Hindawi
Scientific Programming
Volume 2022, Article ID 3518673, 9 pages
https://doi.org/10.1155/2022/3518673

Research Article

Detecting Resource Release Bugs with Analogical Reasoning

Wentao Liang, Lu Wang, Jialuo She, and Yuqing Liu

College of Information Science and Technology, Xidian University, Xi'an 710126, China

Correspondence should be addressed to Lu Wang; wanglu@xidian.edu.cn

Received 23 October 2021; Accepted 5 January 2022; Published 7 February 2022

Academic Editor: Qianchuan Zhao

The resource release bugs are a common type of serious programming bug. However, it is hard to catch them by using static detection for the lacking of comprehensive prior knowledge about the release functions. In this paper, a resource release bug detection method is proposed by introducing analogical reasoning on word vectors. First, the functions of the target source code are encoded into word vectors by the word embedding technique in natural language processing. Second, a two-stage reasoning method is developed for automatically identifying unknown resource release functions according to a few well-known seed functions. 3CosAvg algorithm is employed for the first stage, and a new algorithm is designed for the latter, called 3CosAddExchange. Finally, the identified release functions are translated into static analysis rules to detect potential bugs. The experiment shows that the proposed method is effective and efficient for the large-scale software project. Five unknown resource release bugs are successfully detected in the Linux kernel and confirmed by kernel developers.

1. Introduction

The resource-release-related bug is a common type of programming defect that can lead to improper references to a released resource. Typical release-related defects include use after free and double free. The released resource may be reallocated. Referencing or releasing again the released resource will cause a sensitive information leak or memory corruption, and sometimes, it can result in more serious consequences such as executing arbitrary code. In practice, release-related bugs emerge frequently in various software projects. For example, in 2020, 344 use-after-free vulnerabilities are included in the CVE (common vulnerabilities and exposures) list [1].

Detecting release-related bugs is an important program analysis task. In theory, as long as the resource release functions are known, we can effectively detect release-related bugs using traditional static analysis techniques, for example, data-flow analysis. For example, in the Linux kernel, the logic of many release-related defects is actually very simple, and thus, they can be easily caught if we know the relevant resource release functions. Unfortunately, except for the well-known memory release functions such as *free* and *kfree*, numerous application-specific release-related functions in large-scale software are unfamiliar to the analysts, for example, *devlink_free* in the Linux kernel. Generally, they are known by only a few developers. If the functions are not configured into static analysis tools, it is impossible to effectively detect the release-related defects caused by them. Neglecting potential release-related functions has become one of the main obstacles for detecting the related defects.

In theory, it is possible to discover unknown release-related functions by manual audit. However, manually auditing real-world software projects requires a lot of human effort. It is very difficult, if not impossible, to manually identify unknown release-related functions in large-scale projects. For this reason, how to automatically identify release-related functions has become an urgent problem to solve.

In this paper, the word embedding technique, for example, Word2vec [2], is introduced to address the problem. Word embedding has become one of the most important representation learning methods in natural language processing (NLP). With the help of embedding, the high-dimensional one-hot word vector can be converted to a real number vector of lower dimensionality to more effectively support downstream tasks, for example, semantic search and

sentiment analysis. In addition to direct word similarity measurement, the embedded word vector can support analogical reasoning, answering the questions such as "If *man* is to *woman*, then *king* is to?" We can employ analogical reasoning to get the correct answer. In fact, as shown in Figure 1(a), we can effectively identify *queen* by calculating $V_{\text{King}} - V_{\text{man}} + V_{\text{woman}}$.

Our basic idea is to treat the programming language as a special natural language and the functions as the "words" of the language. The function call sequences of the target project can be used as the training samples (i.e., sentences) to train a word embedding model. Consequently, each function can be encoded into a vector using the model. As shown in Figure 1(b), by leveraging the analogical features of the function vectors, we can infer the unknown resource allocate-release function pairs (**malloc* and **free*) from the known function pairs (e.g., *kmalloc* and *kfree* in Linux kernel). From the inferred pairs, the release-related functions can be identified easily. It should be pointed out that unlike the analogical reasoning in NLP, we will identify potential unknown resource allocate-release function pairs first, instead of reasoning the unknown resource release functions directly.

In view of the above, we propose a resource-release-related bug detection method based on analogical reasoning. First, the function call sequences from the target project are extracted to train a word embedding model and encode all functions as vectors. Second, with a small number of well-known resource allocate-release function pairs, the potential resource-release-related functions are identified via a two-stage reasoning method. Finally, the obtained functions are configured into the detection rule of the static detection tool to find the resource release bugs from the target project. The proposed method has been applied to the Linux kernel. Through the two-stage reasoning, we identified hundreds of potential release-related functions and successfully detected five unknown bugs, which have been confirmed by the kernel developers. The experiment result shows that our method is effective and efficient and can be employed for analyzing and detecting large-scale software projects.

2. Methodology

As shown in Figure 2, the workflow of the proposed method is composed of five main steps. (1) The target project program is sliced, and the function call sequences are extracted from the slices to form a corpus for training an embedding model. (2) The frequent function pairs are mined from the call sequence as the candidate allocate-release function pairs. (3) A word embedding model is trained on the corpus, and the project functions are embedded into vectors. (4) From the known allocate-release function pairs (original seed pairs), the secondary seed pairs are inferred, and the unknown allocate-release pairs are identified subsequently in a two-stage analogical reasoning way. (5) The release-related functions in the identified pairs are converted to the detection rules of the static checker to check the target project for discovering the potential resource release bugs.

2.1. Code Data Preprocessing. To train a word embedding model, we must first prepare a corpus, which should possess sufficient word sequences (sentences). In this study, we use the function call sequence to construct the corpus. Naturally, we regard each function implementation as a semantic unit and extract the function call sequence from it. However, there is not always a relatively close semantic relationship among the function calls in a function. It is improper to directly combine the calls to form a training sample. To this end, we slice the target program and then extract the function call sequences from the slices to construct the corpus. As a result, there is a close relationship, for example, data dependence, among the function calls in a training sample.

Program slicing [3] is a common code preprocessing technique that can obtain a closely related statement subset. To highlight the data relationship among function calls, the program is sliced based on the data dependence graph (DDG). As done in [4], we use the LLVM compiler [5] to parse the target code and generate the corresponding DDG for each function implementation in the LLVM intermediate representation (IR) level. The parameters and variables in the function are used as the slicing criteria to get the corresponding data-dependent slices. From a function, we may get multiple slices. For each slice, we collect the function call instances in it to get a sequence with a depth-first traversal. By using program slicing, we can minimize the interference of irrelevant function calls in a training sample, allowing the word embedding model to effectively learn code semantic features.

We implement the above operation as an optimization pass of LLVM. By this means, LLVM can perform the slicing and collect function call sequences while compiling the target project. Hundreds of thousands of function call sequences can be obtained from a large-scale project such as the Linux kernel. They can effectively support the subsequent frequent function pair mining and model training.

2.2. Frequent Function Pairs Mining. A large-scale software often contains a large number of functions. For example, the Linux kernel has about 300,000 functions. If we compare the seed function pairs with the combinations of any two functions, a combinatorial explosion will be triggered. To avoid unacceptable computation overhead, the comparison is limited to the function pairs whose two elements have a data dependence relationship and appear together more than a certain number of times. To this end, the frequent pattern mining algorithm is leveraged to get the candidate pairs from the extracted call sequences.

There are some on-the-shelf frequent pattern mining algorithms, such as Apriori [6], Eclat [7], and FP-Growth [8]. Among them, FP-Growth is an efficient algorithm for mining frequent item sets. It only needs to traverse the data set twice to obtain frequent patterns that meet the predefined support threshold. In this study, we choose FP-Growth to mine candidate function pairs. After getting the frequent item sets, the corresponding association rules can be easily derived from them. For example, for frequent item sets $\{p1, p2\}$, we can

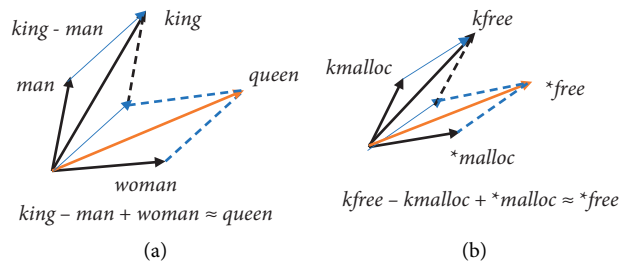

Figure 1: Analogical reasoning on vectors: (a) king – man + woman ≈ queen and (b) kfree – kmalloc + *malloc ≈ *free.

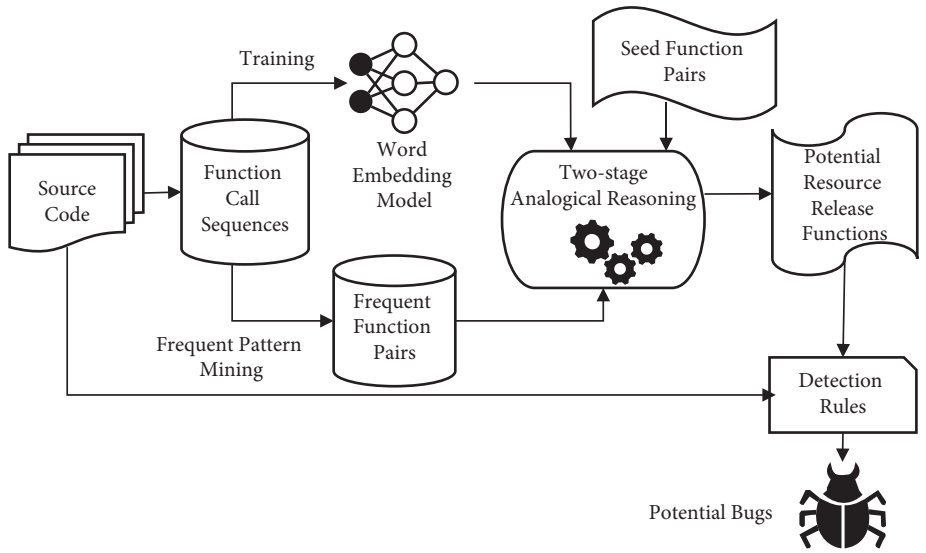

Figure 2: Workflow of bug detection.

obtain an association rule $p1 => p2$ and its confidence c, which means that when $p1$ appears in the data set, the probability that $p2$ also appears is c. With confidence, we can further rank the candidate function pairs to obtain the most closely related ones.

In this study, the frequent pattern length of FP-Growth is set to 2, that is, only frequent function pairs are mined. The mined function pair ($f1$, $f2$) will have the following properties:

(1) The functions $f1$ and $f2$ have a data dependency relationship

(2) The functions $f1$ and $f2$ are called together at least 5 times in the slices

(3) At least half of the slices containing function $f1$ or $f2$ also contain the other one

In a large-scale project, the number of frequent function pairs meeting the above properties may exceed 100,000. We will employ analogical reasoning to identify the pairs of interests from them afterwards.

2.3. Model Training. In NLP, the word embedding model is trained with a great number of natural language sentences/paragraphs from the corpus, such as the Google News data set. In this study, the function call sequences in the slices are used as training samples, like the sentences in the corpus, to train a word embedding model for the program. It can encode a function into a vector.

In recent years, a variety of word embedding models have been proposed, for example, Word2vec [2], FastText [9], and Bert [10]. Word2vec, the most classic model, can effectively support many downstream tasks. However, Word2vec has a significant drawback. Every word is regarded as an indivisible semantic unit, and the semantics of the subword is neglected. In other words, even if the two words are very similar in spelling and have similar semantics, such as "dog" and "dogs," the model will treat them as two unrelated words. In practice, functions with similar functionality may have similar names. For example, in the Linux kernel, both *kfree* and *vfree* are used to release memory blocks, and their names have the same suffix. As a new generation model, Bert has shown excellent performance in a number of NLP tasks and can capture subword semantics, but its training is significantly time-consuming. In contrast, FastText supports subword embedding and can be trained fast, making it applicable to large-scale software projects. In this study, we choose to train a FastText model for embedding functions.

FastText has two training modes: Skip-gram and CBOW. In Skip-gram mode, the model is optimized by using the target word to infer its context. On the contrary, the context

is used to infer the target word in CBOW mode. In general, Skip-gram mode can provide better performance than CBOW. We choose Skip-gram mode to train our model.

The FastText algorithm implemented in the Gensim library is employed. The model is trained to embed the function name as a 300-dimensional vector with the default algorithm parameters (training rounds as 5, context window size as 5, etc.). The training is very fast. For a large software project with hundreds of thousands of function call sequences, for example, Linux kernel, the training can be accomplished in a few minutes on a desktop computer.

2.4. Analogical Reasoning. With the trained model, the function is embedded into a vector that can be used to measure function similarity quantitatively. On this basis, the unknown resource release functions are discovered via analogical reasoning on the function vectors. However, in practice, it is often difficult to use a limited number of known allocate-release function pairs (called initial seed pairs) to fully represent the semantics of various allocate-release operations. If the analogical reasoning is performed directly on the initial seed function pairs, the obtained function pairs may be highly homogenous. As a result, the potential allocate-release pairs with some degree of heterogeneity may be missed. To this end, we designed a two-stage analogical reasoning method. As shown in Figure 3, some strongly confirmed allocate-release function pairs (called secondary seed pairs) are first inferred from the initial seed function pairs, which are expected to cover more allocate-release semantics. With the secondary seed pairs, the analogical reasoning is performed one more time to discover unknown allocate-release functions as many as possible.

In NLP, there are many classic analogical reasoning methods on word vectors, for example, 3CosAdd [2], PairDirections [11], and 3CosMul [11]. These methods can start from just one known word pair (A, B) and a word C to infer a related word D as shown in the following equations:

$$\begin{aligned}\text{3CosAdd}(A, B, C) = \arg\max_{D\in V} \\ \cos\left(v_D, v_C - v_A + v_B\right),\end{aligned} \tag{1}$$

$$\begin{aligned}\text{3CosMul}(A,\ B,\ C) = \arg\max_{D\in V} \\ \frac{\cos\left(v_D, v_C\right)\cos\left(v_D, v_B\right)}{\cos\left(v_D, v_A\right)},\end{aligned} \tag{2}$$

$$\begin{aligned}\text{PairDirection}(A,\ B,\ C) = \arg\max_{D\in V} \\ \cos\left(v_D - v_C, v_B - v_A\right).\end{aligned} \tag{3}$$

In practice, we can often get more than one pair of seed functions from large software projects, such as (*kmalloc, kfree*) and (*vmalloc, vfree*) in Linux. To make the best of seed function pairs and introduce more prior knowledge, we leveraged the idea of 3CosAvg [12] to perform analogical reasoning on multiple pairs of seed functions in the first stage. The word D is inferred as shown in equations (4) and (5), where A and B are two known word sets, whose sizes are n and m, respectively. As demonstrated in [12], 3CosAvg significantly outperforms the above three classical analogical reasoning methods.

$$\text{3CosAvg}(A,\ B,\ C) = \arg\max_{D\in V} \cos\left(v_D, v_C + v_{\text{avg\_offset}}\right), \tag{4}$$

$$v_{\text{avg\_offset}} = \sum_{i=1}^{n} \frac{A_i}{n} - \sum_{i=1}^{m} \frac{B_i}{m}. \tag{5}$$

In addition, the allocation function paired with an arbitrary unknown release function is also unavailable, that is, the word C in the above equations. To perform analogical reasoning, we extract the frequent function pairs from the target software project and calculate the similarity between the seed pairs and them one by one. Specifically, for a frequent function pair ($c1$, $c2$) and n pairs of known seed pairs (*alloc*_1, *free*_1) ... (*alloc_n, free_n*), the analogical similarity between them is computed as shown in the following equations:

$$\begin{aligned}\text{sim}(\text{avg\_offset}, C) = \max\Big(\cos\left(v_{c1}, v_{C2} + v_{\text{avg\_offset}}\right), \\ \cdot\ \cos\left(v_{c2}, v_{C1} + v_{\text{avg\_offset}}\right)\Big),\end{aligned} \tag{6}$$

$$v_{\text{avg\_offset}} = \sum_{i=1}^{n} \frac{v_{\text{free\_}i}}{n} - \sum_{i=1}^{n} \frac{v_{\text{alloc\_}i}}{n}. \tag{7}$$

Because who is for allocating or releasing resources in $c1$ and $c2$ is still unknown, we perform cosine similarity calculations twice and take the higher one as the analogical similarity between ($c1$, $c2$) and the seed pairs. The target function in the calculation with higher output is also identified as the resource release function. For instance, if $\cos(v_{c1}, v_{C2} + v_{\text{avg\_offset}})$ is higher, then $c1$ will be recognized as a release function, and vice versa. In this stage, the function pairs with the highest analogical similarity are selected as the secondary seed pairs (top 10 in this study) for the second stage of reasoning.

Although 3CosAvg has been proven to gain better performance, using it in the second stage may conceal the unique semantics of some secondary seed pairs. For this reason, we employ the idea of 3CosAdd to measure the analogical similarity in the second stage. Besides, we also introduced an axiomatic property of analogy, *the exchange of means*, to optimize the similarity calculation. Using the property can discover the latent semantics [12] more effectively. Formally, in the second stage, the analogical similarity is calculated as shown in equation (8), where $(\widehat{\text{alloc}}, \widehat{\text{free}})$ is a secondary seed pair. We call this similarity calculation 3CosAddExchange.

$$\begin{aligned}\text{sim}((\widehat{\text{alloc}}, \widehat{\text{free}}), (c1,\ c2)) = \max\Big(\cos\Big(v_{\widehat{\text{alloc}}}, v_{\widehat{\text{free}}} + v_{c1} - v_{c2}\Big), \\ \cdot\ \cos\Big(v_{\widehat{\text{alloc}}}, v_{\widehat{\text{free}}} + v_{c1} - v_{c2}\Big), \\ \cdot\ \cos\Big(v_{\widehat{\text{free}}}, v_{\widehat{\text{alloc}}} + v_{c1} - v_{c2}\Big), \\ \cdot\ \cos\Big(v_{\widehat{\text{free}}}, v_{\widehat{\text{alloc}}} + v_{c2} - v_{c1}\Big)\Big).\end{aligned} \tag{8}$$

As can be seen from the equation, four 3CosAdd similarities are computed with different exchange forms for a

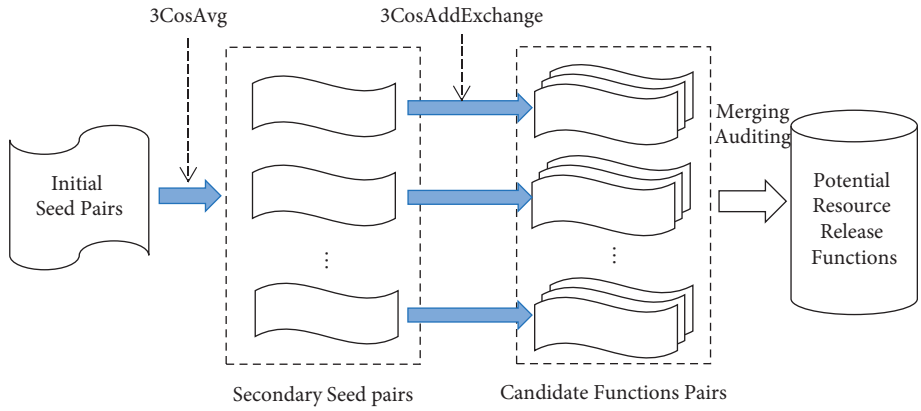

Figure 3: Two-stage analogical reasoning.

frequent function pair ($c1$, $c2$) and a secondary seed pair. The maximum of them is outputted as the analogical similarity between the two pairs, and the potential release-related function is also accordingly identified from ($c1$, $c2$). For each secondary seed pair, the candidate function pairs are selected according to their similarity with the seed. In our experiment on the Linux kernel, the top 200 are selected for each secondary seed pair. However, there may be some pairs that are also similar to the seed but are not in the top 200. To investigate the false negatives caused by the ranking threshold, we also randomly select five pairs with similarities greater than 0.9 from below the top 200 and take them as the candidate function pairs. The candidate function pairs derived from different secondary seed pairs are merged to remove duplicates. Finally, the potentially reliable resource release functions are identified from them by manual auditing. Note that the number of functions being audited in depth is limited with the help of analogical reasoning. We can take just a few hours to identify the true release functions. Such human efforts are completely acceptable for the analysis of large software projects.

2.5. Bug Detection. After identifying the resource release functions, detecting related bugs is not a difficult task. In fact, a variety of Clang checkers have been implemented to detect various C/C++ program bugs. Among them, *MallocChecker* is able to detect use-after-free and double-free bugs. The identified release functions can be easily added to its detection rules. In this way, we can leverage *MallocChecker* to check whether there are potential resource release bugs in the target project, which are caused by the functions identified by our method.

3. Evaluation

In this study, the Linux kernel is chosen as the target of evaluation. Due to the wide use of Linux, the resource release bugs in the Linux kernel may result in serious security risks. Therefore, detecting kernel bugs is an essential and rewarding task.

3.1. Experiment Setup. The experiment is conducted on a laptop with 16 GB memory, a 4-core i7-8565U 1.8 GHz Core processor, 1 TB hard disk, Ubuntu 20.04 operating system, and Python 3.8. The version of LLVM/Clang used for code analysis and bug detection is 12.0.0.

The target project for evaluation is Linux kernel 5.13.7, which contains more than 20,000 source code files. We chose two pairs of well-known allocate-release function pairs, (*kmalloc*, *kfree*) and (*vmalloc*, *vfree*), as initial seeds. The two pairs can be easily found in the classic Linux kernel books, for example, [13].

3.2. Parameter Setting. In the proposed method, the two hyperparameters of the mining algorithm, that is, support and confidence, can lead to different reasoning results. To study the impact of different parameters on performance, we carry out an empirical experiment.

We collect the corresponding result of the first stage under nine different support and confidence settings. Specifically, we tune three different supports (3, 5, and 10) and three different confidences (0.25, 0.5, and 0.75). From Table 1, we can see that the precision of the first stage is perfect when the support is set to 5. All the top 10 pairs in the output are true allocate-release ones and can be directly used as the secondary seed pairs. On this basis, we fix the support to 5 and audit the top 200 of the merged output of the second stage under the three different confidence settings. The precision is 78%, 91%, and 74%, respectively, for the confidences of 0.25, 0.5, and 0.75.

When the confidence is 0.25, the function pairs with low confidence can be top-ranked if they show similar patterns with the seeds in the slices and are identified as similar pairs according to the embeddings. Many such pairs, however, are mismatched or even not correlated to allocation-release operations, making a low precision. On the other hand, when the confidence is set to 0.75, many interesting function pairs, which are semantically close to the seeds but do not show high confidence in the corpus, are eliminated. Hence, dissimilar function pairs are top-ranked, leading to a lower precision.

According to the above empirical experiment, we choose 5 and 0.5 as the support and confidence, respectively.

3.3. Efficiency. In this section, we will discuss the efficiency based on the time cost recorded for each step of our method for the Linux kernel.

Table 1: Result of the first stage under different hyperparameter settings.

	Confidence = 0.25 (%)	Confidence = 0.5 (%)	Confidence = 0.75 (%)
Support = 3	100	80	90
Support = 5	100	100	100
Support = 10	90	80	40

In the experiment, it takes about 22 hours to preprocess the Linux kernel using Clang equipped with our pass of slicing and function extraction. We obtained 874,295 slices in total, which involve 290,276 different functions. Detecting the resource release bugs takes about 20 hours with the configured *MallocChecker*. Note that the steps need to be done only once. For a large software project such as Linux kernel, such time cost is completely acceptable.

Compared with code preprocessing and bug detection, the other steps are extremely fast. About three minutes are needed to mine frequent function pairs using the FP-Growth algorithm, and 107,692 pairs are generated. It is worth mentioning that if we employ Apriori instead of FP-Growth, mining will not end even after several days. Training the FastText model takes about 6 minutes, and the size of the obtained model is about 2.4 GB. The two analogical reasoning stages require about 5 and 11 minutes, respectively.

It can be seen that a normal desktop computer is powerful enough for analyzing large-scale projects with our method.

3.4. Result Analysis. From the results of the first analogical reasoning stage using (*kmalloc, kfree*) and (*vmalloc, vfree*) as the initial seeds, the top 10 pairs with the highest similarity are selected as the secondary seeds as shown in Table 2. We manually audit the new seeds and confirm that all of them are true allocate-release function pairs. In other words, the precision of the secondary seed identification is 100%.

In particular, the identified (*mlxsw_sp_vr_get, mlxsw_sp_vr_put*) is not a trivial resource allocate-release pair. The two functions are involved in the resource reference count. When *mlxsw_sp_vr_put* is invoked, only if the reference count of the target resource indicates that there is not any reference to it, a resource release operation will be triggered. Improperly using this type of function can also introduce subtle release-related bugs. Identifying this type of function as a release function is reasonable. It also shows from one side that we can use word embedding to learn the deep semantics of functions. Identifying such nontypical resource release functions can provide more detection rules and can hit more potential bugs.

In the second stage of analogical reasoning, by using the secondary seeds, 10 groups of candidate function pairs are inferred with the 3CosAddExchange reasoning. From each group, 205 pairs are selected as mentioned in Section 2.4. After merging and duplicate removal, 405 distinct potential allocate-release function pairs are obtained. Among them, 363 pairs (89.63%) are confirmed to be true positives by manual auditing. It is an encouraging result to precisely identify hundreds of potential objective function pairs only with very limited prior knowledge (2 seed pairs).

With *MallocChecker* and the identified release functions, eight suspicious bugs are found in Linux kernel. We have reported them to the Linux kernel developers. Up to now, five of them have been confirmed as real; two are waiting to be confirmed; and the remaining one is a false positive. The confirmed bugs are shown in Table 3. The release functions involved in the bugs rank in the top 200 except for *raid5_release_stripe*, which is randomly selected from the candidate pairs below the top 200. However, a confirmed unknown bug has also been found with it. This suggests that if the selection threshold is appropriately relaxed, we may be able to find more potential bugs.

Figure 4 shows a confirmed bug in *dr_rule_handle_ste_branch*. In its implementation, function *mlx5dr_htbl_put* is called with a pointer *cur_htbl* as the argument. If the current reference count of the object pointed by *cur_htbl* is one, this object will be released by the system. However, after *mlx5dr_htbl_put* returns, *cur_htbl* will be referenced by *mlx5dr_err* again. It is a typical use-after-free bug whose logic is very simple. In theory, it is easy to detect this kind of bug with static code analysis. However, even for such a naive bug, detecting it is also difficult when we do not know that *mlx5dr_htbl_put* is a resource release function. The bug illustrates that it is important for static analysis to effectively identify the bug-related functions.

3.5. Comparison Analysis. We compared our detection result with that of the original *MallocChecker*, which is responsible for release-related defect detection in Clang. The experiment on the Linux kernel shows that the original *MallocChecker* cannot detect any bugs found by our method. By analyzing the source code of *MallocChecker*, we were surprised to find that there are only four release-related functions in its check rule configuration, that is, *free*, *kfree*, *if_freenameindex*, and *g_free*. Such a poor configuration cannot effectively support bug detection in the real world. The comparison further demonstrates the importance of the identification of defect-related functions.

The state-of-the-art study on discovering similar functions pairs for bug detection is SinkFinder [14]. SinkFinder takes about 1 hour to mine frequent pairs, learn the vectors, and reason about the similar pairs with the given seeds. Our work, in contrast, consumes about 20 minutes for mining, embedding, and analogical reasoning, by employing a faster mining algorithm (i.e., FP-Growth) and adopting only one embedding model (i.e., FastText). Preprocessing and bug detection cost much time in our experiments, but they are not an essential part of our approach and can be replaced with the other tools. In summary, our approach achieves comparable efficiency with the existing technique. Besides, we have also compared with SinkFinder on the aspect of

Table 2: Identified secondary seed function pairs.

Rank	Allocation function name	Release function name	3CosAvg similarity
1	*mlxsw_sp_kvdl_alloc*	*mlxsw_sp_kvdl_free*	0.96307445
2	*mlx4_mr_alloc*	*mlx4_mr_free*	0.95947766
3	*mlx5_db_alloc*	*mlx5_db_free*	0.9594188
4	*mlxsw_sp_counter_alloc*	*mlxsw_sp_counter_free*	0.954844
5	*mlx4_db_alloc*	*mlx4_db_free*	0.9546761
6	*nfp_port_alloc*	*nfp_port_free*	0.95387816
7	*mlx4_buf_alloc*	*mlx4_buf_free*	0.95332044
8	*devlink_alloc*	*devlink_free*	0.95299953
9	*mlx4_pd_alloc*	*mlx4_pd_free*	0.9513632
10	*mlxsw_sp_vr_get*	*mlxsw_sp_vr_put*	0.9465025

Table 3: Confirmed release-related flaws.

Buggy function name	Release function name	Bug type
amdgpu_cs_wait_all_fences	*dma_fence_put*	Use after free
raid5_end_write_request	*raid5_release_stripe*	Use after free
dr_rule_handle_ste_branch	*mlx5dr_htbl_put*	Use after free
of_link_to_phandle	*of_node_put*	Double free
drm_gem_prime_import_dev	*dma_buf_put*	Double free

Figure 4: An example of detected bugs.

identifying unknown interesting function pairs. From the Linux kernel, SinkFinder discovers 237 true allocate-release pairs, achieving a precision of 87.13%. Our approach reaches a comparable precision of 90.26%and more true output (389 confirmed pairs). It is demonstrated that only using special-designed analogical algorithms can also effectively identify the functions of interest without requiring to train a classifier.

4. Discussion

4.1. Other Types of Sensitive Functions. The proposed method is not limited to supporting release-related bug detection. For example, if we can find a function similar to the C library function *execve* or *system*, it can be directly used to detect the command injection vulnerability. Unfortunately, unlike the resource release function, the *execve*-like function often does not have a fixed pairing function. Consequently, analogical reasoning cannot be directly leveraged to infer unknown *execve*-like functions. Although word vectors naturally support the "one-to-one" similarity measure, we found that it is not as good as the "pair-to-pair" analogical reasoning. In fact, finding the unknown release functions via one-to-one similarity comparison will produce many false positives. The main reason is that analogical reasoning employing vector pairs can introduce more semantic information, which is essentially a kind of constrained search in the vector space. For example, it is difficult to accurately infer *queen* by *king* alone. However, if *man* and *woman* are introduced as the constraint relationship, *queen* can be identified easily. Designing an analogical reasoning method for unpaired functions, such as *execve*, is one of our future works.

4.2. False Positives. We borrow *MallocChecker* of Clang to detect bugs. However, the checker does not support field-sensitive analysis, resulting in a number of false positives. How to perform the field-sensitive analysis is beyond the scope of this study. We believe that the kind of false positives can be effectively mitigated by improving the static checker. In addition, a suspect we found in the *Btrfs* filesystem of the

Linux kernel is eventually proved to be a false positive by the developer. The resource seems to be released twice in the related implementation. However, the developer informs us that the reference counts of some objects in *Btrfs* are guaranteed to be at least two. As a result, only when the reference count is reduced at least twice, the object may be released. The subtle operation logic is greatly beyond the capability of the normal static program analysis technique. In the future, we plan to investigate whether the special operation logic is widespread. If it is not an isolated case, we will try to explore targeted detection methods.

4.3. False Negatives. In this study, the confidence threshold is set to 0.5 for mining frequent function pairs. If some paired allocate-release functions are not frequently called simultaneously by a caller, it may cause false negatives. In addition, in the second reasoning stage, we mainly selected the top 200 similarity function pairs, and a considerable number of pairs with high similarity (over 0.9) were discarded. This can also lead to false negatives. In practice, when human resources can support larger-scale auditing, the confidence and similarity criteria can be relaxed. In this way, we can detect more resource release functions and related bugs.

4.4. Dynamic Detection. In addition to static analysis, dynamic testing is also a common bug detection method. However, many release-related bugs refer to subtle logic and are difficult to be triggered, especially for the ones involving reference counting. Relatively speaking, as long as the release functions are available, we can find potential release bugs more easily with static analysis.

4.5. Stableness of Reasoning. For the proposed method, there is some indeterminacy in training the embedding model. In other words, although the same training set is employed in multiple training instances, the resulting models may be slightly different. The main reason behind this is the negative samples for training are selected randomly. However, such a little bit of indeterminacy does not make a big difference in detection results. In fact, we find that the allocate-release pair sets identified with these models are almost the same, that is, only a few pairs are hit by one model but missed by another. In practice, we can merge the results from the models trained with different training instances to improve accuracy as far as possible.

5. Related Work

The representative work of embedding code as vectors is Code2vec [15]. It converts the code snippet into a vector to predict the semantic information of the snippet. Code2vec improves the performance of the function name prediction task by 75%. In Code2vec, a neural network is designed to learn the representation of the paths on the code abstract syntax tree (AST) and integrate them into the function representation. Code2vec learns function semantics from the function implementation, while our method focuses on the function context. Considering the diversity of the resource release operation, using the calling context to identify similar functions is more suitable. There are some studies focusing on code structure embedding, such as encoding the code-related graph structure as a vector [16–18] for matching vulnerable code. In addition to high-level language programs, executable code can also be embedded for binary code search [19].

Some existing studies have been carried out to automatically discover the functions of interest. Rasthofer et al. proposed SuSi [20] to identify taint source/sink functions in the Android framework based on machine learning. Using hundreds of known functions labelled manually and dozens of predefined features, they trained a support vector machine (SVM) classifier and successfully discovered a number of unknown taint source/sink functions. However, SuSi requires a considerable number of training samples and relies on feature engineering to determine the classification features. This may lead to over- or underfitting. The closest study is SinkFinder [14]. It employs analogical reasoning to get a training set and trains an SVM classifier to identify unknown sensitive functions. However, our study shows that we can directly identify unknown functions of interest by using the nontraditional 3CosAvg and 3CosAddExchange algorithms and without requiring to train a classifier. In practice, for some target projects, it is difficult to directly infer sufficient samples to support model training.

6. Conclusion

The resource release bug is a common and serious programming defect. Theoretically, static analysis is an effective way to detect the bug. Unfortunately, it cannot work well due to the lack of knowledge about release functions. To address the issue, we propose an unknown release function identification method based on NLP word embedding. By embedding the functions as vectors, we design a two-stage reasoning method to automatically identify unknown resource release functions with a few well-known seed functions. The identified functions are finally configured in static detection rules to detect vulnerabilities in the target system. We apply our method to the Linux kernel and successfully detected five confirmed unknown bugs. The proposed method is proven to be effective and efficient for large-scale software projects.

Data Availability

The data sets used in this paper are available from the corresponding author upon request.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work is supported by the projects (61672401) supported by the National Natural Science Foundation of China and Projects Basic Research Program of Natural Science of Shaanxi (2020JQ-300).

References

[1] "Common vulnerabilities & exposures," https://cve.mitre.org.

[2] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," 2013, https://arxiv.org/abs/1301.3781.

[3] M. Weiser, "Program slicing," *IEEE Transactions on Software Engineering*, vol. se-10, no. 4, pp. 352–357, 1984.

[4] M. Ahmadi, R. M. Farkhani, R. Williams, and L. Lu, "Finding bugs using your own code: detecting functionally-similar yet inconsistent code," in *Proceedings of the 30th USENIX Security Symposium*, Virtual Event, August 2021.

[5] "The LLVM compiler infrastructure," https://llvm.org.

[6] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules," in *Proceedings of the 20th int. conf. very large data bases (VLDB)*, vol. 1215, pp. 487–499, Santiago de Chile, Chile, September 1994.

[7] M. J. Zaki, "Scalable algorithms for association mining," *IEEE Transactions on Knowledge and Data Engineering*, vol. 12, no. 3, pp. 372–390, 2000.

[8] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation," *ACM sigmod record*, vol. 29, no. 2, pp. 1–12, 2000.

[9] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching word vectors with subword information," *Transactions of the Association for Computational Linguistics*, vol. 5, pp. 135–146, 2017.

[10] J. Devlin, M. W. Chang, K. Lee, and K. Toutanova, "Bert: pre-training of deep bidirectional transformers for language understanding," 2018, https://arxiv.org/abs/1810.04805.

[11] O. Levy and Y. Goldberg, "Linguistic regularities in sparse and explicit word representations," in *Proceedings of the eighteenth conference on computational natural language learning*, pp. 171–180, Baltimore, Maryland, June 2014.

[12] Y. Hong and Y. Lepage, "Production of large analogical clusters from smaller example seed clusters using word embeddings," in *Proceedings of the International Conference on Case-Based Reasoning*, pp. 548–562, Stockholm, Sweden, July 2018.

[13] A. Rubini and J. Corbet, *Linux Device Drivers*, O'Reilly Media, Inc., Newton, MA, USA, 2001.

[14] P. Bian, B. Liang, and J. Huang, "SinkFinder: harvesting hundreds of unknown interesting function pairs with just one seed," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1101–1113, Stockholm, Sweden, July 2020.

[15] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "Code2vec: learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, pp. 1–29, 2019.

[16] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp. 480–491, Vienna Austria, October 2016.

[17] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 363–376, Dallas, TX, USA, October 2016.

[18] B. Bowman and H. H. Huang, "VGRAPH: a robust vulnerable code clone detection system using code property triplets," in *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P)*, pp. 53–69, Genoa, Italy, September 2020.

[19] S. H. Ding, B. C. Fung, and P. Charland, "Asm2vec: boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," in *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, pp. 472–489, San Francisco, CA, USA, May 2019.

[20] S. Rasthofer, S. Arzt, and E. Bodden, "A machine-learning approach for classifying and categorizing android sources and sinks," *NDSS*, vol. 14, 2014.