Wuhan University
Journal of Natural Sciences

2019, Vol.24 No.2, 149-160

Article ID 1007-1202(2019)02-0149-12
DOI https://doi.org/10.1007/s11859-019-1380-z

A Method for Software Vulnerability Detection Based on Improved Control Flow Graph

□ **ZHOU Minmin, CHEN Jinfu†, LIU Yisong, ACKAH-ARTHUR Hilary, CHEN Shujie, ZHANG Qingchen, ZENG Zhifeng**

School of Computer Science and Communication Engineering, Jiangsu University, Zhenjiang 212013, Jiangsu, China

Abstract: With the rapid development of software technology, software vulnerability has become a major threat to computer security. The timely detection and repair of potential vulnerabilities in software, are of great significance in reducing system crashes and maintaining system security and integrity. This paper focuses on detecting three common types of vulnerabilities: *Unused_Variable*, *Use_of_Uninitialized_Variable*, and *Use_After_Free*. We propose a method for software vulnerability detection based on an improved control flow graph (ICFG) and several predicates of vulnerability properties for each type of vulnerability. We also define a set of grammar rules for analyzing and deriving the three mentioned types of vulnerabilities, and design three vulnerability detection algorithms to guide the process of vulnerability detection. In addition, we conduct cases studies of the three mentioned types of vulnerabilities with real vulnerability program segments from Common Weakness Enumeration (CWE). The results of the studies show that the proposed method can detect the vulnerability in the tested program segments. Finally, we conduct manual analysis and experiments on detecting the three types of vulnerability program segments (30 examples for each type) from CWE, to compare the vulnerability detection effectiveness of the proposed method with that of the existing detection tool CppCheck. The results show that the proposed method performs better. In summary, the method proposed in this paper has certain feasibility and effectiveness in detecting the three mentioned types of vulnerabilities, and it will also have guiding significance for the detection of other common vulnerabilities.

Key words: software security; software vulnerability; improved control flow graph; vulnerability detection algorithm

CLC number: TP 305

Received date: 2018-07-01

Foundation item: Supported by the National Natural Science Foundation of China (61202110 and 61502205) and the Project of Jiangsu Provincial Six Talent Peaks (XYDXXJS-016)

Biography: ZHOU Minmin, female, Ph. D. candidate, research direction: software analysis and trusted software. E-mail: minminzhou@stmail.ujs.edu.cn

† To whom correspondence should be addressed. E-mail: jinfuchen@ujs.edu.cn

0 Introduction

The security of computer and information systems fundamentally depends on the quality and security of their underlying software system. With the rapid development of software technologies, various types of network attacks and data leakage incidents emerge one after another. The fundamental cause is that software developers are unavoidably making errors in the process of developing applications. Therefore, software vulnerability formed in software can easily be exploited by external attackers. Software vulnerability refers to weaknesses, defects, and errors that are valuable to attackers in the software system. Once attackers exploit such vulnerabilities, they will pose a huge threat to the integrity and security of the information system, even with serious consequences that causing the enormous economic losses [1-4].

The existing approaches to detect software vulnerabilities are commonly divided into static and dynamic [5-8]. Dynamic approaches check the vulnerabilities in run-time by supplying test cases, and they usually cannot completely cover all the execution paths. The widely used dynamic program analysis ranges from simple fuzzy testing [9,10], advanced taint tracking to symbol execution [11,12], and other technologies to further improve the effectiveness of software vulnerability detection. Although these methods can find various defects, they are difficult to operate effectively in practice, and do not produce correct results due to the long running time or the exponential growth of the execution path. On the other hand, static approaches can make up for this deficiency because they consider all the possible execution paths, and even detect vulnerabilities without actually running the program. Many researches have focused on

this topic in the field of software security. Fagan [13] proposed a technique called *design code inspection* that helps software developers detect and fix errors at an early stage of development, which can reduce the costs of correcting software error. Viega *et al* [14] designed a static vulnerability scanning tool named *IST4*. For C and C++ source code, the tool can divide the program into small slices and compare each one with the database to detect vulnerabilities. It is characterized by high detection efficiency, and can be applied to detect vulnerabilities in large-scale projects. However, its defect is that only a few kinds of vulnerabilities are covered. Sands[15] proposed a vulnerability detection method based on theorem proof, in which they firstly converted the program into a logical formula by semantic analysis, and hence proved the validity of the logical formula using the established axioms and rules. Clarke *et al* [16] proposed a model detection technology as a formal verification method for software vulnerability detection. This technology checks whether a given program model meets certain predefined characteristics by traversing the state space, but the defect is that it might just be suitable for small-scale test programs.

In this paper, we propose a vulnerability detection method based on an improved control flow graph to detect three types of software vulnerabilities (*Unused_Variable*, *Use_of_Uninitialized_Variable*, and *Use_After_Free*). The method uses the static semantic analysis technique based on the improved control flow graph to analyze the self-contained information in the vulnerability source code, and hence detects the potential vulnerabilities in the program under test. This paper expands the data flow information into the classic control flow graph and creates a novel form of code icon representation—improved control flow graph (ICFG). In addition, we have formulated a set of vulnerability predicates and grammar rules based on ICFG, for software vulnerability analysis and detection. Finally, we have applied the proposed method to analyze real and specific vulnerability program segments obtained from a community-developed list of common software security weaknesses, known as Common Weakness Enumeration (CWE). Experimental results show that proposed method can play a guiding role in detecting the three types of software vulnerabilities.

The remainder of this paper is structured as follows. The related work is presented in Section 1. The overview of the proposed method framework is given in Section 2. The proposed approach and the methodology of the vulnerabilities detection algorithms are discussed in Section 3. The experimental analysis is detailed in Section 4, and the conclusion and future work are presented in Section 5.

1 Related Work

1.1 Control Flow Graph

Control flow graph (CFG) is a directed graph, and it is a graphical representation of the execution flow of each function in a program [17]. A CFG of a program can represent the control structure information of the function as well as the possible execution path of the program. A CFG corresponding to a program can generally be represented by $G_{\mathrm{CFG}} = (V, E, entry, exit)$, where V is a set of nodes representing the program statements, and E is a set of directed edges that represent the control flow relationships between nodes.

Due to the high availability of control flow graphs in static analysis of program code, the existing techniques for generating control flow graphs are constantly being refined. Rothermel[18] designed an algorithm to obtain the basic "block" from the program using boundary analysis techniques, but this algorithm cannot analyze the nesting between loop statements. Meanwhile, some researchers proposed an algorithm for generating control flow graph using path-sensitive approach, and a streamlined algorithm for generating control flow graph, both of which were devoted to optimizing the generation of control flow graph [19,20]. Gomes *et al* [21] proposed a technique for incremental, modular extraction of control flow graphs, which is suitable for model-checking of temporal control flow safety properties. These research contributions have verified the feasibility of generating control flow graphs, and the continuous development of static analysis technology makes it necessary for us to expand on the existing control flow graphs.

1.2 *Unused_Variable* Software Vulnerability

Unused_Variable is a common software vulnerability [22]. It occurs in situations where a large number of variables are defined during a program development process, but the program does not use all of these variables. The defined but never used variables, also known as *Unused_Variable*, become a dead store that will not be used in the program. This bad programming habit will lead to a common vulnerability model. The consequence is that *Unused_Variable* may increase resistance to software maintenance, or even cause more serious consequences such as memory leaks in the system when at-

tackers exploit those variables.

1.3 *Use_of_Uninitialized_Variable* Software Vulnerability

Use_of_Uninitialized_Variable is a common software vulnerability in C/C++ programming [23-25]. In computer programming, an uninitialized variable is a variable that is declared but not set to a defined value. During program execution, an uninitialized variable will generally have an unpredictable value. For example, after defining a variable in C/C++, if programmers do not assign a value to this variable before using it, it may cause program failure because the program will reference an uninitialized variable. Also, when the programmers released the memory space of the pointer after assigning a value to a dynamically allocated pointer, the value of the memory space will still exist while the link between the pointer and the memory space is disconnected. In this case, if the program uses this pointer variable directly, it may also lead to program failure with reference to the uninitialized variable. The uninitialized variable, once used in a calculation, can quickly propagate throughout the entire program and may affect the program in many ways. The results of each running of the program may be different, which may crash for no apparent reason, or may behave unpredictably. This vulnerability poses a serious threat to software security as it may result in incorrect results, memory violations, unpredictable behaviors, and program failure.

1.4 *Use_After_Free* Software Vulnerability

Referencing memory after it has been freed can cause a program to crash, use unexpected values, or execute code. *Use_After_Free* errors have two common and sometimes overlapping causes: Error conditions and other exceptional circumstances, and confusion over which part of the program is responsible for freeing the memory [26-28]. *Use_After_Free* vulnerabilities caused by the inadvertent use of dangling pointers are a major threat to systems security, which refers to pointers that point to freed memory, and lead to memory safety errors when accessed. A dangling pointer itself does not cause any memory safety problem, but accessing memory through a dangling pointer can lead to unsafe program behaviors and even security compromises, such as control-flow hijacking or information leakage [29]. By taking advantage of the *Use_After_Free* vulnerability in the web browsers or in the document viewers, attackers are able to execute arbitrary code in the context of applications and eventually control the computer or mobile phone remotely. *Use_After_Free* vulnerabilities are a major threat to systems security [30].

2 General Framework

Figure 1 shows the general framework of software vulnerability detection method based on ICFG. This framework provides guidance and specification for the software vulnerability detection based on ICFG. The descriptions of its main function module and process are as follows.

As shown in Fig. 1, we firstly propose the concept of an improved control flow graph by adding the program data flow information into the classical control flow graph. Hence, the subsequent analysis of vulnerability detections is defined based on ICFG. Then, we formulate some vulnerability predicates to describe the vulnerability features. Meanwhile, a set of vulnerability grammar rules are formulated to detect the three types of vulnerabilities (*Unused_Variable*, *Use_of_Uninitilized_Variable*, and *Use_After_Free*). After that, we combine the defined predicates and vulnerability grammar rules, with the ICFG of some particular vulnerability code; and we utilize this combination in our proposed vulnerability detection algorithms. In this paper, we have proposed three vulnerability detection algorithms based on ICFG. The next section provides details of the proposed ICFG, predicates of vulnerability property, vulnerability grammar rule sets and the vulnerability detection algorithms.

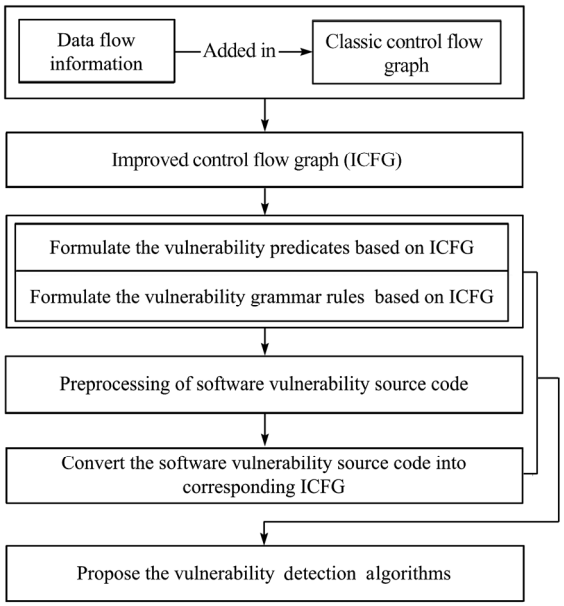

Fig. 1 The general framework of software vulnerability detection based on ICFG

3 The Proposed Approach

3.1 Improved Control Flow Graph

Since the traditional CFG introduced above can only reflect the characteristics of the control flow information contained in the program code, we propose an improved CFG, which also includes the data flow information of programs to compensate for such defect. This paper defines the ICFG as follows.

Definition 1 An improved control flow graph $G_{\mathrm{ICFG}}=(V, E, \lambda, \mu, Entry, Exit)$ is a directed, edge-labeled, and an attributed multi-graph. V is a set of nodes; $E \in V\times V$ is a set of directed edges; and λ is an edge labeling function that assigns a value from the set $l=\{\varepsilon, true, false\}$, which indicates the control flow information on the edge of graph. Properties can be assigned to nodes by the function $\mu=(k, s)$ where k is a set of property keys that assigns a value from the set $k=\{DEF, USE, FREE\}$, and s is the set of property values with $s_i=\{variable|$ *variable* is the variable in node $V_i\}$ (i refers to the number of a node in the graph).

The main feature of the improved control flow graph is that while keeping the original control flow information of the program, it also considers the data flow information in the program segment to make the ICFG contain more semantic information. Therefore, ICFG is more applicable to the static analysis of a program.

For each line in the code sample numbered 1 to 5, and for the code sample given in Fig. 2, hence an example of an improved control flow graph is shown in Fig. 3.

As shown in Fig. 3, there is no definition of a variable in the code statement corresponding to node V_1 in the example code. The variable x is defined at the node V_2 so that the property of the node V_2 is represented as $DEF(2)=[x]$. And the code statement at node V_3 uses variable x, so variable V_3 has the property $USE(3)=[x]$. The property values of the node V_4 are $DEF(4)=[y]$ and

```
1    void foo()
2    {   int x= SUM();
3    if (x < MAX)
4    {   int y=5*x;
5    sink(y);   }
     }
```

Fig. 2 Example code sample

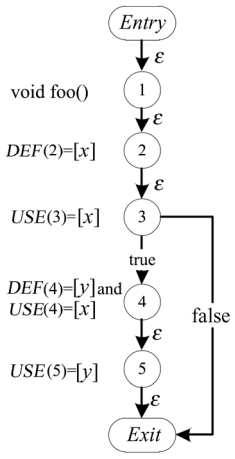

Fig. 3 ICFG representation of code sample in Fig. 2

$USE(4)=[x]$. Node V_5 has the property $USE(5)=[y]$. Meanwhile, we can draw the edge labeling information in the ICFG for the example code. In Fig. 3, $\lambda\langle Entry, V_1\rangle=\varepsilon$, $\lambda\langle V_1, V_2\rangle=\varepsilon$, $\lambda\langle V_2, V_3\rangle=\varepsilon$, $\lambda\langle V_4, V_5\rangle=\varepsilon$, and $\lambda\langle V_5, Exit\rangle=\varepsilon$, because there is no control information transmitted through these five edges, and $\lambda\langle V_3, V_4\rangle=true$, $\lambda\langle V_3, V_5\rangle=false$ for the "if (x < MAX)" statement existing in the code statement corresponding to node V_3.

3.2 The Predicates of Vulnerability Property

To illustrate the feasibility of the proposed vulnerability detection method based on ICFG, this section presents some basic definitions of the ICFG, as well as the related predicates for the three kinds of software vulnerabilities based on ICFG.

Definition 2 The properties of node V_i. Let $k(i)=[s_i]$ denote the property value of node V_i, where $k\subseteq\{DEF, USE, FREE\}$ and $s_i=\{variable|$ refers to the variable that is defined or used at node $V_i\}$.

Definition 3 *DEF~USE pairs*. Use *DEF~USE pair* to describe the definition and use of variables in the program. Let the predicate $DEF(i)\sim USE(j)=[var]$ ($i\leqslant j$ and i, j refer to the node number in the ICFG) represent a *DEF-USE pair* for the variable *var*.

Definition 4 Deep traversal $Traverse_b^a$. Let $Traverse_b^a$ represent a deep traversal from node a to node b, and this traversal returns all nodes at all reachable paths.

Definition 5 Filtering traversal $Filter_p(V)=\{v \in V: p(v)\}$. This traversal returns the set of all nodes that satisfy the condition p, where V refers to the node set of all nodes of the ICFG.

Definition 6 Forward traversal (*FT*)

$FT_l^{k,s}(V)=U_{v\in V}\{u:(v, u)\in E$ and $\lambda(v, u)=l$ and $\mu((v,$

$u), k)=s\}$. Let $FT_l^{k,s}(V)$ represent an ICFG-based forward traversal which returns all nodes reachable over edges with label l and property k, s, where V refers to the node set of all nodes in the ICFG.

Definition 7 Backward traversal (BT)

$BT_l^{k,s}(V)=U_{u\in V}\{v:(v,\ u)\in E$ and $\lambda(v,u)=l$ and $\mu((v,u),k)=s\}$. Let $BT_l^{k,s}(V)$ represent an ICFG-based backward traversal which returns all nodes reachable over edges with label l and property k, s, where V refers to the node set of all nodes in the ICFG.

Definition 8 Counting function $Calculate(\{V\})$. Let the function $Calculate(\{V\})$ count and return the number of elements in the node set $\{V\}$.

Definition 9 Connection operator between multiple operations $\diamondsuit$. The Connection operator between multiple operations regulates the execution from the back to the front. For $x\diamondsuit y$, y is executed firstly, before x.

Definition 10 $FindDefvariable(V_i)$. The function $FindDefvariable(V_i)$ obtains and returns all variables defined at the node V_i.

Definition 11 $FindUsedvariable(V_i)$. The function $FindUsedvariable(V_i)$ obtains and returns all variables used at the node V_i.

Definition 12 $FindFreevariable(V_i)$. The function $FindFreevariable(V_i)$ obtains and returns all variables released at the node V_i.

Definition 13 $FindAllvariable(V)$. The function $FindAllvariable(V)$ obtains and returns all variables in the program.

Definition 14 $Initialize(variable)$. Use $Initialize(variable)$ to check whether the variable *var* has been initialized. The operation returns 1, if *var* has been initialized; otherwise, it will return 0.

3.3 Vulnerability Grammar Rule Sets

The method proposed in this section focuses on detecting three types of vulnerabilities: *Unused_Variable*, *Use_of_Uninitialized_Variable*, and *Use_After_Free*. According to the causes and features of each type of vulnerability, this paper develops a set of vulnerability grammar rules to analyze the three types of vulnerabilities. These vulnerability grammar rules describe and extract the vulnerability features of a program in order to derive and locate the vulnerability existing in the program segment.

1) *Unused_Variable*

The feature of *Unused_Variable* is that some of the defined variables in program segment have never been used or referenced. That is to say, there are no *DEF-USE pairs* for the unused variables in the program segment. The key to detecting *Unused_Variable* is to check whether the *DEF-USE pairs* of each variable is complete. In this paper, we propose a grammar rule for deriving and detecting the *Unused_Variable* based on ICFG. The description of the process is as follows.

Step 1: Use $FindDefvariable(V_i)$ to obtain and return all variables defined at the node V_i, then defined as the collection $\{variable\}$;

Step 2: Execute FT to traverse each element in the collection $\{variable\}$, using the traversal formula $FT_l^{USE,\ variable(i)}(V)=U_{v\in V}\{u:(v,\ u)\in E$ and $\lambda(v,u)=l$ and $\mu(v,USE)=variable\}$. For each element in $\{variable\}$, FT obtains and returns the all nodes in the ICFG that uses the variable, and store them in the collection $\{V_{USE}\}$.

Step 3: For each $\{V_{USE}\}$ obtained in Step 2, determine whether the collection $\{V_{USE}\}$ is empty by executing $Calculate\ \{V_{USE}\}$. If for a $\{V_{USE}\}$, $Calculate(\{V_{USE}\})$ returns 0, it means there exists some defined but never used variables which is an indication of *Unused_Variable* vulnerability; otherwise, it means no existence of *Unused_Variable* in the tested program segment.

Summarizing the three steps above, we can construct the grammar rules for *Unused_Variable* detection as: $Calculate\{FT_l^{USE,variable(i)}(V)\diamondsuit FindDefvariable(V_i)\}$.

2) *Use_of_Uninitialized_Variable*

The vulnerability *Use_of_Uninitialized_Variable* is characterized by the use of variables that have not been defined or initialized before. That is to say, the *DEF-USE pairs* of some variables in the program segment are incomplete or the used variables are declared but are not set to defined values. The key to detecting *Use_of_ Uninitialized_Variable* is to verify that the variables in the program segment have been defined or initialized before they are used. Combining the features of *Use_of_ Uninitialized_Variable* and related predicates, we propose the grammar rules for *Use_of_Uninitialized_ Variable* detection based on ICFG. The derivation process is as follows.

Step 1: Use $FindUsedvariable(V_i)$ to obtain and return all variables used at the node V_i, then defined as the collection $\{variable\}$;

Step 2: Execute BT to traverse each element in the collection $\{variable\}$, using the traversal formula: $BT_l^{DEF,variable(i)}(V)=U_{v\in V}\{u:(v,\ u)\in E$ and $\lambda(v,u)=l$ and $\mu(v,DEF)=variable\}$. For each element in $\{variable\}$, BT obtains and returns all nodes in the ICFG that defines that variable, and store them in the collection $\{V_{DEF}\}$.

Step 3: For each $\{V_{DEF}\}$ obtained in Step 2, deter-

mine whether the collections are empty, by executing *Calculate* ($\{V_{USE}\}$). If for a $\{V_{DEF}\}$, *Calculate* ($\{V_{DEF}\}$) returns 0, it means there exists some used but never defined variables in the tested program segment, indicating the presence of *Use_of_Uninitialized_Variable* vulnerability; otherwise, turn to Step 4.

Step 4: Check further whether the used variables are already initialized. Execute $Initialize(\{variable\})$, if all collection elements are executed and the return of this formula contains 0 which means some variable in the program segment is being used without initialization, and therefore we can detect the *Use_of_Uninitialized_ Variable* in the tested program segment; otherwise, the tested program segment does not include *Use_of_Uninitialized_Variable*.

Summarizing the four steps above, we can construct the grammar rules for *Use_of_Uninitialized_Variable* detection as:

$$Calculate\{BT_l^{DEF,variable(i)}(V)\ FindUsedvariable(V_i)\}$$
$$\begin{cases} = 0, \text{detect the vulnerability} \\ \neq 0 \rightarrow Initialize(\{FindUsedvariable(V_i)\}) = \\ \begin{cases} \text{true, no existence of vulnerability} \\ \text{false, the vulnerability is detected} \end{cases} \end{cases}$$

3) *Use_After_Free*

The value of the node property key k is assigned from the set k= {*DEF*, *USE*, *FREE*}, in which the element *FREE* represents the release of some variable. *Use_After_Free* is characterized by the fact that the program reuses a resource or variable that has already been released. Therefore, the key to detecting *Use_After_Free* is to identify the nodes of the ICFG that have released variables, then to track and confirm whether those variables are being used on subsequent paths. Combining the features of *Use_After_Free* vulnerability and related predicates, we propose a detection method based on ICFG to analyze and locate *Use_After_Free*. The analysis process is detailed as follows.

Step 1: Use $FindFreevariable(V_i)$ to obtain and return all freed variables at the node V_i, then define them as the collection *{variable}*. In addition, record the nodes in which the variables have been released, and denote them as V_{dst};

Step 2: Execute $Traverse_{V_{end}}^{V_{src}}$, where V_{src} represents the node that firstly initializes the *variable*, and V_{end} represents the last node to use the *variable*. This traversal returns all nodes on the path of the *variable* from its initialization to its last use, and then denoted the set as $\{V\}$.

Step 3: Execute filtering traversal $Filter_p(V)$= $\{v \in V : V_{dst} \in \{V\}$, where the node V_{dst} represents the node that has freed variables, and p refers to the condition that node V_{dst} is included in the collection $\{V\}$. That is, condition p is satisfied if $V_{dst} \in \{V\}$ is true. This traversal returns all nodes that satisfy the condition p and define it as a collection $\{V_p\}$.

Step 4: Determine whether the collection $\{V_p\}$ obtained in Step 3 is empty. Execute $Calculate(\{V_p\})$. If its return is not always 0, it means there exist some free variables that are still being used in the program segment, so we can derive that the tested program segment contains *Use_After_Free*; otherwise, it means no existence of *Use_After_Free* in the tested program segment.

Summarizing the four steps above, we can formulate the vulnerability grammar rules for *Use_After_Free* detection as: $Filter_p(V) \diamond Traverse_{V_{end}}^{V_{src}}$.

3.4 The Detection Algorithms for the Software Vulnerabilities

This section gives three detailed detection algorithms (Algorithm 1- Algorithm 3) for detecting the three types of vulnerabilities. We discuss the flow of the algorithm in the detection process, from the input of tested program segment to the output of vulnerability detection results.

1) *Unused_Variable* detection algorithm

In order to detect the *Unused_Variable* vulnerability, we propose an algorithm based on ICFG, which applies the proposed detection process, as shown in Algorithm 1. Algorithm 1 has one input, that is, the program segment to be tested. In Algorithm 1, we create three sets: V, V_{use}, and *variable_D* to store all nodes of the ICFG, the nodes that used variables on the ICFG, and all the variables defined in the program segment, respectively. By converting the program segment into its corresponding ICFG, all nodes in the ICFG can be obtained. The algorithm further determines whether all variables existing in these nodes are defined before they are used. Then the *DEF~USE pair* for each variable determine whether the program segment under test contains *Unused_Variable* vulnerability. After that, the algorithm will output the detection results for which *true* means the detection of *Unused_Variable* vulnerability and *false* means no vulnerability detected.

2) *Use_of_Uninitialized_Variable* detection algorithm

In order to detect the *Use_of_Uninitialized_Variable* vulnerability, we proposed an algorithm shown as Algorithm 2. The algorithm has one input: the program segment to be tested, and three sets V, V_{DEF}, *variable_U* are constructed to store all nodes of the ICFG, all nodes that define variables on the ICFG, and all variables that

used in the program segment, respectively. In addition, a *flag* is used to indicate the result of checking initialization of each variable. The algorithm firstly converts the program segment to be tested into an ICFG. It then obtains all nodes of the ICFG and all variables that exist in the program segment. Then it checks whether the *DEF~USE pair* for each variable is complete. If the *DEF~USE pairs* of variables are not complete, the algorithm will output a detection result, indicating the detection of the *Use_of_Uninitialized_Variable* vulnerability. Otherwise, it further checks the initialization of each variable. If there exists a variable that is not initialized, the algorithm will output *true*, which means *Use_of_Uninitialized_Variable* has been detected.

Algorithm 1 *Unused_Variable* detection algorithm

Input: the program segment to be tested

Output: the detection results (true or false)

1. Construct V ={} to store all the nodes on ICFG;
2. Construct V_{USE}={} to store the nodes using variables;
3. Construct *variable_D*={} to store the defined variables in the program segment;
4. Convert the program segment to be tested into ICFG;
5. *V*[*i*]= *Deep traversal* (ICFG); // To obtain all the nodes in ICFG.
6. For each node V_i in V do
7. v*ariable_D*[*i*]= *FindDefvariable*(V_i);
8. End For
9. For each variable in *variable_D* do
10. V_{USE} = $FT_l^{USE,variable\_D(i)}(V)$;
11. If (*Calculate*(V_{USE}) == 0) Then
12. Return true; // Detect the *Unused_Variable* vulnerability.
13. Else
14. Return false; //No *Unused_Variable* detected.
15. End If
16. End For

Algorithm 2 *Use_of_Uninitialized_Variable* detection algorithm

Input: the program segment to be tested

Output: the detection results (true or false)

1. Construct V ={} to store all the nodes on ICFG;
2. Construct V_{DEF}={} to store the nodes defining variables;
3. Construct *variable_U*={} to store the used variables in the program segment;
4. Construct *flag*: the variable initialization tag (0: Uninitialized; 1: Initialized)
5. Convert the program segment to be tested into ICFG;
6. *V*[*i*]= *Deep traversal* (ICFG); // To obtain all the nodes in ICFG.
7. For each node V_i in V do
8. v*ariable_U*[*i*]= *FindUsedvariable*(V_i);
9. End For
10. For each variable in *variable_U* do
11. V_{DEF} = $BT_l^{DEF,variable\_U(i)}(V)$;
12. If (*Calculate*(V_{DEF}) == 0) Then
13. Return true; // Detect the *Use_of_Uninitialized_Variable* vulnerability.
14. Else
15. For each variable in *variable_U* do
16. *flag* = *Initialize*(*variable*);
17. If (*flag* == 0) Then
18. Return true; // Detect the *Use_of_Uninitialized_Variable* vulnerability.
19. Else
20. Return false; // No *Use_of_Uninitialized_Variable* detected.

```
            End If
          End For
        End If
      End For
```

3) *Use_After_Free* detection algorithm

We propose Algorithm 3 to detect *Use_After_Free* vulnerability. The input of this algorithm is the program segment to be tested. The algorithm firstly generates the corresponding ICFG according to the program code segment to be tested, and it obtains all nodes of the ICFG. The set V_{dst} stores nodes with released variables, and the algorithm stores the released variables in the set *variable_F*. It further checks whether each variable in the set *variable_F* has been released in their path, from the initial definition node to the node in which they were last used. If a variable satisfying this condition is found, it means *Use_After_Free* vulnerability has been detected in the tested program segment.

Algorithm 3 *Use_After_Free* detection algorithm

Input: the program segment to be tested

Output: the detection results (true or false)

1. Construct V ={} to store all the nodes on ICFG;
2. Construct *variable_F*={} to store the freed variables in the program segment;
3. Construct V_{src}={} to store the node in which the variable firstly been defined;
4. Construct V_{end}={} to store the node in which the variable last been used;
5. Construct V_{dst}={} to store the nodes releasing variable;
6. Construct V_{path}={} to store the nodes existing in the path from V_{src} to V_{end};
7. Construct V_p={} to store the nodes that satisfy the condition p;
8. Convert the program segment to be tested into ICFG;
9. $V[i]$= *Deep traversal* (ICFG); // To obtain all the nodes in ICFG.
10. For each node V_i in V do
11. v*ariable_F*[i]= *FindFreevariable*(V_i), V_{dst}= $V[i]$;
12. End For
13. For each variable in *variable_F* do
14. V_{src} = $FT_l^{DEF,variable\_F(i)}(V)$; //To obtain the node in which the variable been defined.
15. V_{end}= $\{BT_l^{USE,variable\_F(i)}(V)\}_{\max}$; //To obtain the node in which the variable been last used.
16. $V_{path}[i]$ = $Traverse_{V_{end}}^{V_{src}}$;
17. V_p= *Filter*$_p$(V_{path}), where p refers to V_{dst} is included in the path from V_{src} to V_{end}
18. If (*Calculate*(V_p) == 0) Then
19. Return true; // Detect the *Use_After_Free* vulnerability.
20. Else
21. Return false; // No *Use_After_Free* detected.
22. End If
23. End For

4 Experimental Analysis

4.1 General Detection Process Based on ICFG

Figure 4 shows the process of the proposed method in vulnerability detection. The detection process begins with the generation of the corresponding ICFG from the vulnerability source code, and then to analyze all nodes in the graph. It further combines the result of predicates analysis with the vulnerability grammar rule set to detect the vulnerabilities in the tested program segment.

4.2 Cases Studies

In this section, we apply the proposed method to analyze and detect the three mentioned types of vulner

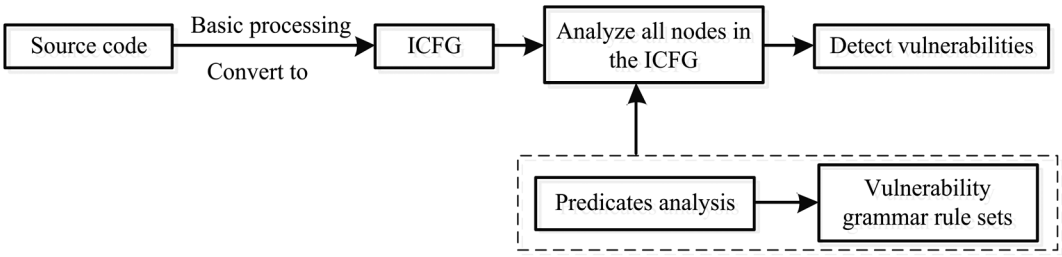

Fig. 4 Vulnerability detection process of the proposed method

abilities using program segments from CWE. The tested program segments are from *CWE*563 *Unused_variable*, *CWE*457 *Use_of_Uninitialized_variable*, and *CWE*416 *Use_After_Free*. The program segments and their corresponding ICFGs are shown in Figs.5-10. The program segments are shown in Fig. 5, Fig. 7, and Fig. 9, and their corresponding ICFGs are given in Fig. 6, Fig. 8, and Fig. 10, respectively. The purpose of cases studies is to verify the validity of the proposed vulnerability detection method in detecting the three mentioned vulnerabilities. The results obtained from cases studies show that the proposed method is feasible and effective for the detection of the three kinds of vulnerabilities. The detailed process of cases studies is as follows.

1) The case study of *Unused_Variable*

```
1   void
    CWE563_Unused_Variable__unused_uninit_variable_
    char_09_bad()
2   {
        char data;
        data = 'C';
3       if(GLOBAL_CONST_TRUE)
        {
            }
    }
```

Fig. 5 The vulnerability program segment of *Unused_Variable*

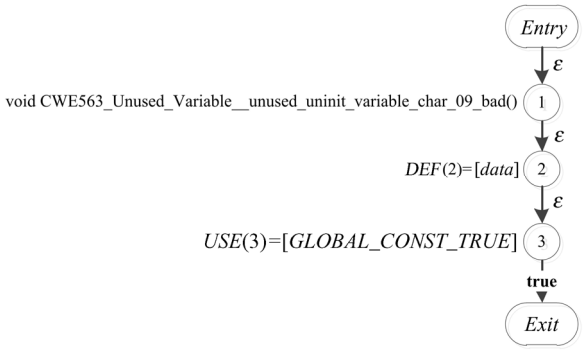

Fig. 6 ICFG for the program segment in Fig. 5

First, we pre-processed the program segment shown in Fig. 5 by labeling its valid code lines with numbers from 1 to 3. We then applied the grammar rule sets of *Unused_variable* to the ICFG presented in Fig. 6; The description of the detection process of *Unused_variable* is as follows.

Step 1: Execute $FindDefvariable(V_i)$ where i= 1, 2, 3, and it returns all the defined variables in the program segment: {*data*}.

Step 2: Execute $FT_l^{USE,data}(V)$ to traverse the collection {*data*}. After that, *FT* returns the use node collection { V_{USE} } for which the variable *data* is $\varnothing$.

Step 3: Determine whether the collection $\{V_{USE}\}$ for each variable obtained in Step 2 is empty. Execute $Calculate(\{V_{USE}\}) = Calculate(\varnothing) = 0$.

After the process of static detection, the proposed method with vulnerability grammar rules can detect *Unused_variable* in the program segment presented in Fig. 5.

2) The case study of *Use_of_Uninitialized_variable*

```
1       void
    CWE457_Use_of_Uninitialized_Variable_double
        _array_declare_no_init_01_bad
2       {
          double * data;
          double dataUninitArray[10];
3       data = dataUninitArray;
4       {
        int i;
5       for(i=0; i<10; i++)
          {
6                 printDoubleLine(data[i]);
              }
          }
        }
```

Fig. 7 The vulnerability program segment of *Use_of_ Uninitialized_variable*

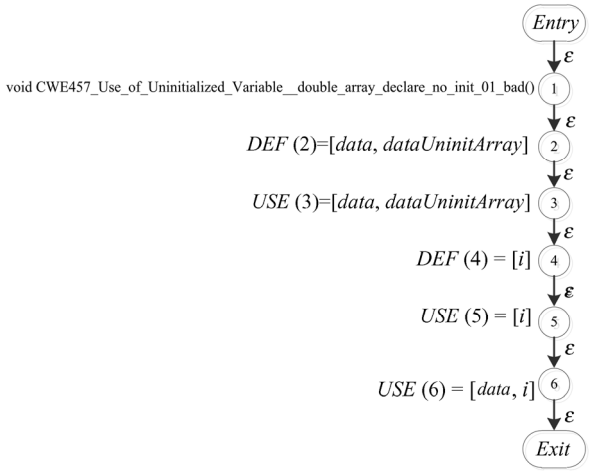

Fig. 8 ICFG for the program segment in Fig. 7

First, we pre-processed the program segment shown in Fig. 7 by labeling its valid code lines with numbers from 1 to 6, and then we applied the grammar rule sets of *Use_of_Uninitialized_variable* to the ICFG presented in Fig. 8; the description of the static detection process of *Use_of_Uninitialized_variable* is as follows.

Step 1: Execute *FindUsedvariable*(V_i) where i= 1, 2, 3, 4, 5, 6, and it returns all the used variables in the program segment: {*data*, *dataUninitArray*, *i*}.

Step 2: Execute $BT_l^{DEF,\{data,dataUninitArray,i\}}(V)$ to obtain the results of the variable definition node collection $\{V_{DEF}\}=\{V_2, V_4\}$ for the three used variables.

Step 3: Determine whether all collections $\{V_{DEF}\}$ obtained in Step 2 are empty. Execute *Calculate* $\left(\{V_{DEF}\}\right)$=*Calculate*($\{V_2, V_4\}$)=2 ≠ 0. Therefore, the analysis should turn to Step 4.

Step 4: Further check whether the used variables are initialized. Execute *Initialize*({*variable*}), in which *variable*= *data*, *dataUninitArray*, *i*. When *variable*= *data*, the *Initialize*({*data*}) returns 0, which means the *Use_of_Uninitialized_variable* is detected.

3) The case study of *Use_After_Free*

For the case of the *Use_After_Free* vulnerability, we pre-processed the program segment shown in Fig. 9 by labeling its valid code lines with numbers from 1 to 9. We subsequently applied the grammar rule sets of *Use_After_Free* to the ICFG presented in Fig. 10; the description of the static detection process of *Use_After_Free* is as follows.

Step 1: Execute *FindFreevariable*(V_i), where i=1, 2, 3,…, 9, and it returns all the freed variables in the program segment: {*data*} as well as the node V_{dst}= {V_7} where the variable *data* has been released.

Step 2: Execute $Traverse_{V_9}^{V_2}$, and it returns all nodes on the path of the *data* from its initialization to its last use: $\{V\}=\{V_2, V_4, V_5, V_6, V_7, V_9\}$.

Step 3: Execute $Filter_p(V)=\{v \in V : V_{dst} \in \{V\}\}$, in which the node $V_{dst}= V_7$, and the condition p refers to the node V_7 is included in $\{V_2, V_4, V_5, V_6, V_7, V_9\}$. After that, we obtain $\{V_p\}=\{V_7\}$.

Step 4: Determine whether the collection $\{V_p\}$ obtained in Step 3 is empty. Execute $Calculate\left(\{V_p\}\right)=Calculate\left(\{V_7\}\right)=1$. This result indicates the detection of *Use_After_Free* in the tested program segment.

```
1    void
     CWE416_Use_After_Free__malloc_free_char_03_bad()
     {
2    char * data;
     data = NULL;
3    if(5==5)
          {
4    data = (char *)malloc(100*sizeof(char));
5    memset(data, 'A', 100-1);
6         data[100-1] = '\0';
7    free(data);
          }
8    if(5==5)
          {
9    printLine(data);
               }
     }
```

Fig. 9 The vulnerability program segment of *Use_After_Free*

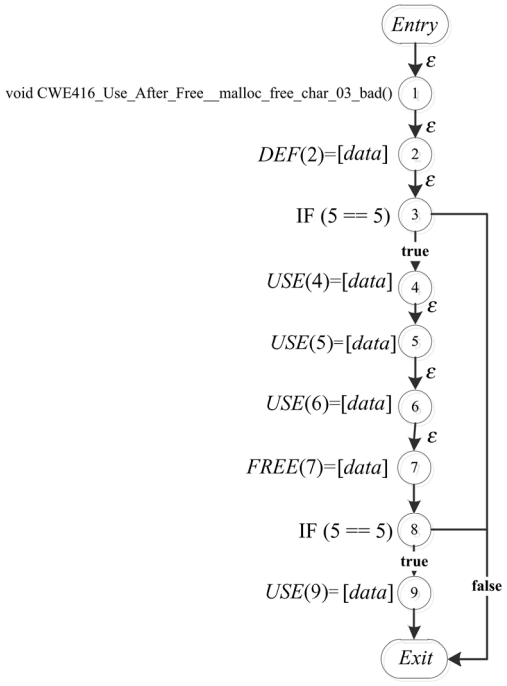

Fig. 10 ICFG for the program segment in Fig. 9

4.3 Experimental Analysis

To verify the effectiveness of the proposed method in detecting the three kinds of vulnerabilities (*Unused_Variable*, *Use_of_Uninitialized_Variable,* and *Use_After_Free*), we conducted experiments based on the three types of vulnerabilities and using program segments from CWE. For each type of vulnerability, we randomly selected 30 program segments to test. Then we applied the method proposed in this paper to manually detect the vulnerabilities in the program segments under test, and compared the results with those of the existing vulnerability detection tool CppCheck. Table 1 presents the detection results of these two detection methods for the three types of vulnerabilities (Vulnerabilities: Vuls., CWE563: *Unused_Variable*, CWE457: *Use_of_ Uninitialized_Variable* and CWE416: *Use_After_Free*).

The results from Table 1 show that, the method proposed in this paper can detect more vulnerabilities than CppCheck, after executing the same number of vulnerability program segments for each type of vulnerability. Compared with CppCheck, the method proposed in this paper achieved improvements of 40%, 43.3%, and 40% in detecting *Unused_Variable*, *Use_of_Uninitialized_Variable*, and *Use_After_Free*, respectively. The experimental results show that the proposed method has strong applicability and effectiveness in the detection of the three mentioned types of vulnerabilities, and it can have guiding significance in the analysis and detection of other types of vulnerabilities.

Table 1 The detection results of the two methods

Name of Vuls.	Total number of Vuls.	The proposed method		CppCheck	
		Number of the detected Vuls.	Detection rate /%	Number of the detected Vuls.	Detection rate /%
CWE563	30	14	46.7	2	6.7
CWE457	30	19	63.3	6	20.0
CWE416	30	21	70.0	9	30.0

5 Conclusion

In this paper, we propose a method for software vulnerability detection based on an improved control flow graph to detect three common vulnerabilities: *Unused_Variable*, *Use_of_Uninitialized_Variable*, and *Use_After_Free*. We firstly proposed the definition of the improved control flow graph, and several predicates of vulnerability properties. For each type of vulnerability, we constructed a corresponding vulnerability grammar rule sets based on ICFG for the software vulnerability detection. In addition, we designed three vulnerability detection algorithms, which can detect specific type of vulnerability in the program segment to be tested. The results of the cases studied in this paper showed that the proposed method is feasible and effective in analyzing and detecting the three mentioned types of vulnerability existing in the tested program segments. Finally, the experimental analysis section compared the vulnerability detection effectiveness of the proposed method with that of an existing vulnerability detection tool CppCheck, by using the vulnerability program segments selected from CWE. The results showed that the proposed vulnerability detection method has a better performance than CppCheck, in detecting the three types of vulnerabilities. The results illustrate that the software vulnerability detection method based on ICFG plays a certain guiding role in the common vulnerability detection field.

Future work of this research mainly includes reducing the generation cost of ICFG, improving the detection accuracy of the method, and further extending the universality of the proposed method.

References

[1] Liu B, Shi L, Cai Z, *et al*. Software vulnerability discovery techniques: A survey[C]// *Proc* 4*th International Conference on Multimedia Information Networking and Security*. Piscataway: IEEE, 2012: 152-156.

[2] Liu P, Su J, Yang X. Research on software security vulnerability detection technology[C]// *Proc* 2*nd International Conference on Computer Science and Network Technology*. Piscataway: IEEE, 2012, **3**: 1873-1876.

[3] Kumar M, Sharma A. An integrated framework for software vulnerability detection, analysis and mitigation: An autonomic system[J]. *Sādhanā*, 2017, **42**(9): 1481-1493.

[4] Rahimi S, Zargham M. Vulnerability scrying method for software vulnerability discovery prediction without a vulnerability database[J]. *IEEE Transactions on Reliability*, 2013, **62**(2): 395-407.

[5] Kim S, Kim R Y C, Park Y B. Software vulnerability detection methodology combined with static and dynamic analysis[J]. *Wireless Personal Communications*, 2016, **89**(3): 777-793.

[6] Chernis B, Verma R. Machine learning methods for software vulnerability detection[C]// *Proc* 4*th ACM International Workshop on Security and Privacy Analytics*. New York: ACM, 2018: 31-39.

[7] Shuai B, Li M, Li H, *et al*. Software vulnerability detection using genetic algorithm and dynamic taint analysis[C]// *Proc* 4*th International Conference on Consumer Electronics*, *Communications and Networks*. Piscataway: IEEE, 2014: 589-593.

[8] Huang C C, Lin F Y, Lin Y S, *et al*. A novel approach to evaluate software vulnerability prioritization[J]. *Journal of Systems & Software*, 2013, **86**(11): 2822-2840.

[9] Kapur P K, Yadavali V S S, Shrivastava A K. A comparative study of vulnerability discovery modeling and software reliability growth modeling[C]// *Proc* 1*st International Conference on Futuristic Trends on Computational Analysis and Knowledge Management.* Piscataway: IEEE, 2015: 246-251.

[10] Bekrar S, Bekrar C, Groz R, *et al*. Finding software vulnerabilities by smart fuzzing[C]// *Proc* 4*th IEEE Fourth International Conference on Software Testing*, *Verification and Validation*. Piscataway: IEEE, 2011: 427-430.

[11] Woo M, Sang K C, Gottlieb S, *et al*. Scheduling black-box mutational fuzzing[C]// *Proc 20th ACM Sigsac Conference on Computer & Communications Security*. New York: ACM, 2013: 511-522.

[12] Avgerinos T, Sang K C, Rebert A, *et al*. Automatic exploit generation[J]. *Communications of the ACM*, 2014, **57**(2): 74- 84.

[13] Fagan M E. Design and code inspections to reduce errors in program development[J]. *IBM Systems Journal*, 2001: **15**(3):182-211.

[14] Viega J, Bloch J T, Kohno Y, *et al*. ITS4: A static vulnerability scanner for C and C++ code[C]// *Proc* 16*th Computer Security Applications*. Piscataway: IEEE, 2000: 257-267.

[15] Sands D. A theorem proving approach to analysis of secure information flow[C]// *Proc* 2*nd International Conference on Security in Pervasive Computing*. Berlin: Springer-Verlag, 2005, **3450**(10): 193-209.

[16] Clarke E M, Grumberg O, Peled D A. Model checking [C]// *Proc 17th International Conference on Foundations of Software Technology & Theoretical Computer Science*. Berlin: Springer-Verlag, 1997, **1346**: 54-56.

[17] Nguyen M H, Nguyen T B, Quan T T, *et al*. A hybrid approach for control flow graph construction from binary code[C]// *Proc* 20*th Asia-Pacofic Software Engineering Conference*. Piscataway: IEEE, 2013, **2**: 159-164.

[18] Rothermel G. Representation and analysis of software[J]. *Angewandte Chemie*, 2005, **46**(31): 5896-900.

[19] Gold R. Control flow graphs and code coverage[J]. *Versita*, 2010, **20**(4): 739-749.

[20] Sun X, Zhongyang Y B, Xin Z, *et al*. Detecting code reuse in Android applications using component-based control flow graph[J]. *IFIP Advances in Information & Communication Technology*, 2016, **428**: 142-155.

[21] Gomes P D C, Picoco A, Gurov D. Sound control flow graph extraction from incomplete Java bytecode programs[C]// *Proc* 17*th International Conference on Fundamental Approaches to Software Engineering*. Berlin: Springer-Verlag, 2014, **8411**: 215-229.

[22] Sasaki S, Tanabe K, Fujita M. Using SpecC program slicing to detect uninitialized variables and unused variables[J]. *Technical Report of Ieice Vld*, 2005, **104**(708): 59-64.

[23] Lin W, Liu J, Wang Q, *et al*. Method of uninitialized variable detecting for C++ Program[J]. *International Journal of Education & Management Engineering*, 2011, **1**(1): 63-67.

[24] Jana A, Naik R. Precise detection of uninitialized variables using dynamic analysis—Extending to aggregate and vector types[C]// *Proc* 19*th Working Conference on Reverse Engineering*. Piscataway: IEEE, 2012: 197-201.

[25] Xu W, Li J, Shu J, *et al*. From collision to exploitation: Unleashing Use-After-Free vulnerabilities in Linux kernel[C]// *Proc* 22*nd ACM Conference on Computer and Communications Security*. New York: ACM, 2015:414-425.

[26] Yan H, Sui Y, Chen S, *et al*. Machine-Learning-Guided typestate analysis for static Use-After-Free detection[C]// *Proc* 33*rd Computer Security Applications Conference*. New York: ACM, 2017: 42-54.

[27] Feist J, Mounier L, Potet M L. Statically detecting use after free on binary code[J]. *Journal of Computer Virology & Hacking Techniques*, 2014, **10**(3): 211-217.

[28] Kouwe E V D, Nigade V, Giuffrida C. DangSan: Scalable Use-after-free detection[C]// *Proc* 12*th European Conference on Computer Systems*. New York: ACM, 2017: 405- 419.

[29] Caballero J, Grieco G, Marron M, *et al*. Undangle: Early detection of dangling pointers in use-after-free and double-free vulnerabilities[C]// *Proc International Symposium on Software Testing and Analysis*. New York: ACM, 2012: 133-143.

[30] Han X, Wei S, Ye J Y, *et al*. Detect use-after-free vulnerabilities in binaries[J]. *Journal of Tsinghua University*, 2017, **57**(10): 1022-1029(Ch).

□