An Optimized Static Propositional Function Model to Detect Software Vulnerability

LANSHENG HAN1, (Member, IEEE), MAN ZHOU1, YEKUI QIAN2, CAI FU1, AND DEQING ZOU1, (Member, IEEE)

1School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China (e-mail: hanlansheng@hust.edu.cn)
24 Jianshedong Road, Zhengzhou, Henan, 450002, China

Corresponding author: Man Zhou (e-mail: zhou_man1125@hust.edu.cn), Yekui Qian (e-mail: qyk1129@hotmail.com)

This work is supported by the National Natural Science Foundation of China under Grant (No. 61272033, 61272045, 61572222); and the National Grand Fundamental Research 973 Program Foundation of China (No. 2014CB340600).

ABSTRACT Due to the lack of appropriate theory to accurately characterize vulnerabilities, the current static detection technologies have two key challenges, i.e., limited applicability, and the problem of state space explosion. In this paper, we put forward a static detection model based on the proposition function. Furthermore, a new program intermediate representation called Vulnerability Executable Path Set (VEPS) is proposed to optimize our model which compresses the program state space distinctly. In addition, in order to confirm the reliability of the static detection model, we conduct three terms of contrast experiments to estimate the results with the vulnerability disclosed by NIST. The results obtained from extensive experiments show that the proposed model effectively detects more Wireshark bugs than NIST, and reveals a higher detection efficiency than FindBugs.

INDEX TERMS Execution Path Optimization, Propositional Function, Static Analysis, Software Vulnerability

I. INTRODUCTION

WITH the development of information technology, the number and scale of softwares are constantly expanding [1]. As a matter of fact, the number of serious new software vulnerabilities is still rising. The Common Vulnerabilities and Exposures (CVE) database reports that there were around 1,000 serious CVEs in 2000, but in 2015 more than 10,000 serious CVEs were registered. The rate of vulnerability growth seems to be increasing [2], [3]. A software vulnerability is a defect in the logical design of the software or an error generated during the writing. This defect or error can be exploited by cyber criminals to steal sensitive or classified information, even obtain system privileges, and then perform various illegal operations [4], [5].

Depending on whether the software programs need to run in the detection process, vulnerability detection technology can be divided into static detection, dynamic detection. From the perspective of program modeling, static detection consists of two categories: static analysis with data modeling and graph-based static analysis [6]. A popular data modeling method is patch matching, which refers to the similarity comparison between the original program and the program with patching to get the difference part [7]. In [8], Z. Xu et al. proposed a scalable binary-level patch analysis framework, named SPAIN, to automatically identify security patches and summarize patch patterns and their corresponding vulnerability patterns. The most significant advantage of data modeling methods for the static detection is that they can achieve a high code coverage of the program, but they can not avoid a challenge of high false positive rate. At the moment, graph-based static detection method makes up for this shortcoming with comprehensive analysis. It has been demonstrated that the static detection method based on graph structure can effectively detect the unknown software vulnerabilities [9]. Graph-based static detection method regards modeling program properties as graphs such as control-flow graphs (CFG), data-flow graphs (DFG) and program-dependence graphs (PDG), etc [10]. In [11], a novel representation of source code called a code property graph was introduced to elegantly model templates for common vulnerabilities with graph traversals. It has been demonstrated its efficacy by identifying 18 previously unknown vulnerabilities in the...
source code of the Linux kernel.

Dynamic detection technology, such as "fuzzers", can acquire specific operating information of the software. Fuzz testing is an automated or semi-automated testing technique which is widely used to discover defects which could not be identified by traditional functional testing methods [12]. In [13], Directed Greybox Fuzzing (DGF) was introduced which generated inputs with the objective of reaching a given set of target program locations efficiently. However, when the dynamic analysis tools interact with the external code, they will face a serious challenge of path explosion. In addition, most of existing dynamic analysis systems can only dig shallow bugs in software. Flaws that lie in deeper logic of an application are more likely be missed by these approaches, and are usually discovered through manual analysis by human experts [14], [15].

Current research considers that the vulnerability is a defect in hardware, software or usage strategy [16]. Recurring events always contain some specific rules, called patterns. In other words, software defects and flaws follow some certain patterns and rules. For example, the buffer overflow in C Programming Language, if the program does not check the buffer size in functions such as strcpy() or memcpy() when it copies a string, there might be a buffer overflow; we can not call or transfer a null pointer, otherwise, there will be a Null Pointer Dereference [17]; the program does not release a resource incorrectly after it is used, which will cause Improper Resource Shutdown or Release (CWE – 404). From our extensive literature review, we observe that software vulnerabilities follow certain behavior patterns, and the identification of vulnerabilities depends on the precise description of the patterns [18].

Whether we use the manual analysis systems or auto-scanning systems, the formalized patterns of vulnerability are precondition and key point for vulnerability detection. The false positives of vulnerability detection are caused by the lack of accurate description of the vulnerability pattern, especially for high-speed detection tools. Therefore, in this paper, we propose a group-based static detection model for discriminating vulnerabilities, and use the defined proposition function to describe the attributes of software vulnerabilities.

Our contributions are as follows:
A. On the basis of the static vulnerability detection model based on propositional function, we develop an optimized static detection prototype system, and describe the vulnerability pattern of nine software vulnerabilities in four typical types with CWE number.
B. In order to avoid state space explosion in traditional static analysis, we propose program intermediate representation called Vulnerability Executable Path Set (VEPS) which compresses the program state space distinctly and achieves an optimized detection model.
C. We carry out three terms of experiments, the results show that our model found more vulnerabilities in the typical open source Wireshark 1.2.0 than NIST announced. Further, the accuracy rate is also higher than that of FindBugs3.0.1 detection on Tomcat 4.0.

This paper is organized into seven sections: Section II introduces some related work. Section III briefly presents our static detection model of software vulnerabilities. In Section IV, we describe the VEPS and optimize the detection model. Section V introduces the detailed design of the framework. In Section VI, we give some contrast examples to verify the framework. Section VII is conclusion and discussion of the future.

II. RELATED WORK

In recent years, with the significant growth of research work, the research emphasis of software vulnerability detection mainly focuses on how to describe vulnerability characteristics and design an efficient detection framework for vulnerability positioning and discrimination [19]–[21]. Below, we discuss the existing works related to our model in terms of the theoretical identification approaches of vulnerability characteristics and mature tools for the vulnerability positioning and discrimination.

A. THEORETICAL APPROACHES

In vulnerability detection process, it is a critical step to appropriately describe vulnerability characteristics. Edmund Clarke proposed a formal software vulnerability testing technology, so that both the model of the system and the specifications are formulated in some precise mathematical language [22]. It is obvious that it can not effectively detect vulnerabilities in large-scale programs. J.Wilander proposed dependence graphs decorated with type and range information as a generic way of modeling security properties of code [23]. Further, research [24] also proposed models to detect both integer overflow and the defect of releasing memory repeatedly, but the accuracy of these models is not high, and the expandability is also not strong.

Various detection frameworks have been proposed for vulnerability positioning and discrimination. Based on vulnerability-cause graphs, D. Byers et al., proposed a structured method for analyzing and recording the causes of software vulnerabilities [25]. This method generates the information needed for improving the process of software development to avoid similar vulnerabilities in future versions. H. Tarik and O. Noura presented an implementation for a deterministic finite automation that detected intrusions in Europay-MasterCard-Visa cards automatically. The implementation is able to do the exact recognition of the intrusions in the inverse of the use of the networks of neurons. This model reduces the false positives to some extent, but all possible executable paths in the program need to be traversed, so the detection efficiency still needs to be improved [26].

B. MATURE TOOLS

J. Viega and J. T. Bloch designed a static analysis tool named ITS4 [27]. In order to discriminate a vulnerability, it maintains a vulnerability database and uses parsing techniques to...
compare programs with the vulnerability database. But it can only detect few vulnerabilities that satisfies the conditions stored in the database, and its false positives rate (FPR) is high.

Compass is a static analysis tool for checking source code designed by ROSE Team [28]. Compass defines and describes the security of some programs, and provides a generic framework to describe few program defects of shallow layer of the program. However, the description of the vulnerability characteristics is not in-depth.

Some lightweight approaches can not find deep layer vulnerabilities and also require manual annotations [29]. FindBugs is a static analysis tool to find defects in Java code but not a style checker. It finds hundreds of defects in large Apps such as Bea WebLogic, but it does not focus on security and with lower tolerance for false positives [30], [31].

H. Li developed a software vulnerability detection mechanism named CLORIFI. It combines the advantage of static and dynamic analysis to detect code clone vulnerability using code clone verification [32]. That reduces false positives, but CVE patch information is not enough to identify sensitive sinks.

Besides, there are many other tools that automatically scan for vulnerabilities, such as Coverity [33], Fortify [2], CodeSonar, and IBM Security AppScan Source [3] (formerly known as Rational). However, due to the lack of a formal description of the vulnerability, thorough scanning requires long-term operation and has a high false positive rate.

In effect, vulnerability discovery becomes an on-going process. Due to the diversity of vulnerable programming practices, security research has largely focused on detecting specific types of vulnerabilities [34].

### III. STATIC DETECTION MODEL BASED ON PROPOSITIONAL FUNCTION

Control Flow Graph (CFG) and program dependency graph (PDG) are two important useful data structures for program static analysis [35]. A CFG is a directed graph in which each node represents a statement and each edge denotes the flow of control between statements within a function. A PDG is a directed graph representing dependencies of all nodes in the CFG towards each other [36]. Here, each node represents a statement and each edge represents the flow of control between statements within a function. The two directed graph is useful structure for static analysis.

In order to describe the vulnerability with a proposition function, we first define the related symbols and definitions in CFG and PDG, as shown in Table 1.

As pointed out in section II, nowadays, the static analysis technology is still not very generic and perfect. Vulnerability static analysis tools such as findbugs, RATs, Klocwork still have much room for improvement in false positives rate (FPR) and false negatives rate (FNR) [37]. In [38], we proposed our static detection model as five tuple Vulnerability = \{n_0, F, S, P, Q\}, where the notations of each signal is expressed in Table 2.

<table>
<thead>
<tr>
<th>Symbols</th>
<th>Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>(E(n_0, n_n))</td>
<td>There is an executable path between (n_0) and (n_n)</td>
</tr>
<tr>
<td>(EP(M))</td>
<td>All the executable paths in program (M)</td>
</tr>
<tr>
<td>(Pred(n_i, n_j))</td>
<td>If (EP(n_i, n_j)), then (n_i) is the predecessor node of (n_j)</td>
</tr>
<tr>
<td>(Succ(n_i, n_j))</td>
<td>If (EP(n_i, n_j)), then (n_j) is the successor node of (n_i)</td>
</tr>
<tr>
<td>(Pred(n))</td>
<td>If (EP(n, n_i)), then the predecessor node set of (n)</td>
</tr>
<tr>
<td>(Succ(n))</td>
<td>If (EP(n, n_j)), then the successor node set of (n)</td>
</tr>
<tr>
<td>(PD(n_j, n_i))</td>
<td>(n_i) is post-dominated by (n_j) if every directed path from (n_i) to (Exit) (not including (n_i)) contains (n_j)</td>
</tr>
<tr>
<td>(PD(n))</td>
<td>All the post-dominator nodes in CFG</td>
</tr>
<tr>
<td>(DD(n_j, n_i, v))</td>
<td>(n_i) is data dependent on (n_j); where (EP(n_i, n_j)) and satisfy the value of (v) at (n_i) has been used during execution of (n_j); and (v) is not redefined on (EP(n_i, n_j))</td>
</tr>
<tr>
<td>(CD(n_i, n_j))</td>
<td>(n_i) is control dependent on (n_j); where (EP(n_i, n_j)) and each node on (EP(n_i, n_j)) from (n_i) to (n_j) (not including (n_i) and (n_j)) is post-dominated by (n_j) and (n_i) is not post-dominated by (n_j)</td>
</tr>
</tbody>
</table>

### TABLE 1: Symbols and definitions in CFG and PDG.

<table>
<thead>
<tr>
<th>Signals</th>
<th>Notations</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n_0)</td>
<td>The initial characteristic nodes of vulnerability, in other words, (n_0) means there may be a vulnerability on the path which contains (n_0). For example, if there are series of buffer functions such as memcpy(), we say there may be a buffer overflow on the execution path of memcpy()</td>
</tr>
<tr>
<td>(F)</td>
<td>The program state space, it contains two parts of the EP in program M, control dependency and data dependency relationships among predecessor nodes, successor nodes and postdominators on each path, and so on</td>
</tr>
<tr>
<td>(S)</td>
<td>Vulnerability syntax rules, that describes the behavioral characteristics of vulnerabilities on EP. The vulnerable nodes set (N) can be deduced by (S). Means that, for any (n_1) on EP, (EP_p(EP_p, \in EP)), if (n_i) conforms to (n_0\rightarrow n_1), then (n_1\in N)</td>
</tr>
<tr>
<td>(P)</td>
<td>Precondition, which means that any node (n(n\in N)) must meet before executing, where (N) is the set of nodes related to vulnerabilities</td>
</tr>
<tr>
<td>(Q)</td>
<td>Post-conditions, which means that any node (n) in (N) must meet these rules after executing</td>
</tr>
</tbody>
</table>

In respect of propositional function, any vulnerability discrimination can be regarded as a decision problem or function problem. So we denote vulnerability detection as \(F: (P)\rightarrow \{Q\}\), which verifies the correctness of programs by Hoare logic. Then, vulnerability detection is converted to judge whether \(N\) meets both \(P\) and \(Q\) on EP. Every node \(n\) in \(N\) must meet \(P\) before executing, and meet \(Q\) after executing. Otherwise, there is a vulnerability.

In [38], we have also presented the two steps of the complete vulnerability detection process: roughly locate and precisely locate. Hence, in this paper, we only demonstrate the propositional functional functions for nine software vulnerabilities in four types with CVE number, as shown in Table 3.

Table 3 summarizes the syntax rules for vulnerability rules database which includes vulnerability syntax rules set \(S\), vulnerability preconditions \(P\) and vulnerability post-conditions \(Q\). It also contains some API functions, such as ResourceAllocationFunctionList, FormatFunctionList, BufferFunctionList, and so on.
TABLE 3: Vulnerability syntax rules S, P, and Q of four types of vulnerability.

<table>
<thead>
<tr>
<th>Vulnerability types</th>
<th>Syntax rules for S, P, Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>Null Pointer Dereference</td>
<td>S = Succ(n₁, n₀) ∪ DD(n₁, n₀, Pointer(v₁)) ∩ USE(Pointer(v₁), n₁) ∩ CHECK(Pointer(v₁), n₂, NotNull)</td>
</tr>
<tr>
<td>P = DD(n₁, n₀, Pointer(v₁)) ∩ CD(n₂, n₁) ∩ CHECK(Pointe(v₁), n₂, NotNull)</td>
<td></td>
</tr>
<tr>
<td>Q = ∅</td>
<td></td>
</tr>
<tr>
<td>Buffer Overflow</td>
<td>S = DD(n₀, n₁, Buffer(v₁, n₀)) ∩ DEF(Buffer(v₁, n₁), n₀) ∩ Pred(n₁, n₀)</td>
</tr>
<tr>
<td>P = CD(n₂, n₁) ∩ DD(n₂, n₁) ∩ CHECK(Buffer(v₁), n₁, input(v₂), n₁, Size) ∪ CH-DEF(Buffer(v₁), n₂, input(v₂), n₀, Size)</td>
<td></td>
</tr>
<tr>
<td>Q = ∅</td>
<td></td>
</tr>
<tr>
<td>Uncontrolled Format String</td>
<td>S = Succ(n₁, n₀) ∩ DEF(v, n₁) ∩ DD(n₀, n₁, v)</td>
</tr>
<tr>
<td>P = CHECK(FF(n₀), n₀, Parameter)</td>
<td></td>
</tr>
<tr>
<td>Q = ∅</td>
<td></td>
</tr>
<tr>
<td>Resource Related Flaws</td>
<td>S = USE(RAF(n₀), n₁)</td>
</tr>
<tr>
<td>P = Pred(n₂, n₁) ∩ IsIn(RF(v), n₁, RAF(n₀))</td>
<td></td>
</tr>
<tr>
<td>Q = ∩ DD(n₀, n₁, RAF(n₀)) ∩ IsIn(RF(n₀), RR(n₁))</td>
<td></td>
</tr>
</tbody>
</table>

IV. MODEL OPTIMIZED BY VEPS

As we know, static analysis technology still needs to traverse all the state space. Many static analysis tools such as Covery and Fortify have not solved this problem thoroughly. In order to avoid state explosion, Covery limits the number of paths in each procedure [33]. Fortify alleviates the bad influence of the path explosion problem by setting the upper limit of time and memory space [2]. It is worthy noting that both solutions partially alleviate the issue, the division on bigger program may cause another complexity.

Fortunately, our model, using propositional function to describe the vulnerability and its discrimination can be used to find the vulnerability executable path set (VEPS). It is clear that every vulnerability must exist in an executable path. It is obvious that checking VEPS will greatly compress state without traversing all the program execution paths. Next we will discuss how to solve VEPS.

A. VEPS SOLUTION

Actually, VEPS is the simplification of CFG. VEPS is the set of the executable paths which contains initial vulnerability nodes set n₀. In other words, VEPS ⊆ EP. In CFG, a VEPS contains all the paths consisting of the nodes between n₀ and exit or between entry and n₀. The propositional function is VEPS = EP(n₀, exit) or EP(entry, n₀). Table 4 summarizes the correspondence between EP and VEPS.

TABLE 4: Correspondence between EP and VEPS.

<table>
<thead>
<tr>
<th>vulnerability types</th>
<th>EP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Null Pointer Dereference</td>
<td>EP(n₀, Exit)</td>
</tr>
<tr>
<td>Buffer Overflow</td>
<td>EP(Entry, n₀)</td>
</tr>
<tr>
<td>Uncontrolled Format String</td>
<td>EP(Entry, n₀)</td>
</tr>
<tr>
<td>Resource Related Flaws</td>
<td>EP(n₀, Exit)</td>
</tr>
</tbody>
</table>

We use Depth-First-Search (DFS) method to obtain VEPS in a CFG, and give the algorithm of EP(n₀, Exit). Let the node in vulnerability initial nodes set (n₀) be the entry node, and use DFS to obtain EP(n₀, Exit) in a CFG. The algorithm of EP(n₀, Exit) is similar to DFS, as shown as following steps:

Step 1. Let v₁ (v₁ ∈ n₀) be the entry node, then visit and mark v₁ (v₁ ∈ n₀).

Step 2. Let v₁ be the current vertex, and then sequentially search each direct successor node of v₁. If the adjacent node v₁ has not been visited, visit and mark it, else, search another adjacent node of v₁;

Step 3. Let v₁ be the current vertex, repeat step 2 until every node in the CFG that has path connected with n₀ is visited;

Step 4. Repeat the above steps until all vulnerability initial nodes in the CFG are visited.

The set of the nodes which are marked from n₀ to exit is VEPS.

From the algorithm of EP(n₀, Exit), it can be seen that the time complexity of the algorithm is O(m + e). Here m represents the number of nodes between n₀ and exit in CFG, and e represents the number of edges. During the time, we solve EP(Entry, n₀), we only need to use predecessor node v₁ to take the place of successor node v₁ and use entry to take the place of exit.

VEPS can generate test cases for software testing. The formal five-tuple model can be optimized as Vulnerability = {n₀, F', S, P, Q}. Where F' represents the program states space in VEPS instead of F in EP and keep others consistent. Then, vulnerability detection is converted into judging whether N meets both P and Q on VEPS.

B. AN EXAMPLE

Figure 1 is a CFG, and it contains six executable paths, as follows:

path 1: <1, 2, 3, 4, exit>; path 2: <1, 2, 3, 4, 5, exit>;
path 3: <1, 2, 5, exit>;
path 4: <1, 6, 7, exit>;
path 5: <1, 6, 7, 8, 9, exit>;
path 6: <1, 6, 8, 9, exit>.

![FIGURE 1: A CFG.](image-url)
TABLE 5: Direct successor nodes set in CFG.

<table>
<thead>
<tr>
<th>Node</th>
<th>DSN</th>
<th>Nodes in VEPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3.5</td>
<td>&lt;2,3,5&gt;;&lt;2,5&gt;</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>&lt;2,3,4&gt;</td>
</tr>
<tr>
<td>4</td>
<td>5,exit</td>
<td>&lt;2,3,4,5&gt;;&lt;2,3,4,exit&gt;</td>
</tr>
<tr>
<td>5</td>
<td>exit</td>
<td>&lt;2,3,4,5,exit&gt;;&lt;2,5,exit&gt;</td>
</tr>
</tbody>
</table>

Firstly, when we find direct successor nodes (DSNs) of node 2 in CFG, we obtain nodes 3 and 5. So, VEPS has two paths: <2,3> and <2,5>; then use DFS to traverse DSNs of node 3, we obtain node 4, and VEPS is <2,3,4>. Let us repeat the step until each path of VEPS contains node exit. We will get two paths of VEPS: <2,3,4,exit> and <2,3,4,5,exit>. Then searching DSNs of node 5, we obtain one path of VEPS: <2,5,exit>. So, all paths in VEPS are as follows:

VEPS1: <2,3,4,exit>;
VEPS2: <2,3,4,5,exit>;
VEPS3: <2,5,exit>.

By contrasting VEPS with EP, we find that the number of paths in VEPS are less than EP. The child nodes on right side of node 1 will be ignored because node 2 is on the other side of node 1, which reduces the workload of path analysis. Considering the worst case in CFG, VEPS is equal to EP when node 1 is the vulnerability initial node. However, the size of VEPS is not directly proportional to the size of software. It depends on the size of vulnerability initial nodes set n0.

This indicates that different programs even in the same size, their EPS(CFG) – VEPS (set difference of EPS and VEPS) may be different, the larger the EPS(CFG) – VEPS is, the more workload will reduced by VEPS.

Therefore, we name our model as propositional function based on VEPS denoted as FOLB⁻ VEPS.

V. DETECTION SYSTEM

Based on this optimized model, we present a static detection system for software vulnerability analysis, as shown in Figure 2. It includes: basic information analysis, VEPS solution, solution of vulnerability node set N, and discrimination of vulnerability.

A. BASIC INFORMATION ANALYSIS

In Figure 3, the basic information analysis module is shown to generate and extract some static information from target program.

Firstly, we use lexical and parsing to extract the vulnerability initial node set n0 from the target program source code. Secondly, the compiler front-end (such as GCC, java compiler) is used to generate the abstract syntax tree (AST), and construct CFG and Call Graph (CG). The n0 and CFG can be used to solve VEPS, and CG is used to do interprocedural analysis in program. Dependent relationships can be extracted from PDG.

B. VEPS SOLUTION

VEPS is an intermediate representation in static detection system. Solving VEPS is a basic work in vulnerability static analysis. Figure 4 shows the process of solving VEPS, it contains: VEPS solver, Interprocedural analysis and Nodes information extraction in VEPS.

1) VEPS Solver
Starting from n0, we use DFS method to get VEPS in CFG. This VEPS is an intraprocedural vulnerability executable paths set.

2) Interprocedural Analysis
Interprodal analysis is a process that analyzes whether there are function call relationships and node dependence relationships between the nodes in-process VEPS and other paths in program. Interprocedual analysis can be achieved by analyzing CG and PDG, where the PDG can be constructed by control dependence sub-graph and data dependence sub-graph, and the System Dependence Graph (SDG) can be constructed by PDG and CG.

An interprodal VEPS needs to be reconstructed when it meets the following two conditions:

a) Analyze the CG, if there is a node n on VEPS_i (VEPS_i ∈ VEPS) which calls a function belonging to

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2019.2943896, IEEE Access
other nodes on other executable paths.

b) Analyze the PDG and SDG, if there is a node in vulnerability initial nodes set $n_0$ control or data dependent on a node $h$ which is in called function $H$.

Condition a) indicates that there must be an interprocedural function call between two different CFGs. Condition b) stipulates that the node in called function $H$ is still control or data dependent on a node in vulnerability initial nodes set $n_0$. Which makes sure that the node in called function $H$ is related to the node in vulnerability initial nodes set $n_0$, and avoids increasing paths unrelated to vulnerabilities.

If the $VEPS_i$ can not meet both of the two conditions above, then we do not need to reconstruct the interprocedural $VEPS$, and the intraprocedural $VEPS$ is the complete $VEPS$. Otherwise, we reconstruct the interprocedural $VEPS$ of the $VEPS_i$. The steps are as follows:

Step 1. Use DFS to traverse the CFG of $H$, we obtain the complete executable paths set of $H$ denoted by $EP_H(entry, exit)$.

Step 2. Extract the executable path from $EP_H(entry, exit)$ which contains $h$, and the set is denoted by $EP_h(entry, exit)$.

Step 3. Regard the $VEPS_i$ as a single linked list, and insert $EP_h(entry, exit)$ into $VEPS_i$. Where $entry$ is the successor node of $n_s$, the predecessor node of $n$, and $exit$ is the predecessor node of $n_p$, the successor node of $n$. Finally, we delete node $n$.

3) Nodes Information Extraction

After interprocedural analysis, we extract information of each node from the $VEPS$ statically, and put $VEPS$ together with these information into the database of the program state space. Extracted information contains: control dependence relationships, data dependence relationships, predecessor nodes, successor nodes, post – dominated relationships, definition – use – check relationships, and so on. We just need to extract information described by vulnerability syntax rules, preconditions and post-conditions to avoid the disclosure of extra information. For example, for Resource Relation Flaws, no control dependence relationship is described in its vulnerability syntax rules set, preconditions and post – conditions. So, we do not need to extract control dependence relationships between two nodes during the analysis of the Resource Relation Flaws.

C. SOLUTION OF VULNERABLE NODES SET $N$

The vulnerable node set $N$ is a set of nodes that may contain vulnerabilities. The solution of $N$ is a rough positioning process for the vulnerability analysis. The solution of vulnerable nodes set $N$ is given in the steps below.

Step 1. The program state space $F$ is searched from the initial vulnerability node set $n_0$.

Step 2. Insert the nodes which conform to vulnerability syntax rules $S$ into vulnerable nodes set $N$.

Algorithm 1 is shown below.

Algorithm 1 The algorithm of solving vulnerable nodes set $N$.

1: Input: $VEPS_i$, $S$, state space $F$
2: Output: vulnerable nodes set $N_{VEPS_i}$
3: Initialization: $N_{VEPS_i} = \emptyset$
4: for each $n \in VEPS_i$, except $n_0$ do
5:  if $(Relation(n_0, n) \subseteq F \& \& Relation(n_0, n) == S)$ then
6:  $N_{VEPS_i} = N_{VEPS_i} \cup n$
7: end if
8: end for
9: return $N_{VEPS_i}$

The time complexity of this Algorithm 1 is $O(m)$, where $m$ is the number of nodes in $VEPS$.

D. DISCRIMINATION OF VULNERABILITY

Vulnerability discrimination is a precise location process of vulnerability analysis which has been described in section III. It is the core of vulnerability analysis. Algorithm 2 highlights the discrimination process as blew.

Algorithm 2 The algorithm of discriminating vulnerability.

1: Input: $VEPS_i$, state space $F$, $N_{VEPS_i}$, preconditions $P$, post-conditions $Q$
2: Output: Vulnerable Nodes;
3: Initialization: Vulnerable Nodes $= \emptyset$
4: for each $n \in N_{VEPS_i}$ do
5:  for each $m \in VEPS_i$, $m \neq n$ do
6:  if $(Relation(n, m) \subseteq F)$ then
7:    Vulnerable Nodes $= Vulnerable Nodes \cup n$
8:  else if $(Relation(n, m) \in F \& \& Relation(n, m) == P) \& \& (Relation(n, m) \neq Q)$ then
9:    Vulnerable Nodes $= Vulnerable Nodes \cup n$
10: end if
11: end for
12: end for
13: return Vulnerable Nodes

From Algorithm 2, it is clear that the time complexity of this algorithm is $O(m^2)$, where $m$ is the number of nodes in $VEPS_i$.

As mentioned in our introduction, the automatic scanning tools do not completely replace human judgment. But time consumption of personal detection is not comparable, so the time complexity is not critical with respect to our new model. As we use $VEPS$ instead of $EP$ to optimize the scanned paths, time complexity analysis of two algorithm here is the only chance to explain our theoretical superiority.

VI. EXPERIMENTS AND EVALUATION

Considering the three main contributions of the paper, to evaluate the effectiveness of our approach, we should focus on operability - how it works, validity - what about the model’s results and superiority - what is the advantage compared with the other method.

We evaluate our method through three terms of experiments from Sep., 2016 to May, 2018. The first is the detection of Virshark 1.2.0 including time consumption contrast of traveling $VEPS$ and traveling CFG ($EPS$) to validate our model. The second is the detection of four open resource softwares and the detection contrast on CWE – 476
(Tomcat4.0) of our method with FindBug 3.0.1 to show the superiority of our model. The third one is the lists of vulnerabilities discovered by our methods. All those resource software are widely used currently and their vulnerabilities are regularly audited disclosed by NIST.

A. A VULNERABILITY DETECTION EXAMPLE

A Java code with resources operation related flaws in it shown as Source code 1, in which the method FunA() called in line 29st has closed resource fos on line 15th, and resource fos is written in line 31rd, so a Use after Free vulnerability occurs; resource fos is allocated by FunB(), but resource fos is not closed in FunB(), so resource leakage occurs.

Source code 1. A source code with resources operation.

```java
4 public class Test {
7 public void FunA(FileOutputStream fos) {
9 try {
11 String s1 = "startdetect...";
12 fos.write(s1.getBytes());
14 } finally {
16 fos.close();
17 } catch (Exception e) {
19 e.printStackTrace();
22 }
25 }
28 public void FunB(int x) {
30 if (x > 0) {
33 System.out.println("x > 0");
35 } else {
38 FileOutputStream fos = null;
41 try {
44 fos = new FileOutputStream("G:\\aaa1.txt");
48 } catch (Exception e) {
51 e.printStackTrace();
53 }
56 }
59 }

The process of vulnerability detection introduced in our model is as follows:

Step 1. Extract the set of initial vulnerability nodes from the source code. Analyzing the source code and searching the resources operation functions, we find that fos = new FileOutputStream ⊆ ResourceAllocateFunctionList in line 27 and there is a DEF(fos, 28). Its predication is n0 = \{the sentence in line 28th\};

Step 2. Construct CFGs for FunA() and FunB(). They are shown as Figure 5.a) and Figure 5.b) respectively.

Step 3. Solve VEPS in CFG of FunB(). VEPS of resource related flaws is EP(n0, exit), and Table 6 shows the DSNs of each node in EP(n0, exit).

Finally, we obtain three vulnerability executable paths of FunB() shown in Table 7.

From Table 7, we find that, comparing with the traditional path analysis methods, the VEPS skips node 25 and its branch node 26 in CFG. So, if branch node 26 has com-
TABLE 8: Paths that contain node 10 and 15 on *FunA*(.).

<table>
<thead>
<tr>
<th>path</th>
<th>nodes of path</th>
</tr>
</thead>
<tbody>
<tr>
<td>path1</td>
<td>[entry, 9, 10, 12, 15, exit]</td>
</tr>
<tr>
<td>path2</td>
<td>[entry, 9, 10, 15, exit]</td>
</tr>
<tr>
<td>path3</td>
<td>[entry, 9, 10, 15, 17, exit]</td>
</tr>
<tr>
<td>path4</td>
<td>[entry, 9, 10, 12, 15, 17, exit]</td>
</tr>
</tbody>
</table>

Secondly, let node 9 be the successor node of node 30 which is the predecessor node of node 31, and let node 10 be the predecessor node of node 32 which is the successor node of node 31 on *VEPS1* and *VEPS2*, respectively. Then we delete node 31, and construct a complete interprocedural *VEPS* shown in Table 9.

TABLE 9: Interprocedural *VEPS* of *FunB*().

<table>
<thead>
<tr>
<th><em>VEPS</em></th>
<th>sub-<em>VEPS</em></th>
<th>nodes of sub-<em>VEPS</em></th>
</tr>
</thead>
<tbody>
<tr>
<td><em>VEPS1</em></td>
<td><em>VEPS1</em></td>
<td>30, 9, 10, 12, 15, 32, 33, exit</td>
</tr>
<tr>
<td></td>
<td><em>VEPS2</em></td>
<td>30, 9, 10, 15, 32, 33, exit</td>
</tr>
<tr>
<td></td>
<td><em>VEPS3</em></td>
<td>30, 9, 10, 12, 15, 32, 33, exit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>30, 9, 10, 12, 15, 32, 33, exit</td>
</tr>
</tbody>
</table>

Step 5. Extract the information of nodes on every vulnerability executable path. The result is shown in Table 10. We avoid the control dependence relationships (depend-relation) because there is no control dependence described in the vulnerability syntax rules set, preconditions, or post conditions of resources operation related flaws. In this way, we avoid extracting the unnecessary information, and compress the state space of program.

Step 6. Solve the set of vulnerable nodes *N* on each *VEPS* (*VEPS1* ∈ *VEPS*), *F*: *n*0 → *n*1, where *S* = *USE*(RAF(*n*0, *n*1)) ∪ *Succ*(*n*1, *n*0) \ *DD*(*n*1, *n*0, RAF(*n*0)). We find the vulnerable node which conforms to vulnerability syntax rules in Table 9.

The nodes set related to vulnerability in *VEPS1* and *VEPS2* are the same, in other words, *NV*-*VEPS1* = *NV*-*VEPS2* = {30, 10, 15, 33};

The nodes set related to vulnerability in *VEPS3* is denoted by *NV*-*VEPS3* = {30}.

Step 7. Discriminate vulnerable nodes set *N* on each vulnerability executable path with precondition *P* and post-condition *Q*.

Where: *P* = *Pred*(n2, n1) \ ~*IsIn*(RR(n2), RAF(*n*0)), *Q* = *PD*(n3, n1) \ *DD*(n3, n1, RAF(*n*0)) \ *IsIn*(RAF(*n*0), RR(n3)) \ *IsIn*(RAF(*n*0), RR(n1)).

The results are shown in Table 11 and Table 12. From Table 11 and Table 12, we see that node 33 in *VEPS1* and *VEPS2* meets neither *P* nor *Q*. The discrimination results are both FALSE. By analyzing source code, we find that node 15 is predecessor node of node 33 releases resource *fos* denoted by RR(15), but node 33 continues to use *fos*. So, it does not meet the preconditions *P* = *Pred*(n2, n1) \ ~*IsIn*(RR(n2), RAF(*n*0)). Node 33 exists *Use After Free*(CWE = 416). Meanwhile, node 33 neither release *fos* on *VEPS1* nor *VEPS2*. So, post conditions *Q* = *IsIn*(RAF(*n*0), RR(n3)) also does not. For *VEPS1*, the program will exit after node 33 executing, so there is a *Memory Leak* (CWE = 401). For *VEPS2*, node 35 which is exception handling sentence is the successor node of node 33, but exception handling sentence do not release resource *fos*, so there is a *Memory Leak* (CWE = 401) and *Improper Resource Shutdown or Release* (CWE = 404). For *VEPS3*, there is a node *n*2 = {sentence28th} which meets *P*, so the discrimination

![Image](https://creativerecomms.org/licenses/by/4.0/)

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2019.2943896, IEEE Access
result is TRUE. There is no node denoted by \( n_3 \) which meets 
\( IsIn(RAF(n_0, RR(n_3)) \) in \( Q \), so discrimination result is FALSE. By analyzing source code, we find that node 30 allocates a resource \( f_{os} \), but does not release it on node 35 when program exits on \( VEPS_3 \). It is similar to \( VEPS_2 \). Node 30 in \( VEPS_3 \) has both Memory Leak (CWE – 401) and Improper Resource Shutdown or Release (CWE – 404).

B. DETECTION RESULT OF WIRESHARK AND TIME CONSUMPTION CONTRAST

Wireshark is an open source network packet analysis software. We use Joern (http://mlsec.org/joern) to get \( CFG \) and then solve \( VEPS \) according to our model. The number of \( CFG \) and \( VEPS \) in Wireshark 1.2.0 as well as their traveling time consumption are shown in Table 13. The detection results are compared with the vulnerabilities disclosed by NIST, and is shown in Table 14.

### TABLE 13: Paths and traveling time consumptions of \( CFG \) and \( VEPS \) in Wireshark.

<table>
<thead>
<tr>
<th>types</th>
<th>The number of initial nodes</th>
<th>the number of paths in ( CFG ) Traveling time consumption(ms)</th>
<th>the number of paths in ( VEPS ) Traveling time consumption(ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>NPD1</td>
<td>6910</td>
<td>10897/8717620</td>
<td>8822/7057621</td>
</tr>
<tr>
<td>BF2</td>
<td>1211</td>
<td>4699/3571240</td>
<td>2609/1995885</td>
</tr>
<tr>
<td>UFS3</td>
<td>822</td>
<td>691/525160</td>
<td>854/649040</td>
</tr>
<tr>
<td>RRF4</td>
<td>436</td>
<td>2794/2267278</td>
<td>2073/1683276</td>
</tr>
<tr>
<td>sum⁴</td>
<td>9379</td>
<td>19081/1508274</td>
<td>14358/1138582</td>
</tr>
</tbody>
</table>

Notations:
1. The Null Pointer Dereference vulnerability.
2. The Buffer Overflow vulnerability.
3. The Use Format String vulnerability.
4. The Resources Related Operation Flow.
5. To develop the traveling algorithm more easily, we store paths as adjacent chains.

From Table 13, we find that the number of paths in \( VEPS \) (14358) is smaller than the number of paths which contain initial vulnerability nodes in a \( CFG \) (19081). As vulnerability detection is divided into two phases of work, the time consumption is also divided into two parts: the positioning time consumption and the confirmation time consumption. The confirm technology is simple and relatively mature. Therefore, the time of vulnerability detection is mainly spent in exploring the position of vulnerabilities. And different detection methods consume different time. It is known that the analysis and locating technologies of vulnerabilities are not completely automatic, so it is difficult to accurately compare the consumption time of \( VEPS \) with other methods. However, if we divide the whole process into much smaller segments, we can accurately compare it with different detection methods on each segment.

Vulnerability positioning is in fact traveling the execution path \( CFG \) (\( EPS \)) of the program. So the time consumption contrast of traveling \( VEPS \) and traveling \( CFG \) (\( EPS \)) indicates the superiority of \( VEPS \). Thus we adopt DFS method to traversal paths of \( CFG \) and \( VEPS \) of Wireshark 1.2.0, and sum all the consumption time of every path of \( CFG \) and \( VEPS \), respectively, as shown in Table 13. The result shows that \( VEPS \) does save much time.

### TABLE 14: Detection results of Wireshark.

<table>
<thead>
<tr>
<th>Vtype¹</th>
<th>NISTnum²</th>
<th>FOLBnum³</th>
<th>Muntualnum⁴</th>
<th>FPnum⁵ /FNnum⁶</th>
<th>FPR/FNR(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>NPD</td>
<td>26</td>
<td>43</td>
<td>20</td>
<td>23/6</td>
<td>53.423</td>
</tr>
<tr>
<td>BF</td>
<td>106</td>
<td>121</td>
<td>99</td>
<td>22/7</td>
<td>18.2/6.6</td>
</tr>
<tr>
<td>UFS</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0/0</td>
<td>0/0</td>
</tr>
<tr>
<td>RRF</td>
<td>6</td>
<td>8</td>
<td>6</td>
<td>2/0</td>
<td>5/0</td>
</tr>
<tr>
<td>sum⁷</td>
<td>138</td>
<td>172</td>
<td>125</td>
<td>47/13</td>
<td>27.39/4</td>
</tr>
</tbody>
</table>

¹ Notation: 1. The type of vulnerability.
2. The number disclosed by NIST.
3. The number detected by FOLB·\( VEPS \).
4. The number of mutual vulnerability.
5. False positives number/false negatives number,\( FPnum=FOLBnum\)- Muntualnum.
6. \( FNnum=NISTnum\)- Muntualnum.
7. FPR/FNR.

From Table 14, it can be seen that the detection results of buffer overflow have a lower FPR and FNR. There are two false positives of resource related flaws, but no false negatives. In general, both of the FPR and FNR of our method are low. However, there is an exception that FPR of Null Pointer Dereference is higher than the data provided by NIST. The reason is that the preconditions in this paper require checking whether each pointer is not null explicitly before it is called. But in real program, some software may check the return value from some functions and handle it appropriately, which prevents it from detecting unexpected states and conditions. Hence, these functions can not check the validity of the pointer address when calling these pointers. However, checking the return value from some functions can not prevent \( NPD \), since no one can make sure the pointer states when it is called.

C. ACCURACY OF FOLB·\( VEPS \) AND CONTRAST WITH FINDBUGS

We have verified and confirmed the vulnerabilities of four open source projects disclosed by NIST, shown as Table 15.

### TABLE 15: Verification and validation of the vulnerabilities disclosed by NIST.

<table>
<thead>
<tr>
<th>Vtype</th>
<th>NPD</th>
<th>BF</th>
<th>UFS</th>
<th>RRF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chrome 5.0</td>
<td>NISTnum</td>
<td>FOLBnum</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Wireshark1.8</td>
<td>NISTnum</td>
<td>FOLBnum</td>
<td>1</td>
<td>43</td>
</tr>
<tr>
<td>ABM1.0</td>
<td>NISTnum</td>
<td>FOLBnum</td>
<td>0</td>
<td>58</td>
</tr>
<tr>
<td>Asterisk10.2</td>
<td>NISTnum</td>
<td>FOLBnum</td>
<td>0</td>
<td>49</td>
</tr>
</tbody>
</table>

| The accuracy of FOLB·\( VEPS \) | 100% | 88.9% | 100% | 92.8% |

Notation: The accuracy of FOLB·\( VEPS = FOLBnum/NISTnum \).

As shown in Table 15, the result indicates that the vulnerabilities of RRF, NPD and UFS have high detection accuracy, whereas, BF has low FPR and FNR. Thus, our method can accurately and efficiently detect most of vulnerabilities.
A Java project tomcat 4.0 is a Servlet container developed by the Jakarta project under the Apache Software Foundation, whose vulnerabilities are also disclosed by NIST. This paper uses FOLB & VEPS to detect Null Pointer Dereference (CWE – 476) and compare the result with static analysis tool FindBugs 3.0.1. The detection result is described in Table 16.

TABLE 16: The comparison of detection results of NullPointer Dereference (CWE – 476) between FindBugs 3.0.1 and FOLB & VEPS.

<table>
<thead>
<tr>
<th>Detection tools</th>
<th>Detection result</th>
<th>confirm as False positives number</th>
<th>False positives rate</th>
<th>Detection rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>FindBugs F’V</td>
<td>36</td>
<td>22</td>
<td>61.1%</td>
<td>5.6%</td>
</tr>
<tr>
<td></td>
<td>42</td>
<td>21</td>
<td>50%</td>
<td>30.9%</td>
</tr>
</tbody>
</table>

The result indicates that the detection rate of FOLB & VEPS (F’V) is higher than that of FindBugs and the FPR is also lower than that of FindBugs. The main reason is that, in our model, the syntax rules for vulnerability rules database are described accurately by the propositional function. Even in a bug-free application, the vulnerability syntax rules can not be matched exactly, thus the FPR naturally decreased.

D. A VULNERABILITY DETECTION EXAMPLE

Table 17 presents some vulnerabilities in the real applications discovered by our methods in past several years.

In Table 10, the vulnerabilities named ’Unreported’ and ’Waiting for’ were discovered by the method of this paper, and have not been reported. The vulnerability named ’CVE-2013-1221’ was reported to Microsoft by our laboratory in 2013. These findings are based on the formalization of propositional functions in this paper. In addition, compared with other traditional methods, the main advantages of our model are faster positioning and high accuracy, which establishes the foundation for the automatic detection.

VII. LIMITATIONS

Based on propositional function, the paper gives formal description of the four main types of vulnerabilities. In theory, it can be applied to any static open source program code. However, further research is needed to explore our model’s effectiveness, in depth. There are still some challenges need to be address are highlighted below.

A. The model is only used for static detection;

B. Our prototype system only gives a specific description of the normalization of vulnerabilities in programs by C and Script languages such as JAVA, VB or ASP, and the normalization of corresponding vulnerabilities in other programming languages has not yet been done;

C. We present the conditions for vulnerabilities and give a formal description of these conditions in terms of propositional function, in order to achieve the complete automatic detection on open source programs. Currently, we have not developed our own program to draw CFG. For short programs, we do it manually, and for long complex programs, we still use the Joern (http://mlsec.org/joern) to get their CFG.

VIII. CONCLUSIONS AND FUTURE WORK

Auto or intelligent check is a trend of vulnerability detection. In this paper, we analyzed the principles and defects of existing static vulnerability detection tools, and pointed that the root cause of their defects was lack of precise definition and formal description of properties of vulnerabilities. Therefore, we presented a formal description of a static vulnerability detection model based on propositional functions. Firstly, we defined and described the existing preconditions, characteristics and decision rules of the vulnerability. Secondly, we optimized our model by VEPS to avoid the space explosion, which only needs check the executable path set of vulnerability without traversing all the executable path of software program. Following this, we designed a static detection prototype system, and performed three terms of experiments. The experimental results shown that our static detection model not only accurately described the vulnerability, but also eliminated the unreachable path and reduced the false positive rate.

In Future, we have two key directions. The first one is to keep expanding the propositional function of vulnerability syntax rules and discriminant conditions, and verify the correctness of more open-source codes. The second one is to improve our prototype system and compare with more other static detection systems.

REFERENCES


TABLE 17: Vulnerabilities discovered by our methods.

<table>
<thead>
<tr>
<th>Name or ID</th>
<th>Address or Version</th>
<th>Vulnerability types and Possible consequence</th>
<th>Time and Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVE-2017-15265</td>
<td>Linux 4.14-rc5/dve/snd/seq/ snd_seq_soc1_create_port() 1271row</td>
<td>Conditional competition leading to a UAF that can be used to upgrading system permissions</td>
<td>(CNVD-2017-30251, CNVND-201710-230) C language</td>
</tr>
<tr>
<td>CVE-2013-1221</td>
<td>Win2k sp4 Winamp5.11 (Chinese version) Address of function Load library: 77e80221 Address of function system: 78018bf</td>
<td>Buffer overflow, may cause upgrading system permissions</td>
<td>2013-10 C Language</td>
</tr>
</tbody>
</table>
LANSHENG HAN received the Ph.D. degree in computer information security from the Huazhong University of Science and Technology (HUST), Wuhan, China, in 2006. He has been an Associate professor of Information Security in Huazhong University of Science and Technology (HUST), China. His main interest includes security of network, malicious code identification, network virus tracking, security in Big Data, et al.

MAN ZHOU is a graduate student and candidate for PhD with School of Computational Science and Technology, Huazhong University of Science and Technology (HUST), China. Her research interests include network space security, Internet of Thing security, malicious behavior identification.

YEKUI QIAN received the Ph.D. degree in Zhengzhou air defense academy, Zhengzhou, China. He is a professor, his main research direction is network space security, telecommunications technology, Internet technology, and building science and engineering.

CAI FU received the Ph.D. degree in computer information security from the Huazhong University of Science and Technology (HUST), Wuhan, China. He is a professor of Information Security in Huazhong University of Science and Technology, China. His research interests include mobile network security, network behavior analysis, malicious code and cryptographic theory and technology.

DEQING ZOU received the Ph.D. degree from the Institute of Computational Science, Huazhong University of Science and Technology (HUST), Wuhan, China, in 2004. He is a Professor of Huazhong University of Science and Technology, China. His major research interests include system security, trusted computing, virtualization, and cloud security.