NLP-EYE: Detecting Memory Corruptions via Semantic-Aware Memory Operation Function Identification

Jianqiang Wang
wjq.sec@gmail.com
Shanghai Jiao Tong University

Siqi Ma (✉)
siqi.ma@csiro.au
CSIRO DATA61

Yuanyuan Zhang (✉)
yyjess@sjtu.edu.cn
Shanghai Jiao Tong University

Juanru Li (✉)
jarod@sjtu.edu.cn
Shanghai Jiao Tong University

Zheyu Ma
zheyuma@mail.nwpu.edu.cn
Northwestern Polytechnical University

Long Mai
root_mx@sjtu.edu.cn
Shanghai Jiao Tong University

Tiancheng Chen
sec_mxgc@sjtu.edu.cn
Shanghai Jiao Tong University

Dawu Gu
dwgu@sjtu.edu.cn
Shanghai Jiao Tong University

Abstract

Memory corruption vulnerabilities are serious threats to software security, which is often triggered by improper use of memory operation functions. The detection of memory corruptions relies on identifying memory operation functions and examining how it manipulates the memory. Distinguishing memory operation functions is challenging because they usually come in various forms in real-world software. In this paper, we propose NLP-EYE, an NLP-based memory corruption detection system. NLP-EYE is able to identify memory operation functions through a semantic-aware source code analysis automatically. It first creates a programming language friendly corpus in order to parse function prototypes. Based on the similarity comparison by utilizing both semantic and syntax information, NLP-EYE identifies and labels both standard and customized memory operation functions. It uses symbolic execution at last to check whether a memory operation causes incorrect memory usage.

Instead of analyzing data dependencies of the entire source code, NLP-EYE only focuses on memory operation parts. We evaluated the performance of NLP-EYE by using seven real-world libraries and programs, including Vim, Git, CPython, etc. NLP-EYE successfully identifies 27 null pointer dereference, two double-free and three use-after-free that are not discovered before in the latest versions of analysis targets.

1 Introduction

The memory-unsafe programming languages, such as C and C++, provide memory operation functions in the standard library (e.g., `malloc` and `free`) to allow manipulating the memories. During the development process, developers could implement dynamic memory operation functions by their own memory management policies to achieve higher performance, or by wrapping the standard memory operation functions with additional operations to fulfill other purposes (e.g., print debugging information).

Mistakes made by misusing the memory operations lead to well-seen memory corruption vulnerabilities such as buffer-overflow and double-free in real-world software and their number is steadily increasing. For customized memory operation functions, some private memory operation functions are poorly implemented and thus carry some memory vulnerabilities at birth. On the other hand, developers keep making common mistakes, such as using the memory after it has been released (i.e., the use-after-free vulnerability), during the development process. Both cases aggravate the emerging of memory corruption vulnerabilities, which endow the attackers higher chance of compromising a computer system. A recent report of Microsoft demonstrated that around 70 percent of vulnerabilities in their products are memory safety issues [14].

To identify memory corruptions, various analysis methods using different kinds of techniques have been proposed. For instance, code similarity detection and information flow analysis are proposed to identify memory safety issues in source code [29] [44] [41]. Some tools such as AddressSanitizer [40], Dr. Memory [22] can also detect memory corruptions in binary code by instrumentation. These analyses require to abstract the usage of memory, and then extract certain patterns that are related to memory corruption. Otherwise, analyzing a program with millions of lines of code is inefficient and error-prone.

Customized memory operations could not help to decrease the chance of memory corruption at all, and moreover, the customized functions cause great difficulty in memory corruption analysis. Previous works, such as CRED [44], Pinpoint [41] and Dr. Memory [22], only consider the memory operation functions defined in the standard library. They are unable to identify customized memory operation functions, and thus disregard vulnerabilities caused by customized functions. Manual efforts can be involved to identify and label those functions, but it is exhausted and time consuming.

To address the above problems, we propose NLP-EYE, a source code-based security analysis system that adopts natural language processing (NLP) to detect memory corruptions. NLP-EYE will only parse the function prototypes instead of analyzing implementation of the functions. It then applies

symbolic execution to check whether the corresponding memory usages are correct. Unlike the other tools [1], the accuracy of NLP-EYE in memory operation function identification helps reduce the time cost by only analyzing partial code snippets and facilitate a better detection performance.

NLP-EYE reports typical memory corruption vulnerabilities, i.e., null pointer de-reference, double-free and user-after-free in seven open source software, such as Vim and Git.NLP-EYE has found 49 unknown vulnerabilities from their latest versions. For source code with more than 60 thousand of function prototypes, NLP-EYE is able to parse every ten thousand functions in one minute and finish the memory operation checking within an hour.

Contributions. Major contributions of this paper include:

- We proposed a source code-based analysis system that detects vulnerabilities by only analyzing a few function implementations, i.e., function prototypes and comments. Since these information are usually available, it is helpful for analysts and developers to build secure software with limited details.
- We implemented a vulnerability detection tool, NLP-EYE, that discovers memory corruption vulnerabilities effectively and efficiently. By combining NLP and symbolic execution, NLP-EYE labels both standard and customized memory operation functions and records states of the corresponding memory regions.
- We analyzed the latest versions of seven libraries and programs with NLP-EYE, and identified 49 unknown memory corruption vulnerabilities with 32 of them caused by customized memory operation functions. It demonstrates that the semantic-aware identification of NLP-EYE helps find new vulnerabilities that are unseen before.

Structure. The rest of the paper is organized as below: Section 2 lists the challenges of identifying memory corruptions caused by customized memory operation functions, and provide corresponding insights to solve these challenges. Section 3 details the design of NLP-EYE. In Section 4, we reported new vulnerabilities found by NLP-EYE, and illustrated the experiment results covering both vulnerability detection accuracy and performance comparison with the other tools. Section 5 discusses related works. We conclude this paper in Section 6.

2 Background

We give a concrete example of memory corruption vulnerability in Figure 1. Followed by that, we point out some challenges that hinders the detection of such vulnerabilities, and give corresponding insights to address those challenges.

2.1 Running Example

Detecting a memory corruption vulnerability (e.g., use-after-free) requires three significant steps: 1) identifying memory

```
1  //functions are provided by TTL module to operate dynamic memory
2  void  TTLreleaseMem2Pool(Pool *pool, MemRegion p)
3  {
4      return pool->destroy_func(p);
5  }
6  MemRegion  TTLretrieveMemFromPool(Pool *pool, size_t len)
7  {
8      return pool->alloc_func(len);
9  }
10
11 //memory pool used to provide dynamic memory region manipulation
12 extern Pool globalPool;
13
14 int main(int argc, char **argv, char **env)
15 {
16     char content[100];
17     scanf("%s",content);
18     char* buf = (char*)TTLretrieveMemFromPool(&globalPpool,1000);
19     int ret = processContent(content,buf);
20     if(!ret)
21     {
22         err("error occurs during process content!");
23         TTLreleaseMem2Pool(&globalPool,(MemRegion)buf);
24         goto clean;
25     }
26     ...
27     clean:
28         TTLreleaseMem2Pool(&globalPool,(MemRegion)buf);
29 }
```

Figure 1: Double-free vulnerability caused by the customized memory operation functions

operation functions and labeling dynamically allocated memory regions; 2) tracing the allocated memory regions to understand how they are operated; and 3) detecting incorrect operations on allocated memory regions. However, existing vulnerability detection techniques barely consider *customized memory operation functions*, and thus fail to detect vulnerabilities triggered by them.

The customized memory operation functions has caused the memory corruption vulnerability in Figure 1. Instead of using the standard memory operation functions provided by C standard library, functions `TTLretrieveMemFromPool` and `TTLreleaseMem2Pool` are used to allocate a dynamic memory (Line 18) and release the corresponding allocated memory (Line 23), respectively. While executing, `TTLreleaseMem2Pool` releases the memory if the function `processContent` returns a null value (Line 20); then, a duplicate release (Line 28) causes a double-free vulnerability. Consider this double-free vulnerability, it cannot be detected by simply analyzing standard memory operation procedures because of the customized memory operation functions (i.e., `TTLreleaseMem2Pool` and `TTLretrieveMemFromPool`). [1]

Generally, whether a function is a memory operation function, we can observe whether it calls C standard library memory operating functions, or compare the similarity with other memory operation function implementations. In either case, it requires the function implementation which is usually not

[1] Actually we have applied typical tools such as Cppcheck [2] and VisualCodeGrepper [18] to detect the vulnerability in this sample and found that none of them could detect this vulnerability.

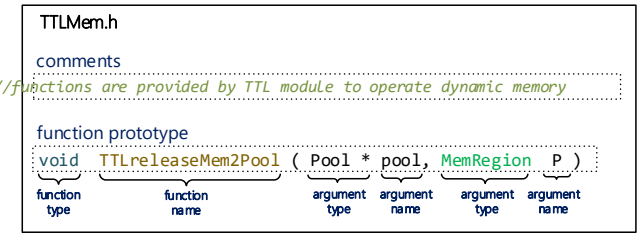

Figure 2: A function prototype with comments

available. For example, the declared memory operation function, `alloc_func()` (Line 8) might be implemented externally and only its binary is available. Under such circumstance, the semantic information in a function prototype (i.e., function declaration) becomes the only reference for the memory operation function identification.

As Figure 2 depicts, a function prototype consists of a function type, a function name, argument types for arguments, and (optionally) names of arguments. While defining a function prototype, developers prefer to use meaningful function name and proper data types for this function. Besides, developers may add comments to describe in more details.

In most cases, function prototypes and comments help us to determine the semantics without knowing function implementations. Therefore, we can analyze prototype structures to retrieve meanings of those words.

2.2 Challenges

Most challenges lie in understanding the function semantics and identify memory operation functions accurately.

Challenge I: Irregular Representations. Searching for specific words in the source code is the common strategy to identify functions, such as locating the keyword *memory* to identify memory operation functions. While plenty of abbreviations and informal terms are used in function prototypes, it is difficult to extract the semantic information effectively by only applying a keyword-based searching strategy.

Consider the function prototype `TTLreleaseMem2Pool` in Figure 1. An abbreviation `Mem2` represents *memory to* in an informal way. The abbreviation `Mem` is unable to be located by using the word *memory*, and the number `2` makes it harder to understand the semantics of the phrase.

Challenge II: Ambiguous Word Explanations. Since the context collected from function prototypes is insufficient, it makes the semantics extraction more challenging. Although some function prototypes may use the same word, the actual function semantics can be different because of their various naming formats.

Considering two function names, `PyObject_Malloc` and `_PyObject_DebugMallocStats`, in the source code of CPython[2], the former function is for allocating a dynamic memory while the latter one is for outputting debugging information of memory allocator.

[2]CPython is the reference implementation of Python.

Even though the lexical analysis with a specific dictionary can help split the word *malloc* from those two function names, the corresponding function semantics cannot be inferred precisely. For the function `PyObject_Malloc`, the word *malloc* does represent memory allocation; however in function `PyObject_DebugMallocStats`, *malloc* is used to qualify the object, that is *Stats*, to illustrate the status of the memory allocator. Therefore, we need not only analyze the meaning but also the format of the word to construct the function prototype.

Challenge III: Diverse Type Declaration. Diversified data types declaration in C/C++ programming makes it harder to compare two function prototypes. For instance, both `short` and `unsigned short int`, are used to represent the Integer type. Besides, C/C++ has provided a type re-define feature (i.e., `typedef`) that programmers can shorten the name of a complex type.

2.3 Insights

Fortunately, function prototypes are constructed by following some certain formats in programming. We utilize these formats to extract the semantic information.

Adaptive Lexical Analysis. Irregular representations make the function prototype segmentation even harder. A natural language corpus is not suitable for word segmentation in computer programming. Thus, we construct an adaptive corpus to address the problem of the lexical analysis in Challenge I. The corpus consists of natural language used in computer science, common keywords in the programming language (e.g., `proc`, `ttl`) and comments in the source code. Common keywords reveal the words that are often used in programming, and comments in the source code suggest some semantic information of a function.

Grammar-free Comparison. By examining the function prototypes manually, we observe that developers do not usually follow English grammars when naming a function. However, they still use similar words (e.g., *get*, *acquire*, *alloc*) with similar grammatical order (i.e., the order of words), such as `AcquireVirtualMemory` and `getMemfromPool`. We then propose a grammar-free analysis, which performs an NLP-based comparison, to solve Challenge II.

To identify the semantic information of each function prototype, we create a set of reference functions (e.g., standard memory operation functions), whose semantics are known. Then, we compare the function name and argument names of each function prototype with the corresponding names in reference functions. If the similarity between a function prototype and a reference function is higher than a threshold, we label this function prototype as a potential memory operation function, and proceed with the type comparison to confirm.

Various Types Clustering. NLP-based comparison only helps decide whether a function prototype is a potential memory operation function. We design a type comparison scheme to handle its declared return type and argument types. Because of the diversity of function types, we first normalize

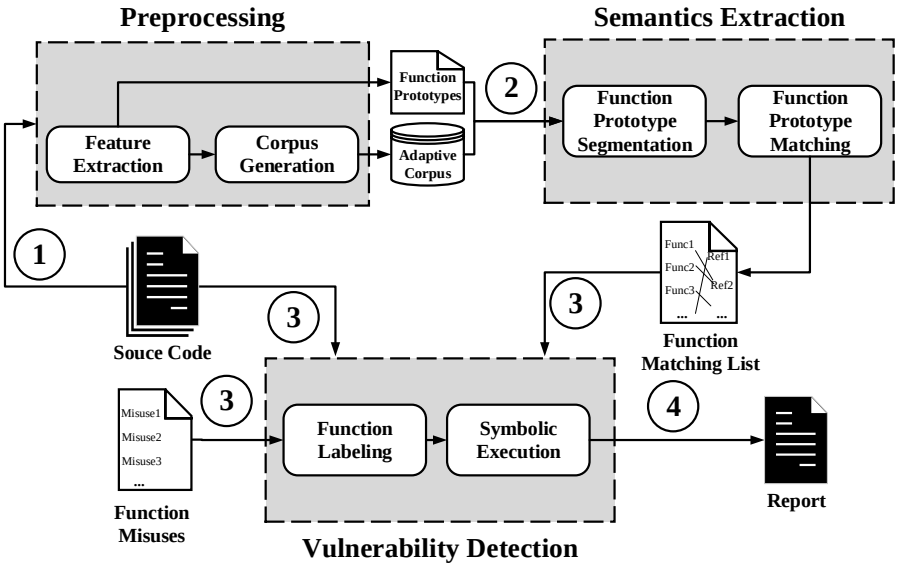

Figure 3: System overview of NLP-EYE

those types in aliases (e.g., types defined by `typedef`) by using their original forms, which solve Challenge III. Having the pair of a function prototype and its matched reference function, we then compare their return types and argument types. We assume a function prototype as a memory operation function if both names and types are matched.

3 Design of NLP-EYE

We propose NLP-EYE, a source code analysis system that utilizes NLP to retrofit the process of the memory corruption vulnerability detection. There are three phases: **preprocessing**, **semantics extraction**, and **vulnerability detection** in NLP-EYE. Figure 3 illustrates the overview of NLP-EYE. It takes source code files as inputs, i.e., the analysis target. The preprocessing phase extracts function prototypes and comments to generate an adaptive corpus. The semantics extraction phase uses the adaptive corpus to build a matching list by collecting all the possible memory operation functions in the analysis target. Vulnerability detection phase labels memory operation functions in the target and feeds it to the symbolic execution to facilitate the vulnerability detection. We introduce the working details of each phase below.

3.1 Preprocessing

NLP-EYE takes a batch of source code as inputs and generates *function prototype*s and an *adaptive corpus* to perform adaptive lexical analysis. First, NLP-EYE extracts function prototypes and comments from source code. Then, it comprises comments with the other two corpuses to construct an adaptive corpus. Details are presented below.

3.1.1 Feature Extraction

Feature extraction component of NLP-EYE is built on top of Clang Static Analyzer plugin [1], which provides an interface for users to scan the declaration of each function. Given the source code, NLP-EYE uses this plugin to extract all function prototypes in the format of "Type@Name", including those functions that are imported from other libraries. For comments from source code, NLP-EYE uses regular expressions to match comment symbols in C language.

3.1.2 Corpus Generation

After collecting those comments, NLP-EYE constructs an adaptive corpus to perform adaptive lexical analysis. The adaptive corpus includes three parts, that are Google Web Trillion Word Corpus (GWTWC) [7], MSDN library API names [21] [20], and comments from source code.

The GWTWC is a popular corpus created by Google, containing more than one trillion words extracted from public web pages. It can be applied to identify common words used in natural languages. With the help of MSDN library API names and comments, NLP-EYE can process programming languages. The MSDN library provides normalized APIs in Camel-Case format. Therefore, it is easy to divide each function name into words/abbreviations through capital letters. For example, function `GetProcAddress` can be divided into [`"Get"`,`"Proc"`,`"Address"`]. While processing comments from source code, NLP-EYE first filters the symbol characters (e.g., `#%!`), and then splits text by applying regular expressions. Numbers and words appeared in GWTWC are excluded.

Since abbreviations are commonly used in programming, we set the appearance frequency of MSDN APIs to be higher than the appearance frequency of comments, to provide them a higher priority. We further assume that a word, who is a

substring of another word in MSDN API names, should have a lower frequency than its parent word. For example, `arm` in function `mallocWithAlarm` is a substring of `Alarm`, obviously `Alarm` is to be regarded as a whole; then we assign a lower frequency for `arm` than for `Alarm`.

3.2 Semantics Extraction

NLP-EYE compares each function prototype with a set of *reference function*s (e.g., `malloc`, `free`), and generates a *function matching list*. When a match was found in the function matching list, we can infer the semantics from the function prototype that it has the similar semantics with the reference function.

NLP-EYE processes the data type and the function name, arguments name separately to identify memory operation functions in two steps. First, it divides the function name and arguments name into a serial of words. Next, it performs NLP-based comparison to select the potential function prototypes with memory operation functionalities and confirm the results by applying type comparison.

3.2.1 Function Prototype Segmentation

To proceed function prototype segmentation, NLP-EYE applies Segaran *et al.*'s word segmentation algorithm [39] to select the *segmentation list* with the highest *list frequency*.

Given a function prototype (FP), with n letters, NLP-EYE first creates 2^{n-1} possible combinations of these letters and constructs 2^{n-1} segmentation lists. Each segmentation list reserves the original order of these letters appeared in the FP. NLP-EYE then computes the list frequency for each segmentation list. It compares each word (w_i) in a segmentation list (SL) with words in the adaptive corpus, and returns the following list frequency (LF):

$$LF = \prod_{i=1}^{|SL|} freq(w_i)$$

where $|SL|$ represents how many words are contained in SL, and $freq(w_i)$ is the frequency of w_i in the adaptive corpus. Finally, NLP-EYE considers the segmentation list with the highest list frequency as its segmentation result.

3.2.2 Function Prototype Matching

Due to the diversity of type declaration, NLP-EYE processes names (i.e., function names and argument names) and types (i.e., return types and argument types) separately. It performs NLP-based comparison to identify those names that are related to memory operation functionalities. NLP-EYE then applies type comparison to determine memory operation functions and generates a function matching list.

NLP-based Comparison. Natural language processing (NLP) has been widely used to identify the connection between two words for semantic similarity matching. To measure the word similarity, a *context corpus* is required to extract the taxonomy information. Words in the context corpus are then represented by sets of vectors in Word2vec [34] model. The cosine distance between two words positively related to their semantic similarity, and a higher cosine distance represents a higher similarity between two words.

To extract the semantic meaning of an unknown name, we generates a set of reference functions manually, which contains standard memory operation functions provided by C/C++ and other known memory operation functions. Having those reference functions, NLP-EYE compares the name of an unknown function with the names of the reference functions and calculate their similarity scores. If a similarity score is higher than a *threshold*, NLP-EYE labels this unknown function as *similar* to the reference function, that is, the corresponding function is a potential memory operation function.

We address function names and argument names individually, since the comparison results of function names and argument names may interfere each other while applying the NLP-based comparison. Consider a function with only abbreviations for function names, but complete words for argument names, its similarity score may not achieve the threshold. Although the similarity score of the argument names is the highest, the total similarity score will be impacted by the low similarity of function names. Therefore, we set different similarity threshold, *fn-similarity* and *arg-similarity*, as the threshold of function names and argument names, respectively. Only when fn-similarity and arg-similarity are both satisfied, NLP-EYE will label the function. For function arguments, NLP-EYE compares each argument of the reference function with every argument of the target function and generates similarity score. Then NLP-EYE chooses the most similar one as the corresponding arguments regardless of the number of arguments.

Type Comparison. Given the potential memory operation functions and their matched reference functions, NLP-EYE compares their data types correspondingly. We use Clang Static Analyzer to classify the data types into several categories to address the type diversity.

First, NLP-EYE normalizes data types. Some data types are re-defined as aliases by `typedef`. Thus, NLP-EYE uses the original data types to replace those aliases. Second, we define some coarse grained categories based on the basic data types in C programming. NLP-EYE finally suggests the correct category for each data type. For example, `unsigned int` and `signed short` are assigned to the category of `Integer`. `void *` and `char *` belong to the category of `pointer`.

We compare the return type and corresponding argument types of the potential memory operation function with data types of the matched reference function. If their types are assigned to the same category, the unknown function is a memory operation function, and it is assumed to have the same semantics as the corresponding reference function. Each pair of a function prototype and its matched reference function is inserted to the function matching list.

3.3 Vulnerability Detection

NLP-EYE creates a vulnerability report for each source code by comparing the usages of memory operation functions with the pre-defined *function misuses*. NLP-EYE first labels memory operation functions in the source code; then, NLP-EYE checks whether there exists any function misuse.

3.3.1 Function Labeling

NLP-EYE takes the function matching list as an inputs to identify memory operation functions. It compares functions in the source code with the functions in the function matching list. If a function appears in the function matching list, NLP-EYE labels this function as a memory operation function.

3.3.2 Symbolic Execution

The code, that can be compiled independently, is regarded as an unit. NLP-EYE first generates the call graph for each unit and then executes each unit from top to bottom one by one by adopting symbolic execution.

The output of semantics extraction is a function matching list which maps the standard memory operation functions and its corresponding customized memory operation functions. Given this function matching list, NLP-EYE dynamically instruments stubs before function calls memory operation and memory access points in advance to record and revise memory region states. NLP-EYE identifies memory operation function calls by simply comparing the called function name and the function names in the function matching list. The stubs are extra code snippets that are executed before the symbolic execution engine measuring the instrumented statements. We manually made up a coarse function misuse list which contains general function misuse implementations, such as a memory region can not be released more than once and a memory region can not be accessed after being released. Given this list, once symbolic execution reaches any memory access point or any function call site of a memory operation function, NLP-EYE executes the instrumented stub and checks misuses. If it meets a misuse, NLP-EYE will report this misuse as a vulnerability. Otherwise, the corresponding memory state will be updated (i.e., allocated or released) based on the function call of the memory operation function. For instance, the source code in figure 1, NLP-EYE instruments before line 18, 23 and 28 since the called functions are identified as memory operation functions. Then during symbolic execution, NLP-EYE records that a memory region is allocated in line 18 and released in line 23. When symbolic execution reaches line 28, it recognizes that a memory region (i.e., buf) is to be released twice which is one of the given function misuses, therefore NLP-EYE reports a double-free vulnerability.

	Lines of code	# of functions	# of memory operation functions
Vim-8.1 [17]	468,133	16,012	73
ImageMagick-7.0.8-15 [9]	514,472	14,636	79
CPython-3.8.0a0 [3]	556,950	12,000	66
Git-2.21.0 [4]	289,532	8,788	32
GraphicsMagick-1.3.31 [8]	369,569	7,406	29
GnuTLS-3.6.5 [6]	488,654	5,433	11
LibTIFF-4.0.10 [11]	85,791	1,326	4
Total	2,773,101	65,601	294

Table 1: Lines of code, number of functions and number of memory operation functions collected from each library/program.

4 Evaluation

In this section, we report the results of four experiments. The first experiment assesses the performance of function prototype segmentation. The second demonstrates the accuracy of NLP-EYE while identifying memory operation functions, and whether the context corpus has any impact on the identification accuracy. The third experiment looks into the vulnerability detection ability of NLP-EYE, and the last experiment discusses its runtime performance.

4.1 Experiment Setup

Dataset. We collected the latest version of seven popular open source libraries and programs written in C/C++ programming language with a total of 65,601 functions by December 2018 (see Table 1 for more details).

Due to the lack of open source labeled memory operation functions, we created our benchmarks. For identifying memory operation functions, we asked a team of annotators (3 programmers), all with more than seven years of programming experience in C/C++ to examine the implementations of memory operation functions. We first required team members to label memory operation functions independently, and then all members checked the results together. If there were any function with different labels, team members would discuss an agreement to label this function before it could be included in the dataset. In this procedure, we found 294 memory operation functions in total.

Implementation. We evaluated NLP-EYE on a Ubuntu 16.04 x64 workstation with an Intel Core i7-6700 CPU (four cores, 3.40 GHz) and 64 GB RAM. For the function prototype segmentation, we used NLTK [32], a natural language processing toolkit, to create the adaptive corpus for segmentation. We used the WordSegment [15] module in Python to split function prototypes. Gensim [37] is set up for NLP-based comparison, which conducts the similarity comparison based on the context corpus. Finally, we adopted Clang Static Analyzer [1] to perform type comparison and symbolic execution. Clang Static Analyzer is a a source code analysis tool which adopts symbolic execution to analyze each translation unit. It provides a framework that developers can intercept the symbolic execution process at specific points such as function call and memory access. In addition, Clang Static Analyzer provide

useful programming interfaces that can be used by developers to interact with the data type.

4.1.1 Experiment Design

To evaluate the effectiveness and efficiency of NLP-EYE, we present the designed four experiments in details below.

EX1 (Prototype Segmentation). To evaluate the effectiveness of prototype segmentation, we measured the Levenshtein-inspired distance [45] [38] of the segmentation results as our evaluation metrics. The distance between two segmentation lists i and j of string s is given by d_{ij}, which can be calculated as:

$$d_{ij} = \sum_{k=1}^{|s|-1} (vec_i[k] \quad xor \quad vec_j[k])$$

where $|s|$ represents the length of string s. Segmentation lists i and j are converted into vectors, vec_i and vec_j. In each vector, zero is regarded as "without split", and one is "split".

For the Levenshtein-inspired distance, a lower distance with the correct one indicates that the segmentation list requires fewer edit operations (i.e., split and merge) to be adjusted to the correct one. Thus, a lower distance specifies a better segmentation result.

EX2 (Memory Operation Function Identification). NLP-EYE identifies memory operation functions by using NLP-based comparison and type comparison. We evaluated the function identification performance by using precision, recall and F-measure as the evaluation metrics.

EX3 (Vulnerability Detection). We targeted on typical memory corruption vulnerabilities in this paper, i.e., double-free, use-after-free, and null pointer de-reference against real world software products such as Vim and CPython. To evaluate the effectiveness of NLP-EYE, we further compared it with the other four vulnerability detection tools (MallocChecker [13], Cppcheck [2], Infer [10] and SVF [42]), and counted the number of vulnerabilities that are correctly detected.

EX4 (Runtime Performance). We evaluated the average time cost of each phase in NLP-EYE, including preprocessing, semantics extraction, and vulnerability detection.

4.2 Ex1: Prototype Segmentation

Before we start, we manually split the function names we collected as the ground truth. We counted the number of function names that are correctly segmented, and then calculated the Levenshtein-inspired distance to evaluate the performance of each segmentation. Further, we compared the segmentation results that are generated by the adaptive corpus of NLP-EYE with the corresponding results generated by Google Web Trillion Word Corpus (GWTWC). It assesses the result accuracy while applying the adaptive corpus.

NLP-EYE correctly segments 230 out of 350 function names. Levenshtein-inspired distances of those function names are zero. Figure 4 demonstrated the average distance of each library and program by using the adaptive

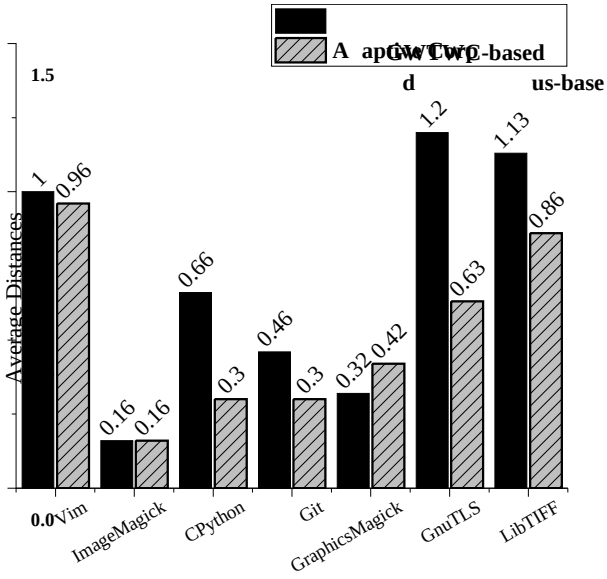

Figure 4: Segmentation results of function names by using NLP-EYE and GWTWC

corpus of NLP-EYE and GWTWC. NLP-EYE segments function names of Vim, ImageMagick, CPython, Git, GraphicsMagick,GnuTLS and LibTIFF accurately with the Levenshtein-inspired distance as 0.96, 0.16, 0.3, 0.3, 0.42, 0.63 and 0.86 respectively. The results for LibTIFF and Vim are worse than the others, because lots of function names involve single letters, and NLP-EYE cannot distinguish those letters from a word.

Except for GraphicsMagick and ImageMagick, the adaptive corpus-based segmentation performs better than the GWTWC-based segmentation. According to our manual inspection, we found that GWTWC is not a programming language-based corpus and it cannot proceed programming abbreviations. Thus, most of its segmentation results are worse than the results of the adaptive corpus-based segmentation. However, this conclusion is not satisfied on GraphicsMagick, because some function names are incorrectly divided into abbreviations by the adaptive corpus. Taken a function name `preview` as an example, it is divided into ["`pre`", "`view`"] instead of "`preview`", because the frequencies of those two abbreviations are higher in comments. For the ImageMagick, most function names are declared in normalized words, which are easy for GWTWC and the adaptive corpus to distinguish each word.

4.3 Ex2: Memory Operation Function Identification

We counted the number of memory operation functions that are correctly detected by NLP-EYE and computed the precision, recall, and F-measure on the entire dataset. To conduct this experiment, we separately set the thresholds (fn-similarity and arg-similarity) as (0.3, 0.4, 0.5) for function names and argument names, and found that NLP-EYE performs the best when fn-similarity and arg-similarity are 0.4 and 0.5, respectively.

	# of identified functions	# of correctly identified functions	Precision	Recall	F-measure
Vim	304	42	13%	57%	21%
ImageMagick	137	44	32%	55%	40%
CPython	131	48	36%	72%	48%
Git	46	8	17%	25%	20%
GraphicsMagick	69	16	23%	55%	32%
GnuTLS	74	4	5%	36%	8%
LibTIFF	8	0	0	0	0
Total	769	162	21%	55%	30%

Table 2: Memory operation function identification results of NLP-EYE

	NPD		**DF**		**UAF**	
	Detected	Confirmed	Detected	Confirmed	Detected	Confirmed
Vim	17	**17**	2	**1**	8	**2**
CPython	10	**4**	1	**1**	8	**1**
Git	1	**1**	0	0	0	0
GraphicsMagick	6	**5**	0	0	0	0
Total	34	**27**	3	**2**	16	**3**

Table 3: Detection results of null pointer de-reference (NPD), double-free (DF) and use-after-free (UAF). Note that this result only shows the vulnerabilities caused by customized memory operation functions.

Function Identification Results. We applied the StackOverflow corpus for NLP-based comparison. All the posts from the StackOverflow forum [16] are included in the StackOverflow corpus. Table 2 shows the best identification result with the number of identified functions and the number of memory operation functions that are correctly identified. We also computed precision, recall, and F-measure of NLP-EYE. NLP-EYE correctly identifies 162 memory operation functions out of the 769 identified functions, with precision, recall, F-measure value of 21%, 55%, and 30%, respectively. For LibTIFF, NLP-EYE cannot detect any memory operation functions because many single letters are used to name a function argument. For example, "s" is commonly used to express "size" that causes the recognition of memory operation functions even harder if the thresholds are too high. We then determine a balance between the thresholds (i.e., fn-similarity and arg-similarity) and the identification accuracy.

Within millions of functions, NLP-EYE narrows down the number of functions that need to be analyzed, and the total number of functions for manual analysis is acceptable. Furthermore, the false positive and the false negative are reasonable.

Context Corpus Selection. We further applied NLP-based comparison on two extra context corpuses (i.e., Wikipedia corpus, and customized corpus) to assess the identification performance. The Wikipedia corpus contains all webpages from Wikipedia [19]. Alternatively, the customized corpus consists of: 1) Linux man pages [12]; 2) Part of GNU Manuals [5]; and 3) two programming tutorials, i.e., C++ Primer [31] and C Primer Plus [36].

Based on the Wikipedia corpus, NLP-EYE only identifies no more than ten memory operation functions in each library and program with a precision value of 7%, and a worse recall value. While using customized corpus as the context corpus, the precision and recall of NLP-EYE are 42% and 19%, respectively. Although its precision is acceptable, it still causes too many false negatives. By manually analyzing the results, we found that Wikipedia corpus is insensitive to the programming language, and most identified functions are unrelated to memory operation. For the customized corpus, it fails to identify functions that use abbreviations, which cause exceptions if words are not found in the corpus.

4.4 Ex3: Vulnerability Detection

We tested NLP-EYE on the seven libraries and programs to examine whether there is any unknown memory corruption vulnerability. Note that the seven collected libraries and programs are the latest versions (collected in December 2018).

Vulnerabilities Detected by NLP-EYE. NLP-EYE detects 49 vulnerabilities from these libraries and programs in total. While only considering vulnerabilities caused by customized memory operation functions, four libraries and programs are involved. The detection result is shown in Table 3. By manually verifying these results, NLP-EYE successfully detects 32 vulnerabilities, including 27 null pointer de-reference, two double-free, and three use-after-free, existed in customized memory operation functions. To further verify the correctness of our results, we reported the manual-confirmed vulnerabilities to developers, and they have confirmed and patched ten

	NLP-EYE	MallocChecker	Cppcheck	Infer	SVF
Vim	3.82	2.77	18.90	51.28	50.92
ImageMagick	6.16	5.00	28.00	64.25	0.25
Cpython	8.31	7.70	1.47	23.43	0.26
Git	3.11	2.80	0.88	13.52	2.36
GraphicsMagick	2.08	1.45	11.83	8.75	0.15
GntTLS	2.75	2.33	9.65	11.13	0.11
LibTIFF	0.91	0.87	0.93	3.55	0.04
Total	27.14	22.92	71.66	175.91	54.09

Table 4: Runtime performance comparison (minutes)

null pointer de-reference and all the double-free, use-after-free vulnerabilities. Each customized memory operation function may cause vulnerabilities, since NLP-EYE failed to identify a part of them, this may lead to a false negative of vulnerability detection result. Besides the successfully detected vulnerabilities, NLP-EYE made false positive as well listed in Table 3. However, after we manually inspected the false positive, we found that none of them are caused by the wrong identification result.

There are two reasons that cause the false positive: 1) symbolic execution engine proceeds the expression with indexes in a loop as a static expression. For instance, the engine may report a double free on an array with different index in a loop since the engine regard the array element with different index as the same value; 2) While processing a conditional statement with a complex logic, the symbolic execution engine executes every path without considering the constraints defined in the conditional statements.

Detection Effectiveness Assessment. To assess the detection effectiveness of NLP-EYE, we applied four detection tools, MallocChecker, Cppcheck, Infer and SVF, to the entire dataset for comparison. MallocChecker and Infer claim to detect all three kinds of vulnerabilities. Cppcheck and SVF are designed to detect vulnerabilities of use-after-free and double-free. For the null pointer de-reference vulnerability, MallocChecker and Infer correctly reported 11 and 30 vulnerabilities, respectively. However, they can only report those misuses caused by standard memory allocation functions, while NLP-EYE can detect both standard and customized memory allocation functions. Even worse, none of these tools can detect vulnerabilities of use-after-free and double-free correctly.

We analyzed false positives caused by these tools. Similar to NLP-EYE, the symbolic execution engine of MallocChecker cannot identify the index of an array in a loop. Although Cppcheck can detect use-after-free vulnerabilities, it became inaccurate when lots of variables are declared to operate dynamic memories. Infer checks all returned pointers, which cause many false positives. It even reported a use-after-free vulnerability existed in an integer statement. SVF performed the worst by reporting hundreds of double-free vulnerabilities, which causes lots of errors.

4.5 Ex4: Runtime Performance

We evaluated the time cost of each phase (i.e., preprocessing, semantics extraction, and vulnerability detection) of NLP-EYE. Additionally, we tested the runtime of the other detection tools to assess the efficiency of NLP-EYE.

Before vulnerability detection, we collected all the posts on StackOverflow forum with the size of 17GB to create the context corpus, and it costs 56 hours to generate the model file. This step processes only once because we can repeatedly use the context corpus in further analysis.

Table 4 shows the total runtime cost of NLP-EYE and the other tools while analyzing our dataset. NLP-EYE preprocesses each library and program, and constructs the corresponding adaptive corpus within one seconds. It further spends 36.601s on average to identify memory operation functions in each library and program. NLP-EYE spends 70.917s on ImageMagick, but no more than 6s on LibTIFF, because ImageMagick has 14,636 functions and LibTIFF only includes 1,326 functions.

By comparing with the other tools, the runtime performance of NLP-EYE and MallocChecker are similar, since they use the same symbolic execution engine. SVF sacrifices the detection accuracy to achieve a higher runtime performance. Unfortunately, it is unhelpful for programmers to pinpoint vulnerabilities. Cppcheck and Infer analyze the entire source code to ensure a complete coverage, which costs much time.

4.6 Limitations

NLP-EYE successfully detects some memory corruption vulnerabilities other tools cannot detect. The results of function identification and vulnerability detection indicate that NLP-EYE understands the function semantics well with only limited information. However, we still have the following limitations that cause detection failures.

1. When a function implementation is complex, the symbolic execution engine in NLP-EYE cannot correctly analyze the data flow and control flow.

2. NLP-EYE cannot handle single letters involved in the function prototypes which may causes false positive and false negative.

File1: GraphicsMagick/magick/memory.c

```
1 MagickExport void * MagickMalloc(const size_t size){
2       if (size == 0)
3           return ((void *) NULL);
4       MEMORY_LIMIT_CHECK(GetCurrentFunction(),size);
5       return (MallocFunc)(size);
6 }
```

File2: GraphicsMagick/coders/pdb.c

```
1   static Image *ReadPDBImage(const ImageInfo
2   *image_info,ExceptionInfo *exception){
3       ...
4       comment=MagickAllocateMemory(char *,length+1);
5       p=comment;
6       p[0]='\0';
7       ...
8   }
```

Figure 5: A null pointer de-reference vulnerability in GraphicsMagick

4.7 Case Study

We discuss two representative vulnerabilities found in the GraphicsMagick library and the CPython interpreter, respectively.

GraphicsMagick is a library that was derived from the ImageMagick image processing utility in November 2002. GraphicsMagick is securely designed and implemented after being tested by Memcheck and Helgrind[3]. Also, AddressSanitizer (ASAN) [40], the most mature redzone-based memory error detector, proves it to be secure against memory errors. Nevertheless, NLP-EYE detects six null pointer de-reference vulnerabilities from its latest version. An example is presented in Figure 5. The function `MagickAllocateMemory` is declared to allocate memories. If the dynamic memory is insufficient and a null pointer is returned by this function (Line 4 of File2), a segmentation fault will be triggered (Line 6 of File2).

To detect this vulnerability, a detector should recognize the customized memory allocation function `MagickAllocateMemory`, which is a macro definition of the `MagickMalloc` function. For `MagickMalloc`, its implementation is defined in File1, and a customized memory allocation function `MallocFunc` is declared in this function. Besides analyzing the standard memory operation functions, NLP-EYE first identifies the macro definition, `MagickAllocateMemory` in Line4 of File2, and uses its original function `MagicMalloc` in File1 to replace it. By proceeding the preprocessing and semantics extraction phases, NLP-EYE labels those functions as memory operation functions, and finally locates function misuses. In comparison, other detection tools (e.g., MallocChecker) cannot distinguish those customized functions (i.e, `MallocFunc`, `MagicMalloc`, and `MagickAllocateMemory`), and thus fail to detect the flaw.

[3]Memcheck is a memory error detector for C and C++ programs. Helgrind is a tool for detecting synchronisation errors in C, C++ and Fortran programs that use the POSIX pthreads threading primitives. They are all based on Valgrind [35]

File1: CPython/Objects/obmalloc.c

```
1 void PyMem_Free(void *ptr){
2     _PyMem.free(_PyMem.ctx, ptr);
3 }
```

File2: CPython/Modules/_randommodule.c

```
1   static PyObject *
2   random_seed(RandomObject *self, PyObject *args){
3       ...
4       res = _PyLong_AsByteArray((PyLongObject *)n,
5                                 (unsigned char *)key,
6                                  keyused * 4,
7                                 PY_LITTLE_ENDIAN,
8                                 0); /* unsigned */
9       if (res == -1) {
10          PyMem_Free(key);
11          goto Done;
12      }
13      ...
14      Done:
15          PyMem_Free(key);
16          return result;
17  }
```

Figure 6: A double-free vulnerability in CPython

Another sample code snippet with double-free vulnerability is shown in Figure 6, which is detected from CPython interpreter. Apparently, function `PyMem_Free` in File1 is a memory de-allocation function. If the variable *res* is -1, the variable *key* will be freed twice (Line 10 and 15 of File2, respectively) . To our surprise, this simple vulnerability was found neither by manual audit nor automated source code analysis. According to the feedback of CPython developers, the corresponding host function has been tested for many times, but the vulnerability still exists. Based on this feedback, we would say that identifying customized memory operation functions is suitable to memory corruption detection. NLP-EYE is very helpful in this scenario.

5 Related Work

There are prior efforts of vulnerability detection, in this section, we introduce these works based on their analysis approaches, i.e., source code-based analysis and binary code-based analysis.

5.1 Source Code-based Analysis

Previous studies detect vulnerabilities by applying program analysis on source code to extract pointer information [41] [44] and data dependencies [33], [24], [29], [28].

To analyze C programming source code, CRED [44] detects use-after-free vulnerabilities in C programs. It extracts points-to information by applying a path-sensitive demand-driven approach. To decrease false alarms, it uses spatio-temporal context reduction technique to construct use-after-free pairs

precisely. However, the pairing part is time consuming that every path in the source code is required to be analyzed and memorized. Instead of analyzing the entire source code, Pinpoint [41] applies sparse value-flow analysis to identify vulnerabilities in C programs, such as use-after-free, double-free. To reduces the cost of data dependency analysis, Pinpoint analyzes local data dependence first and then performs symbolic execution to memorize the non-local data dependency and path conditions.

Similar to the above, some other tools detect vulnerabilities by compare data-flows with some pre-defined rules/violations. CBMC [28] is a C bounded model checker, which examines safety of the assertions under a given bound. It translates assertions and loops into a formula. If this formula satisfies any pre-defined violations, then a violated assertion will be identified. Coccinelle [29] finds specific bug by comparing the code with a given pattern written in Semantic Patch Language (SmPL).

Source code-based analysis has also been applied to Linux kernel. Due to the large amount of kernel code in Linux, DR. CHECKER [33] and K-Miner [24] are designed to be more effective and efficiency. DR. CHECKER employs a soundy approach based on program analysis. It is capable of conducting large-scale analysis and detecting numerous classes of bugs in Linux kernel drivers. K-Miner finds vulnerabilities by setting up a virtual kernel environment and processing syscalls separately.

Those proposed tools perform well to detect vulnerabilities implemented under standard programming styles, such as calling standard library APIs, designing standard implementation steps. They cannot proceed those customized functions just like how NLP-EYE does.

Instead of applying program analysis, both VulPecker [30] and VUDDY [27] detects vulnerabilities based on the code similarity. VulPecker builds a vulnerability database by using diff hunk features collected from each vulnerable code and its corresponding patch code. VUDDY proceeds each vulnerable function as an unit, and then abstracts and normalizes vulnerable functions to ensure that they are able to detect clones with modifications. However, similarity-based techniques require a massive database that can be learnt from.

5.2 Binary Code-based Analysis

Instead of analyzing source code, binary code can also be adopted to identify memory corruption vulnerabilities on stacks and allocated memories [22], [35], [40], [26], [43], [25], [23].

Memory shadowing helps to track the memory status at runtime. It also causes large memory consumption. Dr.Memory [22] conducts memory checking on Windows and Linux. It uses memory shadowing to track the memory status and identifies stack usage within heap memory. Dr.Memory is flexible and lightweight by using an encoding for callstacks to reduce memory consumption. AddressSanitizer [40] minimizes the memory consumption by creating a compact shadow memory, which achieves a a 128-to-1 mapping. By implementing a specialized memory allocator and code instrumentation in the compiler, AddressSanitizer analyzes the vulnerabilities on stack, head, global variables. HOTracer [26] discovers heap overflow vulnerabilities by examining whether a heap access operation can be controlled by an attacker. HOTracer finds vulnerabilities by giving an accurate definition to buffer overflow and it uses a heuristic method to find memory allocation functions. HOTracer is able to identify memory allocation functions with a higher accuracy, and several unknown overflow vulnerabilities are detected.

Unfortunately, detecting memory corruptions through binary code-based analysis requires proper inputs, that can precisely trigger the corresponding memory operation. It might cause some false negatives because of the incomplete code coverage.

6 Conclusion

We propose an NLP-based automated approach to detect memory corruption vulnerabilities. A detection tool, NLP-EYE, is developed to identify vulnerabilities of null pointer dereference, use-after-free, double free. The novelty of our approach is that we retrieve the function semantics accurately based on a little function information, i.e., function prototypes and comments, instead of using the entire function implementations. With the help of NLP-based and type-based analyses, NLP-EYE identifies memory operation functions accurately. Our approach is also adaptable since NLP-EYE generates an adaptive corpus for different dataset by extracting their comments from source code and various programming styles.

In this work, we only focused on memory corruption vulnerabilities. We plan to extend NLP-EYE in future with additional reference functions to identify the other vulnerabilities. We also open source NLP-EYE to help analysts and developers to improve software security.

7 Acknowledgments

The authors would like to thank the anonymous reviewers for their feedback and our shepherd, Dongpeng Xu, for his valuable comments to help improving this paper.

This work was supported by the General Program of National Natural Science Foundation of China (Grant No.61872237), the Key Program of National Natural Science Foundation of China (Grant No.U1636217) and the National Key Research and Development Program of China (Grant No.2016YFB0801200).

We especially thank Huawei Technologies, Inc. for the research grant that supported this work, Ant Financial Services Group for the support of this research within the *SJTU-AntFinancial joint Institute of FinTech Security*, and Nanjing Turing Artificial Intelligence Institute with the internship program.

References

[1] Clang Static Analyzer. http://clang-analyzer.llvm.org.

[2] Cppcheck. http://cppcheck.sourceforge.net.

[3] CPython. https://www.python.org.

[4] Git. https://git-scm.com.

[5] GNU Manuals Online. https://www.gnu.org/manual/manual.en.html .

[6] GnuTLS. https://www.gnutls.org.

[7] Google Web Trillion Word Corpus. http://googleresearch.blogspot.com/2006/08/all-our-n-gram-are-belong-to-you.html.

[8] GraphicsMagick. http://www.graphicsmagick.org.

[9] ImageMagick. https://www.imagemagick.org.

[10] Infer. https://fbinfer.com.

[11] LibTIFF. http://www.libtiff.org.

[12] Linux man pages online. http://man7.org/linux/man-pages/index.html.

[13] MallocChecker. https://clang-analyzer.llvm.org/.

[14] Microsoft: 70 percent of all security bugs are memory safety issues. https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/.

[15] Python Wordsegment. https://pypi.org/project/wordsegment/.

[16] Stackoverflow. https://stackoverflow.com.

[17] Vim. https://www.vim.org.

[18] VisualCodeGrepper. https://github.com/nccgroup/VCG.

[19] Wikipedia. https://www.wikipedia.org.

[20] Windows 8 APIs References. https://docs.microsoft.com/en-us/windows/desktop/apiindex/windows-8-api-sets.

[21] Windows Driver API references. https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/.

[22] Bruening, Derek and Zhao, Qin. Practical Memory Checking with Dr. Memory. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2011.

[23] Dinakar Dhurjati and Vikram Adve. Efficiently Detecting All Dangling Pointer Uses in Production Servers. In *Proceedings of International Conference on Dependable Systems and Networks (DSN)*, 2006.

[24] David Gens, Simon Schmitt, Lucas Davi, and Ahmad-Reza Sadeghi. K-miner: Uncovering Memory Corruption in Linux. In *Proceedings of 2018 Network and Distributed System Security Symposium (NDSS)*, 2018.

[25] Niranjan Hasabnis, Ashish Misra, and R Sekar. Light-weight Bounds Checking. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization (CGO)*, 2012.

[26] Xiangkun Jia, Chao Zhang, Purui Su, Yi Yang, Huafeng Huang, and Dengguo Feng. Towards Efficient Heap Overflow Discovery. In *Proceedings of 26th USENIX Security Symposium USENIX Security (USENIX Security)*, 2017.

[27] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery. In *Proceedings of 2017 IEEE Symposium on Security and Privacy (SP)*, 2017.

[28] Daniel Kroening and Michael Tautschnig. CBMC–C Bounded Model Checker. In *Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2014.

[29] Julia Lawall, Ben Laurie, Ren'e Rydhof Hansen, Nicolas Palix, and Gilles Muller. Finding Error Handling Bugs in Openssl Using Coccinelle. In *Proceedings of 2010 European Dependable Computing Conference (EDCC)*, 2010.

[30] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Hanchao Qi, and Jie Hu. VulPecker: an Automated Vulnerability Detection System Based on Code Similarity Analysis. In *Proceedings of the 32nd Annual Conference on Computer Security Applications (ACSAC)*, 2016.

[31] Stanley B. Lippman. *C++ Primer*. 2012.

[32] Edward Loper and Steven Bird. NLTK: the Natural Language Toolkit. *arXiv preprint cs/0205028*, 2002.

[33] Aravind Machiry, Chad Spensky, Jake Corina, Nick Stephens, Christopher Kruegel, and Giovanni Vigna. DR.CHECKER: A Soundy Analysis for Linux Kernel Drivers. In *Proceedings of 26th USENIX Security Symposium USENIX Security (USENIX Security)*, 2017.

[34] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient Estimation of word representations in Vector Space. *arXiv preprint arXiv:1301.3781*, 2013.

[35] Nicholas Nethercote and Julian Seward. Valgrind: a Framework for Heavyweight Dynamic binary instrumentation. In *Proceedings of ACM Sigplan notices*, 2007.

[36] Stephen Prata. *C Primer Plus*. 2014.

[37] Radim Rehurek and Petr Sojka. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks (LREC)*, 2010.

[38] Edward J Schwartz, Cory F Cohen, Michael Duggan, Jeffrey Gennari, Jeffrey S Havrilla, and Charles Hines. 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] Toby Segaran and Jeff Hammerbacher. *Beautiful Data: the Stories Behind Elegant Data Solutions*. 2009.

[40] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC)*, 2012.

[41] Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. Pinpoint: Fast and Precise Sparse Value Flow Analysis for Million Lines of Code. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2018.

[42] Yulei Sui and Jingling Xue. SVF: Interprocedural Static Value-flow Analysis in LLVM. In *Proceedings of the 25th international conference on compiler construction (CC)*, 2016.

[43] Erik Van Der Kouwe, Vinod Nigade, and Cristiano Giuffrida. Dangsan: Scalable use-after-free Detection. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys)*, 2017.

[44] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. Spatio-Temporal Context Reduction: a Pointer-Analysis-Based Static Approach for Detecting Use-After-Free Vulnerabilities. In *Proceedings of 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, 2018.

[45] Li Yujian and Liu Bo. A Normalized Levenshtein Distance Metric. *IEEE transactions on pattern analysis and machine intelligence (TPAMI)*, 2007.