SparrowHawk: Memory Safety Flaw Detection via Data-driven Source Code Annotation

Yunlong Lyu[1], Wang Gao[2], Siqi Ma[3], Qibin Sun[1], and Juanru Li[2]

[1] University of Science and Technology of China
`lyl2019@mail.ustc.edu.cn`
[2] Shanghai Jiao Tong University
`gaowang@sjtu.edu.cn`, `jarod@sjtu.edu.cn`
[3] The University of Queensland
`slivia.ma@uq.edu.au`

Abstract. Detecting code flaws in programs is a vital aspect of software maintenance and security. Classic code flaw detection techniques rely on program analysis to check whether the code logic violates certain pre-define rules. In many cases, however, program analysis falls short of understanding the semantics of a function (e.g., the functionality of an API), and thus is difficult to judge whether the function and its related behaviors would lead to a security bug. In response, we propose an automated data-driven annotation strategy to enhance the understanding of the semantics of functions during flaw detection. Our designed SparrowHawk source code analysis system utilizes a programming language aware text similarity comparison to efficiently annotate the attributes of functions. With the annotation results, SparrowHawk makes use of the Clang static analyzer to guide security analyses.
To evaluate the performance of SparrowHawk, we tested SparrowHawk for memory corruption detection, which relies on the annotation of customized memory allocation/release functions. The experiment results show that by introducing function annotation to the original source code analysis, SparrowHawk achieves more effective and efficient flaw detection, and successfully discovers 51 new memory corruption vulnerabilities in popular open source projects such as FFmpeg and kernel of OpenHarmony IoT operating system.

Keywords: Objective function recognition, Programming language understanding, Neural network, Vulnerability discovery.

1 Introduction

Due to a variety of cyber attacks targeting on software flaws, pursuing secure programming becomes one of the most essential requirements for all programmers. However, a software is commonly comprised by thousands of lines of code, which is not easy for programmers to be aware of all flaws timely. In the real world, hackers attack softwares every 39 seconds, averagely 2,244 times per day[4].

[4] https://www.varonis.com/blog/cybersecurity-statistics/

The software with security breaches may be exploited by the hackers and finally data breaches will expose sensitive information and vulnerable users to hackers. Hence, it is crucial to identify and fix software flaws in time.

To reduce human efforts on analyzing project source code, automated approaches are propose to explore software flaws. Two types of techniques for source code analysis, program analysis [29] [40] [50] [52] and machine learning [37] [24] [25], are mainly introduced. For program analysis based approaches, they commonly analyze the entire source code and learn the control/data dependencies by conducting abstract interpretation, pattern matching, symbolic execution to identify. Although such a technique can ensure a significant code coverage, it is inefficient to construct control/data dependencies among functions when a large amount of code with complex dependencies are involved. To solve this issue, some researchers proposed machine learning algorithms to learn patterns of the vulnerable code and then rely on the trained models to discover software flaws. Different from the program analysis techniques that have to be executed every time of flaw detection, model training is a one-time effort; thus it only needs to be trained once and then used for the following detection.

However, the existed machine learning based approaches have a common drawback — a vast dataset of millions of open source functions that are labeled appropriately. Since the programming languages are unlike natural languages, it is impractical to understand how a function behaves by simply regarding each function as a bag of words. Generally two steps are proceeded: 1) extracting inter- and intra-dependencies at a fine-grained level. 2) taking the dependency graph as input for model training. Even though the model training is a one-time effort, it is time consuming to label millions of open source functions manually and study the inter- and intra-dependencies of every function.

To address the limitations of the previous machine learning based approaches, we observe that operations implemented in the function bodies can be inferred via the function prototypes. Hence, we propose an automated function annotating inspired approach for flaw detection. Since function prototypes consist of multiple informal terms (e.g., abbreviation, programming-specified words), we first construct a programming corpus with the posts from StackOverflow [42]. Within the programming corpus, it not only contains the informal terms used in programming languages, but also includes natural languages that are commonly used in project programming. In order to extract meaningful word units (subwords) from programming corpus, we further utilize Byte Pair Encode (BPE) [39] and BPE-dropout [34] algorithms to collect a subword collection with occurrence frequencies. According to the subword collection, function prototypes are segmented with meaningful subwords through a Probabilistic Language Model (PLM). Then we train a Siamese network [3] to embed function prototypes into vectors, and the annotations of unknown function prototypes will be obtained by comparing the vectors with a certain type of function prototypes.

Based on the function annotating inspired approach, we build a flaw detection tool, SparrowHawk. To validate the effectiveness of SparrowHawk, we conducted experiments targeting on memory-specified flaws, namely, null pointer dereference and double free. We labeled functions that are collected from real-world open source projects including OpenHarmony [33] IoT operating system and FFmpeg [12] and evaluated the performance of SparrowHawk. With the enhancement of function annotation, SparrowHawk successfully reported 51 previously unknown memory corruption flaws. We also evaluated whether the performance of SparrowHawk was influenced when various input data were provided. We found SparrowHawk still annotated functions effectively and efficiently even if only a small amount of training material (3,579 functions) were provided.

Contributions of this paper:

– We proposed an automated annotation-based analysis system that recognizes the targeted functions accurately without the need of analyzing the corresponding function implementations. While training the annotation model, only a few labeled dataset are required, which is helpful to reduce the involved human efforts.
– We implemented an efficient flaw detection tool, SparrowHawk, to explore certain types of flaws based on function annotation. Instead of analyzing the entire source code of a project, SparrowHawk pinpoints specific target functions by checking their function names and further identifies whether the target functions are properly invoked. This function annotation based flaw detection is data driven and flexible.
– We evaluated the performance of function annotation by providing various amount of input data and observed that SparrowHawk could still identify memory operation functions effectively. Moreover, SparrowHawk reported 51 previously unknown flaws from eight open source projects, which indicates that function annotation enhances classical flaw detection.

Availability. We provided the SparrowHawk executable, instructions of our experiments, and the tested projects at `https://sparrowhawk.code-analysis.org`.

2 Motivation

The existing program analysis based approaches are heavyweight while analyzing the program source code. We aim to design a system that can annotate each function accurately without checking the corresponding implementations of the function body. Lack of the semantic information of function prototypes, the following challenges are required to be addressed to implement an efficient and effective annotation based flaw detection.

2.1 Challenges

In order to annotate the targeted functions from source code, the following three aspects are generally processed:

- **Function Name.** By analyzing the semantic meaning of the function name, it is easily to determine what operations might be performed in its body.
- **Function Arguments.** According to the input arguments including argument types and argument names, the attributes of operation objects and operation types can be retrieved.
- **Function Body.** Reference to each function body including the implementation and annotations, the implemented functionalities can be determine.

Although the three aspects are precisely defined for function recognition, several challenges need to be resolved to determine whether a function is relevant to memory operation automatically. Details are demonstrated as below.
Challenge I: Natural Language Gap. Instead of using the completed and formal semantic words, function names are normally comprised by abbreviations, informal terms, programming-specific terms and project-specific terms. As these characters are barely appeared in the natural languages, it is difficult to determine the semantic meaning of a function name automatically.
Challenge II: Function Prototype Correlations. Since there exist strong correlations between each part of function prototype, it makes the entire semantics extraction form function prototype even more challenging. Even though some association patterns exist in function prototype, the workload for modeling the relationships for each type of function by human effort can be unacceptable. Therefore, for extracting the entire semantics from function prototype automatically, we need a method which can capture different relationships exist in different function prototypes.
Challenge III: Complex logical structures in function implementation. Sometimes, determining the functionality of a function only by its function prototype is not enough, and the complex logical structures in function implementation hinder automated tools to identify its main functionality.

2.2 Insights

Programming Language Aware Word Segmentation. The variety of naming styles and the usage of informal terms make it difficult to segment each function name into meaning units. To address Challenge I, we construct a programming corpus which contains not only the context in natural languages, but also programming-specified terms. Such a programming corpus provides a channel to connect the programming-specified terms with the natural language context.

Additionally, function names commonly consist of multiple terms. We further learn how function names are constructed by utilizing a pair encoding algorithm, which learns the frequent word units appeared in the programming corpus. Based on the frequent word units, we adopt PLM to conduct function segmentation.
Self-Attention Based Function Prototype embedding. For each word unit of the function prototype, it is inaccurate and inefficient to extract its semantic meaning by designing a rule to match the word unit with natural language context. To address this issue (Challenge II), we propose a self-attention based neural network encoder to generate function prototype embedding.

Semantic-Aware Call Graph. Generally, analyzing the function body and determining its main functionality is a hard work. But fortunately, by analyzing the function implementation manually, we observe that the functionality information about a function can be conveyed by its callee functions. As the semantics of function prototype can be extracted by the self-attention neural network and the nodes in call graph structure are function prototypes, thus we can give the call graph with some semantic information.

Therefore, to solve Challenge III, we propose a method to annotate targeted function in call graph, and utilize these annotations to understand the implementation of function.

3 SparrowHawk

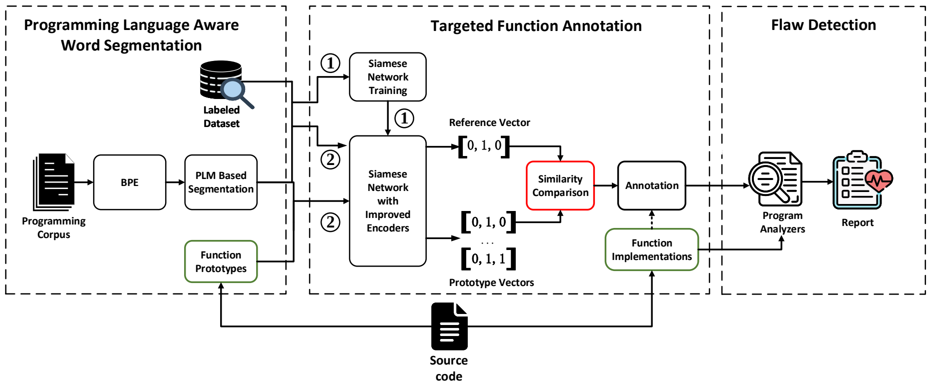


Fig. 1: Workflow of SparrowHawk

For most original flaw detection tools, they generally identify the specific flaws by analyzing the entire source code, which is inefficient. To resolve this issue, we propose an annotating inspired detection system, SparrowHawk, which automatically learns the functionality of each function through the function prototype and further identifies flaws by analyzing the source code of the target function.

3.1 Overview

The workflow of SparrowHawk is shown in Figure 1, which include three components, *Programming Language Aware Word Segmentation*, *Targeted Function Annotation*, *Flaw Detection*. We introduce each component in detail as below:
Programming Language Aware Word Segmentation. Functions are named variously and each function name might consist of multiple informal terms and programming-specific words, thus it is difficult to learn the functionality of a function via its name precisely (Challenge I). Instead of analyzing the function name as a whole, SparrowHawk takes as input a programming corpus to build

a segmentation model. It further splits each function name into several units of words (subwords).

Targeted Function Annotation. Taken the subwords as input, SparrowHawk trains a function annotation model and generates a *reference vector* from the target functions in labeled dataset. To annotate unknown function prototypes, SparrowHawk executes the annotation model to generate function prototype vectors. It then computes cosine similarities between the *reference vector* and the function prototype vectors. If the cosine similarities are higher than a threshold, or their function implementations are matched by annotation rules, SparrowHawk labeled them as targeted functions.

Flaw Detection. After recognizing the targeted functions, SparrowHawk conducts a source code based program analysis to examine whether the input source code files contain potential code flaws.

3.2 Programming Language Aware Word Segmentation

SparrowHawk first takes as input the function prototypes and splits them into subwords for the following semantic analysis. To segment function prototypes accurately, it is essential to build a corpus that includes the informal terms and programming languages used for naming functions. Therefore, we collect the posts of StackOverflow forum containing the context of programming languages from StackExchange Archive Site [41] which contains rich lexical information of programming languages. Figure 2 depicts the detailed process of word segmentation.

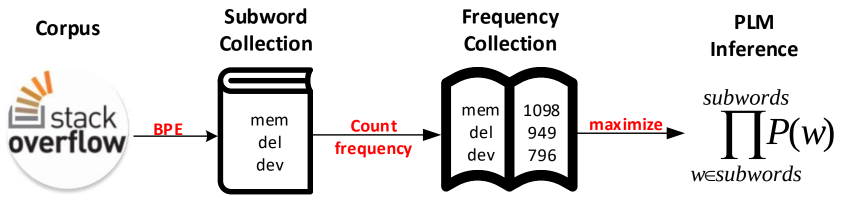
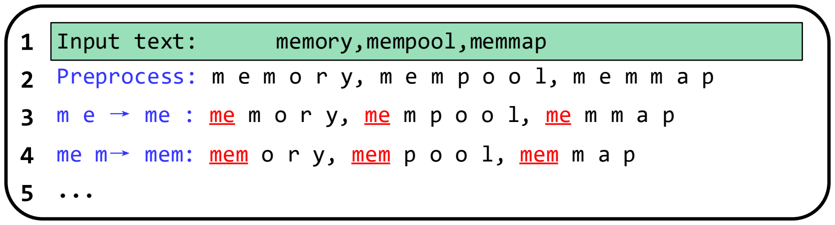


Fig. 2: Programming language aware word segmentation

Programming Corpus Construction The significant difference between programming language and natural language determines that SparrowHawk could not rely on the materials with natural languages to guide the following segmentation. However, directly using source code as corpus is neither suitable since it only contains limited semantic information. Therefore, we created a programming corpus for SparrowHawk using the posts of StackOverflow forum, which contains both meaningful natural language materials and programming language texts.

Subwords Collection With the created corpus, SparrowHawk collects meaningful units from it as subwords. Note that a subword may not be a vocabulary, thus SparrowHawk utilizes the BPE algorithm to collect subwords from the corpus by merging the most frequent items at character level. BPE initializes the input with a sequence of characters and iteratively replace each occurrence of the most frequent pair with a new item. Figure 3 gives an example of BPE merge operation. In this example, input text contains three words: *memory*, *mempool*, and *memmap*. The BPE processing first splits each word to separate characters, then merges the most frequent item *mem* and adds the item and its occurrence frequency to the subword collection.

For efficiency consideration, SparrowHawk only returns a subword that contains less than 15 characters. To provide a robust subword collecting, SparrowHawk additionally adopt BPE-dropout [34] algorithm to add stochastic noise during BPE merge operation.



```
Input text:      memory,mempool,memmap
Preprocess: m e m o r y, m e m p o o l, m e m m a p
m e → me : me m o r y, me m p o o l, me m m a p
me m→ mem: mem o r y, mem p o o l, mem m a p
...
```

Fig. 3: An example of BPE merging operations

PLM based Word Segmentation The collection of subwords (and their occurrence frequencies) is used by SparrowHawk to employ a PLM based word segmentation. SparrowHawk first splits a function prototype using the item appeared in the subword collection. If there exists only one segmentation result, then this result is accepted. Otherwise, SparrowHawk uses Equation (1) to determine which segmentation result should be chosen. For instance, for a segmentation result with subwords A, B, and C, the occurrence frequency probability of each subword is multiplied to obtain the probability of the segmentation result. Then SparrowHawk chooses the one with the highest probability.

$$P(segmentation) = \prod_{w \in subwords}^{subwords} P(w) \tag{1}$$

3.3 Targeted Function Annotation

SparrowHawk *identifies* certain types of a function (e.g., crypto function, encoding function) with an automated function annotation. To annotate a function , SparrowHawk compares its prototype to a labeled dataset, which contains manually labeled target functions and non-target functions. SparrowHawk first trains a Siamese network combined with two identical Transformer encoders [45], who share the same parameters. The training starts from randomly generating either a *target pair* (two prototypes of target functions) or a *non-target pair* (one prototype of target function and the other non-target function) from the labeled dataset. Pairs of prototypes are sent to two encoders to calculate the similarity between two functions.

After the training, SparrowHawk uses one of the improved encoder to generate embedding vectors for all target functions in labeled dataset, and derives a *reference vector* by computing the mean vector of these embedding vectors. This reference vector helps SparrowHawk efficiently identify a new target function: if the prototype vector of an analyzed function is similar to the reference vector (using cosine similarity, 0.5 as the similarity threshold), it belongs to the type of labeled dataset and is annotated as a target function by SparrowHawk.

In the following, we illustrate the annotation process in detail.

Siamese Architecture Given a set of function prototype pairs (f_i, f_i') with ground truth pairing information $y_i \in \{+1, -1\}$, where $y_i = +1$ indicates that f_i and f_i' are similar, or $y_i = -1$ otherwise. We define the embedding of function prototype f_i as $\vec{e}_i$, and the output of Siamese architecture for each pair as

$$cosine(f, f') = \frac{\vec{e}^{\top}\vec{e}\,'}{||\vec{e}|| \times ||\vec{e}\,'||} \tag{2}$$

Then the parameters of function prototype encoder will be trained by minimize the Mean Squared Error Loss Function [32].

Function Prototype Embedding A function prototype generally consists of four parts: return type, function name, argument types and argument names. And all these four parts changing the semantic of function prototype to different degrees. In order to encode a function prototype to a meaningful embedding vector, SparrowHawk adopts the Transformer encoder as the function prototype encoder.

Transformer is a powerful attention model whose attention mechanism can learn the association of words forwardly and backwardly in a sequence. However, the output of the Transformer Encoder is a context matrix, different to Recurrent Neural Network, and it does not provide a sentence embedding directly. To address this issue, we add a pooling layer after the output layer of Transformer Encoder, introduced by the CLS-token pooling strategy in Sentence-BERT [36], to take a function prototype as the input, and output a function prototype embedding.

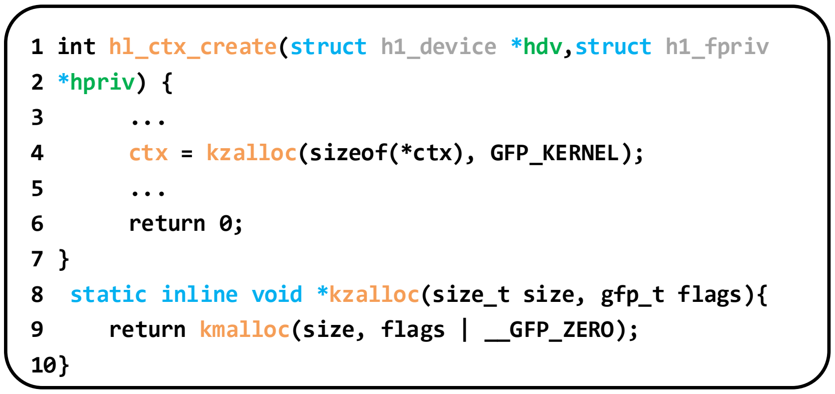
Similarity Inference After the Siamese network training is completed, we generate the embedding vectors $e_1, ..., e_n$ for all target function prototypes in the label dataset, and compute their arithmetic mean $\vec{e}_m$ as *reference vector*. For a given new function prototype f_t and its embedding vector $\vec{e}_t$, we obtain its similarity score by calculating cosine similarity with *reference vector*.

$$Score(f_t) = cos(\vec{e}_t, \vec{e}_m) = \frac{\vec{e}_t^{\top}\vec{e}_m}{||\vec{e}_t|| \times ||\vec{e}_m||} \in [-1, 1] \tag{3}$$

Targeted function Annotation Based on the similarity scores, SparrowHawk provides two ways to annotate targeted functions. The straightforward way only needs function prototype to make inference, whereas another way needs function body and customize the heuristic rules but provides more accurate annotation.

With Only Function Prototype. SparrowHawk simply makes inference by comparing the similarity scores of the given functions with the threshold *infer-threshold*. If the similarity scores are greater than the threshold, then SparrowHawk annotates the these functions as targeted functions.

With Function Implementation and Heuristic Rules. As our observation that the functionality information of a function can be conveyed through its callee function, and SparrowHawk has the ability to attribute a function only by its function prototype, thus SparrowHawk is able to achieve a more accurate annotation with well-designed heuristic rules on call graph.



```
int hl_ctx_create(struct hl_device *hdv,struct hl_fpriv
*hpriv) {
      ...
      ctx = kzalloc(sizeof(*ctx), GFP_KERNEL);
      ...
      return 0;
}
 static inline void *kzalloc(size_t size, gfp_t flags){
    return kmalloc(size, flags | __GFP_ZERO);
}
```

Fig. 4: An example of user-defined memory allocation function

Figure 4 is an intuitive example for the memory allocation function annotation task. As we observe the body of function `hl_ctx_create`, it is easy to find that its main functionality of memory allocation is implemented by its callee function `kzalloc`. As seeing the body of function `kzalloc`, `kzalloc` also calls a memory allocation function `kmalloc` to allocate memory. This common phenomenon exists in many projects, that because developers usually hope to wrapper lower level functions to achieve performance improvements and bring convenience by using custom memory allocators and de-allocators,

Therefore, we can take advantage of this property to design some heuristic rules and provide more accurate annotation about memory operation functions. More specifically, we set two different similarity thresholds, *recall-threshold* and *precision-threshold*. The functions whose similarity score lower than *recall-threshold* are filtered out, and the remaining functions are annotated as targeted functions only if they have a callee function whose similarity score is greater than *precision-threshold*.

3.4 Flaw Detection

SparrowHawk creates a flaw report for source code projects by comparing the usages of targeted functions with the predefined function misuse rules of program analyzers. First, SparrowHawk extracts function prototypes from source code files and generates call graphs. Then, according to the function prototypes and call graphs, SparrowHawk annotates the targeted functions and passes them to program analyzers to guide the function misuses detection. Here, SparrowHawk implements a program analyzer to detect null pointer dereference and double free vulnerabilities in source code.

Having the targeted functions, SparrowHawk performs a flow-sensitive and inter-procedural static analysis based on symbolic execution. SparrowHawk maintains two symbolic variables sets, allocation set and deallocation set, to record the status of memory chunks during the symbolic execution. Once symbolic execution reaches a memory operation function, the symbolic variables of allocated or deallocated memory chunks will be added to allocation set or deallocation set, respectively. When the same symbolic variable is added to deallocation set more than once, SparrowHawk will report a double free vulnerability. Or the dereference operation related symbolic variable exists in allocation set and its value of constraint solving equals to zero, then SparrowHawk will report a null pointer dereference vulnerability.

3.5 Implementation

We relied on several existing tools and modules to fulfil the certain functionalities in SparrowHawk. Clang [7] is embedded as part of the function prototype extractor to distinguish function prototypes during compiling. The used programming corpus is an 80 GB raw XML dataset and the size of meaningful text is 18 GB after our normalization. To retrieve subwords, SparrowHawk uses the *CharBPETokenizer* module in Tokenizers [44] which relies on the BPE and BPE-Dropout algorithms to segment and regularize words into sequences of subword units. Having the frequency vocabulary, SparrowHawk further executes WordSegment [48] to segment the function prototype. The Siamese network is trained relying on Gensim [14] with the implemented Word2vec [31] for subword embedding training. We further built the Siamese network in TensorFlow [2]. Once the interested function is retrieved, we adopted Clang Static Analyzer [1] to analyze the source code of each software and identify potential vulnerabilities.

4 Real-world Evaluation

We evaluated SparrowHawk from three perspective, function segmentation, function annotation, and flaw detection. In specific, the following three research questions (RQs) are answered:

RQ1: Function Prototype Segmentation. The first step of SparrowHawk is to segment function prototypes, thus we are curious about how accurate SparrowHawk is during function prototype segmentation.

RQ2: Function Annotation. As SparrowHawk relies on the customized memory operation functions, how effective and efficient is SparrowHawk in recognizing customized memory operation functions, i.e., memory allocation functions and memory deallocation functions.

RQ3: Flaw Detection. As SparrowHawk is introduced to detect the specific flaws (i.e., null pointer dereference and double free), how effective is SparrowHawk in detecting these flaws?

Since the goal for each research question is different, we collected different sets of dataset to conduct our experiment.

4.1 RQ 1: Function Prototype Segmentation

To evaluate the segmentation accuracy of SparrowHawk, we compared its segmentation result with a state-of-the-art tool, NLP-EYE [47], which is proposed with function prototype segmentation.

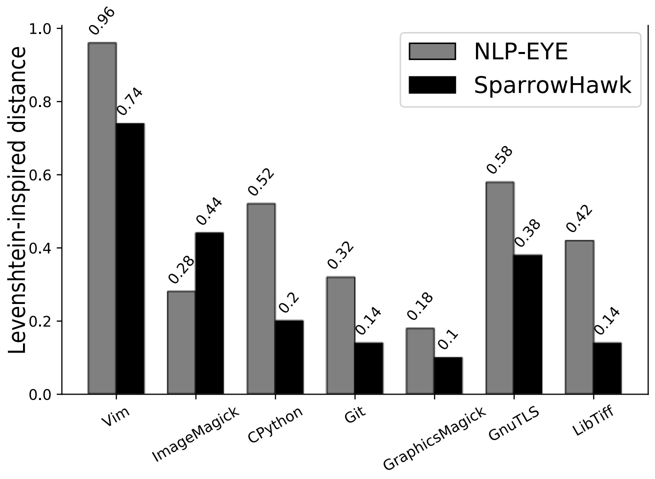


Fig. 5: Function name segmentation results comparison between NLP-EYE and SparrowHawk

Setup. We randomly collected 350 function names from seven programs, i.e., Vim [46], ImageMagick [21], GraphicsMagic [18], CPython [8], LibTIFF [28], GnuTLS [16], and Git [15], 50 function names from each program. Given the 350 function names, we built our ground truth by manually segmenting each function name. Then we evaluated the segmentation accuracy of SparrowHawk and NLP-EYE relying on Levenshtein-inspired distance [23] [38] in which a lower distance represents a higher accuracy.

Results. The segmentation results of SparrowHawk and NLP-EYE are demonstrated in Figure 5. We observed that SparrowHawk achieves a lower Levenstein distance, i.e., performs better than NLP-EYE. By manually inspecting the segmentation results, we realized that NLP-EYE fails to distinguish the function names with abbreviation, information terms and programming-specification words. Although a large corpus (i.e., GWTWC [17]) and an adaptive corpus with a number of program annotations is being used for segmentation, the semantic meanings of certain informal terms are unable to be learned precisely. For SparrowHawk, we firstly constructed the programming corpus and then applied BPE and BPE-Dropout to collect subwords, which enable SparrowHawk to segment function name precisely with the knowledge of programming language.

4.2 RQ 2: Function Annotation

SparrowHawk aims to identify the customized memory operation function, i.e., memory allocation functions and memory deallocation functions. We first evaluated the effectiveness of function annotation and then assessed the improvement with designed heuristic rules.

Setup. We collected 35,794 functions from the source code of ten Linux kernel drivers including `bluetooth`, `devfreq`, `mm`, `memory`, `media`, `memstick`, `message`, `mfd`, `misc`, and `mmc`. Obviously, it is time-consuming and infeasible to manually verify all functions to build ground truth, Hence, we conducted a semi-automatic annotating approach which takes the following three steps:

1. We first manually labeled 591 memory allocation functions and 778 memory deallocation functions in 5,342 functions (15% of the entire functions) as the initial labeled dataset, and utilized them to train the Siamese network of SparrowHawk.
2. Next, we randomly chose 19% (5,800/30,492) unlabeled functions and executed SparrowHawk to generate similarity scores for these functions. For functions with similarity scores smaller than -0.9 (around 90% in our experiment), SparrowHawk labeled them as non-target functions but need to inspect their function prototypes to select the possible target functions and exam their implementations manually. And the left functions were verified by both examining their prototypes and implementations manually.
3. With the labeled 5342+5,800 functions, we then repeated step 2 again. This time we sent the rest unlabeled 24,652 functions as inputs of SparrowHawk.

Finally, all 35,795 functions were labeled which include 2,008 memory allocation functions and 3,001 memory deallocation functions, and the other functions as non-target functions. Although our semi-automatic annotation may not strictly reflect the ground truth (85% of the functions were annotated relying on a computer-aided labeling), it significantly increases the scale of the labeled dataset by introducing a small portion of inaccuracy. Given the labeled dataset, all the labelled 35,794 functions were used to train the Siamese network again and the evaluation of function annotation of SparrowHawk was performed on the trained Siamese network.

Effectiveness As different developers might have various styles to name functions in their projects, we investigated whether the previous trained Siamese

Table 1: Comparison of memory operation function annotation with and without heuristic rules of SparrowHawk

	Allocation function annotation		**Deallocation function annotation**		**Memory operation function annotation**	
Function type	**Allocation**	**Others**	**Deallocation**	**Others**	**target**	**Others**
# of functions	117	2,883	135	2,865	252	5,748
# of correct annotation	73 / 86	2,853 / 2,875	123 / 127	2,786 / 2,815	196 / 213	5,639 / 5,690
# of error annotation	44 / 31	30 / 8	12 / 8	79 / 50	56 / 39	109 / 58
Precision	70.8% / 91.4%		60.8% / 71.7%		64.2% / 78.9%	
Recall	62.9% / 73.5%		91.1% / 94.0%		77.7% / 84.5%	
F1-score	**66.3% / 81.5%**		**72.9% / 81.4%**		**70.3% / 81.4%**	

∗ The **left** side of slash represent the results **with only function prototype**.
∗ The **right** side of slash represent the results **with function implementations and heuristic rules.**

network can annotate memory operation functions in a different project. Therefore, we set up a testing dataset by randomly selecting 3,000 functions from the OpenHarmony [33] IoT operating system and labeled them manually. As a result, 117 memory allocation functions and 135 memory deallocation functions were identified.

The experiment result is listed in Table 1. SparrowHawk successfully annotated 196 memory operation functions out of the 252 memory operation functions, with precision of 62.4%, recall of 77.7% and F1-score of 70.3%. Specifically, SparrowHawk separately achieved F1-score of 66.3% and 72.9% when it identified memory allocation functions and memory deallocation functions, respectively. The accuracy to annotate memory allocation functions is lower because the implementations of memory allocation functions are more complicated.

By analyzing the function prototypes collected from Linux kernel and OpenHarmony OS, we found that the performance drop is mainly caused by the inconsistent naming style. Consider the word "get" as an example, it indicates to fetch an object from a structure in Linux kernel, whereas developers of OpenHarmony use it to allocate a memory chunk sometimes. Alternatively, the word "release" in Linux kernel functions usually represents deallocating a memory space. However in OpenHarmony OS, it is usually used to release a lock, clean up an object, or set a flag bit to zero.

Improvement with Heuristic Rules Due to the inconsistent naming style among projects, SparrowHawk cannot annotate functions accurately based on function prototypes only. To resolve this issue, we improved SparrowHawk by embedding customized heuristic rules which analyzes function prototypes as well as each function bodies. In order to balance the candidate retrieving and precision improvement, we set the *recall-threshold* and *precision-thresholds* as -0.9 and 0.95, respectively.

The results in Table 1 show that the effectiveness of SparrowHawk is improved significantly with the help of the customized heuristic rules. In terms of the precision, recall, and F1-score of memory operation, SparrowHawk with the customized heuristic rules improves over SparrowHawk by 22.9%, 8.7% and 15.7%. Relying on the recall threshold, SparrowHawk can label more functions as the potential memory operation functions; thus it achieves a higher recall value. Besides, the precision threshold filtered out the function that did not invoke any memory operation functions in its function body.

Time cost We conducted our experiment on a server running 64-bit Ubuntu 18.04 with an AMD 3970X CPU (32 cores) running at 2.2GHz, 256 GB RAM and a GeForce GTX 2080Ti GPU card. We computed the efficiency of SparrowHawk by considering the worst case. Hence, we trained the Siamese network by using 90% of labeled functions. Finally, SparrowHawk averagely cost 5 hours 7 minutes to train the model for memory allocation function and 7 hours 21 minutes to train the model for memory deallocation functions. The time cost for model training is reasonable because it can be completed within one day and it is a one-time effort.

4.3 RQ 3: Flaw Detection

According to the annotated memory operation functions, SparrowHawk analyzes the corresponding functions to check whether there is any memory-related flaws, i.e., null pointer dereference and double free.

Table 2: Details about collected projects

	Number of Functions	**Number of Allocators**	**Number of De-allocator**
OpenHarmony	17,893	539	930
Cpython	11,347	436	228
FFmpeg	19,905	227	469
Gnutls	4,478	27	137
Vim	6,090	113	237
BusyBox	4,134	82	134
Curl	2,877	120	327
Gravity	916	60	62

∗All collected in the master branch in May 2021.

Setup. We executed SparrowHawk on eight open source projects, i.e., OpenHarmony [33], Cpython [8], FFmpeg [12], Gnutls [16], Vim [46], BusyBox [5], Curl [9], Gravity [19]. SparrowHawk first pinpointed the customized memory operation functions. Table 2 lists the result of the annotated memory operation functions

Table 3: Detection results

	Null Pointer Dereference			**Double Free**		
	Reported	Identified	Confirmed	Reported	Identified	Confirmed
OpenHarmony	41	16	**12**	128	0	0
Cpython	16	5	**5**	0	0	0
FFmpeg	37	9	**5**	43	0	0
Gnutls	4	2	**2**	10	3	**1**
Vim	41	0	0	46	4	**1**
BusyBox	114	1	**1**	15	1	**1**
Curl	0	0	0	6	1	0
Gravity	87	9	-	8	0	0
Total	340	42	**25**	256	9	**3**

in each project. Reference to the customized memory operation functions, SparrowHawk conducted code analysis to detect flaws.

Results. The detection result is presented in Table 3. In total, SparrowHawk reported 596 vulnerabilities from the eight projects including 340 null pointer dereference and 256 double free. By manually inspected the results, we identified 42 null pointer dereference vulnerabilities and 9 double free vulnerabilities.

To further verify the identification correctness, we contacted the project developers and reported the manual-confirmed vulnerabilities. Finally, 28 vulnerabilities (i.e., 25 null pointer dereference and 3 double free) are confirmed by SparrowHawk.

Case Study We demonstrated a representative example to discuss how SparrowHawk detects flaws. The source code snippet of Vim is shown in Figure 6 which contains a double free flaw.

Given the File1, SparrowHawk first extracted all function prototypes and corresponding function implementations and conducted function annotation to identify memory operation functions. As a result, it identified a memory allocation functions (i.e., `mem_realloc` (line 1)) and two memory deallocation functions (i.e., `mem_realloc` (line 1) and `vim_free` (line 8)).

Having the identified memory operation functions, SparrowHawk executed Clang Static Analyzer to analyze File2 and identified whether the identified memory operation functions were being properly invoked. Since there exists a feasible execution path from `vim_realloc` (line 20) to `vim_free` (line 24) and the two deallocation function freed the same memory chunk, where the macro function `vim_realloc` is expanded to function `mem_realloc` in File3, SparrowHawk reports a double free vulnerability.

As we observed that if the argument `bufno` is can be controlled, function `vim_realloc` will free the variable `buf_list` and return `NULL`, thus the same memory address will be freed twice by function `vim_free`. According to the feedback of Vim developers, the argument `bufno` can be controlled by a netbeans command, and the vulnerability is patched with a patch number 8.2.1843.

5 Related Work

We classify the related prior work into two categories: deep learning based flaw detection and program analysis based flaw detection.

5.1 Deep learning Based Flaw Detection

Recently, deep learning based approaches are being widely used to detect code flaw automatically. These approaches aim to learn syntactic and semantic representations [10], [37], [26], [27] or learn graph structure representations [11], [53] from source code, and then utilize these representations to detect code flaws.

To learn syntactic and semantic representations, Khanh Dam *et al.* [10] parses methods of Java source files into sequences of code tokens and uses Long Short-Term Memory networks [20] to generate syntactic and semantics features of a file. Russell *et al.* [37] creates a custom C/C++ lexer to tokenize source code and adopts Convolutional Neural Network (CNN) to learn function-level representations. In order to provide more fine-grained detection, VulDeePecker [27] and SySeVR [26] extract code slices based on data dependency and convert them to vector of symbolic representation, and then apply deep learn models to predict vulnerabilities.

As the aforementioned approaches have limitations on capturing logic and structure from source code, some works have attempted to learn representations from graph structures. Based on Code Property Graph (CPG) [49], VulSniper [11] utilize attention mechanism to encode CPG to a feature tensor and Devign [53] uses Graph Neural Network [22] to learn node representations.

However, these deep learning based approaches need the heavy efforts of gathering and labeling a large number flaw dataset, and can not give the precise reasons about how flaws are caused.

5.2 Program Analysis Based Flaw Detection

Program analysis based methods find flaws in source code by detecting unexpected behaviors. K-Miner [13] utilizes data-flow analysis to uncover memory corruption vulnerabilities in Linux kernel. It requires human effort to mark memory operation functions and performs a source-sink analysis on marked memory operation functions. Dr.checker [30] focus on control flow and has found diverse bugs in Linux kernel drivers by using a soundy pointer and taint analysis based on abstract representation. Moreover, SVF [43] is a static analysis framework which applies sparse value-flow analysis to detect flaws. Developers can use SVF to write their own checkers and detect flaws in source code.

To reduce the false positive of static analysis, symbolic execution based approaches utilize constraint solving to reason feasible paths. As the number of feasible paths in programs grows exponentially with an increase in program size, whole-program symbolic execution [6] could encounter the problem of path explosion. Thus, under-constrained methods like UCKLEE [35], sys [4] and UBITECT [51] are proposed to overcome this problem by executing individual functions instead of whole programs.

6 Conclusion

In this paper, we present SparrowHawk, an automated annotation-based source code flaw detection system. SparrowHawk includes a function prototype segmentation tool with the state-of-the-art accuracy, a targeted function annotation model requires only a few labeled dataset and an efficient source code flaw detection tool for detecting null pointer dereference and double free vulnerabilities. We demonstrated that SparrowHawk successfully identified 51 unknown flaws with the help of annotated memory operation functions. Furthermore, SparrowHawk is not limited to detect memory corruptions. Developers can easily customize SparrowHawk to annotate other types of function, and thus detect new types of flaws efficiently.

Acknowledgment

We would like to thank the anonymous reviewers for their helpful comments. This work was partially supported by the National Natural Science Foundation of China (U19B2023), the National Key Research and Development Program of China (Grant No.2020AAA0107800), and the National Natural Science Foundation of China (Grant No.62002222).

References

1. Clang Static Analyzer. http://clang-analyzer.llvm.org
2. Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, e.a.: Tensorflow: A system for large-scale machine learning. In: Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation. p. 265–283. USENIX Association (2016)
3. Bromley, J., Guyon, I., LeCun, Y., Säckinger, E., Shah, R.: Signature verification using a "siamese" time delay neural network. In: Proceedings of the 6th International Conference on Neural Information Processing Systems. p. 737–744. Morgan Kaufmann Publishers Inc.
4. brown, F., deian stefan, dawson engler: Sys: A static/symbolic tool for finding good bugs in good (browser) code. In: 29th USENIX Security Symposium (USENIX Security 20). pp. 199–216. USENIX Association (2020)
5. Busybox. `https://github.com/mirror/busybox`
6. Cadar, C., Dunbar, D., Engler, D.: Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation. p. 209–224. USENIX Association (2008)
7. Clang. `https://clang.llvm.org/`
8. Cpython. `https://github.com/python/cpython`
9. Curl. `https://github.com/curl/curl`
10. Dam, H.K., Tran, T., Pham, T., Ng, S.W., Grundy, J., Ghose, A.: Automatic feature learning for vulnerability prediction. arXiv preprint arXiv:1708.02368 (2017)

11. Duan, X., Wu, J., Ji, S., Rui, Z., Luo, T., Yang, M., Wu, Y.: Vulsniper: Focus your attention to shoot fine-grained vulnerabilities. In: Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19. pp. 4665–4671. International Joint Conferences on Artificial Intelligence Organization (2019)
12. Ffmpeg. https://github.com/FFmpeg/FFmpeg
13. Gens, D., Schmitt, S., Davi, L., Sadeghi, A.R.: K-miner: Uncovering memory corruption in linux. (2018)
14. Gensim. https://radimrehurek.com/gensim/
15. Git. https://github.com/git/git
16. Gnutls. https://gitlab.com/gnutls/gnutls/
17. Google web trillion word corpus. http://googleresearch.blogspot.com/2006/08/all-our-n-gram-are-belong-to-you.html
18. Graphicsmagick. http://www.graphicsmagick.org/
19. Gravity. https://github.com/marcobambini/gravity
20. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural Comput. p. 1735–1780 (1997)
21. Imagemagick. https://github.com/ImageMagick/ImageMagick
22. Li, Y., Tarlow, D., Brockschmidt, M., Zemel, R.: Gated graph sequence neural networks. arXiv preprint arXiv:1511.05493 (2017)
23. Li, Y., Liu, B.: A normalized levenshtein distance metric. IEEE transactions on pattern analysis and machine intelligence (TPAMI) (2007)
24. Li, Z., Zou, D., Tang, J., Zhang, Z., Sun, M., Jin, H.: A comparative study of deep learning-based vulnerability detection system. IEEE Access **7**, 103184–103197 (2019)
25. Li, Z., Zou, D., Xu, S., Jin, H., Qi, H., Hu, J.: Vulpecker: an automated vulnerability detection system based on code similarity analysis. In: Proceedings of the 32nd Annual Conference on Computer Security Applications. pp. 201–213 (2016)
26. Li, Z., Zou, D., Xu, S., Jin, H., Zhu, Y., Chen, Z.: Sysevr: A framework for using deep learning to detect software vulnerabilities. arXiv preprint arXiv:1807.06756 (2018)
27. Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z., Zhong, Y.: Vuldeepecker: A deep learning-based system for vulnerability detection (2018)
28. Libtiff. http://www.libtiff.org/
29. Ma, S., Thung, F., Lo, D., Sun, C., Deng, R.H.: Vurle: Automatic vulnerability detection and repair by learning from examples. In: European Symposium on Research in Computer Security. pp. 229–246. Springer (2017)
30. Machiry, A., Spensky, C., Corina, J., Stephens, N., Kruegel, C., Vigna, G.: DR. CHECKER: A soundy analysis for linux kernel drivers. In: 26th USENIX Security Symposium (USENIX Security 17). pp. 1007–1024. USENIX Association (2017)
31. Mikolov, T., Sutskever, I., Chen, K., Corrado, G.S., Dean, J.: Distributed representations of words and phrases and their compositionality. In: Advances in Neural Information Processing Systems 26, pp. 3111–3119 (2013)
32. Mean squared error. https://en.wikipedia.org/wiki/Mean_squared_error
33. Openharmony. https://openharmony.gitee.com/openharmony
34. Provilkov, I., Emelianenko, D., Voita, E.: BPE-dropout: Simple and effective subword regularization. In: Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics. pp. 1882–1892. Association for Computational Linguistics (2020)

35. Ramos, D.A., Engler, D.: Under-constrained symbolic execution: Correctness checking for real code. In: 24th USENIX Security Symposium (USENIX Security 15). pp. 49–64. USENIX Association (2015)
36. Reimers, N., Gurevych, I.: Sentence-bert: Sentence embeddings using siamese bert-networks. In: Proceedings of the 20‘ 19 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP). pp. 3982–3992. Association for Computational Linguistics (2019)
37. Russell, R., Kim, L., Hamilton, L., Lazovich, T., Harer, J., Ozdemir, O., Ellingwood, P., McConley, M.: Automated vulnerability detection in source code using deep representation learning. In: 2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA). pp. 757–762. IEEE (2018)
38. Schwartz, E.J., Cohen, C.F., Duggan, M., Gennari, J., Havrilla, J.S., Hines, C.: Using logic programming to recover C++ classes and methods from compiled executables. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS) (2018)
39. Sennrich, R., Haddow, B., Birch, A.: Neural machine translation of rare words with subword units. In: Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). pp. 1715–1725. Association for Computational Linguistics (2016)
40. Shen, Z., Chen, S.: A survey of automatic software vulnerability detection, program repair, and defect prediction techniques. Security and Communication Networks **2020** (2020)
41. Stackexchange archive site. `https://archive.org/download/stackexchange/stackoverflow.com-Posts.7z`
42. Stackoverflow forum. `https://stackoverflow.com/`
43. Sui, Y., Xue, J.: Svf: Interprocedural static value-flow analysis in llvm. In: Proceedings of the 25th International Conference on Compiler Construction. p. 265–266. Association for Computing Machinery (2016)
44. Tokenizers. `https://github.com/huggingface/tokenizers`
45. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L.u., Polosukhin, I.: Attention is all you need. In: Advances in Neural Information Processing Systems 30, pp. 5998–6008 (2017)
46. Vim. `https://github.com/vim/vim`
47. Wang, J., Ma, S., Zhang, Y., Li, J., Ma, Z., Mai, L., Chen, T., Gu, D.: Nlp-eye: Detecting memory corruptions via semantic-aware memory operation function identification. In: 22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019). pp. 309–321. USENIX Association (2019)
48. Wordsegment. `https://github.com/grantjenks/python-wordsegment`
49. Yamaguchi, F., Golde, N., Arp, D., Rieck, K.: Modeling and discovering vulnerabilities with code property graphs. In: 2014 IEEE Symposium on Security and Privacy. pp. 590–604. IEEE (2014)
50. Yan, H., Sui, Y., Chen, S., Xue, J.: Spatio-temporal context reduction: A pointer-analysis-based static approach for detecting use-after-free vulnerabilities. In: 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE). pp. 327–337. IEEE (2018)
51. Zhai, Y., yzhai: Ubitect: A precise and scalable method to detect use-before-initialization bugs in linux kernel. In: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020). ACM (2020)

52. Zhang, Y., Ma, S., Li, J., Li, K., Nepal, S., Gu, D.: Smartshield: Automatic smart contract protection made easy. In: 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER). pp. 23–34. IEEE (2020)
53. Zhou, Y., Liu, S., Siow, J., Du, X., Liu, Y.: Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In: Advances in Neural Information Processing Systems 32, pp. 10197–10207 (2019)

File1:Vim/src/misc2.c

```
void *mem_realloc(void *ptr, size_t size){
    void *p;
    mem_pre_free(&ptr);
    p = realloc(ptr, size);
    mem_post_alloc(&p, size);
    return p;
}
void vim_free(void *x){
    ...
    free(x);
    ...
}
```

File2:Vim/src/netbeans.c

```
static nbbuf_T *nb_get_buf(int bufno){
    buf_list_size = 100;
    ...
    if (bufno >= buf_list_size){
        nbbuf_T *t_buf_list = buf_list;
        incr = bufno - buf_list_size + 90;
        buf_list_size += incr;
        buf_list = vim_realloc(buf_list,
           buf_list_size * sizeof(nbbuf_T));
        if (buf_list == NULL)
          {
           vim_free(t_buf_list);
           buf_list_size = 0;
           return NULL;
          }
     }
}
```

File3:Vim/src/vim.h

```
# define vim_realloc(ptr, size)
mem_realloc((ptr), (size))
```

Fig. 6: A double free vulnerability in Vim