TSINGHUA SCIENCE AND TECHNOLOGY
ISSN 1007-0214 05/13 pp604–613
DOI: 10.26599/TST.2019.9010068
Volume 25, Number 5, October 2020

A Memory-Related Vulnerability Detection Approach Based on Vulnerability Features

Jinchang Hu, Jinfu Chen*, Lin Zhang, Yisong Liu, Qihao Bao, Hilary Ackah-Arthur, and Chi Zhang

Abstract: Developing secure software systems is a major challenge in the software industry due to errors or weaknesses that bring vulnerabilities to the software system. To address this challenge, researchers often use the source code features of vulnerabilities to improve vulnerability detection. Notwithstanding the success achieved by these techniques, the existing studies mainly focus on the conceptual description without an accurate definition of vulnerability features. In this study, we introduce a novel and efficient Memory-Related Vulnerability Detection Approach using Vulnerability Features (MRVDAVF). Our framework uses three distinct strategies to improve vulnerability detection. In the first stage, we introduce an improved Control Flow Graph (CFG) and Pointer-related Control Flow Graph (PCFG) to describe the features of some common vulnerabilities, including memory leak, double-free, and use-after-free. Afterward, two algorithms, namely Vulnerability Judging algorithm based on Vulnerability Feature (VJVF) and Feature Judging (FJ) algorithm, are employed to detect memory-related vulnerabilities. Finally, the proposed model is validated using three test cases obtained from Juliet Test Suite. The experimental results show that the proposed approach is feasible and effective.

Key words: vulnerability feature; Control Flow Graph (CFG); Memory Leak (ML); Double-Free (DF); Use-After-Free (UAF)

1 Introduction

Recent years have witnessed a significant interest in vulnerability analysis from different perspectives, such as vulnerability prediction, classification, and vulnerability detection based on data collected from open-source repositories. Generally, the detection-based approaches rely on source-code features for this detection[1, 2]. Currently, some progress has been made with regards to the research on software vulnerabilities, but the research on software vulnerability features or characteristics is still limited. Previous studies mainly focus on the conceptual description[2] of vulnerability without an accurate definition of vulnerability feature. In addition, there are limited studies on the formal description of vulnerability features. Memory leak[3], double-free[4], and use-after-free[4, 5] are closely related to the use of dynamic memory, and are the major causes of vulnerabilities. The study on vulnerability features will promote the research on the prevention and detection of vulnerabilities. To further improve the body of the literature on vulnerability analysis, this study introduces a novel and efficient Memory-Related Vulnerability Detection Approach using Vulnerability Features (MRVDAVF).

Our framework employs three distinct strategies based on vulnerability features to improve vulnerability detection. In the first stage, we introduce an improved Control Flow Graph (CFG) and Pointer-related Control Flow Graph (PCFG) to describe the features of memory leak, double-free, and use-after-free vulnerabilities. Next, two algorithms, namely

• Jinchang Hu, Jinfu Chen, Lin Zhang, Yisong Liu, Qihao Bao, Hilary Ackah-Arthur, and Chi Zhang are with the School of Computer Science and Communication Engineering, Jiangsu University, Zhenjiang 212013, China. E-mail: 2211708037@stmail.ujs.edu.cn; jinfuchen@ujs.edu.cn; 1228175948@qq.com; liuyisong@ujs.edu.cn; qihaobao@stmail.ujs.edu.cn; hilaryaa@ujs.edu.cn; 2211708035@stmail.ujs.edu.cn.
∗ To whom correspondence should be addressed.
Manuscript received: 2019-10-24; accepted: 2019-11-05

Vulnerability Judging algorithm based on Vulnerability Feature (VJVF) algorithm and Feature Judging (FJ) algorithm, are utilized to detect memory-related vulnerabilities. Furthermore, we benchmark the model against four vulnerability detection tools, namely MRVDAVF, cppcheck[6], flawfinder[7], and splint[8, 9]. The main contributions of this paper are as follows.

(1) We conducted a meta-data analysis of memory-related vulnerabilities to determine the characteristics of memory leak, double-free, and use-after-free vulnerabilities.

(2) We propose an improved CFG – PCFG model using vulnerability features to improve vulnerability feature definition.

(3) We propose an efficient and effective vulnerability detection tool called MRVDAVF to detect memory leak, double-free, and use-after-free vulnerabilities.

The remainder of this paper is organized as follows: Section 2 discusses related work. Section 3 describes the formalization of vulnerability feature. Section 4 details the vulnerability detection framework of MRVDAVF. Section 5 reports the experimental analysis. Finally, Section 6 provides a conclusion to this paper.

2 Related Work

The research on software vulnerability mainly focuses on the causes, classification, and detection of vulnerabilities. There is little authoritative research on vulnerability feature definition and feature formalization. The representative studies on the features of memory leak, double-free, and use-after-free vulnerabilities are as follows.

Caballero et al.[4] pointed out that one important feature of double-free and use-after-free is that the occurrence of such vulnerabilities is a result of the creation and use of dangling pointers; hence, the focus is the creation of dangling pointers. Yamaguchi et al.[10] studied the features of memory leak and use-after-free using control-flow vulnerability description. The findings show that the proposed model is effective. The major limitation of their study is that the technique mainly focuses on exploiting vulnerability features to detect vulnerabilities based on graph traversal; however, the description of vulnerability features is not detailed enough. Zeng et al.[11] studied the features of memory-related vulnerabilities, including memory leak and use-after-free. They designed a detection method for memory-related vulnerabilities based on memory information management and memory block lifecycle. Their method makes good use of the features of memory information for the memory-related vulnerabilities. Nevertheless, the vulnerability features were not formalized. Wang et al.[12] formally defined the features of software vulnerabilities, which include the initial vulnerability node set, state space, vulnerability syntax rule set, and pre- and post-determination conditions. However, their description of vulnerability features is relatively general. Liu et al.[13] defined a formal description method for the semantic features of memory leak vulnerabilities and defined syntax detection rules based on the formal features, but their method can only describe memory leak formally. Han et al.[14] studied the features of use-after-free and showed that the feature of use-after-free is allocating memory, releasing and using the released memory in the program, and these three operations appear in order. However, the vulnerability feature described in their study is not formalized in depth.

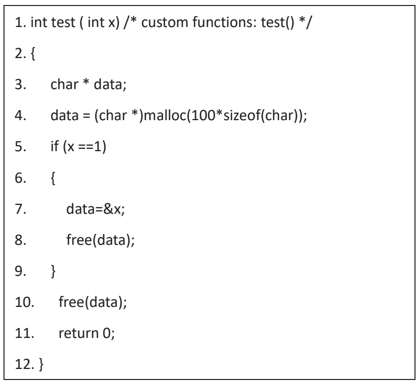
Generally, great achievements have been made in the research on the features of memory leak, double-free, and use-after-free, but the existing research lacks an authoritative method to define and describe the features of the various vulnerabilities. Therefore, analyzing the vulnerability features in-depth and formalizing the definition and description of a vulnerability feature are of great significance, as this can further promote vulnerability detection.

3 Vulnerability Feature Formalization

The feature of software vulnerability is the special quality that can distinguish one kind of software vulnerability from others. The unified definition and formal description of a software vulnerability feature will promote vulnerability model construction and vulnerability detection.

3.1 Definition of vulnerability feature

Software vulnerabilities are hidden in the source code of the software. Although software vulnerabilities have various types and forms, they often have lexical features, syntax features, and semantic features from the viewpoint of the source code. Taking the program code in Fig. 1 as an example, if the parameter x with the value of 1 is passed to the function test (), a double-free vulnerability can occur. In Fig. 1, pointer variables (such as pointer variable "data" declared in Line 3), malloc function (Line 4), and free function (Lines 8 and 10) constitute the lexical features of



```
1. int test ( int x) /* custom functions: test() */
2. {
3.     char * data;
4.     data = (char *)malloc(100*sizeof(char));
5.     if (x ==1)
6.     {
7.        data=&x;
8.        free(data);
9.     }
10.    free(data);
11.    return 0;
12. }
```

Fig. 1 Example program code.

vulnerability, which are named as vulnerability risk factors ξ in this paper. The pointer definition statement (Line 3), the memory allocation statement (Line 4), and the memory release statement (Lines 8 and 10) are syntax features of vulnerabilities, which are called vulnerability-related risk nodes N in this paper. The dependencies on memory allocation, memory release, and pointer definition constitute the semantic features of vulnerability, which are called vulnerability-related constraints C in this paper. Therefore, the vulnerability feature of a program can always be described based on lexical, syntax, and semantic perspectives.

Definition 1 defines vulnerability feature σ for three vulnerabilities: memory leak, double-free, and use-after-free.

Definition 1 Vulnerability Feature σ: σ(VulType) $= \{N, C\}$. VulType represents the vulnerability type, N represents the vulnerability-related risk nodes, and $N = \{N_1, N_2, \dots, N_i\}, i = 0, 1, 2, \dots$. For a program Prog, we call all program statements containing vulnerability risk factors ξ as vulnerability-related risk nodes N. C is a set of vulnerability-related constraints, which represents the constraints among elements in vulnerability-related nodes N. $C = C_1 || C_2 || \cdots || C_j || \cdots, j = 0, 1, 2, \dots$. That is, the feature of vulnerability may include multiple vulnerability-related constraints.

3.2 PCFG

At present, researchers worldwide usually design specific vulnerability detection algorithms to detect memory-related vulnerabilities based on the CFG[15, 16] of the program. The CFG was first proposed by Frances E. Allen in 1970. It shows the control-flow information of the program in the form of a graph. Each node in a CFG corresponds to a statement in the program. The deficiency of the CFG is that the information contained in the graph is limited, and consequently, the vulnerability detection algorithm is not optimally implemented. This paper proposes an improved CFG called PCFG. While CFG contains nodes of program statements, PCFG only contains the nodes of pointer definition, pointer access, and memory allocation and release, as well as related nodes, which can ensure the complete structure of the program. Each node in a PCFG graph has a feature property

Definition 2 PCFG: PCFG = $(N, E, \text{Entry}, \text{Exit})$. Here, N is a set of nodes, and $\forall n \in N$, $n = $ (id, ln, nextn_id, nf), where id denotes the number of Node n in the PCFG, ln denotes the line number of Node n in the source program, nextn_id denotes the id of the next node in the PCFG that n points to, and nf denotes the feature of the node that stores the pointers and feature operations involved. Moreover, E is a set of edges, and it is used to represent the pointing relation between nodes. For example, $\langle n_i, n_j \rangle$ indicates that there is a control flow from n_i to n_j. Entry is the entry node of PCFG, and Exit is the exit node of PCFG.

Definition 3 Executable Path (Path): In PCFG, if a program path is a sequence of statements represented as $\langle n_0, n_1, n_2, \dots, n_m \rangle$, and it satisfies $(n_{i-1}, n_i) \in E$, where $i = 1, 2, \dots, m$, then this program path can be called an executable path of PCFG.

Definition 4 Data Dependency (DataDep): For any two nodes n_x and n_y of PCFG, if there is an executable path from n_x to n_y, and there is a variable var defined in n_x and used in n_y, where var is not redefined anywhere else in the path from n_x to n_y, it is said that n_y is data-dependent on n_x and is denoted as DataDep(n_x, n_y, var).

Definition 5 Successor Node: For any two nodes n_x and n_y of PCFG, if there is an executable path from n_x to n_y, it is said that n_y is the successor node of n_x, and their relationship is denoted as Suc(n_x, n_y).

3.3 Formalization of vulnerability features

3.3.1 Related definition

The heap memory can be dynamically allocated for a program if necessary when the program is running. A programmer can request a certain size of memory from the heap by calling the corresponding function to release memory after usage; otherwise, the allocated memory would be leaked. Definitions 6–10 give the elements needed for vulnerability feature formalization.

Definition 6 DefPointerVariable(pv): DefPointerVariable(pv) is used for the declaration or definition of a pointer variable pv, abbreviated as DPV(pv).

Definition 7 MemoryMallocFunction(pv): MemoryMallocFunction(pv) is used to represent the memory allocation function, abbreviated as MMF(pv). Common memory allocation functions include malloc, calloc, and realloc of C programming language, and new, new [ ] of C++ programming language.

Definition 8 MemoryFreeFunction(pv): MemoryFreeFunction is used to represent the memory release function, abbreviated as MFF(pv). Common memory release functions include free of C programming language, delete, and delete [ ] of C++ programming language. In this paper, MMF(pv)/MFF(pv) is used to represent the match of the memory allocation and release functions and MMF(pv)/*MFF(pv) is used to represent the mismatch of the memory allocation and release functions.

Definition 9 UsePointerVariable(pv): UsePointerVariable(pv) is used to denote the use of pointer variable pv, abbreviated as UPV(pv).

Definition 10 Has(*n*, Operation/Function): Has(n, Operation/Function) denotes that the current node n contains certain defined operations or functions. The operations include DPV(pv), UPV(pv), MMF(pv), and MFF(pv). For example, $(\exists\, n \in \text{Prog}) \wedge \text{Has}(n, \text{DPV(pv)})$ indicates that a pointer variable pv is defined in a node n of a program Prog.

3.3.2 Memory leak

Memory Leak (ML) (CWE-401)[3,17,18] refers to the situation where a part of the heap memory allocated to a program is not released until the program ends, resulting in a reduction of system memory. Memory leak may eventually cause the program to run slowly or even lead to the collapse of the computer system. According to the definition of vulnerability feature, the feature of memory leak is $\sigma(\text{ML}) = \{N, C\}$. For memory leak, the specific formal descriptions of vulnerability-related risk nodes N and vulnerability feature constraint C are given below.

(1) $N = \{N^{\text{DPV}}, N^{\text{MMF}}, N^{\text{MFF}}\}$, $N^{\text{MFF}} = \{n \mid (\exists n \in \text{Prog}, \exists \text{pv} \in \text{Prog}) \bigwedge \text{Has}(n, \text{MFF(pv)})\}$.

Interpretation: N includes all nodes of N^{DPV}, N^{MMF}, and N^{MFF}. N^{DPV} includes all the nodes that declare or define the pointer variables, N^{MMF} includes all the nodes that allocate memory for pointer variables, and N^{MFF} includes all the nodes that release the allocated memory to which their pointer variables point.

(2) $C = C_1 || C_2 || C_3$.

(a) Unreleased Allocated Memory: $C_1 = (\exists \text{pv} \in \text{Prog}) \bigwedge ((\exists n_i, n_j \in \text{Prog}) \bigwedge \text{Has}(n_i, \text{DPV(pv)}) \bigwedge \text{Has}(n_j, \text{MMF(pv)}) \bigwedge \text{Suc}(n_i, n_j) \bigwedge \text{DataDep}(n_i, n_j, \text{pv})) \bigwedge \neg ((\exists n_k \in \text{Prog}) \bigwedge \text{Has}(n_k, \text{MFF(pv)}) \bigwedge \text{Suc}(n_j, n_k) \bigwedge \text{DataDep}(n_i, n_k, \text{pv}))$.

Interpretation: The constraint C_1 represents a situation where in an executable path, a pointer variable pv is defined in n_i, and then the program allocates memory for pv in n_j (n_j is the successor node of n_i and n_j is data-dependent on n_i). However, there is no node that releases the allocated memory that pv points to until the program ends.

(b) Pointer Reassigning: $C_2 = (\exists \text{pv} \in \text{Prog}) \bigwedge ((\exists n_i, n_j \in \text{Prog}) \bigwedge \text{Has}(n_i, \text{DPV(pv)}) \bigwedge \text{Has}(n_j, \text{MMF(pv)}) \bigwedge \text{Suc}(n_i, n_j) \bigwedge \text{DataDep}(n_i, n_j, \text{pv})) \bigwedge ((\exists n_k \in \text{Prog}) \bigwedge \text{Has}(n_k, \text{MMF(pv)}) \bigwedge \text{Suc}(n_j, n_k) \bigwedge \text{DataDep}(n_i, n_k, \text{pv}))$.

Interpretation: The constraint C_2 represents a situation where in an executable path, a pointer variable pv is defined in n_i, and then the program allocates memory for pv in n_j (n_j is the successor node of n_i and n_j is data-dependent on n_i). However, the program reallocates memory for pv in a node n_k (n_k is the successor node of n_j and n_k is data-dependent on n_i) of the program instead of releasing the initial allocated memory that pv points to, so the initial allocated memory that pv points to would be leaked.

(c) Mismatched Memory Allocation and Release: $C_3 = (\exists \text{path} \in \text{Prog}) \bigwedge (\exists \text{pv} \in \text{path}) \bigwedge ((\exists n_i, n_j \in \text{path}) \bigwedge \text{Has}(n_i, \text{DPV(pv)}) \bigwedge \text{Has}(n_j, \text{MMF(pv)}) \bigwedge \text{Suc}(n_i, n_j) \bigwedge \text{DataDep}(n_i, n_j, \text{pv})) \bigwedge ((\exists n_k \in \text{path}) \bigwedge \text{Has}(n_k, *\text{MFF(pv)}) \bigwedge \text{Suc}(n_j, n_k) \bigwedge \text{DataDep}(n_i, n_k, \text{pv}))$.

Interpretation: The constraint C_3 represents a situation where in an executable path, a pointer variable pv is defined in n_i, and then the program allocates memory for pv in n_j (n_j is the successor node of n_i and n_j is data-dependent on n_i), and finally, the memory release function is called to release the allocated memory pointed by pv in node n_k (n_k is the successor node of n_j and n_k is data-dependent on n_i). However, the memory release function does not match the memory allocation function; thus, the allocated memory pointed by pv would be leaked.

3.3.3 Double-free

Double-Free (DF) (CWE-415)[4,19] refers to the situation where in the process of running a program, the allocated memory pointed by a pointer is released, but the release function to the pointer is later called again. This causes a data structure error in the memory management and eventually leads to unknown defects in the program or the system. According to the definition of vulnerability feature, the feature of double-free is $\sigma(\text{DF}) = \{N, C\}$. For double-free, the specific formal descriptions of vulnerability-related risk nodes N and vulnerability feature constraint C are given below.

(1) $N = \{N^{\text{DPV}}, N^{\text{MMF}}, N^{\text{MFF}}\}$, and $N^{\text{DPV}} = \{n \mid (\exists n \in \text{Prog}, \exists \text{pv} \in \text{Prog}) \bigwedge \text{Has}(n, \text{DPV}(\text{pv}))\}$, $N^{\text{MMF}} = \{n \mid (\exists n \in \text{Prog}, \exists \text{pv} \in \text{Prog}) \bigwedge \text{Has}(n, \text{MMF}(\text{pv}))\}$, $N^{\text{MFF}} = \{n \mid (\exists n \in \text{Prog}, \exists \text{pv} \in \text{Prog}) \bigwedge \text{Has}(n, \text{MFF}(\text{pv}))\}$.

Interpretation: N includes all nodes of N^{DPV}, N^{MMF}, and N^{MFF}. N^{DPV} includes all the nodes that declare or define the pointer variables, N^{MMF} includes all the nodes that allocate memory for pointer variables, and N^{MFF} includes all the nodes that release the allocated memory to which their pointer variables point.

(2) $C = (\exists \text{pv} \in \text{Prog}) \bigwedge ((\exists n_i, n_j, n_k \in \text{Prog}) \bigwedge \text{Has}(n_i, \text{DPV}(\text{pv})) \bigwedge \text{Has}(n_j, \text{MMF}(\text{pv})) \bigwedge \text{Has}(n_k, \text{MFF}(\text{pv})) \bigwedge \text{Suc}(n_i, n_j) \bigwedge \text{DataDep}(n_i, n_j, \text{pv}) \bigwedge \text{Suc}(n_j, n_k) \bigwedge \text{DataDep}(n_i, n_k, p_v)) \bigwedge ((\exists n_u \in \text{Prog}) \bigwedge \text{Has}(n_u, \text{MFF}(\text{pv})) \bigwedge \text{Suc}(n_k, n_u) \bigwedge \text{DataDep}(n_i, n_u, \text{pv}))$.

Interpretation: The constraint C represents the situation where in an executable path, a pointer variable pv is defined in n_i; after allocating memory for pv in n_j (n_j is the successor node of n_i and n_j is data-dependent on n_i) and then releasing the allocated memory pointed by pv in n_k (n_k is the successor node of n_j and n_k is data-dependent on n_i), pv points to the unknown memory address. However, the program calls the memory release function to pv again in n_u (n_u is the successor node of n_k and n_u is data-dependent on n_i), which would lead to double-free.

3.3.4 Use-after-free

Use-After-Free (UAF) (CWE-416)[5,20–22] refers to the situation where a pointer is not to null after releasing the allocated memory for the pointer; hence, accessing the pointer may affect the running of the program or lead to unpredictable consequences. According to the definition of vulnerability feature, the feature of use-after-free is $\sigma(\text{UAF}) = \{N, C\}$. For use-after-free, the specific formal descriptions of the vulnerability-related risk nodes N and vulnerability feature constraint C are given below.

(1) $N = \{N^{\text{DPV}}, N^{\text{MMF}}, N^{\text{MFF}}\}$, and $N^{\text{DPV}} = \{n \mid (\exists n \in \text{Prog}, \exists \text{pv} \in \text{Prog}) \bigwedge \text{Has}(n, \text{DPV}(\text{pv}))\}$, $N^{\text{MMF}} = \{n \mid (\exists n \in \text{Prog}, \exists \text{pv} \in \text{Prog}) \bigwedge \text{Has}(n, \text{MMF}(\text{pv}))\}$, and $N^{\text{MFF}} = \{n \mid (\exists n \in \text{Prog}, \exists \text{pv} \in \text{Prog}) \bigwedge \text{Has}(n, \text{MFF}(\text{pv}))\}$.

Interpretation: N includes all nodes of N^{DPV}, N^{MMF}, N^{MFF}, and N^{UPV}. N^{DPV} includes all the nodes that declare or define the pointer variables, N^{MMF} includes all the nodes that allocate memory for pointer variables, N^{MFF} includes all the nodes that release the allocated memory to which their pointer variables point, and N^{UPV} includes all the nodes that access the pointer variables.

(2) $C = (\exists \text{pv} \in \text{Prog}) \bigwedge ((\exists n_i, n_j, n_k \in \text{Prog}) \bigwedge \text{Has}(n_i, \text{DPV}(\text{pv})) \bigwedge \text{Has}(n_j, \text{MMF}(\text{pv})) \bigwedge \text{Has}(n_k, \text{MFF}(\text{pv})) \bigwedge \text{Suc}(n_i, n_j) \bigwedge \text{DataDep}(n_i, n_j, \text{pv}) \bigwedge \text{Suc}(n_j, n_k) \bigwedge \text{DataDep}(n_i, n_k, p_v)) \bigwedge ((\exists n_u \in \text{Prog}) \bigwedge \text{Has}(n_u, \text{MFF}(\text{pv})) \bigwedge \text{Suc}(n_k, n_u) \bigwedge \text{DataDep}(n_i, n_u, \text{pv}))$.

Interpretation: The constraint C represents the situation where in an executable path, a pointer variable pv is defined in n_i; after allocating memory for pv in n_j (n_j is the successor node of n_i and n_j is data-dependent on n_i) and then releasing the allocated memory pointed by pv in n_k (n_k is the successor node of n_j and n_k is data-dependent on n_i), pv points to the unknown memory address. However, the program tries to access pv in n_u (n_u is the successor node of n_k and n_u is data-dependent on n_i), which would lead to use-after-free.

4 Vulnerability Detection Framework

4.1 Module analysis

As shown in Fig. 2, the detection framework of MRVDAVF consists of four modules: Abstract Syntax Tree (AST) generation module, node feature extraction

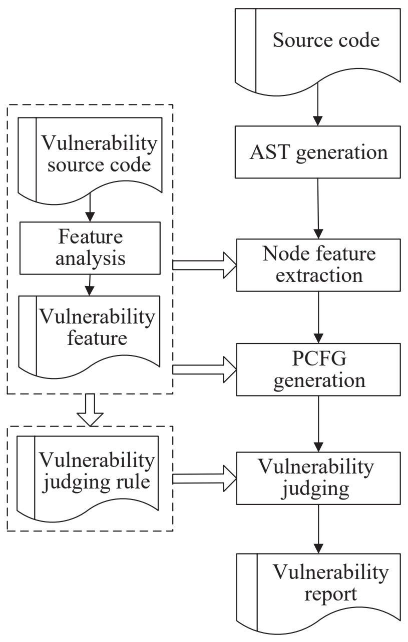


Fig. 2 Detection framework of MRVDAVF.

module, PCFG generation module, and vulnerability judging module. The details of these modules are given below.

(1) AST Generation Module: The AST generation module mainly uses the front-end of the GCC compiler to parse the source code and store the AST of the source code in text form. Since directly extracting PCFG from AST in text form is relatively inefficient, storing AST in the designed data structure is necessary.

(2) Node Feature Extraction Module: Node feature extraction module extracts the key information (the pointers and feature operations involved) of the node from AST and finally takes the extracted information as the feature property of the node in PCFG.

(3) PCFG Generation Module: According to the definition of PCFG, PCFG only contains pointer definition, pointer access, and memory allocation and release, as well as the related nodes to ensure the complete structure of the program. Figure 3 shows the data structure of the PCFG node. PCFG can be generated by traversing AST recursively, and only considering pointer-related nodes (the nodes of pointer definition, pointer access, and memory allocation and release, as well as related nodes, which ensure the complete structure of the program) should be considered when adding nodes. The consideration of pointer-related nodes would reduce the overhead of analyzing unnecessary nodes in the subsequent vulnerability judging module.

(4) Vulnerability Judging Module: The vulnerability judging module traverses the executable path of the PCFG of the program. In the traversal process, it analyses whether the nodes on the current path satisfy the vulnerability feature, and then judges whether there exists a vulnerability according to the vulnerability Judging Rule 1 discussed below.

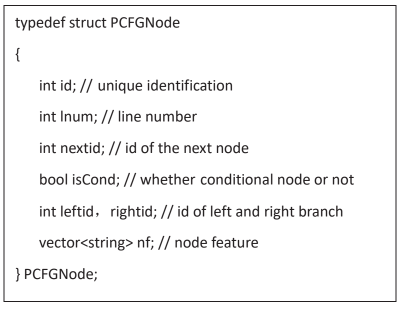
4.2 Vulnerability judging algorithm based on vulnerability feature

Based on the formalization of the vulnerability feature and PCFG of the target program, we can judge whether vulnerabilities exist in a program. The following judging rule would be used in the analysis process:

Judging Rule 1: For a program Prog, if the vulnerability-related risk nodes N for an executable path $p \in$ Path is not NULL, and there are elements in the vulnerability-related risk point N that satisfy the vulnerability-related constraint C, then vulnerability exists in the program.

Based on Judging Rule 1, VJVF is proposed, which is given in Algorithm 1. The main idea of VJVF is to traverse PCFG, record vulnerability-related risk nodes N during traversal, and judge whether vulnerabilities exist in the FJ algorithm.

The input of the VJVF algorithm is the PCFG of the



```
typedef struct PCFGNode
{
    int id; // unique identification
    int lnum; // line number
    int nextid; // id of the next node
    bool isCond; // whether conditional node or not
    int leftid，rightid; // id of left and right branch
    vector<string> nf; // node feature
} PCFGNode;
```

Fig. 3 Data structure of PCFG node.

Algorithm 1 VJVF algorithm

Input: PCFG

Output: err /*output the set of information including all vulnerability-related risk nodes ***N*** ***/**

```
root = PCFG.head; /* Obtain the head node of PCFG */
vector<unsigned int> pointers; /* Pointer set */
vector<stack<unsigned int>> deps; /* Record all vulnerability-related risk nodes */
VulTraverse(root.id); /* Traverse PCFG recursively */
VulTraverse(id); /* The body of VulTraverse () */
begin
if (id = 0) then
    return;
end if
node = PCFG.getNodeById(id);
flag = has(pointers, node); /* Judge whether pointers associated with PCFGNode exist in pointers */
if (flag) then
    pos = index(pointers, node);
    if (isMMF(node) || isMFF(node) || isUPV(node)) then
        deps[pos].push(id);
    end if
else
    stack<unsigned int> newStack;
    newStack.push(id);
    deps.push_back(newStack);
    pointers.push(id);
end if
err=FJ(deps); /* Check for vulnerabilities with FJ algorithm */
VulTraverse(node.left);
TraverseBack(); /* Roll back of PCFG nodes */
VulTraverse(node.next);
TraverseBack(); /* Roll back of PCFG nodes */
VulTraverse(node.right);
return err;
end
```

target program, and the output is err. The err records the set of information including all the vulnerability-related risk nodes N that causes the vulnerability. Lines 1–3 in the Algorithm 1, declare the relevant variables, and Line 4 calls the VulTraverse() function to traverse the PCFG and judge whether a vulnerability exists. Lines 5–30 are the main body of the VulTraverse() function. Lines 7–22 traverse the nodes of PCFG and record the information of vulnerability-related risk nodes N. Lines 23–28 judge whether there exists vulnerability in PCFG recursively. Line 29 returns all the information of vulnerability-related risk nodes N that causes the vulnerability. The execution time of the VJVF algorithm is mainly spent on node traversal; hence the execution time is linear with the number of nodes traversed. Since the total number of the nodes analyzed will not be more than the total number of PCFG nodes, the time complexity of the VJVF algorithm is $O(n)$, where n is the total number of PCFG nodes.

4.3 Feature judging algorithm

The FJ algorithm, given in Algorithm 2, is a sub-algorithm called by the VJVF algorithm. Its main task is to analyze the information of vulnerability-related risk nodes N by traversing in the program, and then judging whether there exist elements in vulnerability-related risk nodes N that satisfy the vulnerability-related constraint C. If the condition is met, we can conclude that there is a corresponding vulnerability in the program.

The input of the algorithm is deps, which records the set of all vulnerability-related risk nodes N; and the output is st, which records the information of the vulnerability-related risk node n that leads to a vulnerability. Lines 3 and 4 judge whether there is use-after-free vulnerability; Lines 5 and 6 judge whether there is double-free vulnerability; and Lines 7 and 8 judge whether there is memory leak vulnerability.

Algorithm 2 FJ algorithm

Input: deps /* The set of all vulnerability-related risk nodes *N/**

Output: st /* Stack information, which records the information of the vulnerability-related risk node *n* that leads to the vulnerability*/

```
1: for (int $i = 0$; $i <$ deps.size() $-1$; $i++$) do
2:   st = deps[$i$];
3:   if (the nodes in st satisfy the vulnerability-related constraints of use-after-free) /* Judge whether there exists a use-after-free */ then
4:     return UseAfterFreeError(st); /* Throw corresponding exception information */
5:   else if (the nodes in st satisfy the vulnerability-related constraints of double-free) /* Judge whether there exists double-free */ then
6:     return DoubleFreeError(st); /* Throw corresponding exception information */
7:   else if (the nodes in st satisfy the vulnerability-related constraints of memory leak) /* Judge whether there exists a memory leak */ then
8:     return MemoryLeakError(st); /* Throw corresponding exception information */
9:   end if
10: end for
```

5 Experimental Analysis

Comparative experiments among four vulnerability detection tools (cppcheck, flawfinder, splint, and MRVDAVF) were carried out. Three types of test case sets, consisting of CWE401_Memory_Leak, CWE415_Double_Free, and CWE416_Use_After_Free, were selected from Juliet Test Suite for C/CPP[23–25] and employed in the experiments. The details of the preprocessed test case sets are shown in Table 1.

The total number of vulnerabilities in the original test case set is recorded as TVN, the total number of vulnerabilities in the tool report is recorded as TRN, the number of true positives in the tool report is recorded as TTP, and the number of false negatives in the tool report is recorded as TFN, where $\text{TRN} = \text{TTP} + \text{TFN}$. According to the definition of False Negative Rate (FNR) and False Positive Rate (FPR), we know that $\text{FNR} = (\text{TVN} - \text{TTP})/\text{TVN}$ and $\text{FPR} = \text{TFN}/\text{TRN}$. The lower the FNR and FPR, the better the detection ability of the detection tool.

Table 2 presents the detection results of flawfinder, cppcheck, splint, and MRVDAVF on three test case sets.

Compared with flawfinder, cppcheck, and splint, MRVDAVF has the least FNR and FPR in detecting memory leak and use-after-free. In detecting double-free, cppcheck has the lowest FNR, but MRVDAVF has the lowest FPR. Therefore, compared with the other three tools, MRVDAVF generally has a better ability to detect memory leak, use-after-free, and double-free vulnerabilities.

Table 1 Test case sets selected from juliet test suite.

Test case set	Number of files	Number of total lines	TVN
ML	1032	141 760	1032
DF	560	73 624	560
UAF	398	654 99	398

Table 2 Detection results of four detection tools on three test case sets.

Test case set	TVN	Tool	TRN	TTP	TFN	FNR (%)	FPR (%)
ML	1032	Cppcheck	311	122	189	88.18	60.77
		Flawfinder	2304	260	2044	74.81	88.72
		Splint	340	132	208	87.21	61.18
		MRVDAVF	926	630	296	38.95	31.97
DF	560	Cppcheck	500	426	74	23.93	14.80
		Flawfinder	560	0	560	100	100
		Splint	228	114	114	79.64	50.00
		MRVDAVF	218	218	0	61.07	0
UAF	398	Cppcheck	293	0	293	100	100
		Flawfinder	434	0	434	100	100
		Splint	197	108	89	72.86	45.18
		MRVDAVF	386	245	141	38.44	36.53

6 Conclusion

Memory leak, double-free, and use-after-free vulnerabilities all have certain features. The study on vulnerability features will promote research on the prevention and detection of vulnerabilities. This study analyzes in-depth the features of three memory-related vulnerabilities (including memory leak, double-free, and use-after-free), and MRVDAVF is proposed. First, we define the vulnerability feature by analyzing numerous source codes containing memory-related vulnerabilities. Then we propose PCFG, which consists only of the nodes of pointer definition, pointer access, and memory allocation and release, as well as related nodes to ensure the complete structure of the program. We also formalize the features of three vulnerabilities based on feature definition and PCFG. The study describes the detection framework of MRVDAVF and details VJVF algorithm and FJ algorithm. To validate the effectiveness of vulnerability feature analysis and the feasibility of the MRVDAVF, comparative experiments are carried out among four vulnerability detection tools (MRVDAVF, cppcheck, flawfinder, and splint). Compared with cppcheck, flawfinder, and splint, MRVDAVF has a better detection ability for memory leak, use-after-free, and double-free vulnerabilities. The experimental results show that MRVDAVF is feasible and effective.

Acknowledgment

This study was funded by the National Natural Science Foundation of China (Nos. U1836116 and 61872167), the Project of Jiangsu Provincial Six Talent Peaks (No. XYDXXJS-016), and the Graduate Research Innovation Project of Jiangsu Province (No. KYCX17 1807).

References

[1] W. R. Fitriani, P. Rahayu, and D. I. Sensuse, Challenges in agile software development: A systematic literature review, in *Proc. of the 8th International Conference on Advanced Computer Science and Information Systems*, Bali, Indonesia, 2017, pp. 155–164.

[2] L. J. Liu, Y. Q. Shi, and R. Tao, The research of component-based software development application on data management in smart education, *Advances in Intelligent Systems and Computing*, vol. 279, no. 7, pp. 1099–1108, 2014.

[3] Z. B. Xu, J. Zhang, and Z. X. Xu, Melton: A practical and precise memory leak detection tool for C programs, *Frontiers of Computer Science*, vol. 9, no. 1, pp. 34–54, 2015.

[4] J. Caballero, G. Grieco, M. Marron, and A. Nappa, Undangle: Early detection of dangling pointers in use-after-free and double-free vulnerabilities, in *Proc. of International Symposium on Software Testing and Analysis*, Minneapolis, MN, USA, 2012, pp. 188–195.

[5] H. Yan, Y. L. Sui, S. P. Chen, and J. L. Xue, Spatio-temporal context reduction: A pointer-analysis-based static approach for detecting use-after-free vulnerabilities, in *Proc. of International Conference on Software Engineering*, Gothenburg, Sweden, 2018, pp. 327–337.

[6] J. S. Liu, Y. S. Chen, L. X. Zhang, J. Deng, and W. X. Zhang, The evaluation of the embedded software quality based on the binary code, in *Proc. of IEEE International Conference on Software Quality, Reliability and Security Companion*, Vienna, Austria, 2016, pp. 167–170.

[7] J. C. Liu, L. Q. Chen, L. M. Dong, and J. Wang, UC Bench: A user-centric benchmark suite for C code static analyzers, in *Proc. of International Conference on Information Science and Technology*, Wuhan, China, 2012, pp. 230–237.

[8] H. Shahriar, H. M. Haddad, and I. Vaidya, Buffer overflow patching for C and C++ programs: Rule-based approach, *ACM Sigapp Applied Computing Review*, vol. 13, no. 2, pp. 8–19, 2013.

[9] C. Chahar, V. S. Chauhan, and M. L. Das, Code analysis for software and system security using open source tools, *Information Security Journal: A Global Perspective*, vol. 21, no. 6, pp. 346–352, 2012.

[10] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, Modeling and discovering vulnerabilities with code property graphs, in *Proc. of IEEE Symposium on Security and Privacy*, San Jose, CA, USA, 2014, pp. 590–604.

[11] J. P. Zeng, Q. H. Yang, H. L. Wang, B. P. Xu, and W. Huang, Design and implementation of memory leak detection tool of C/C++ based on dynamic instrumentation, (in Chinese), *Application Research of Computers*, vol. 32, no. 6, pp. 1737–1741, 2015.

[12] T. Wang, L. S. Han, C. Fu, D. Q. Zhou, and M. Liu, Static software vulnerability detection model and detection framework, (in Chinese), *Computer Science*, vol. 43, no. 5, pp. 80–86, 2016.

[13] Z. Q. Liu, B. Xu, D. Liang, C. Liu, Z. J. Jiang, and C. L. Du, Semantics-based memory leak detection for C programs, in *Proc. of International Conference on Fuzzy Systems and Knowledge Discovery*, Changsha, China, 2016, pp. 2283–2287.

[14] X. H. Han, S. Wei, J. Y. Ye, C. Zhang, and Z. Y. Ye, Detect use-after-free vulnerabilities in binaries, (in Chinese), *Journal of Tsinghua University*, vol. 57, no. 10, pp. 1022–1029, 2017.

[15] K. S. Kumar and D. Malathi, A novel method to find time complexity of an algorithm by using control flow graph, in *Proc. of International Conference on Technical Advancements in Computers and Communications*, Melmaurvathur, India, 2017, pp. 66–68.

[16] A. V. Phan, M. L. Nguyen, and L. T. Bui, Convolutional neural networks over control flow graphs for software defect prediction, in *Proc. of International Conference on Tools with Artificial Intelligence*, Boston, MA, USA, 2017, pp. 45–52.

[17] Q. Gao, Y. F. Xiong, Y. Q. Mi, L. Zhang, W. K. Yang, Z. P. Zhou, B. Xie, and H. Mei, Safe memory-leak fixing for C programs, in *Proc. of International Conference on Software Engineering*, Firenze, Italy, 2015, pp. 459–470.

[18] X. H. Sun, S. H. Xu, C. K. Guo, J. Xu, N. P. Dong, X. J. Ji, and S. Zhang, A projection-based approach for memory leak detection, in *Proc. of Computer Software and Applications Conference*, Tokyo, Japan, 2018, pp. 430–435.

[19] Y. Chen, M. Khandaker, and Z. Wang, Pinpointing vulnerabilities, in *Proc. of ACM Asia Conference on Computer and Communications Security*, New York, NY, USA, 2017, pp. 334–345.

[20] D. Dewey, B. Reaves, and P. Traynor, Uncovering use-after-free conditions in compiled code, in *Proc. of International Conference on Availability, Reliability and Security*, Washington, DC, USA, 2015. pp. 90–99.

[21] J. Feist, L. Mounier, and M. L. Potet, Statically detecting use after free on binary code, *Journal of Computer Virology and Hacking Techniques*, vol. 10, no. 3, pp. 211–217, 2014.

[22] S. Liu and X. J. Qin, Parallelly refill SLUB objects freed in slow paths: An approach to exploit the use-after-free vulnerabilities in linux kernel, in *Proc. of International Conference on Parallel and Distributed Computing, Applications and Technologies*, Taipei, China, 2017, pp. 387–390.

[23] NSA center for assured software, Juliet test suite 1.2 for C/C++, https://samate.nist.gov/SRD/around.php#juliet_documents, 2018.

[24] A. Ibing and A. Mai, A fixed-point algorithm for automated static detection of infinite loops, in *Proc. of IEEE International Symposium on High Assurance Systems Engineering*, Daytona Beach, FL, USA, 2015, pp. 44–51.

[25] A. Wagner and J. Sametinger, Using the Juliet test suite to compare static security scanners, in *Proc. of International Conference on Security and Cryptography*, Vienna, Austria, 2014, pp. 244–252.

Jinchang Hu is a master student in the School of Computer Science and Communication Engineering, Jiangsu University, China. He received the BE degree from Jiangsu University, China, in 2017. His research interests include software testing and service computing.

Jinfu Chen received the BE degree from Nanchang Hangkong University, Nanchang, China, in 2004, and the PhD degree from Huazhong University of Science and Technology, Wuhan, China, in 2009. He is a professor in the School of Computer Science and Communication Engineering, Jiangsu University, Zhenjiang, China. His major research interests include software testing, software analysis, and trusted software. Prof. Chen is a member of the ACM and China Computer Federation.

Lin Zhang is a master student in the School of Computer Science and Communication Engineering, Jiangsu University, China. She received the BEng degree from Jiangsu University, China, in 2015. Her research interests include software testing and service computing.

Yisong Liu received the BE degree from Hunan University, Changsha, China, in 1988, the MS degree from Jiangsu University, Zhenjiang, China, in 1999, and the PhD degree from Nanjing University of Science and Technology, Nanjing, China, in 2009. He is a professor in the School of Computer Science and Communication Engineering, Jiangsu University, Zhenjiang, China. His major research interests include computer graphics, human-computer interaction techniques, and software engineering.

Qihao Bao is a master student in the School of Computer Science and Communication Engineering, Jiangsu University, China. He received the BE degree from Jiangsu University, China, in 2017. His research interests include software testing and service computing.

Hilary Ackah-Arthur received the BS degree from the University of Cape Coast, Ghana, in 2007, the MS degree from HAN University of Applied Sciences, The Netherlands, in 2011. Since 2011, he has been a lecturer with the Computer Science Department, Takoradi Technical University, Ghana. He is currently pursuing the doctorate degree at the School of Computer Science and Communication Engineering, Jiangsu University, China. His research interests include software analysis and testing, service computing, and information systems and security. Mr. Ackah-Arthur is a member of IEEE Computer Society.

Chi Zhang is a master student in the School of Computer Science and Communication Engineering, Jiangsu University, China. He received the BE degree from Jiangsu University, China, in 2017. His research interests include software testing and service computing.