proceedings of the 2nd Summit on Advances in Programming Languages, SNAPL 2017, May 7-10, 2017*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Asilomar, CA, USA, 18:1–18:14.

[50] Andrzej Wasylkowski, Andreas Zeller, and Christian Lindig. 2007. Detecting Object Usage Anomalies. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ESEC/FSE '07, September 3-7, 2007 (ESEC/FSE '07)*. ACM, Dubrovnik, Croatia, 35–44.

[51] Fabian Yamaguchi, Christian Wressnegger, Hugo Gascon, and Konrad Rieck. 2013. Chucky: Exposing missing checks in source code for vulnerability discovery. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, November 4-8, 2013*. ACM, Berlin, Germany, 499–510.

[52] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. 2017. Machine-Learning-Guided Typestate Analysis for Static Use-After-Free Detection. In *Proceedings of the 33rd Annual Computer Security Applications Conference, December 4-8, 2017*. ACM, Orlando, FL, USA, 42–54. https://doi.org/10.1145/3134600.3134620

[53] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. 2006. Using model checking to find serious file system errors. *ACM Trans. Comput. Syst.* 24, 4 (Nov. 2006), 393–423.

[54] Wei You, Peiyuan Zong, Kai Chen, XiaoFeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. 2017. SemFuzz: Semantics-based Automatic Generation of Proof-of-Concept Exploits. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, October 30 - November 03, 2017*. ACM, Dallas, TX, USA, 2139–2154. https://doi.org/10.1145/3133956.3134085

[55] Insu Yun, Changwoo Min, Xujie Si, Yeongjin Jang, Taesoo Kim, and Mayur Naik. 2016. APISan: sanitizing API usages through semantic cross-checking. In *Proceedings of the 25th USENIX Security Symposium, USENIX Security 16, August 10-12, 2016*. USENIX Association, Austin, TX, USA, 363–378.

[56] Jian Zhang and Xiaoxu Wang. 2001. A Constraint Solver and Its Application to Path Feasibility Analysis. *International Journal of Software Engineering and Knowledge Engineering* 11, 2 (2001), 139–156. https://doi.org/10.1142/S0218194001000487