Raisin: Identifying Rare Sensitive Functions for Bug Detection

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

Jianglei Nie
School of Information, Renmin University of China
Beijing, China
rucnjl@ruc.edu.cn

Yuanjun Gong
School of Information, Renmin University of China
Beijing, China
gongyuanjun@ruc.edu.cn

Wei You
School of Information, Renmin University of China
Beijing, China
youwei@ruc.edu.cn

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

Pan Bian
Huawei Technologies CO., LTD.
Beijing, China
bianpan@huawei.com

ABSTRACT

Mastering the knowledge about the bug-prone functions (i.e., *sensitive functions*) is important to detect bugs. Some automated techniques have been proposed to identify the sensitive functions in large software systems, based on machine learning or natural language processing. However, the existing statistics-based techniques are not directly applicable to a special kind of sensitive functions, i.e., the *rare* sensitive functions, which have very few invocations even in large systems. Unfortunately, the rare ones can also introduce bugs. Therefore, how to effectively identify such functions is a problem deserving attention.

This study is the first to explore the identification of rare sensitive functions. We propose a context-based analogical reasoning technique to automatically infer rare sensitive functions. A *1+context* scheme is devised, where a function and its context are embedded into a pair of vectors, enabling pair-wise analogical reasoning. Considering that the rarity of the functions may lead to low-quality embedding vectors, we propose a weighted subword embedding method that can highlight the semantics of the key subwords to facilitate effective embedding. In addition, frequent sensitive functions are utilized to filter out reasoning candidates. We implement a prototype called Raisin and apply it to identify the rare sensitive functions and detect bugs in large open-source code bases. We successfully discover thousands of previously unknown rare sensitive functions and detect 21 bugs confirmed by the developers. Some of the rare sensitive functions cause bugs even with a solitary invocation in the kernel. It is demonstrated that identifying them is necessary to enhance software reliability.

CCS CONCEPTS

• **Security and privacy → Software and application security**.

*Corresponding author.

KEYWORDS

Rare sensitive function, Bug detection, Analogical reasoning, Embedding

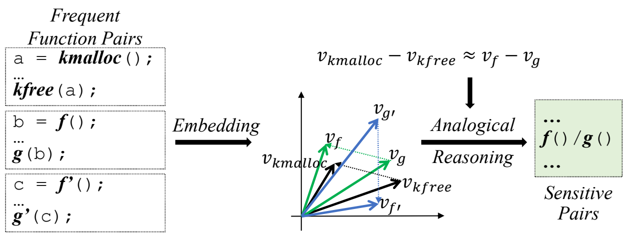
ACM Reference Format:
Jianjun Huang, Jianglei Nie, Yuanjun Gong, Wei You, Bin Liang, and Pan Bian. 2024. Raisin: Identifying Rare Sensitive Functions for Bug Detection . In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597503.3639165

1 INTRODUCTION

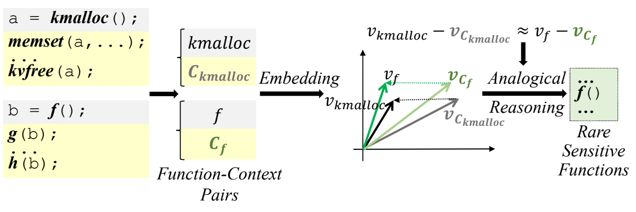
Bug detection is crucial for ensuring software reliability. An important and effective detection method is static analysis, which is often driven by a set of detection rules. These rules usually have an integral component called *sensitive functions* [6, 38], which are functions whose uses are prone to bugs. For instance, *free()* is a sensitive function that frees a memory chunk and is commonly involved in rules to detect double-free and use-after-free bugs. However, identifying a comprehensive list of sensitive functions is highly challenging, especially in large software systems, where there are a huge number of application-specific sensitive functions.

To mitigate the need for manually collecting sensitive functions, researchers have proposed a few data-mining and machine-learning based techniques for their identification [6, 35]. *SuSi* [35] trains an SVM classifier to determine if a function is sensitive. It however requires a large number of training samples, which may not be satisfiable in practice. *SinkFinder* [6] proposes a more practical method. It embeds functions to vectors and leverages *analogical reasoning* in NLP [32, 33] to infer unknown sensitive functions, using only a pair of known sensitive operations (i.e., one-shot learning). Analogical reasoning answers questions similar to the following "*If Germany is to Berlin, then France is to ?*" [32]. SinkFinder devises a pair-to-pair reasoning method, and identifies hundreds of frequent sensitive function pairs in the Linux kernel (short as *Linux*), such as *dma_fence_get()*/*dma_fence_put()*, from the one-shot example *kmalloc()*/*kfree()*.

Figure 1a shows the workflow of analogical reasoning for pair-wise functions in Linux. Two well-known sensitive functions, i.e., *kmalloc()* and *kfree()* in the top-left corner, are embedded to $v_{kmalloc}$ and v_{kfree} (the black solid vectors in the middle of Figure 1a). In



(a) Analogical reasoning for function-function pairs in SinkFinder



(b) Analogical reasoning for function-context pairs in Raisin

Figure 1: Basic idea of two kinds of analogical reasoning.

analogical reasoning, the correlations of the two functions can be approximated by the difference of the two vectors ($v_{kmalloc} - v_{kfree}$, i.e., the black dotted arrow). Other pairs of functions are searched for, for which the difference between the embedding vector resemble the difference between $v_{kmalloc}$ and v_{kfree}. In Figure 1a, the function pair ⟨*f()*, *g()*⟩ satisfies the condition, i.e., $v_{kmalloc} - v_{kfree} \approx v_f - v_g$, and is recognized as a potential allocation/deallocation function pair. In contrast, ⟨*f'()*, *g'()*⟩ is quite different and excluded.

This kind of pair-to-pair analogical reasoning can produce more precise result than a one-to-one method, because more information is introduced in the reasoning. However, it requires target functions to be frequent and pairwise. Unfortunately in practice, many sensitive functions are rare in a software system (e.g., only invoked few times). We term them *rare functions*. In this study, we consider functions with fewer than ten invocations as rare. When Linux v5.19 was released, it had more than 230K rare ones among the 400K functions. What is worse, rare functions often do not have co-occurring paired functions as the frequent ones do, and function pair-based reasoning techniques like SinkFinder tend to miss them. Therefore, there is a necessity to design a reasoning method specially for identifying rare sensitive functions.

It is also notable that, though few in number, rare sensitive functions should not be ignored in bug detection. For example, a vulnerability (CVE-2022-3542) detected in this study is related to the function *elfcorehdr_alloc()*, which is invoked only once in Linux. SinkFinder could not identify the function. According to our study in Section 4.1, SinkFinder can identify only 30 among thousands of rare allocation/deallocation functions.

In this paper, we propose a novel *context-based analogical reasoning* method, Raisin, to infer rare sensitive functions using a single function (not a pair). As far as we know, we are the first to focus on the problem of identifying rare sensitive functions for bug detection. Figure 1b illustrates the basic idea. Given a function, such as *kmalloc()*, Raisin generates a pair consisting of the function and its context denoted as $C_{kmalloc}$. Informally, the context in this study includes the data correlated function calls around all call sites of the function (with itself excluded). In Figure 1b, for example, the data correlated function calls of *kmalloc()* are ⟨*memset()*, *kvfree()*⟩, i.e., the ones data dependent on *kmalloc()*. We can then perform analogical reasoning based on function-context pairs. In Figure 1b, *f()* is considered similar to *kmalloc()* due to the similarity of the difference vectors (i.e., $v_{kmalloc} - v_{C_{kmalloc}} \approx v_f - v_{C_f}$ as denoted by the black and green dotted arrows).

To realize the idea, we address two critical technical challenges. First, context-based analogical reasoning still requires accurate embeddings of function names (including the rare ones). However, rare function names have poor embeddings. Subword-based embedding techniques can improve embeddings to some extent by emitting similar embedding vectors for functions with similar subwords, but the word segmentation method used in traditional methods like *fastText* [8] is not semantics-related and hence yields sub-optimal results. For example, fastText uses a fix-sized window to segment a word in an n-gram fashion to obtain the subwords. Although embedding based on such segmentation is acceptable for frequent functions that have sufficient training samples, it cannot produce high-quality embedding for rare functions according to our experiment.An ideal way is to split a function name into multiple semantic units, which may have various sizes. Besides, the subword indicating the operational logic of function is crucial (e.g., '*alloc*' in *elfcorehdr_alloc()* and '*uninit*' in *auxiliary_device_uninit()*), but traditional techniques do not treat such key subwords differently. To this end, we propose a weighted subword embedding technique. It leverages WordPiece [44] to split a function name to potential semantic units and introduces a weight for each unit when computing the embedding, emphasizing the operational key subword(s).

Second, there may be a large number of rare functions in a complex software project. Many of them are not sensitive. However, they could happen to be associated to function-context pairs that have close difference with the query, due to the estimation-based embedding. As such, they may be mistakenly classified as sensitive functions. We call them *noise functions*. We need to eliminate these noise functions. To address the issue, we first identify frequent sensitive functions via function-context pair based analogical reasoning, and then extract operational key subwords from them such as *alloc* and *free*. We then filter out those (noise) functions that do not contain the key subwords.

We apply Raisin to identify five kinds of rare sensitive functions, i.e., allocation, deallocation, lock, unlock and format string. The identified functions are further used to detect bugs. Our experiments on Linux v5.19, the FreeBSD kernel v13.1 (short as *FreeBSD*), OpenSSL v3.1.1, FFmpeg v6.0 and QEMU v8.1.0 show an average identification precision of 91% with a total of 16,897, whereas SinkFinder can identify zero and even not support to identify format string functions. In total, 27 bugs have been reported, with 21 confirmed by the developers. Raisin and the data are available at https://github.com/jlgithub66/rarefunctions.

Our work makes the following contributions:

- We reveal the importance of identifying rare sensitive functions. To the best of our knowledge, we are the first to explore automated identification of these functions.

```
386 // drivers/net/wireless/ath/ath11k/mhi.c
387 //   ath11k_mhi_register(struct ath11k_pci*)
388 mhi_ctrl = mhi_alloc_controller();
389 if (!mhi_ctrl)
390     return -ENOMEM;
391 ath11k_core_create_firmware_path(ab, ATH11K_AMSS_FILE,
392                  ab_pci->amss_path,
393                  sizeof(ab_pci->amss_path));
394 ret = ath11k_mhi_get_msi(ab_pci);
395 if (ret) {
396   ath11k_err(ab, "failed to get msi for mhi\n");
397   mhi_free_controller(mhi_ctrl);
398   return ret;
399 }
400 if (test_bit(ATH11K_FLAG_FIXED_MEM_RGN, &ab->dev_flags)) {
401   ret = ath11k_mhi_read_addr_from_dt(mhi_ctrl);
402   if (ret < 0)                                          +
403   - return ret;-     {
                            mhi_free_controller(mhi_ctrl);
                            return ret;
404 }                    }
405 // ...
```

Figure 2: A resource leak in Linux, in which the sensitive allocation function *mhi_alloc_controller* occurs only once in total.

- We propose a new context-based analogical reasoning method, which is further augmented with weighted subword embedding, context expansion and key subwords-based filtering.
- We develop a prototype RAISIN and evaluate it on five large code bases, Linux, FreeBSD, OpenSSL, FFmpeg and QEMU. Each has 1 ~ 33 MLoCs. RAISIN reports 16,897 rare sensitive functions with 91% accuracy. In contrast, the baseline SinkFinder identifies only 30 rare sensitive functions.
- Twenty-seven new bugs are detected in the popular systems, among which 21 bugs related to resource allocation/deallocation have been confirmed by the kernel developers.

2 MOTIVATION

We use a real resource leak bug we found in Linux (v5.19) to motivate our method. Figure 2 illustrates the simplified code snippet related to the bug. The function *mhi_alloc_controller()* at line 388 allocates a memory chunk and stores its address to *mhi_ctrl*. Along the execution path, when a read operation at line 401 fails, the program directly returns with an error code (line 403), without releasing the memory chunk. It can cause a resource leak. The bug can be fixed by replacing line 403 with the green box in Figure 2, i.e., freeing the memory before return.

The allocation function is rare, with only one occurrence in Linux. The corresponding deallocation function *mhi_free_controller()* has four invocations in total and is also a rare operation. It is hence very difficult for existing mining based approaches [3, 7, 40] to identify these functions. Moreover, analogical reasoning based methods like SinkFinder [6] cannot infer that *mhi_alloc_controller()* and *mhi_free_controller()* are a pair of allocation/deallocation functions since they do not satisfy the required co-occurrence frequency. Without the knowledge about the rare sensitive functions, we cannot leverage a static analyzer such as the Clang static analyzer [2] to detect the resource leak.

In contrast, RAISIN is able to infer that *mhi_alloc_controller()* is a rare sensitive function with a one-shot example *kmalloc()*. First, frequent functions are directly embedded into vectors and function-context based analogical reasoning is leveraged to infer frequent sensitive functions, denoted as F. Second, operational key subwords are extracted from F, e.g., '*alloc*', based on which candidate rare functions are collected, e.g., *mhi_alloc_controller()*. Third, we segment rare functions by WordPiece [44], embed the subwords, and then associate a frequency-based weight to each subword. A rare function embedding is computed by aggregating its involved weighted subword embeddings. By this means, the embedding of *mhi_alloc_controller* is dominated by '*alloc*', making it close to an allocation operation. Fourth, through a data flow analysis, data correlated functions in the neighborhood are collected as its context. Many of these functions may be rare too. To further improve effectiveness, rare functions in context are (recursively) inlined until frequent functions are yielded. In Figure 2, *mhi_free_controller()* at line 397 and *ath11k_mhi_read_addr_from_dt()* at line 401 use the data defined by *mhi_alloc_controller()* and thus are correlated with it. The two functions have only four and two invocations, respectively. They are hence inlined, leading to 25 correlated frequent functions included in the context of *mhi_alloc_controller()*, such as *kfree()*, *vfree()*, etc. Finally, analogical reasoning is performed between the query function-context pair and the target pair. We leverage the PairDirection [24], using cosine similarity to measure the pair-wise similarity. The highest similarity exceeds a predefined threshold and thus *mhi_alloc_controller()* is reported as a potential allocation function. With a set of allocation/deallocation functions configured, the Clang static analyzer can detect the leak at line 403.

3 APPROACH

3.1 Overview

RAISIN is a context-based analogical reasoning method to identify rare sensitive functions. Figure 3 shows the workflow. RAISIN takes a known sensitive function as the *query* (i.e., the one-shot in the top-right corner) and infers whether another given function (i.e., the *target*) is semantically similar to the query (❺), even when the target function is rare. Confirmed rare sensitive functions are then included in downstream static analyzers to detect bugs (❼).

Source code preprocessing (❶), i.e., extracting the sequences of function calls (*function sequences*), rare functions, call relations and other necessary information for later analysis, is omitted in this paper. Identifying sensitive functions is split into two parts, for frequent functions (in the upper dashed box) and for rare ones (in the lower dashed box), respectively. In the upper half of Figure 3, functions sequences are directly used to train a fastText [8] embedding model (❷) that has proven to generate acceptable embeddings for frequent functions. The context of each frequent function is clustered (❹), allowing every clustered context to concentrate on a kind of semantically similar operational environment. The vector difference between a target function and its context is compared with that for the query via analogical reasoning (❺), and functions analogous to the query are considered as frequent sensitive functions. The

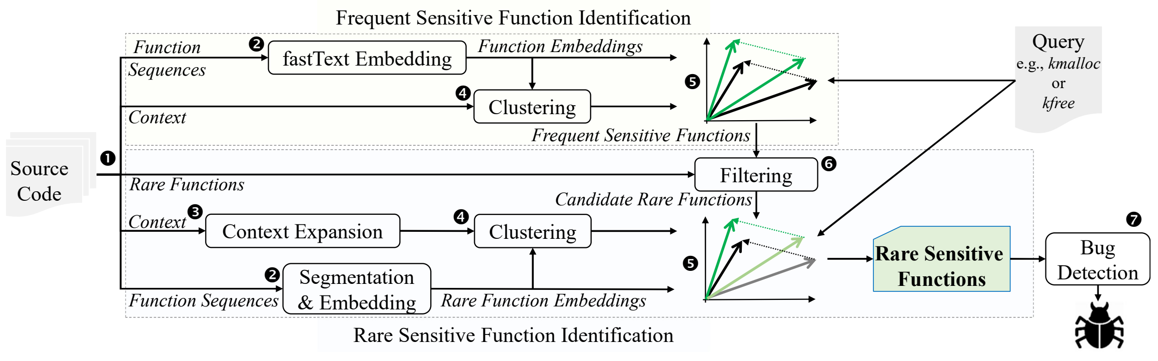


Figure 3: Overview workflow of Raisin.

lower half of Figure 3 has similar steps for context clustering (❹) and analogical reasoning (❺). The differences are three fold. First, the context of a rare function is expanded (❸) so that infrequent functions are (recursively) inlined until only frequent functions are included. Second, rare functions are not directly embedded. Instead, they are segmented by WordPiece [44], and their weighted subword embeddings are then aggregated to yield the function embeddings (❷). Third, rare noise functions are filtered out according to the operational key subwords collected from the previously identified frequent sensitive functions (❻). Similar analogical reasoning (❺) is then performed to infer rare sensitive functions.

We term our approach ***1+context*** analogical reasoning, where '*1*' denotes a function *f()* and '*context*' contains function calls around all call sites of *f()*. Note that, a function call must be data correlated with *f()* to be included in the context. In this study, two function calls (*f()* and *g()*) are considered *data correlated* if one of the following criteria is satisfied.

- *f()* is data dependent on *g()* (i.e., there is an argument in *f()* that uses a value defined by *g*),
- *g()* is data dependent on *f()*,
- *f()* and *g()* appear in the same control-flow path and refer to the same data (i.e., the same value is used in both function calls).

3.2 1+Context

In this section, we elaborate the *1+context* method, in which context clustering (❹) and analogical reasoning (❺) have the same procedure for both frequent and rare functions. Identifying rare sensitive functions takes an extra step for context expansion (❸).

3.2.1 Context Clustering. Various correlated functions can be included into the context of a function. Straightforwardly using them as a whole is not a good choice as different kinds of operations can interleave each other. Figure 4 shows a simplified example of the context for *kmalloc()*. Varied operations, such as *free*, *read*, *create* and so on, are correlated with *kmalloc()* at different call sites. However, a target allocation function may only be correlated with few types of these operations and thus its function-context difference would be far away from that for *kmalloc()*. For example, the operations such as *init*, *create* and many outliers (e.g., *printk()*) in Figure 4 do not exist in the context of *mhi_alloc_controller()*.

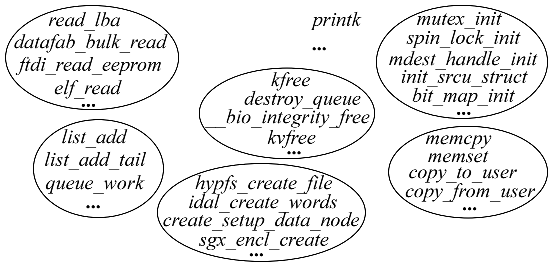


Figure 4: Simplfied context for *kmalloc()*.

Including them in an all-in-one context embedding will prevent *mhi_alloc_controller()* from being inferred analogous to *kmalloc()*.

To address the issue, we cluster the context according to function semantics which have been represented in embedding vectors. Without any prior knowledge about the number of function clusters, we adopt DBSCAN [14], a density-based clustering algorithm, to group the context of a function into sub-contexts, each concentrating a kind of operation. A function-context pair is then transformed to multiple function-sub-context pairs.

Lines 2 ~ 6 in Algorithm 1 describe the above process, where **E** holds the embedding vectors of the functions after step ❷ in Figure 3. Clustering is performed on the context functions and generates a group of sub-contexts for each target and query (lines 3 and 4). A centroid for each sub-context is computed to denote the sub-context, which is used to form the function-sub-context pair (lines 5 and 6).

3.2.2 Analogical Reasoning. Analogical reasoning is hence performed on the function-sub-context pairs to identify sensitive functions, as presented at Lines 7 ~ 10 in Alg. 1. All pairs associated with target f are compared with pairs for query q and the most similar two pairs are gathered as FC_f^a and FC_q^b (line 7). If their similarity is higher than a predefined threshold τ, we report f as a potential sensitive function that is analogous to q (lines 8 and 9). In this study, we set $\tau = 0.5$.

We use PairDirection [24] as the analogy metric. PairDirection measures cosine similarity between pairs and has been proven to be effective in measuring the relational similarity of word pairs in

Algorithm 1 Identifying if f is analogously similar to the query q given a similarity threshold τ. Contexts of functions are in $\mathbf{C}$ and the function embeddings are denoted by $\mathbf{E}$.

```
procedure IDENTIFY( $f, q, \tau, \mathbf{C}, \mathbf{E}$ )
    $v_f, v_q = \mathbf{E}[f], \mathbf{E}[q]$;
    $G_f = \text{DBSCAN}\{\mathbf{E}[i] \mid \forall i \in \mathbf{C}[f]\}$;
    $G_q = \text{DBSCAN}\{\mathbf{E}[j] \mid \forall j \in \mathbf{C}[q]\}$;
    $FC_f = \{\langle v_f, v_{C_f}\rangle \mid \forall C_f \in G_f, v_{C_f} = \text{avg}(\sum_{\forall i \in C_f} \mathbf{E}[i])\}$;
    $FC_q = \{\langle v_f, v_{C_q}\rangle \mid \forall C_q \in G_q, v_{C_q} = \text{avg}(\sum_{\forall j \in C_q} \mathbf{E}[j])\}$;
    $FC_f^a, FC_q^b = arg\max\{sim(FC_f^i, FC_q^j)\}$;
    if $sim(FC_f^a, FC_q^b) > \tau$ then
        $f$ is a potential sensitive function analogous to $q$;
    end if
end procedure
```

NLP [24]. Eq. 1 shows how the similarity at line 7 is calculated for two function-sub-context pairs.

$$sim(\langle v_f, v_{C_f^i}\rangle, \langle v_q, v_{C_q^i}\rangle) = \cos(v_{C_f^i} - v_f, v_{C_q^i} - v_q) \quad (1)$$

3.2.3 Context Expansion. We expand the context only for rare functions. A rare function has a very limited number of invocations, resulting in very few functions in its context, with some contextual functions being rare too. Take Figure 2 as an example. The function *mhi_alloc_controller()* has only one invocation and three correlated functions, i.e., *mhi_free_controller()*, *ath11k_mhi_read_addr_from_dt()* and *mhi_register_controller()* (not present in Figure 2 due to space limit). The three functions have four, two and two calls, respectively, meaning that they are also rare. Such a condition makes the context unreliable in sensitive function identification.

We propose a context expansion scheme to augment the context by recursively gathering the correlated frequent functions of a target rare function. Algorithm 2 shows the workflow for a single occurrence of a rare function f, and outputs a list of collected frequent functions as the context $\mathbb{C}$ (line 25). Initially, the *caller* of f is acquired. We set $\mathcal{V} = \emptyset$ for the target function f, meaning that all the arguments and return value at the *callsite* will be tracked by *get_related_func_in_caller()* at line 5. The analysis collects all calls in the function that are data correlated with f at the *callsite*. From line 6 to 15, we inspect each call in $\mathbb{T}$. A frequent call is added into $\mathbb{C}$ (line 9), while a rare one is inlined and the identified context $\mathbb{D}$ is appended to $\mathbb{C}$ (lines 11~13). Line 11 gets the formal parameters of interest in *callee* and line 12 recursively invokes the algorithm to collect the context in *callee*. As we go into the callee to collect function calls correlated to the parameters, tracking back to any call site of the callee will include uncorrelated context or repeat the analysis, and should be forbidded (lines 16~18). To this end, we use a *null* call site to perform the expansion, making the algorithm focus on the given function only (line 3). Besides, a nonempty $\mathcal{V}'$ will force the related function collector at line 5 to care about only the provided variables (i.e., the formal parameters).

We also collect correlated functions at those points that indirectly invoke f (lines 19 ~ 24). Every call site of *caller* will be checked to see if correlated variables exist. For example, f may use a parameter of *caller* or *caller* returns a value that is defined by f. The data

Algorithm 2 Context collection and expansion for a rare function f at a specific *callsite*, given the call graph *cg*. The variables in $\mathcal{V}$ will be tracked if $\mathcal{V}$ is not empty.

```
procedure CTX_EXPAND( $f$, callsite, cg, $\mathcal{V}$ )
    $\mathbb{C}$ = [];
    caller = (callsite ≠ null) ? get_caller(cg, callsite) : $f$;
    cfg = get_cfg(caller);
    $\mathbb{T}$ = get_related_func_in_caller(cfg, callsite, $\mathcal{V}$);
    for cs ∈ $\mathbb{T}$ do
        callee = get_invoke_target(cs);
        if callee is a frequent function then
            $\mathbb{C}$ = concat($\mathbb{C}$, callee);
        else
            $\mathcal{V}'$ = extract_correlated_vars(callsite, cs);
            $\mathbb{D}$ = CTX_EXPAND(callee, null, cg, $\mathcal{V}'$);
            $\mathbb{C}$ = concat($\mathbb{C}$, $\mathbb{D}$);
        end if
    end for
    if callsite == null then
        return $\mathbb{C}$;
    end if
    for cs ∈ get_callsites(cg, caller) do
        if ($\mathcal{V}'$ = extract_correlated_vars(cs, callsite)) ≠ ∅ then
            $\mathbb{D}$ = CTX_EXPAND(caller, cs, cg, $\mathcal{V}'$);
            $\mathbb{C}$ = concat($\mathbb{C}$, $\mathbb{D}$);
        end if
    end for
    return $\mathbb{C}$;
end procedure
```

correlation can potentially introduce substantial functions via a recursive expansion at line 21.

3.3 Rare Function Embedding

Rare functions are analogous to the out-of-vocabulary words in NLP. Due to their rarity, existing word embedding techniques (e.g., Word2vec [32] and fastText [8]) could not generate as high-quality embeddings as for frequent functions in the task of identifying rare sensitive functions. Fortunately, we observe that, though the sensitive functions are rare, they usually contain a similar or the same operational subword to indicate their sensitive behaviors, compared to the frequent ones. For example, *mhi_alloc_controller()* includes *alloc* that denotes an allocation and appears commonly in both frequent and rare functions. In other words, the sensitivity-oriented operational subwords (i.e., *key subwords*) are general despite the function rarity.

To highlight the key subwords in the embedding, we present a weighted subword embedding technique (❷ in the lower half in Figure 3). Specifically, we use a mature algorithm, WordPiece [44] that is also leveraged by BERT [11], to segment the function names into potential semantic units (i.e., subwords) and then remove the prefix and postfix '_' in each subword. A subword-based Word2vec embedding model is trained. We use Word2vec because it is lightweight to run on a normal PC and there is not a need to care about further subword information in the segmented corpus. Subword frequency based weight is then applied to each unit when computing the embedding of a rare function. Given the subword corpus O and the corresponding subword embedding model S, the embedding of a rare function f is computed as Eq. 2.

$$v_f = \frac{\sum_{s \in f} \mathrm{S}[s] \times N(s, O)}{\sum_{s \in f} N(s, O)} \tag{2}$$

In Eq. 2, $\mathrm{S}[s]$ retrieves the embedding vector of the input subword that is a unit in f, and $N(s, O)$ counts the number of occurrences of subword s in the segmented corpus. According to the computation, a rare function's embedding is dominated by the most frequent subword. Therefore, sensitive functions that contain usual operational key subwords will be spotted for sensitivity identification.

Take *mhi_alloc_controller()* in Figure 2 as an example. It is split into three subwords, '*mhi*', '*alloc*' and '*controller*'. The counts of the subwords are 1634, 54207 and 2308, respectively, through the whole Linux. As a result, the operational key subword '*alloc*' should have the largest impact (>90%) on the function embedding to highlight its sensitivity. In other words, the embedding vector of *mhi_alloc_controller()* should be dominated by the action '*alloc*', while the module '*mhi*' and the object '*controller*' have little impact in preventing it being analogous to an allocation function.

3.4 Key Subwords-based Filtering

We filter the target functions (❻) to obtain the most likely candidates for rare sensitive function identification, to avoid the impact of noise functions. We manually audit the reported frequent sensitive functions and determine whether a function is a sensitive one of interest by inspecting how it is implemented and used. The operational key subwords (see Section 3.3) are extracted from the confirmed sensitive functions. Manually extracting the key subwords is a limitation of our approach, but the manual process can emit accurate key subwords that do represent the core actions of the functions during auditing.

Using *kfree()* as a query for Linux, we can extract many operational key subwords such as *free* in *tee_shm_free()*, *put* in *gss_put_ctx()*, *destroy* in *damon_destroy_ctx()*, *release* in *op_release()* and so on. For *kmalloc()*, we have *alloc* in *kzalloc()*, *get* in *usb_get_phy()*, *create* in *debugfs_create_dir()*, etc. Though which keywords are extracted is project-specific, the extraction is general and can be applied to other projects. For example, we can extract *delete* from *delete_unrhdr()* in FreeBSD, *new* from *EVP_CIPHER_CTX_new()* in OpenSSL, *get* from *ff_get_video_buffer()* in FFmpeg, and *release* from *qcow2_cache_table_release()* in QEMU.

The extracted key subwords are then used to filter the rare functions. As a result, only a rare function containing some key subword will be collected as a candidate for analogical reasoning. For instance, *mhi_alloc_controller()* is a candidate allocation function, and *mcba_usb_get_free_ctx()* is a candidate allocation for *kmalloc()* or a candidate deallocation for *kfree()*. Further analogical reasoning will determine whether they are actually allocation or deallocation functions.

Table 1: Large software systems of evaluation

TOE	LoC	#Functions	#Rare Functions
Linux v5.19	31,317,255	456,086	241,366
FreeBSD v13.1	8,399,769	194,913	116,843
OpenSSL v3.1.1	1,048,122	14,206	9,836
FFmpeg v6.0	1,600,078	17,417	14,168
QEMU v8.1.0	22,479,680	86,980	75,443

3.5 Bug Detection

Identified rare sensitive functions are ranked according to their analogy similarity in a descending order for auditing. A simple heuristic is adopted to pair the functions to form, for example, allocation/deallocation and lock/unlock pairs. During auditing the frequent sensitive functions, we gather the knowledge about which two keywords are usually paired, e.g., *alloc*/*free*, *get*/*put*, etc. Then, two functions are paired if their names contain the paired key subwords and have the same prefix and postfix. The pairing results can facilitate us to audit the identified rare sensitive functions.

Confirmed rare sensitive functions will be used to detect bugs, involving both automated static detection and manual examination.

We choose the wide-used open-source Clang static analyzer [2] (short as *Clang* afterwards) as the automated detector. The Clang checkers are implemented in its frontend and detect bugs along with compilation. Clang does not require the sensitive functions to be paired. To detect a memory leak, for example, Clang tracks the memory chunk returned by an allocation function and reports a leak if the chunk is not passed to any function in the deallocation function list. It is a limitation, but enhancing the Clang checkers is beyond the scope of this paper.

We also manually examine the top ranked or randomly selected rare sensitive functions to see if bugs exist, as automated static detector like Clang may have some limitations in analyzing complicated software systems like Linux. In addition, some projects (e.g., FreeBSD) may not be compilable in our working environment and automatic detection is hence infeasible.

4 EVALUATION

In this section, we evaluate Raisin on large software systems to answer the following four questions.

RQ1: How effective is the rare sensitive function identification (Section 4.1) ?

RQ2: Are the proposed filtering, weighted subword embedding and context expansion necessary in Raisin (Section 4.2)?

RQ3: Is the bug detection effective with the identified rare sensitive functions (Section 4.3)?

RQ4: Is the approach efficient (Section 4.4) ?

All experiments are conducted on a desktop computer with 16 GB memory, an Intel Core i5-10400F CPU @ 2.9GHz and Ubuntu 20.04. Source code parsing and lightweight data flow analysis are implemented in Java on top of *fuzzyc2cpg* v1.1.19 [1], which allows to preprocess code bases even if they are not compilable in the experiment environment, e.g., FreeBSD. The static analyzer is LLVM/Clang 12.0.0. All other components, including clustering, expansion, training and reasoning, etc., are implemented in Python.

Table 2: Identification result. '#RSF' for number of rare sensitive functions and 'P' for precision.

Category	Linux			FreeBSD			OpenSSL			FFmpeg			QEMU		
	Query	#RSF	P	Query	#RSF	P	Query	#RSF	P	Query	#RSF	P	Query	#RSF	P
Alloc	*kmalloc*	3,223	83%	*malloc*	2,342	78%	*OPENSSL_malloc*	244	92%	*av_malloc*	127	94%	*g_malloc*	1,204	82%
Dealloc	*kfree*	4,990	92%	*free*	1,759	91%	*OPENSSL_free*	228	97%	*av_free*	73	96%	*g_free*	352	92%
Lock	*mutex_lock*	763	65%	*pthread_ mutex_lock*	603	76%	*CRYPTO_THREAD _lock_new*	11	91%	*pthread_ mutex_lock*	2	100%	*qemu_mutex _lock*	38	90%
Unlock	*mutex_ unlock*	338	90%	*pthread_ mutex_unlock*	239	93%	*CRYPTO_THREAD _lock_free*	6	100%	*pthread_ mutex_unlock*	2	100%	*qemu_mutex _unlock*	23	100%
FormatStr	*sprintf*	46	96%	*sprintf*	257	78%	*sprintf*	10	100%	*snprintf*	3	100%	*sprintf*	14	100%
Overall		9,360	85%		5,200	83%		499	96%		207	98%		1,631	93%

We have evaluated RAISIN on five large open-source software systems, as shown in Table 1, two kernels and three popular projects. Each has 1 MLoC (million lines of code) to 31 MLoC and 14K to 456K functions. The number of rare functions (with fewer than ten invocations) ranges from 9.8K to 241K.

4.1 Effectiveness

We apply RAISIN to identify five categories of rare sensitive functions in the five software systems, using the most popular sensitive function in each system as a query (e.g., *OPENSSL_malloc()* in OpenSSL). Table 2 shows the queries and the reported numbers (*#RSF*) of rare sensitive functions. It is impractical to manually confirm all the reported functions, so we inspect at most 100 for each query to a system. For a query with more than 100 reported functions, we *randomly* select 100. Their similarity range from 0.503 to 0.927. The auditing took one-person day.

RAISIN identifies 9360, 5200, 499, 207 and 1631 rare sensitive functions, respectively, in five systems and the average precision ranges from 83% to 98%. There are only a few rare sensitive functions for lock/unlock/format string categories in OpenSSL and FFmpeg, but the numbers are relatively large in the kernels that have much more rare functions. QEMU contains more lines of code than FreeBSD, but fewer functions (see Table 1) leads to much fewer rare sensitive ones. The result shows that RAISIN is effective in identifying rare sensitive functions.

Comparison with SinkFinder [6]. We compare with SinkFinder, which analogically reasons about frequent function pairs from a one-shot pair, using its default setting. SinkFinder leverages GCC to compile and preprocess the source code and fails to analyze FreeBSD that is not compilable on Ubuntu. In addition, SinkFinder cannot work for the format string category, which does not have paired functions as allocation/deallocation or lock/unlock.

We use the same query functions to form query pairs and identify allocation/deallocation pairs. SinkFinder successfully identifies 285 sensitive function pairs, with 15/4, 5/3, 0/0, 2/1 distinct rare allocation/deallocation functions, respectively, in Linux, OpenSSL, FFmpeg and QEMU. SinkFinder can discover the 30 rare functions because it extracts up to 100 control flow paths in each function, which could have a chance to increase the frequency of co-occurring rare sensitive function pairs in the mining corpus. However, the result demonstrates that analogical reasoning for frequent function pairs is ineffective to identify rare sensitive functions compared to RAISIN. The reasons are twofold. First, the pairs often may not occur in the same function. And second, even if the frequency of a function pair is inflated by considering more paths, the frequency may not satisfy the minimum requirement of SinkFinder.

Table 3: Ablation study on Linux. The symbols √ and × indicate whether a technique, i.e., filtering (*Fil.*), weighted subword embedding (*WSE*) or context expansion (*Exp.*), is enabled.

ID	Fil.	WSE	Exp.	Precision			Recall			F1
				Alloc	Dealloc	Avg.	Alloc	Dealloc	Avg.	
1	×	×	×	11.8%	13.4%	12.6%	9.9%	6.0%	8.0%	9.8%
2	×	×	√	15.1%	13.0%	14.1%	25.6%	16.4%	21.0%	14.9%
3	×	√	×	24.6%	13.5%	19.1%	71.6%	76.7%	74.2%	30.4%
4	×	√	√	32.9%	12.5%	22.7%	90.4%	94.5%	92.5%	36.6%
5	√	×	×	71.0%	80.5%	75.8%	9.9%	6.0%	8.0%	11.1%
6	√	×	√	82.2%	92.5%	87.4%	25.6%	16.4%	21.0%	33.9%
7	√	√	×	75.4%	85.9%	80.7%	71.6%	76.7%	74.2%	77.3%
8	√	√	√	80.5%	90.7%	85.6%	90.4%	94.5%	92.5%	88.9%

4.2 Ablation Study

In this section, we evaluate the effectiveness of the three techniques in RAISIN, i.e., key subword-based filtering, weighted subword embedding and context expansion. We apply eight combinations of the techniques to identify rare allocation/deallocation functions in Linux. In total, more than 50K rare sensitive functions are reported. It takes two authors about one week to confirm 2,869 allocation and 4,788 deallocation functions. These confirmed functions are used to estimate the ground truth. The precision, recall and F1 for each group are then computed.

In Table 3, we can see that, the filtering makes a great contribution to the precision (e.g., #4⟶#8) and the weighted subword embedding helps RAISIN find much more rare sensitive functions compared with the full-name Word2vec embedding (e.g., #6⟶#8). Context expansion generally increases both precision and recall (e.g., #7⟶#8). Group #6 achieves the highest precision but very low recall (21.0%) with only 1,740 functions reported. Group #4 has the same highest recall with group #8 but too many reported functions (>46K) emit a relatively low precision (22.7%). By measuring the overall performance, we observe that, any single technique cannot produce sufficiently high F1 (#2, #3 and #5) and the combinations of two techniques can at most produce an F1 of 77.3% (#7). Compared to the other groups, RAISIN (i.e., #8) reports over 8,000 functions and achieves the best overall F1 score (88.9%).

Though context expansion does not show as great contribution to precision or recall as the other two techniques, it is necessary to find more true rare sensitive functions and eliminate false positives. The function *mhi_alloc_controller()* in Figure 2 is a good example.

Table 4: Detected bugs that have been confirmed by the maintainers.

ID	Target Systems	Rare Sensitive Function	#Occurrences	Bug Type	Confirmation
1	Linux	*mhi_alloc_controller*	1	Memory Leak	[Github]: 43e7c350
2	Linux	*sfp_alloc*	1	Memory Leak	[Github]: 0a18d802
3	Linux	*xhci_alloc_stream_ctx*	1	Memory Leak	[Github]: 7e271f42
4	Linux	*audioreach_alloc_graph_pkt*	1	Memory Leak	[Github]: df5b4aca
5	Linux	*elfcorehdr_alloc*	1	Memory Leak	[Github]: 12b9d301
6	Linux	*aq_nic_init*	2	Memory Leak	[Github]: 65e5d27d
7	Linux	*nfp_cpp_area_alloc*	3	Memory Leak	[Github]: c56c9630
8	Linux	*auxiliary_device_uninit*	1	Use After Free	[Github]: 1c11289b
9	Linux	*nouveau_bo_del_ttm*	3	Use After Free	[Github]: 540dfd18
10	Linux	*amdgpu_vm_init*	2	Resource Leak	[Github]: c3c48339
11	Linux	*hfi1_alloc_ctxt_rcv_groups*	2	Memory Leak	[Github]: aa2a1df3
12	Linux	*init_mr_info*	2	Memory Leak	[Github]: b3236a64
13	Linux	*damon_new_ctx*	3	Memory Leak	[Github]: 188043c7
14	Linux	*init_rx_sa*	1	Resource Leak	[Github]: c7b205fb
15	Linux	*init_tx_sa*	1	Resource Leak	[Github]: c7b205fb
16	Linux	*hsi_claim_port*	3	Resource Leak	[Github]: b28dbcb3
17	Linux	*bnx2x_frag_alloc*	3	Memory Leak	[Github]: b43f9acb
18	Linux	*sec_queue_aw_alloc*	1	Inconsistent Argument	[Github]: 32c0f7d4
19	Linux	*mcba_usb_get_free_ctx*	2	Resource Leak	[Lore]: 20221124144532. 6u3hnvb6b2ninlxy@pengutronix.de
20	FreeBSD	*ext4_ext_alloc_meta*	2	Memory Leak	[Bugzilla]: 265071
21	FreeBSD	*bhnd_alloc_pmu*	3	Resource Leak	[Bugzilla]: 265147

#1 ~ #18: Submitted patches have been applied to the master branch of Linux, which can be retrieved by appending the commit ID to https://github.com/torvalds/linux/commit/. #19: The patch was confirmed and applied to linux-can by the maintainer, reachable by appending the email to https://lore.kernel.org/all/. #20 and #21: FreeBSD uses Bugzilla to track the issues and the two confirmations can be reached by searching the issue ID in https://bugs.freebsd.org/bugzilla.

Without context expansion, the function is not recognized as a sensitive one because of a low similarity (0.31) in group #7. However, in group #8, the similarity increases to 0.65, making it flagged sensitive. In addition, some functions may be considered sensitive due to the operational key subwords in group #7. For example, *db2k_initialize_tmrs()* and *pqm_get_user_queue()* contain the subwords *init* and *get*, respectively. In group #7, both are reported because their similarity values are 0.55 and 0.57, above the threshold. But they are actually insensitive. In group #8, the similarity is reduced to 0.35 and 0.31, with the support of context expansion. Therefore, they are eliminated.

Overall, the ablation study demonstrates the necessity of the proposed techniques. All three techniques can improve rare sensitive function identification.

4.3 Bug Detection

We configure the Clang static analyzer [2] with the identified rare allocation/deallocation functions to detect resource/memory-related bugs. In addition, we manually inspect 1,000 randomly selected functions for Linux and 100 for FreeBSD. The inspection takes only two days for two authors since each function has very few occurrences.

We have reported 21 and six bugs to the Linux and FreeBSD maintainers, respectively. Among them, 19 and two are confirmed by the developers. Table 4 lists the confirmed bugs. The last column shows the confirmation information, e.g., the commit IDs.

Note that, bug #18 in Linux was spotted by manual inspection as Clang does not support this kind of bug. The developer uses an inconsistent argument when calling the sensitive function, i.e., '*W*' in *SEC_QUEUE_A**W**_FROCE_NOALLOC* was misspelled as '**R**', which is consistent to another sensitive function.

From the *#Occurrences* column, we can see that nine (47.4%) of the Linux bugs result from sensitive functions that are invoked only once in Linux. All the other bugs are associated with rare functions that have at most three occurrences.

The real bugs have demonstrated the performance of bug detection with the identified rare sensitive functions. It also shows that rare sensitive functions can lead to many bugs. Therefore, identifying them is a crucial step for bug detection. As a contrast, SinkFinder [6] cannot identify these rare sensitive functions and thus these bugs would be missed.

4.4 Efficiency

We break down the time cost of one complete pass step by step. Table 5 shows the time for identifying rare allocation functions and detecting related bugs using Clang. Automated bug detection on FreeBSD is not applicable and the total time includes only the other steps. Note that, filtering is a human effort and for each target system, we take 15~30 minutes to extract the operational key subwords.

In addition to the Clang-based bug detection, preprocessing costs the most time for each target. This step involves parsing source code and extracting context with an intra-procedural data flow analysis.

Table 5: Time cost for identifying rare allocation functions and detecting related bugs.

Target	Preprocessing	Frequent Functions		Rare Functions				Bug Detection	Total
		Embedding	Analogy	Filtering	Embedding	Expansion	Analogy		
Linux	8h43m	3m25s	5m36s	30m	5m11s	15m28s	1m21s	11h37m	21h21m
FreeBSD	3h42m	46s	16s	30m	2m13s	1m43s	12s	N/A	4h17m
OpenSSL	46m23s	18s	5s	15m	40s	21s	2s	14m18s	1h17m
FFmpeg	1h24m	22s	7s	15m	1m17s	21s	5s	39m24s	2h21m
QEMU	5h19m	1m9s	38s	30m	3m19s	4m38s	14s	3h24m	9h23m

```
923 // driver/hsi/clients/ssi_protocol.c
924 //   in ssip_pn_open(struct net_device *)
925 err = hsi_claim_port(cl, 1);
926 if (err < 0) {
927   dev_err(&cl->device, "SSI port already claimed\n");
928   return err;
929 }
930 err = hsi_register_port_event(cl, ssip_port_event);
931 if (err < 0) {
932   dev_err(&cl->device, "Register HSI port event failed
          (%d)\n", err);
933 +     hsi_release_port(cl);
934   return err;
935 }
```

Figure 5: Claimed port not released in Linux.

Embedding rare functions requires more time than that for frequent ones, as the former contains three steps (i.e., segmentation, training and weighted subword embedding) while the latter consists of only a one-step training. However, we notice that, analogical reasoning (including context clustering) for rare sensitive functions is faster than that for frequent ones. The filtering has significantly reduced the number of rare functions for analogy and the rare functions have fewer clustered contexts than frequent ones.

Recall that the time cost in Figure 5 is only for allocation function identification. When a different query is chosen, we can reuse the proprocessing result, embedding vectors and expanded contexts. In other words, we need just to redo analogy for frequent and rare functions plus a filtering step, which can be quickly finished according to Table 5. Hence, we believe the time cost is acceptable for large software systems.

4.5 Case Study

In this section, we present two more cases.

Case 1: Claimed port is not released. Figure 5 presents a resource leak bug (#16 in Table 4). The rare sensitive function *hsi_claim_port()* claims a port for an HSI (High speed synchronous Serial Interface) client at line 925. The claim process involves incrementing a reference count, setting flags for sharing, etc. When the port is no longer needed at line 934, it should be released with *hsi_release_port()* as shown in the green box. Analogous to *kmalloc()* and *kfree()*, *hsi_claim_port()* allocates a kind of special resource and *hsi_release_port()* deallocates that resource. In Linux v5.19, both functions are rare, with only three and four calls, respectively. What is worse, *hsi_claim_port()* does not call any explicit memory allocation functions such as *kmalloc()*, making static analyzers unable to detect the leak even with powerful inter-procedural analysis. Raisin identifies that *hsi_claim_port()* is a kind of allocation operation. With the knowledge, Clang can report the bug via an intra-procedural analysis.

Case 2: Initialized virtual memory (vm) instance is not finalized. Figure 6 shows a simplified code snippet for another resource leak in Linux. The function *amdgpu_vm_init()* performs a series of initializations to the given AMD GPU vm instance, which has two invocations in total. In the presented code snippet, a memory chunk is allocated before the initialization at line 1151 and deallocated at line 1222 if any subsequent operation fails (e.g., ❷) or the execution succeeds (❶). Along the normal work flow (❶), the vm instance will be finalized via *amdgpu_vm_fini()* at line 1217 and freed at line 1222. However, when the immediate operation (line 1155) after the initialization fails, the error handling (③) does not call *amdgpu_vm_fini()* to tear down the vm instance, leading to a resource leak. Later, if mapping meta data at line 1161 fails, error handling (❹) can well address the finalization. Therefore, the bug is caused because of a wrong execution flow (③).

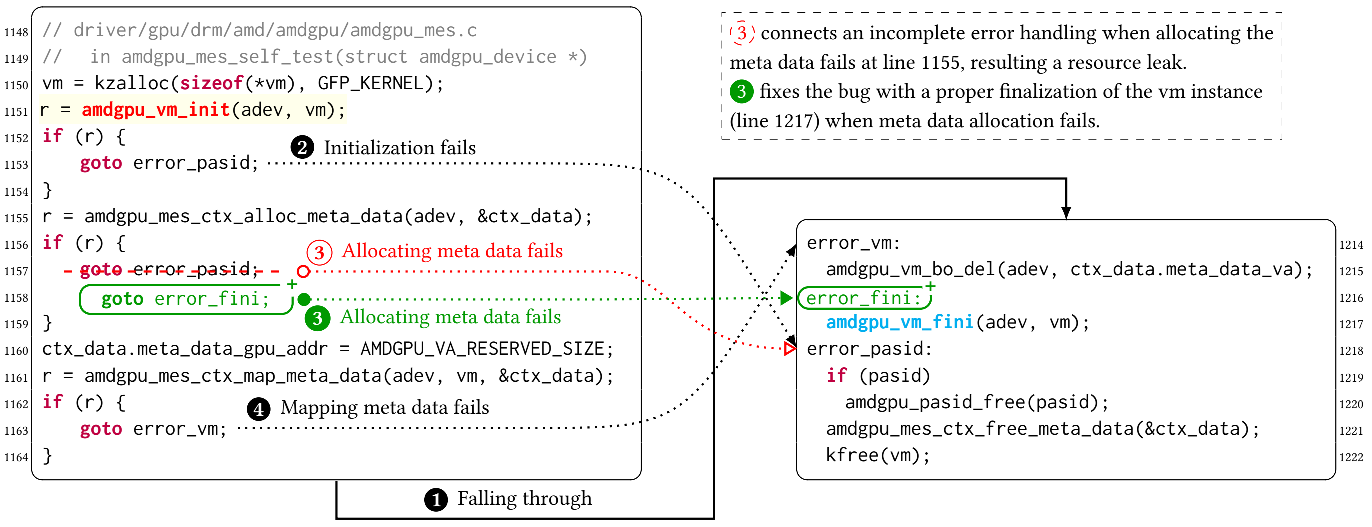
Raisin successfully highlights that *amdgpu_vm_init()* is a kind of resource allocation and therefore the bug in Figure 6 is detected. Fixing the bug is to insert a label (as in the right green box) and route the error handling to the label (❸, i.e., replacing the crossed line 1157 with the code in the left green box).

5 DISCUSSION

Though we have demonstrated the effectiveness of Raisin and bug detection, there are some points to be further discussed.

Sensitivity-oriented tokenizer. WordPiece [44] is a generic tokenizer and it does not pay special attention to the semantics of function subwords. We have also tried another popular segmentation algorithm in NLP, i.e., BPE [37], but it shows even worse results. In fact, existing popular word segmentation algorithms such as WordPiece [44] and BPE [37] do not fully fit the requirement of highlighting the key behavior for sensitive function identification. The statistics-based techniques do not take the semantics of the key subwords into account, and they improperly or even incorrectly segment certain function names, which impedes subsequent embedding and reasoning. Therefore, a subword semantics-oriented segmentation algorithm can improve Raisin. We will address this big challenge in our future work.

Large code models. While we trained a lightweight model specific for each given software system, we think that the large code models [15, 17, 18] pre-trained on many different code projects


❷ Initialization fails
③ Allocating meta data fails
❸ Allocating meta data fails
❹ Mapping meta data fails
❶ Falling through

```
1148 // driver/gpu/drm/amd/amdgpu/amdgpu_mes.c
1149 //   in amdgpu_mes_self_test(struct amdgpu_device *)
1150 vm = kzalloc(sizeof(*vm), GFP_KERNEL);
1151 r = amdgpu_vm_init(adev, vm);
1152 if (r) {
1153     goto error_pasid;
1154 }
1155 r = amdgpu_mes_ctx_alloc_meta_data(adev, &ctx_data);
1156 if (r) {
1157   - goto error_pasid;
1158     goto error_fini; +
1159 }
1160 ctx_data.meta_data_gpu_addr = AMDGPU_VA_RESERVED_SIZE;
1161 r = amdgpu_mes_ctx_map_meta_data(adev, vm, &ctx_data);
1162 if (r) {
1163     goto error_vm;
1164 }
```

```
1214 error_vm:
1215   amdgpu_vm_bo_del(adev, ctx_data.meta_data_va);
1216 error_fini: +
1217   amdgpu_vm_fini(adev, vm);
1218 error_pasid:
1219   if (pasid)
1220     amdgpu_pasid_free(pasid);
1221   amdgpu_mes_ctx_free_meta_data(&ctx_data);
1222   kfree(vm);
```

③ connects an incomplete error handling when allocating the meta data fails at line 1155, resulting a resource leak.
❸ fixes the bug with a proper finalization of the vm instance (line 1217) when meta data allocation fails.

Figure 6: A case of non-finalized vm instance in Linux and the corresponding simplified execution flows.

may be helpful in generating higher-quality embedding vectors and augment sensitive function identification. However, lack of the aforementioned tokenizer may also reduce a large model's capability of recognizing analogous rare sensitive functions due to the out-of-vocabulary problem, compared to their achievement in other downstream tasks, e.g., code clone detection, code search, etc.

A better static checker. We use Clang static analyzer to perform bug detection with the discovered rare sensitive functions. However, the general-purpose checker does not show great performance on large and complicated software systems like Linux. It emit 427 warnings for Linux but manual inspection confirms only 21 real bugs. In addition, some special code patterns are not supported by the checker, e.g., transmitting a deallocation function as an argument of a function. We believe a customized static checker is necessary and will be developed for our future studies.

The order of filtering and reasoning. Filtering the target functions before the analogical reasoning introduces the potential to miss some highly semantically similar (HSS) functions that have completely different names. However, the impact is very limited. We conduct a study to perform the filtering after the reasoning with a filtering threshold as 0.75 to ensure the HSS functions can be kept. Above the threshold, there are 96 allocation functions in Linux. Among them, 39 do not contain any key subwords for allocation, with 38 confirmed false positives and one uncertain (i.e., *walk_down_tree()*) due to its complicated logic. Therefore, the filtering strategy in RAISIN can miss very few, if not zero, HSS functions. Moreover, RAISIN takes only 80 seconds in the analogy step, while more than ten minutes are needed if filtering is done after the reasoning.

6 RELATED WORK

Engler et al. [13] and Kremenek et al. [22] defined semantic properties that sensitive functions should satisfy. Ganapathy et al. [16] leveraged the domain knowledge to cluster resource manipulations as the fingerprints of sensitive operations. Saha et al. [36] and Emamdoost et al. [12] discover allocation functions by heuristically checking the returned pointers and then infer the corresponding deallocation functions in the control-flow paths. SuSi [35] prepared hundreds of known taint sources and sinks in Android framework, based on which an SVM classifier is trained. The classifier consumes the extracted code features and predicts source/sink functions to facilitate taint analysis on Android apps. RAISIN chooses another path compared with the above techniques. Instead of depending on predefined heuristics or a mass of prior knowledge about target system, RAISIN requires only a few known sensitive functions and usually a single one is enough. SinkFinder [6] utilized a more recent NLP-inspired technique, i.e., analogical reasoning, to automatically identify hundreds of application-specific security functions using only one pair of query functions. We have demonstrated that SinkFinder is not suitable for identifying rare sensitive functions. However, inspired by SinkFinder, we design RAISIN particularly to effectively identify rare sensitive function.

Various techniques to discover the programming rules with machine learning for bug detection have also been proposed [4, 7, 9, 21, 23, 25, 26, 28, 29, 39–41]. Some studies also explicitly identify sensitive operations, with or without prior knowledge. Based on a set of security check functions, AutoISES [38] infers the security rules, i.e., which sensitive operations should be protected by security checks. RRFinder [42] heuristically and iteratively mines resource acquiring-releasing functions and the specifications for object-oriented programming languages. PR-Miner [25] and NAR-Miner [7] mines the specification for sequential function usages. APISan [46], FICS [3] and Crix [31] leveraged majority-voting to infer common behaviors and detect inconsistencies as bugs. Wu et al. [43] identifies paired functions involved in error handling paths, but some rare functions are not called in pairs due to various reasons. For example, the deallocation function of *xhci_alloc_stream_ctx()* in Table 4 is forgotten, leading to a resource leak. Nguyen et al. [34] embedded APIs using Word2vec to mine the API pairs sharing same usage relations. Our work addresses the problem of identifying rare sensitive functions using limited prior knowledge. The function rarity makes pattern mining non-applicable.

In fact, many bug detection techniques do require the knowledge of sensitive functions in advance. For example, Chelf et al. [10] and his colleagues [13, 19] developed rule specifications to detect bugs and the rules require not only the sensitive functions but also their semantics. Privacy leak detection on Android apps [5, 20] must have a predefined set of source/sink functions. Some recent studies that try to identify security checks [30], detect double-fetch bugs [45], or detect other security bugs based on paired path inconsistencies [27, 47] have explicitly claimed that they manually collect sensitive functions of interest. The knowledge generated by our work can be employed by the above techniques. Researchers can extend their rules to the sensitive functions to effectively detect more bugs.

7 CONCLUSION

We are the first to present the problem of analogically reasoning about rare sensitive functions that occur very few times through a large software system. Despite their rarity, many bugs are related to such functions. In this paper, we propose Raisin, which presents a *1+context* scheme to make constrained pairs for effective analogical reasoning. Each pair consists of a function and its context, i.e., data correlated calls around it. We devise a weighted subword embedding to highlight the operational semantics for sensitive function identification, and expand the context recursively by inlining rare functions. Key subwords in frequent sensitive functions are also identified as a filter criterion to screen out the identification candidates. Experiments on five large open-source code bases have shown the efficiency and effectiveness of Raisin. Besides, 21 bugs resulting from rare sensitive functions have been confirmed by the developers, further demonstrating the importance of our work.

ACKNOWLEDGMENTS

The authors would like to thank the reviewers for their constructive comments. The work is supported in part by National Natural Science Foundation of China (NSFC) under grants 62272465, 62002361 and U1836209, and the Fundamental Research Funds for the Central Universities and the Research Funds of Renmin University of China under grant 22XNKJ29, and the CCF-Huawei Populus euphratica Innovation Research Funding under grant CCF-HuaweiSE2021002.

REFERENCES

[1] 2019. fuzzyc2cpg: A fuzzy parser for C/C++ that creates semantic code property graphs. https://github.com/ShiftLeftSecurity/fuzzyc2cpg.

[2] 2023. Clang Static Analyzer. https://clang-analyzer.llvm.org/.

[3] Mansour Ahmadi, Reza Mirzazade Farkhani, Ryan Williams, and Long Lu. 2021. Finding Bugs Using Your Own Code: Detecting Functionally-similar yet Inconsistent Code. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, Michael Bailey and Rachel Greenstadt (Eds.). USENIX Association, 2025–2040. https://www.usenix.org/conference/usenixsecurity21/presentation/ahmadi

[4] Glenn Ammons, Rastislav Bodík, and James R. Larus. 2002. Mining specifications. In *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002.* ACM, 4–16.

[5] 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, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 259–269. https://doi.org/10.1145/2594291.2594299

[6] 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 *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020.* ACM, 1101–1113.

[7] 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, Lake Buena Vista, FL, USA, November 04-09, 2018*, Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu (Eds.). ACM, 411–422. https://doi.org/10.1145/3236024.3236032

[8] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomás Mikolov. 2017. Enriching Word Vectors with Subword Information. *Trans. Assoc. Comput. Linguistics* 5 (2017), 135–146.

[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, London, UK, July 9-12, 2007.* ACM, 163–173.

[10] 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, Charleston, South Carolina, USA, November 18-19, 2002.* ACM, 51–60.

[11] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, Jill Burstein, Christy Doran, and Thamar Solorio (Eds.). Association for Computational Linguistics, 4171–4186. https://doi.org/10.18653/v1/n19-1423

[12] Navid Emamdoost, Qiushi Wu, Kangjie Lu, and Stephen McCamant. 2021. Detecting Kernel Memory Leaks in Specialized Modules with Ownership Reasoning. In *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021.* The Internet Society. https://www.ndss-symposium.org/ndss-paper/detecting-kernel-memory-leaks-in-specialized-modules-with-ownership-reasoning/

[13] Dawson R. Engler, David Yu Chen, and Andy Chou. 2001. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *Proceedings of the 18th ACM Symposium on Operating System Principles, SOSP 2001, Chateau Lake Louise, Banff, Alberta, Canada, October 21-24, 2001*, Keith Marzullo and Mahadev Satyanarayanan (Eds.). ACM, 57–72. https://doi.org/10.1145/502034.502041

[14] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. 1996. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD-96), Portland, Oregon, USA*, Evangelos Simoudis, Jiawei Han, and Usama M. Fayyad (Eds.). AAAI Press, 226–231. http://www.aaai.org/Library/KDD/1996/kdd96-037.php

[15] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020 (Findings of ACL, Vol. EMNLP 2020)*, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for Computational Linguistics, 1536–1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139

[16] Vinod Ganapathy, Dave King, Trent Jaeger, and Somesh Jha. 2007. Mining Security-Sensitive Operations in Legacy Code Using Concept Analysis. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007.* IEEE Computer Society, 458–467. https://doi.org/10.1109/ICSE.2007.54

[17] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022*, Smaranda Muresan, Preslav Nakov, and Aline Villavicencio (Eds.). Association for Computational Linguistics, 7212–7225. https://doi.org/10.18653/v1/2022.acl-long.499

[18] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021.* OpenReview.net. https://openreview.net/forum?id=jLoC4ez43PZ

[19] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson R. Engler. 2002. A System and Language for Building System-Specific, Static Analyses. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*, Jens Knoop and Laurie J. Hendren (Eds.). ACM, 69–82. https://doi.org/10.1145/512529.512539

[20] Jianjun Huang, Xiangyu Zhang, and Lin Tan. 2016. Detecting sensitive data disclosure via bi-directional text correlation analysis. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su (Eds.). ACM, 169–180. https://doi.org/10.1145/2950290.2950348

[21] Samantha Syeda Khairunnesa, Hoan Anh Nguyen, Tien N. Nguyen, and Hridesh Rajan. 2017. Exploiting implicit beliefs to resolve sparse usage problem in usage-based specification mining. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 83:1–83:29.

[22] 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, Seattle, WA, USA*, Brian N. Bershad and Jeffrey C. Mogul (Eds.). USENIX Association, 161–176. http://www.usenix.org/events/osdi06/tech/kremenek.html

[23] Ivo Krka, Yuriy Brun, and Nenad Medvidovic. 2014. Automatic mining of specifications from invocation traces and method invariants. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey (Eds.). ACM, 178–189. https://doi.org/10.1145/2635868.2635890

[24] Omer Levy and Yoav Goldberg. 2014. Linguistic Regularities in Sparse and Explicit Word Representations. In *Proceedings of the Eighteenth Conference on Computational Natural Language Learning, CoNLL 2014, Baltimore, Maryland, USA, June 26-27, 2014*, Roser Morante and Wen-tau Yih (Eds.). ACL, 171–180. https://doi.org/10.3115/v1/w14-1618

[25] Zhenmin Li and Yuanyuan Zhou. 2005. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, Michel Wermelinger and Harald C. Gall (Eds.). ACM, 306–315. https://doi.org/10.1145/1081706.1081755

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

[27] Dinghao Liu, Qiushi Wu, Shouling Ji, Kangjie Lu, Zhenguang Liu, Jianhai Chen, and Qinming He. 2021. Detecting Missed Security Operations Through Differential Checking of Object-based Similar Paths. In *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi (Eds.). ACM, 1627–1644. https://doi.org/10.1145/3460120.3485373

[28] V. Benjamin Livshits and Thomas Zimmermann. 2005. DynaMine: finding common error patterns by mining software revision histories. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, Michel Wermelinger and Harald C. Gall (Eds.). ACM, 296–305. https://doi.org/10.1145/1081706.1081754

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

[30] Kangjie Lu, Aditya Pakki, and Qiushi Wu. 2019. Automatically Identifying Security Checks for Detecting Kernel Semantic Bugs. In *Computer Security - ESORICS 2019 - 24th European Symposium on Research in Computer Security, Luxembourg, September 23-27, 2019, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 11736)*, Kazue Sako, Steve A. Schneider, and Peter Y. A. Ryan (Eds.). Springer, 3–25. https://doi.org/10.1007/978-3-030-29962-0_1

[31] Kangjie Lu, Aditya Pakki, and Qiushi Wu. 2019. Detecting Missing-Check Bugs via Semantic- and Context-Aware Criticalness and Constraints Inferences. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, Nadia Heninger and Patrick Traynor (Eds.). USENIX Association, 1769–1786. https://www.usenix.org/conference/usenixsecurity19/presentation/lu

[32] Tomás Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. In *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*.

[33] Tomás Mikolov, Wen-tau Yih, and Geoffrey Zweig. 2013. Linguistic Regularities in Continuous Space Word Representations. In *Human Language Technologies: Conference of the North American Chapter of the Association of Computational Linguistics, Proceedings, June 9-14, 2013, Westin Peachtree Plaza Hotel, Atlanta, Georgia, USA*, Lucy Vanderwende, Hal Daumé III, and Katrin Kirchhoff (Eds.). The Association for Computational Linguistics, 746–751. https://aclanthology.org/N13-1090/

[34] 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, Buenos Aires, Argentina, May 20-28, 2017*. IEEE / ACM, 438–449.

[35] 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, San Diego, California, USA, February 23-26, 2014*. The Internet Society.

[36] Suman Saha, Jean-Pierre Lozi, Gaël Thomas, Julia L. Lawall, and Gilles Muller. 2013. Hector: Detecting Resource-Release Omission Faults in error-handling code for systems software. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Budapest, Hungary, June 24-27, 2013*. IEEE Computer Society, 1–12. https://doi.org/10.1109/DSN.2013.6575307

[37] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural Machine Translation of Rare Words with Subword Units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*. The Association for Computer Linguistics. https://doi.org/10.18653/v1/p16-1162

[38] Lin Tan, Xiaolan Zhang, Xiao Ma, Weiwei Xiong, and Yuanyuan Zhou. 2008. AutoISES: Automatically Inferring Security Specification and Detecting Violations. In *Proceedings of the 17th USENIX Security Symposium, July 28-August 1, 2008, San Jose, CA, USA*. USENIX Association, 379–394.

[39] 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, Auckland, New Zealand, November 16-20, 2009*. IEEE Computer Society, 283–294. https://doi.org/10.1109/ASE.2009.72

[40] Song Wang, Devin Chollak, Dana Movshovitz-Attias, and Lin Tan. 2016. Bugram: bug detection with n-gram language models. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, David Lo, Sven Apel, and Sarfraz Khurshid (Eds.). ACM, 708–719. https://doi.org/10.1145/2970276.2970341

[41] Westley Weimer and George C. Necula. 2005. Mining Temporal Specifications for Error Detection. In *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3440)*, Nicolas Halbwachs and Lenore D. Zuck (Eds.). Springer, 461–476. https://doi.org/10.1007/978-3-540-31980-1_30

[42] Qian Wu, Guangtai Liang, Qianxiang Wang, Tao Xie, and Hong Mei. 2011. Iterative mining of resource-releasing specifications. In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011*, Perry Alexander, Corina S. Pasareanu, and John G. Hosking (Eds.). IEEE Computer Society, 233–242. https://doi.org/10.1109/ASE.2011.6100058

[43] Qiushi Wu, Aditya Pakki, Navid Emamdoost, Stephen McCamant, and Kangjie Lu. 2021. Understanding and Detecting Disordered Error Handling with Precise Function Pairing. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, Michael Bailey and Rachel Greenstadt (Eds.). USENIX Association, 2041–2058. https://www.usenix.org/conference/usenixsecurity21/presentation/wu-qiushi

[44] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. 2016. Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. *CoRR* abs/1609.08144 (2016). arXiv:1609.08144 http://arxiv.org/abs/1609.08144

[45] Meng Xu, Chenxiong Qian, Kangjie Lu, Michael Backes, and Taesoo Kim. 2018. Precise and Scalable Detection of Double-Fetch Bugs in OS Kernels. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. IEEE Computer Society, 661–678. https://doi.org/10.1109/SP.2018.00017

[46] Insu Yun, Changwoo Min, Xujie Si, Yeongjin Jang, Taesoo Kim, and Mayur Naik. 2016. APISan: Sanitizing API Usages through Semantic Cross-Checking. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, Thorsten Holz and Stefan Savage (Eds.). USENIX Association, 363–378. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/yun

[47] Qingyang Zhou, Qiushi Wu, Dinghao Liu, Shouling Ji, and Kangjie Lu. 2022. Non-Distinguishable Inconsistencies as a Deterministic Oracle for Detecting Security Bugs. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi (Eds.). ACM, 3253–3267. https://doi.org/10.1145/3548606.3560661