An Static Propositional Function Model to Detect Software Vulnerability

Lansheng Han, Man Zhou(✉), and Cai Fu

School of Computer Science and Technology,
Huazhong University of Science and Technology, Wuhan, China
{hanlansheng,zhou_man1125}@hust.edu.cn

Abstract. Due to lacking proper theory to accurately describe characteristics of vulnerability, the existing static detection models are designed for specific vulnerability is hard to be expanded and the latter often encounters the state space explosion and with higher false positive rate. This paper proposes a static detection model of a five-tuple $(n_0; F; S; P; Q)$: the vulnerability initial nodes set, program state space, Vulnerability Syntax Rules, preconditions of vulnerability, and post-conditions of vulnerability are accurately described. We design a testing prototype system for the static detection model and carry out experiments to evaluate the results with the vulnerabilities disclosed by NIST. Our model find more vulnerabilities of Wireshark than published by NIST and shows higher detection efficiency than that of FindBugs. Formal accurately description is prerequisite of auto-detection of vulnerability.

Keywords: Software vulnerability · Propositional function · Static analysis · State space explosion

1 Introduction

With the advent of information society and the popularization of software applications, more security problems of computer are arising from software vulnerabilities. Software vulnerability is weakness in software systems that may cause the application crash or be exploited by a threat to gain unauthorized access to information [1,2]. So software vulnerability detection has been a research focus of information security in recent years [3]. And various detection approaches are put forward [4,5].

1.1 Motivation and Contributions

All these existing approaches are falling into three main categories: static, dynamic, and integrated analysis systems. But due to lacking proper theory to accurately describe characteristics of vulnerability, they are imprecise, resulting

© ICST Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2019
Published by Springer Nature Switzerland AG 2019. All Rights Reserved
J. Li et al. (Eds.): SPNCE 2019, LNICST 284, pp. 564–575, 2019.
https://doi.org/10.1007/978-3-030-21373-2_46

in a large amount of false positives. Dynamic analysis systems, such as "fuzzers", can provide conditional inputs. However, they suffer from exhaustive test cases.

Current researches agree that every software vulnerability is caused by some flaws or defects of the software [3]. Most software defects and flaws are parts of software inherent attributes and they always occur regularly [6]. So we believe that software vulnerabilities follow certain patterns and can be identified by them if the patterns are accurately be described [7].

Many years of the research on the vulnerabilities detection make us believe false positives are caused by suspicion or misunderstanding, the both are due to the lack of an accurate formal description of the vulnerability especially for some high speed detecting tools.

In this paper, we propose a vulnerability static detection model by abstracting the characteristics of a variety of vulnerabilities in form of propositional function. We focus on software source code detection and try to formalize patterns of vulnerability. If there is a violation of patterns in a program, there will be software vulnerability. We discriminate and describe a variety of software vulnerabilities formally by this model.

1.2 Related Work

In the following, we briefly review the prior work most closely related to our model in two groups: theoretical approaches and mature tools.

Clarke proposed a formal software vulnerability testing technology which can judge whether a given program meets the pre-defined characteristics or not by traversing the state space [8]. Obviously, there will be a state space explosion when it is used to detect large-scale programs.

Describing vulnerability characters appropriately is a critical step for its detection. Wilander proposed a generic way to model the security characteristics of codes by vulnerability dependency graph [9].

B. Liang and K.K. Hou proposed an expanded finite-state machine model which can traverse the possible executable paths in a program statically and identify the current operation. This model reduces the false positives to some extent, but all possible executable paths in the program need to be traversed, so the detect efficiency still needs to be improved [10].

Compass is a static analysis tool for checking source code designed by ROSE Team [11]. It does not describe the characteristics of vulnerabilities in depth.

Some lightweight approaches include Rats [12], Prefast [13] as well as Splint [14], they can not find deep layer vulnerabilities and also require manual annotations. FindBugs is a static analysis tool to find defects in Java code but not a style checker.

There are some other tools, like Coverity, Fortify, CodeSonar, and IBM Security AppScan Source (formerly Rational). Due to the auto scanning, those tools can make thorough analysis with configurable rule sets. Lack of formal description of vulnerability, thorough scanning need long run time and the false positive rate is still high [15].

2 Static Detection Model Based on Propositional Function

CFG (Control Flow Graph) and PDG (program dependency graph) are two important useful data structures for program static analysis [16,17]. A CFG is a directed graph that shows all paths might be traversed through a program during its execution, whose edges represent possible flow of control between statements [18]. Amed at describing vulnerability conveniently by the propositional function, we define the related concepts in CFG and PDG at first.

2.1 Related Definitions in *CFG*

Let n_i and n_j be two nodes on CFG:

Definition 1. *In a CFG, if there is sequence* $p = <n_0, ..., n_m>$ *which meets* $(n_{i-1}, n_i) \in E$, *where* $i = 1, 2...m$. *Then there is an executable path between* n_0 *and* n_m, *denoted by* $EP(n_0, n_m)$. *The set of all the executable paths in program denoted by* EP.

Definition 2. *If there is an executable path* $EP(n_i, n_j)$ *between* n_i *and* n_j, *then* n_i *is the predecessor node of* n_j, *denoted by* $Pred(n_i, n_j)$; n_j *is the successor node of* n_i, *denoted by* $Succ(n_i, n_j)$. *Let* n *be a sentence, and the set of all its predecessor nodes called the precursor node set of* n, *denoted by* $Pred(n)$. *The set of all its successor nodes is called the successor node set of* n, *denoted by* $Succ(n)$.

Definition 3. *In a CFG,* n_i *is post-dominated by* n_j *if every directed path from* n_i *to Exit(not including* n_i*) contains* n_j, *denoted by* $PD(n_j, n_i)$. *It should be noted that* n_i *is not the post-dominator of itself. Let* n *be a sentence, and the set of all its post-dominator is called the post-dominator set of* n, *denoted by* $PD(n)$.

Definition 4. *There is an executable path* $EP(n_i, n_j)$. n_j *is data dependent on* n_i, *denoted by* $DD(n_j, n_i, v)$ *if*

(1) there is a variable v, *the value of* v *at* n_i *has been used during execution of* n_j.
(2) v *is not redefined on* $EP(n_i, n_j)$.

Definition 5. *There is an executable path between* n_i *and* n_j. n_j *is control dependent on* n_i, *denoted by* $CD(n_i, n_j)$ *if*

(1) each node on $EP(n_i, n_j)$ *from* n_i *to* n_j *(except* n_i *and* n_j*) is post-dominated by* n_j.
(2) n_i *is not post-dominated by* n_j.

2.2 Vulnerability Static Detection Model Based on Propositional Function

Following the above definitions, we can construct our detection model based on propositional function.

Definition 6. *The detection model is defined as a five-tuple denoted as* $Vulnerability = \{n_0, F, S, P, Q\}$*. It includes the vulnerability initial nodes set, program state space, Vulnerability Syntax Rules, preconditions of vulnerability, and post-conditions of vulnerability. The followings are the detailed description of the five-tuple.*

Vulnerability initial nodes set n_0. n_0 is the initial characteristic nodes of vulnerability which is the entrance node of vulnerability detection. For a program M, its sentence is finite. So, the vulnerability initial nodes set n_0 is finite and certain.

Program state space F. F is the program state space extracted from source code, CFG and PDG. It contains the EP in program, control dependency and data dependency among nodes. F is an Intermediate Representation which contains all necessary information for vulnerability detection, and it can not be empty.

Vulnerability Syntax Rules set S. S is a set of vulnerability syntax rules which are state transition rules between vulnerability initial nodes set n_0 and vulnerable nodes set N on EP.

Precondition P. P is Precondition which means that any node $n(n \in N)$ must meet these state conditions before executing, where N is the set of nodes related to vulnerabilities. Otherwise, there will be a vulnerability.

Post-conditions Q. Q is Post-conditions which means that any node n in N must meet these rules after executing. Otherwise, there will be a vulnerability.

With the model above, the complete process of vulnerability detection can be described as F :{P}$n_0 \xrightarrow{s} N$\{Q\}. This process has two steps:

Step 1. Locate vulnerabilities roughly. $n_0 \xrightarrow{s} N$ means that we find the vulnerable node which conforms to the Vulnerability Syntax Rules set S from n_0 on EP. For any $\mathrm{EP}_i(\mathrm{EP}_i \in F)$, if there is a node n_1 conforms to $n_0 \xrightarrow{s} n_1$, n_1 is a vulnerability related node, $n_1 \in N$.

Step 2. Locate vulnerabilities precisely. {P}N{Q} means that we detect the vulnerable nodes set N by Precondition P and Post-conditions Q. For any $EP_i(\mathrm{EP}_i \in F)$, if there is node n_2 conforms to $n_1 \xrightarrow{p} n_2$ before n_1 executing, and there is node n_3 conforms to $n_1 \xrightarrow{Q} n_3$ after n_1 executing, the detection result is TRUE, and n_1 does not have a vulnerability. Before n_1 executing, if $\neg\forall n_2$ conforms to $n_1 \xrightarrow{p} n_2$, or after n_1 executing, $\neg\forall n_3$ conforms to $n_1 \xrightarrow{Q} n_3$, detection result is FALSE and n_1 has a vulnerability.

Next we will use propositional function to describe some types of software vulnerabilities.

2.3 Formal Description of Software Vulnerability Based on Propositional Function

In this paper, we focus on describing and detecting nine software vulnerabilities in four types with CWE number which are the most prone to general programs, as shown in Table 1. Before formulating these software vulnerabilities, we also need some definitions in the form of propositional function.

Table 1. Software vulnerabilities

Vulnerability	CWE number
Null Pointer Dereference	CWE-476; CWE-690
Buffer Overflow	CWE-119; CWE-120
Uncontrolled Format String	CWE-134
Resource Relation Flaws	CWE-401; CWE-404; CWE-415; CWE-416

Definition 7. *The way to use variable* v *can be described as definition-use-check relationships.* $DEF(v, n)$ *means the statement, definition or assignment of* v *at sentence* n*;* $USE(v, n)$ *means* v *is used or cited on node* n*;* $CHECK(v, n, Statement)$ *means detect the statement of* v *on node* n*. For example,* $CHECK(v, n, Null)$ *means detecting whether the statement of* v *on* n *is Null or not, and the check result will be True or False.*

Definition 8. *The type of a parameter in program M can be described by corresponding propositional functions. For example: pointer variable* $v = \{v|\exists v \in M,$ *type of* v *is Pointer*$\}$ *is denoted by* $Pointer(v)$*; function* f *is denoted by* $Function(f)$*, etc.*

Definition 9. *Use the* $ResourceAllocateFunctionList$ *to denote the function set related to resource allocation. In C Programming Language the common resource allocation functions are* $malloc()$*,* $fopen()$*,* $calloc()$*,* $new()$*, etc. The* $ResourceAllocateFunctionList(n)$ *denotes resource allocation functions on node* n*, abbreviated as* $RAF(n)$*.* $ResourceRelease(n)$ *means to release resources related to resource allocation functions* $RAF(n)$ *on node* n*, abbreviated as* $RR(n)$*.*

Definition 10. *The format functions are denoted by* $FormatFunctionList$*. In C Programming Language common format functions include* $printf()$*,* $strncpy()$*,* $fwprintf()$*,* $snwscanf()$*,* $fprint()$*,* $printf()$*, etc.* $FormatFunction(n)$ *means format function which is called on node* n*, abbreviated as* $FF(n)$*.*

Definition 11.
The buffer related functions are denoted by $BufferFunction(n)$*. The common buffer related API functions include* $memcpy()$*,* $strcpy()$*,* $sprint()$*,* $vsprintf()$*,* $gets()$*,* $scanf()$*,* $strcat()$*, etc.*

Definition 12. *Propositional function* $CallFunction(n)$ *means the information of functions called on node* n.

Definition 13. *Propositional function* $SharedResource(v, n)$ *means shared resources on node* n *in program* v. *Propositional function* $SharedResource(v)$ *means the set of all shared resources in program* v.

Definition 14. *Propositional function* $IsIn(n_1, n_2)$ *means* $n_1 \subseteq n_2$ *and propositional function* $\neg IsIn(n_1, n_2)$ *means* $n_1 \subsetneq n_2$.

With these formal definition we can present formal propositional function for software vulnerability. We summarize the characteristics of these vulnerabilities and achieve its propositional function.

(1) Null Pointer Dereference. For a target program M, the set of sentence n which defines or declares pointers in M is vulnerability initial nodes set denoted by n_0. Its propositional function is:

$$n_0 = \{n | \exists n \in M \wedge DEF(Pointer(v), n)\}. \tag{1}$$

On any executable path $EP_i (EP_i \in EP)$, if there is a successor node n_1 of n_0 which calls $Pointer(v)$ and is data dependent on n_0 with $Pointer(v)$, n_1 is a vulnerable node s. Its propositional function is:

$$S = Succ(n_1, n_0) \wedge DD(n1, n0, Pointer(v)) \wedge USE(Pointer(v), n_1). \tag{2}$$

On this executable path $EP_i (EP_i \in EP)$, if there is a node n_2 which is data dependent on n_0 with $Pointer(v)$ and vulnerable node n_1 is control dependent on n_2, and $Pointer(v)$ is Null on n_2, n_1 does not have any vulnerability. Otherwise, n_1 has vulnerabilities. Its propositional function is:

$$P = DD(n_2, n_0, Pointer(v)) \wedge CD(n_2, n_1) \wedge CHECK(Pointer(v), n_2, NotNull), \tag{3}$$

Q on NPD is Null.

(2) Buffer overflow. For a target program M, the set of sentence n which calls the buffer related functions is vulnerability initial nodes set denoted by n_0. Its propositional function is:

$$n_0 = \{n | \exists n \in M \ wedgeCallFunction(n) \subseteq BufferFunctionList\}. \tag{4}$$

On any executable path $EP_i (EP_i \in EP)$, if n_0 is data dependent on $Buffer(v_1)$ which is defined on the predecessor node n_1 of n_0, n_0 is a vulnerable node. Its propositional function is:

$$S = DD(n_0, n_1, Buffer(v_1, n_0)) \wedge DEF(Buffer(v_1), n_1) \wedge Pred(n_1, n_0). \tag{5}$$

On this executable path $EP_i (EP_i \in EP)$, if there is a node n_2 that n_0 is control dependent on, and n_1 is the postdominator of n_2, and both buffer size

and input data length are matching, n_2 does not have vulnerability. Otherwise, n_2 has vulnerabilities. Its propositional function is:

$$\begin{aligned} P =&CD(n_2, n_0) \wedge PD(n_2, n_1) \wedge CHECK(buffer(v_1), input(v_2), n_1, Size) \cup \\ &CHECK(buffer(v_1), input(v_2), n_0, Size), \end{aligned} \tag{6}$$

Q on Buffer Overflow is null.

(3) Uncontrolled Format String. For a target program M, the set of sentence n which calls $FormatFunctionList$ is vulnerability initial nodes set. Its propositional function is:

$$n_0 = \{n | \exists n \in M \wedge CallFunction(n) \subseteq FormatFunctionList\}. \tag{7}$$

On any executable path $EP_i(EP_i \in EP)$, if n_0 is data dependent on variable v defined on its predecessor node n_1, n_0 is a vulnerable node. Its propositional function is:

$$S = Succ(n_1, n_0) \wedge DEF(v, n_1) \wedge DD(n_0, n_1, v). \tag{8}$$

On this EP_i, if both the type and the number of parameters in a format string function are matching on node n_0, the result is TRUE, which means n_0 does not have vulnerability. Otherwise, n_0 has vulnerabilities. Its propositional function is:

$$P = CHECK(FF(n_0), n_0, Parameter), \tag{9}$$

and Q is null.

(4) Resource Related Flaws. In target program M, the set of sentence n which defines or declares a variable v belonging to resource allocation functionlist is a vulnerability initial nodes set n_0. Its propositional function is:

$$n_0 = \{n | \exists n \in M \wedge DEF(v, n) \subseteq ResourceAllocateFunctionList\}. \tag{10}$$

On any executable path EP_i, if node n_1 which is the successor node of n_0 calls resource allocation functions on n_0 denoted by $RAF(n_0)$ and is data dependent on n_0 which is the precursor node of n_1 with $RAF(n_0)$, n_1 is a vulnerable node. Its propositional function is:

$$S = USE(RAF(n_0), n_1) \cup Succ(n_1, n_0) \wedge DD(n_1, n_0, RAF(n_0)). \tag{11}$$

On this EP_i, if there is not resource release operation $RR(n_2)$ corresponding resource allocation functions $RAF(n_0)$ on node n_2 which is the precursor node of n_1,the precondition P is True. Its propositional function is:

$$P = Pred(n_2, n_1) \wedge IsIn(RR(n_2), RAF(n_0)). \tag{12}$$

On this EP_i, if there is node n_3 which is the post-dominator of n_1 and is data dependent on its predecessor node n_0 with $RAF(n_0)$, and there is resource release

operation $RR(n_1)$ or $RR(n_3)$ corresponding $RAF(n_0)$, the post-conditions Q is True. Its propositional function is:

$$Q = PD(n_3, n_1) \wedge DD(n_3, n_1, RAF(n_0)) \wedge (IsIn(RAF(n_0), RR(n_3)) \cup IsIn(RAF(n_0), RR(n_1))). \tag{13}$$

We must consider the case that if n_0 can not deduce n_1 by the vulnerability syntax rules set S, the variables or functions defined on n_0 belong to redundant code.

3 Detection Algorithm

Based on this model, we design a static detection process for software vulnerability analysis, as shown in Fig. 1. It includes: basic information analysis and rules.

3.1 Basic Information Analysis

The basic information analysis module is used to generate and extract some basic static information from target program, as shown in Fig. 1.

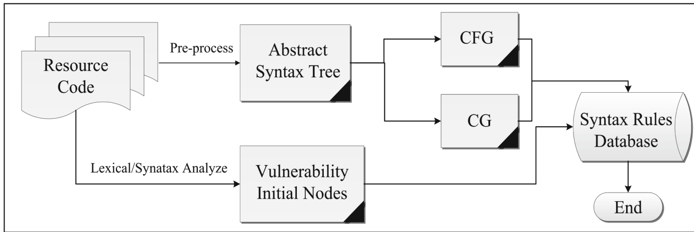

Fig. 1. Basic processing module flow chart

Firstly, use lexical and syntax analysis to extract vulnerability initial nodes set n_0 from target program source code. Secondly, use compiler front-end (such as GCC, java compiler) to generate abstract syntax tree AST, and construct CFG and CG (call graph).

3.2 Solution of Vulnerable Nodes Set *N*

Vulnerable nodes set N is a set of nodes which may contain vulnerabilities. Solving N is the coarse locating process of vulnerability analysis which we described in Sect. 3.2. Steps of Solving N are as follows:

Step 1. Search the program state space F starting from vulnerability initial nodes set n_0.

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

Algorithm 1 is shown as follow:

Algorithm 1. The algorithm of solving vulnerable nodes set N

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

Algorithm 2. The algorithm of discriminating vulnerability

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

3.3 Vulnerability Syntax Rules Database

Vulnerability rules database is a database which includes vulnerability syntax rules set S, vulnerability preconditions P and vulnerability post-conditions Q. It also contains some API functions, such as $ResourceAllocation FunctionList$, $FormatFunctionList$, $BufferFunctionList$, and so on.

According to Definitions 2–5 in Sect. 2.1, Definition 6 in Sect. 2.2 and Definitions 7, 12, 14 in Sect. 2.3, we summarize the vulnerability syntax rules for S, P, Q in Table 2.

4 Experiments and Evaluation

To evaluate the effectiveness of our approach, we proceed to evaluate our method by carrying on experiments from Sep., 2013 to May, 2016 on 4 open resource software and contrast of detection on CWE-476($Tomcat$4.0) of our method with FindBug 3.0.1.

We have verified and confirmed the vulnerabilities of 4 open source projects which was disclosed by NIST, shown as Table 3.

Table 2. Vulnerability syntax rules S, P, Q of the four types of vulnerability

Vulnerability types	Syntax rules for S, P, Q
Null Pointer Dereference	$S = Succ(n_1, n_0) \wedge DD(n1, n0, Pointer(v)) \wedge USE(Pointer(v), n_1)$ $P = DD(n_2, n_0, Pointer(v)) \wedge CD(n_2, n_1) \wedge CHECK(Pointer(v), n_2, NotNull)$ $Q = \emptyset$
Buffer Overflow	$S = DD(n_0, n_1, Buffer(v_1, n_0)) \wedge DEF(Buffer(v_1), n_1) \wedge Pred(n_1, n_0)$ $P = CD(n_2, n_0) \wedge PD(n_2, n_1) \wedge CHECK(buffer(v_1), input(v_2), n_1, Size)$ $\cup\, CHECK(buffer(v_1), input(v_2), n_0, Size)$ $Q = \emptyset$
Uncontrolled Format String	$S = Succ(n_1, n_0) \wedge DEF(v, n_1) \wedge DD(n_0, n_1, v)$ $P = CHECK(FF(n_0), n_0, Parameter)$ $Q = \emptyset$
Resource Flaws	$S = USE(RAF(n_0), n_1) \cup Succ(n_1, n_0) \wedge DD(n_1, n_0, RAF(n_0))$ $P = Pred(n_2, n_1) \wedge IsIn(RR(n_2), RAF(n_0))$ $Q = PD(n_3, n_1) \wedge DD(n_3, n_1, RAF(n_0)) \wedge (IsIn(RAF(n_0), RR(n_3))$ $\cup\, IsIn(RAF(n_0), RR(n_1)))$

Table 3. Verify and confirm the vulnerabilities disclosed by NIST

Vtype		NPD	BF	UFS	RRF
Chrome 5.0	NISTnum	0	1	0	5
	FOLBnum	0	1	0	5
Wireshark1.8	NISTnum	1	43	1	0
	FOLBnum	1	40	1	0
ABM1.0	NISTnum	0	58	8	9
	FOLBnum	0	49	8	8
Asterisk10.2	NISTnum	5	14	0	0
	FOLBnum	5	14	0	0
The accuracy of FOLBˆEPS		100%	88.9%	100%	92.8%

Notation: The accuracy of FOLBˆEPS = FOLBnum/NISTnum

As shown in Table 3, the open source projects that we test covering all types of vulnerabilities we describe in this paper. Results show that resource operations flaws, null pointer dereference and format string have high detection accuracy. And buffer overflow has low false positives rate and false negatives rate. Therefore, the method that we propose can be applied to detect most of vulnerabilities.

A Java project tomcat 4.0 is a real software system widely used as serverlet container, whose vulnerabilities are also disclosed by $NIST$. Here we use our method to detect $Null\ Pointer\ Dereference$($CWE$-476) and compare the result with that of FindBugs 3.0.1. Although FindBugs has been around for a long time, due to its universality and openness, we select the newer FindBugs 3.0.1 as the experiment tool. The result is shown in Table 4.

Table 4. Null Pointer Dereference (CWE-476) detection results contrast on Tomcat 4.0 between FindBugs 3.0.1 and FOLBˆEPS

Detection tools	Detection result	Confirmed	Confirmed as false positives number	Can't confirmed	False positives rate	Detection rate
FindBugs	36	2	22	12	61.1%	5.6%
FOLBˆEPS	42	13	21	8	50%	30.9%

The result shows that the detection rate of our method is higher than that of FindBugs and the False positives rate is also lower than that of FindBugs.

5 Conclusions and Future Work

In this paper, we proposed a static vulnerability detection model based on propositional function. Firstly, we defined and described the existing preconditions, characteristics and properties of vulnerabilities, and gave corresponding discriminant formula in terms of propositional function. Then constructed our detection model with a five-tuple. Then we designed a static detection process according to the new model, and used propositional function to described four types of disclosed software vulnerabilities in CWE. Finally, we carried out experiments, to verify that our model based on the propositional function realized more accurate description of vulnerability, improved detection rate and reduced the false alarm rate.

Acknowledgement. This work was supported in part 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). We also thank students of class 2013, 2014 and 2015 majoring information security in our university for their hard work of analysis and collecting vulnerability information of many $C/C{+}{+}/Java$ open sources. They filtered 7,168 vulnerabilities and achieved 1,761 vulnerabilities, patches and attribute information see the attachment1.

References

1. Martins, E., Morais, A., Cavalli, A.: Generating attack scenarios for the validation of security protocol implementations. In: The 2nd Brazilian Workshop on Systematic and Automated Software Testing (SBES 2008 -SAST), Brazil, October 2008
2. Du, W., Mathur, A.: Vulnerability testing of software system using fault injection. In: Proceeding of the International Conference on Dependable Systems and Networks (DSN 2000), Workshop on Dependability Versus Malicious Faults (2000)
3. Chen, Z.Q., Zhang, Y., Chen, Z.R.: A categorization framework for common vulnerabilities and exposures. Comput. J. Arch. **53**(5), 551–580 (2010)
4. Perl, H., Dechand, S., Smith, M., et al.: VCCFinder: finding potential vulnerabilities in open-source projects to assist code audits. In: ACM SIGSAC Conference on Computer and Communications Security, pp. 426–437. ACM (2015)
5. Czibula, G., Marian, Z., Czibula, I.G.: Software defect prediction using relational association rule mining. Inf. Sci. **264**(183), 260–278 (2014)
6. Li, P., Cui, B.J.: A comparative study on software vulnerability static analysis techniques and tools. In: IEEE International Conference on Information Theory and Information Security, pp. 521–524. IEEE Press, Beijing (2010)
7. Zeng, F.P., Chen, A.Z., Tao, X.: Study on software reliability design criteria based on defect patterns. In: Proceedings of the 8th International Conference on Reliability, Maintainability and Safety (ICRMS 2009), pp. 723–727. IEEE, Chengdu (2009)
8. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)
9. Wilander, J.: Modeling and visualizing security properties of code using dependence graphs. In: Proceedings of 5th Conference on Software Engineering Research and Practice in Sweden, pp. 65–74. ACM Press, Vasteras (2005)
10. Meland, P., Spampinato, D., Hagen, E., Baadshaug, E., Krister, K., Velle, K.: SeaMonster: providing tool support for security modeling. In: National Conference on Information Security, NISK 2008, November 2008
11. Quinlan, D., Panas, T.: Source code and binary analysis of software defects. In: 5th Annual Workshop on Cyber Security and Information Intelligence Challenges and Strategies, pp. 1–4. AMC Press, New York (2009)
12. Rough Auditing Tool for Security (RATS). https://code.google.com/p/rough-auditing-tool-for-security/. Accessed Jan 2015
13. PREfast Analysis Tool. https://msdn.microsoft.com/enus/library/ms933794.aspx. Accessed Jan 2015
14. Splint Annotation-Assisted Lightweight Static Checking. http://splint.org/. Accessed Jan 2015
15. Coverity Scan — Static Analysis. https://scan.coverity.com/. Accessed Aug 2015
16. Allence, F.E.: Control flow analysis. ACM SIGPLAN Not. **5**(7), 1–19 (1970)
17. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. **9**(3), 319–349 (1987)
18. Horwitz, S., Reps, T., Binkley, D.: Interprocedural slicing using dependence graphs. In: ACM/SIGPLAN88 Conference on Programming Language Design and Implementation, pp. 26–60. ACM Press, Atlanta (1988)