Open Source Software Vulnerability Propagation Analysis Algorithm based on Knowledge Graph

[1]Wenhui Hu
National Engineering Research Center for Software Engineering
Peking University
Beijing, Chinaline
huwenhui@pku.edu.cn

[1]Yu Wang
ChinaSoft International Co., Ltd
Beijing China
wangyu@chinasoftit.com

[*] Xueyang Liu
National Engineering Research Center for Software Engineering
Peking University
Beijing, Chinaline
liuxueyang@pku.edu.cn

Jinan Sun
National Engineering Research Center for Software Engineering
Peking University
Beijing, Chinaline
sjn@pku.edu.cn

Qing Gao
National Engineering Research Center for Software Engineering
Peking University
Beijing, Chinaline
gaoqing@pku.edu.cn

Yu Huang
National Engineering Research Center for Software Engineering
Peking University
Beijing, Chinaline
hy@pku.edu.cn

***Abstract*—With the extensive reuse of open source components, the scope of vulnerability impact will have cascade expansion. At the level of vulnerability data analysis, aiming at the vulnerability propagation problem, this thesis proposes a hierarchical propagation path search algorithm based on open source software vulnerability knowledge graph, at the same time, proposes a heuristic search strategy in both component layer and class layer to reduce the search space complexity, which is optimized from exponential down to polynomial. Furthermore, we propose the optimal blocking concept to represent the cost of repairing the entire propagation path, in order to measure the severity of the project's vulnerability. As for the purpose of providing effective suggestions on vulnerability repairing, we model the optimal blocking calculation as the network flow minimal separate problem, then calculate the network maximal flux to obtain the key dependencies with risks. Finally, multiple case studies with various vulnerability dependent risks show that the proposed algorithm can find software vulnerabilities affecting specific projects effectively.**

Keywords—open source software, vulnerability propagation analysis, knowledge graph, optimal blocking analysis, propagation difficulty, lazy strategy

I. INTRODUCTION

Defect management and code version management are two major activities in the life cycle management of open source software. In general, a vulnerability repair can be achieved with one or more code submissions. When vulnerability is disclosed, there will be a series of code created and submitted to fix the vulnerability. Organic integration and analysis of vulnerability information and vulnerability-related code information is of significant value for vulnerability data analysis. With the extensive reuse of open source components, the scope of vulnerability impact will have cascade expansion. For example, in Maven project[1], the POM file in org.wso2.carbon.security.policy defines a dependency on the third-party component org.apache.derby. The vulnerability CVE-2015-1832[2] exists in org.apache.derby, causing this project to have potential vulnerability threat.

From the above Fig. 1, it can be shown that when project versions and vulnerabilities are linked to form a knowledge graph[3], we can obtain the products and related Maven projects that are directly affected by the published vulnerability and have access to all projects affected by indirect vulnerability with the dependency inherent in the Maven project.

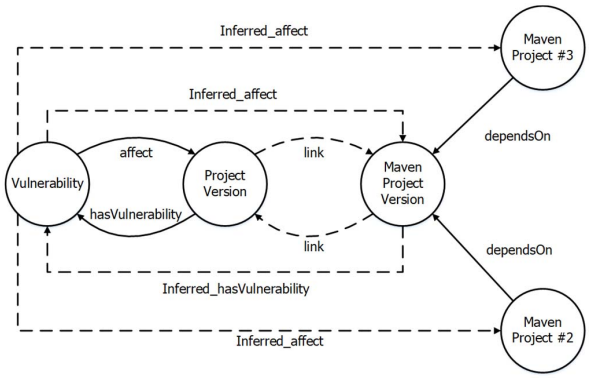

Fig. 1. Schematic diagram of direct and indirect vulnerability hazards to project

The path solving space is enormous in scale and the complexity increases exponentially. In this paper, a hierarchical propagation path search algorithm is proposed based on the source software vulnerability knowledge mapping; and the macro component search algorithm based on lazy strategy at the macro component level and a search algorithm for microcode class based on propagation difficulty at the micro code class level are proposed. After optimization of the algorithm, the complexity of search space is optimized from exponential level to polynomial level depending on the number of relationships. It has a vast improvement in the computational efficiency of vulnerability analysis. In order to further measure the vulnerability threat of a specific project, in this paper, the recommendation of using costs from blocking vulnerability propagation paths to support repair proposals is considered. The blocking operations are divided into two types. The first one is to replace a dependency of a code package or class in a project to an alternative component that is not affected by vulnerability. The second one is to update a version of a code class. Furthermore, an optimal blocking analysis algorithm for vulnerability propagation path is proposed in this paper, the optimal blocking calculation is modeled as the network flow minimum cut problem, and by solving the maximum flow algorithm, the key dependency with risk are obtained, which provides an effective support on the cost-benefit maximization for repairing critical vulnerability impacts.

II. RELATED RESEARCH

978-1-7281-5506-7/19/$31.00 ©2019 IEEE
DOI 10.1109/SmartCloud.2019.00030

At present, relevant research and papers have been made on the vulnerability analysis and detection in source code. Plate et al. [4] proposed a dynamic analysis technique. This technique carries out vulnerability impact analysis with the introduction of code changes based on security fixes. The method verifies whether vulnerable code has been executed in a given project depending on dynamic analysis. The limitation of the method is that its scope of application is limited to specific projects. It is unable to effectively analyze dependencies within and across more general projects. Its analysis result is also imperfect due to that the accuracy of analysis depends mainly on code coverage. Nguyen et al. [5] proposed an automation method to identify vulnerable code based on the old version of the software system. This method first scans each previous version of the code base to find fragments containing vulnerable code. Therefore, the greatest issue is that all vulnerable codes are based on old vulnerabilities. It lacks of sufficient flexibility when a new form of attack or code utilization is transformed. Mircea et al. [6] introduced a method in their vulnerability alert service (VAS) to inform users while vulnerability is reported by the software system. The operation of VAS mainly depends on open source dependency checking tool provided by OWASP (Dependency-Check[7]). VAS reports vulnerable items identified by OWASP tools. Therefore, it also lacks of support for transmission dependency analysis of vulnerable components.

Based on the dependencies between different projects, software vulnerabilities can be propagated incrementally. The quantity of vulnerability propagation paths also increases exponentially with the increasing number of projects. Targeted at the vast search space of vulnerability propagation path, an efficient search algorithm needs to be designed for the effective vulnerability risk analysis of the actual project. Given the cost of fixing vulnerability effect, effective repair guidance should be provided on the premise of maximizing cost-benefit.

III. Vulnerability Propagation Path Analysis

We need to analyze the path of vulnerability propagation to fully discuss the behavior of a vulnerability propagating through multiple hops (that is, through dependencies between components and jar packages) in multiple open source projects to analyze projects those are indirectly affected by vulnerabilities, or the direct or indirect impact of all vulnerabilities on a project[8].

For the purposes of discussion, vulnerabilities, projects, artifacts, and more microscopic code classes are considered as nodes in a directed graph, the direct impact of vulnerabilities on code classes, the direct dependence on code classes at the micro level, and the dependency between projects and components at the macro level are considered as a directed edge between the dependent party (image party) and the dependent party (image party). Then, the problem of vulnerability propagation path analysis turns to be the solution problem for the related paths on the directed graph[9].

Let us focus the goal on the analysis of a particular project $Proj_0$. This project can be either an existing open source project or a hypothetical closed source project that depends on a series of open source projects. For this project, we use component sets:

$$Comp_0 = \{comp_{0,1}, \dots comp_{0,|Comp_0|}\} \quad (1)$$

and key class sets

$$Class_0 = \{class_{0,1}, \dots, class_{0,|Class_0|}\} \quad (2)$$

It is then described, that is: $Proj_0 =< Comp_0, Class_0 >$. It can be extended. For all open source projects that may be involved in the problem domain $Proj_i$, we can all define its internal structure in a similar way:

$$Comp_i = \{comp_{i,1}, \dots comp_{i,|Comp_0|}\}$$
$$Class_i = \{class_{i,1}, \dots, class_{i,|Class_0|}\} \quad (3)$$
$$Proj_i =< Comp_i, Class_i >$$

For target project $Proj_0$, we expect to solve all vulnerability sets V that directly or indirectly affect the project through other projects$\{Proj_i\}$. The most direct idea is to solve the problem through reverse breadth-first searching. However, based on the foregoing analysis, we learnt that the number of open source projects covered by only two hops has reached thousands of orders of magnitude. This allows the time-space complexity required for analysis algorithms only at the macro-component level to rapidly expand to full space (that is, the whole open source software space). The time and space complexity of such algorithms will be unacceptable.

The breadth-first search optimization is mainly based on the following methods.

A. Macro component search based on lazy strategy

The vulnerability propagation path is analyzed first at component level. Apparently, a path that propagates vulnerabilities through class calls must correspond to a path that propagates vulnerabilities through component dependencies. This is different reflection of the same path at the micro and macro levels. Therefore, preliminary analysis of vulnerability propagation path can be carried out based on component level, and then detailed discussion can be made on the propagation path of a specific vulnerability when making a specific class call. This is conducive to minimize searches for propagation paths that obviously do not exist[10].

The number of propagation paths is still exponential at the component level. One optimization idea is to transform the problem into solving the dependency and direct propagation relationships that may exist in the propagation path instead of focusing on the specific set of paths[11]. That is, the component set $Comp = \cup Comp_{0\sim N}$and vulnerability set V in all the projects $proj_0 \sim proj_N$, are considered as node set Ns of directed digraph, and dependency relationship between components and the direct propagation relationship between components and a vulnerability are considered as set of connected edges of directed digraph E_s, finally, N_s and subset N_0 and E_s and subset E_0 is solved. For all $n \in N_0$, there is at least one vulnerability propagation path p, and $n \in Node(p)$ is satisfied. $Node(p)$ represents a set of nodes for path p. For all $e \in E_0$, there is at least one vulnerability propagation path p, and $e \in Edge(p)$ is satisfied. $Edge(p)$ represents a set of connected edges for path p. Through point set N_0 and edge set E_0, all potential dependency propagation paths in the project can be described, and it can also be understood as to remove the invalid paths that occupy the vast majority of the original total graph.

Component level macro search is based on the idea of depth first. It is noteworthy that the lazy label method is used to avoid multiple extensions to the same component. That is,

after completing the search for a component, by marking whether the component can eventually be propagated directly/indirectly by vulnerability to avoid repeated searches for trying component extension next time.

The overall algorithm flow is as follows:

Step 1: Traversal the components in the component set Comp_0 of the target project in turn, and recursively perform depth-first search.

Step 2: Traversal all dependent components $c' \sim dep_{c,1\sim|dep_c|}$ in turn for current traversal component c.

Step 3: Check the label of component c'. If the label of component c' is true, it indicates that at least one path is eventually propagated by a vulnerability through component c, and the dependence of component c to c'must form part of a propagation path. Therefore, component c can be added to the set N_0, and dependency $c \rightarrow c'$ can be added to the set E_0.

Step 4: If the label of any dependency is true, mark the current component as true. Otherwise, mark it as untrue.

Through the above algorithm, the final point set N_0 and edge set E_0 obtained can describe all potential dependency propagation paths in the project. Compared with the general depth-first search method for finding all paths, the time complexity is reduced from exponential to $O(|N_S| + |E_S|)$, and spatial complexity is $O(|N_S| + |E_S|)$.

B. Microcode class search based on propagation difficulty

Based on the macro search reduced dependency graph, we can further refine the component dependency path to code class invocation path. The path space is still massive (exponential); therefore, it is a better approach to obtain a certain number of paths through the difficulty of calling paths. When discussing possible paths, it is a complex situation in which the length is proportional to the difficulty of the call instead of forming a path of length 1 through a single class call. It can obtain the path with lower communication difficulty first to avoid meaningless growth of search space[12].

Sort the currently expandable nodes according to the difficulty of communication from low to high. For every expansion, priority should be given to expanding the nodes with lower communication difficulty. Difficulty of propagation is synthetically generated based on the parameters of the class and the project, such as the number of calls, versions, classes, packages and etc. In other words, the vulnerability propagation path is relatively simple for projects with low dissemination difficulty. Expanding such nodes first helps to generate shorter and simpler vulnerability propagation paths first.

The overall algorithm flow is as follows:

Step 1: Construct set of expanded nodes S, and S is maintained with priority queue. The key word is the difficulty of propagation based on code class and project parameters $key = f(class_i)$. There are only target project $proj_0$ in S at the beginning of the algorithm.

Step 2: Construct vulnerability set V. V is an empty set at the beginning of the algorithm.

Step 3: Take the code class node with the highest priority $class_i$ from S. When S is empty, end of the algorithm. Otherwise continue with step 4.

Step 4: To expand $class_i$. Add all code classes called by $class_i$ to collection S, and calculate the difficulty of calling these code classes $f(class_i)$. In order to calculate the specific propagation path, minimum cost of maintaining each code class $cost_j$ and $path_i$ corresponding to minimum cost, check whether $cost_i + f(class_j) < cost_j$ is valid when using $class_i$ to expand $class_j$. If it is valid, then: $cost_j = cost_i + f(class_j)$, $path_j = path_i + \{j\}$. However, only if $cost_j$ is within a certain range, expanding can be carried out.

Step 5: Traversal all the vulnerabilities V_{class_i} that directly affect the code class $class_i$, add vulnerabilities to $V$and mark the propagation path of the vulnerability.

Step 6: Update Set S, that is, to maintain with priority queue. Jump to step 3.

Through the above algorithm, the propagation path searched at the micro level is saved at the final set V, and its specific path is also marked in the process of algorithm.

In practical terms, this algorithm is similar to the Shortest Path Faster algorithm (SPFA) for solving the shortest path. Compared with general breadth-first search, prioritize the expansion of project nodes with low complexity can reduce the exponential growth of time complexity to the $O(\lambda M)$ level, which is proportional to the number of dependencies. In the general sparse graph, λ can be regarded as a small constant, and the worst time complexity in dense graph is also polynomial-level $O(NM)$. It can be said that the breadth-first search has been greatly improved.

IV. Optimal Blocking Analysis of Vulnerability Propagation Path

Various code packages and code classes contained in a target project may be affected by multiple vulnerabilities. The cost of blocking the path of vulnerability propagation can be considered to measure the threat to the project. The blocking operations are divided into two types: the first one is to replace a dependency of a code package or class in a project to an alternative component that is not affected by vulnerabilities. The second one is to solve the problem that the dependent component is affected by vulnerabilities through version updates to a code class.

Optimal blocking analysis of vulnerability propagation path aims to see an optimal scheme for the above objectives to complete vulnerability repair for target project with minimal blocking operation[13]. The cost of the optimal scheme can measure the extent to which the project is currently affected by vulnerability to a certain extent, that is, risk level.

In the optimal blocking analysis of vulnerability propagation path, the problem is considered as the minimal cut problem on the directed graph, that is, vulnerabilities are considered as source points on the directed graph, and the set of source points $S = \{s_1, s_2, \dots, s_{|S|}\}$ represents all vulnerabilities that can affect the target project. Specific code classes in each project are considered as intermediate nodes, represented by collection $M = \{m_1, m_2, \dots, m_{|M|}\}$. All code classes contained in the target project belong to set M, that is, $T = \{t_1, t_2, \dots, t_{|T|}\} \subseteq M$. Relationships between classes, projects and components that directly affect vulnerabilities, and dependencies between classes are considered as edges between nodes $E = \{e_1, e_2, \dots, e_{|E|}\}$.

For two different blocking methods, we can abstract our problems as: select minimal edge sets as possible $F \subseteq M * M$ to enable any path from the source set S to the target set M contains at least one edge in the edge set F. Or, select minimal sets as possible $G \subseteq M + T$ to enable any path from the source set S to the target set M contains at least one node in point set G.

Both of the above problems can be transformed into minimal cut problems on the directed graph, and the optimal solution is solved with maximum flow algorithm for network flows. For different blocking methods, different splitting points should be used to refine the drawings. The first blocking method is introduced, that is, network construction under the method of replacing a dependency of a code package or class in a project to an alternative component that is not affected by vulnerabilities:

Step1: Construct total source point S_0

Introduce one edge with positive infinite capacity from S_0 to all defect nodes $s \in S$. The reason for setting its capacity to infinite is to avoid full capacity on that edge and to become part of the minimum cut. The edge is virtual, and this point represents the set of vulnerabilities that an attacker may exploit to a certain extent in order to merge the set of source points.

Step2: Constructing direct propagation traffic

Traversal all defect nodes $s \in S$, construct an edge with positive and infinite capacity $< s, n, +\infty >$, $n \in N_s$ for all code class nodes that are directly affected $N_s \subseteq M + T$. The reason for setting its capacity to infinite is that removing its dependence on a defect cannot block a class, and this edge needs to be avoided as part of the smallest cut.

Step3: Construct indirect communication traffic

For existing code classes depending on $c_1 \rightarrow c_2$, that is, code class c_2 depends on code class c_1, for the corresponding code class nodes n_{c_1} and n_{c_2}, construct an edge with capacity of $1 < n_{c_1}, n_{c_2}, 1 >$. The set E' composed of these edges actually corresponds to the set E on the mailbox map. From the perspective of the minimum cut algorithm, if the edge is a full flow, then the edge is replaced by a dependency, otherwise, the dependency remains unchanged. Therefore, by setting the capacity to 1, the optimal scheme E_{best} of the final minimum cut must belong to E'. That is, for each edge $< n_{c_1}, n_{c_2}, 1 > \in E_{best}$, and dependencies from c_2 to c_1 is replaced to complete optimal block.

Ste4: Construct convergence point T_0

Introduce an edge of infinite capacity to T_0 from code class nodes $t \in T$ of all target projects. The reason for setting its capacity to infinite is to avoid full capacity on that edge and to become part of the minimum cut. Since the edge is virtual, and this point represents the whole target project to a certain extent in order to merge the collection of convergence points.

The second blocking method is to avoid further vulnerability propagation from the component through version updates to a component. From the perspective of network construction, in fact, the minimum cut algorithm for edges is transformed into the minimum cut algorithm for points. Therefore, the node can be dismantled based on the previous network construction method, and the specific network construction can be described as:

Step1: Construct total source point S_0

Introduce one edge with positive infinite capacity from S_0 to all defect nodes $s \in S$. The reason for setting its capacity to infinite is to avoid full capacity on that edge and to become part of the minimum cut. The edge is virtual, and this point represents the set of vulnerabilities that an attacker may exploit to a certain extent in order to merge the set of source points.

Step2: Constructing direct propagation traffic

Traversal all defect nodes $s \in S$, construct an edge with positive and infinite capacity $< s, n', +\infty >$, $n' \in N_s'$ for all component nodes that are directly affected $N_s' \subseteq M + T$. The reason for setting its capacity to infinite is that removing its dependence on a defect cannot block a class, and this edge needs to be avoided as part of the smallest cut.

Step3: Construct indirect communication traffic

For existing components depending on $c_1' \rightarrow c_2'$, that is, component c_2' depends on component c_1', for the corresponding code class nodes $n'_{c_1'}$ and $n'_{c_2'}$, construct an edge with capacity of $< n'_{c_1'}, n'_{c_2'}, +\infty >$ to avoid minimum cut to deal with the dependencies between components.

Step4: Construct dismantling point flow: split all component nodes $n' \in M$ into two nodes n'_p, n'_q. Node n'_p accepts all incoming edges of the original component node n', and node n'_q accepts all outgoing edges of the original component node n', and an edge with capacity of 1 is introduced from n'_p. Throughout the journey, only the split edges are of limited capacity, so the edges in the minimum cut must come from the set of splitting edges. From the perspective of the minimum cut algorithm, if the edge is a full flow, it indicates to fix and update the component, otherwise, the component remains unchanged.

Step5: Construct convergence point T_0

Introduce an edge of infinite capacity to T_0 from code class nodes $t' \in T$ of all target projects. The reason for setting its capacity to infinite is to avoid full capacity on that edge and to become part of the minimum cut. Since the edge is virtual, and this point represents the whole target project to a certain extent in order to merge the collection of convergence points.

After the construction of network flow, the maximum flow of the network can be directly calculated through maximum flow algorithm, and the optimal blocking number of vulnerability propagation paths in the network can be obtained directly through the theorem of maximum flow and minimum cut. By checking whether the traffic on each side of the network is the same as its capacity, the minimum cut scheme of the network can be further obtained. If they are the same, it means that the edge is part of the minimum cut.

V. Analysis of Experimental Results

CVE vulnerability information and Maven project information are imported into knowledge map, the relevant statistical information of CVE vulnerability and Maven project are as follows:

TABLE I. CVE VULNERABILITY AND MAVEN PROJECT STATISTICAL INFORMATION

Item	number
Number of all entities	335,326

Number of all relationships	2,252,560
Number of CVE vulnerabilities	97,901
Number of product versions affected by vulnerabilities	186,352
Product quantity	34,192
Number of Maven project versions	2,383,247
Number of Maven projects	221,994

In the following three aspects, the macro level, micro level and the optimal blocking of vulnerability propagation path, the corresponding experimental analysis of open source project components affected by vulnerabilities is carried out.

A. *Analysis of Vulnerability Propagation Effect Based on macro components*

The analysis show that a total of 1,584 Maven projects were affected by the published CVE vulnerability, and a total of 22,613 project versions were affected by the vulnerability. Once the construction of software vulnerability knowledge map is completed, through algorithm analysis, it can be seen intuitively that which projects and their corresponding versions are directly affected by open vulnerabilities, for example, which open source projects are directly affected by published vulnerabilities, which vulnerabilities affect a project at the same time, and which vulnerabilities affect multiple releases of a product.

However, open source projects have multiple related dependency libraries and third-party components in a direct or indirect way. When the libraries and components are faced with security vulnerabilities, all projects that directly or indirectly depend on this library or component are potentially threatened by vulnerabilities. Therefore, the vulnerability propagation analysis based on open source project indirect dependency transfer is crucial to evaluate the extent of vulnerability and its propagation scope etc.

There is a progressive hierarchy of dependencies among open source projects. With the expansion of dependency hierarchies among projects, the number of indirect dependencies related to projects increases step by step, and the number of potential vulnerabilities and hazards will also increase gradually.

The time-space complexity of the analysis algorithm at the macro-component level increases exponentially and rapidly expands to the full space (that is, whole open source software space) with the gradual increase of transfer hierarchy. In this experiment, on the basis of lazy strategy-based macro component search algorithm, different examples are provided to illustrate how to identify potential risk components or projects by dependency transfer. Such as the analysis example of the indirect impact of vulnerabilities based on component dependencies in TABLE II, we can see that, some open source projects or components have been exposed to one or more security vulnerabilities. For example, as part of the Eclipse Foundation, Jetty is an open source servlet container, providing a runtime environment for Java-based web containers, such as JSPs and servlets. We identified that, in jetty 7.0.0, there are five CVE vulnerabilities: “CVE-2009-1523”, “CVE-2009-4609”, “CVE-2009-4610”, “CVE-2009-4611” and “CVE-2011-4461”. In level 1, 540 projects have direct vulnerability risks. In level 2 and level 3, there are 169 and 260 dependent components, respectively.

TABLE II. EXAMPLE OF VULNERABILITY INDIRECT IMPACT ANALYSIS BASED ON COMPONENT DEPENDENCIES

#	Component Name	# of Vulnerabilities	CVE-IDs	Number of dependents based on level(L)		
				L1	*L2*	*L3*
P1	wicket 7.0.0	3	CVE-2016-6806 CVE-2014-7808 CVE-2014-3526	198	6	13
P2	groovy 2.3.0	1	CVE-2015-3253	120	250	300
P3	jetty 7.0.0	5	CVE-2009-1523 CVE-2009-4609 CVE-2009-4610 CVE-2009-4611 CVE-2011-4461	540	169	260
P4	log4j 2.0	1	CVE-2017-5645	139	890	3028
P5	tomcat 7.0	4	CVE-2017-12615 CVE-2016-1240 CVE-2016-9774 CVE-2016-9775	1289	3415	5115

For some very frequently used components, the potential damage caused is quite severe in case the security vulnerability is discovered. Such as Log4j. In Java project, Log4j has a high use frequency. For example, for a CVE vulnerability “CVE-2017-5645” found in the log4j 2.0 version, in Level 1, it has 139 affected projects; in Level, it has 890 affected projects; and even more surprisingly, in level, it has 3, 028 affected projects. We can imagine that with the deeper level, projects exposed to the potential impact of vulnerabilities will increase exponentially.

B. *Analysis of vulnerability propagation effect based on microscopic code class*

Generally, a release version of an open source project contains dozens of running-dependent code packages. Each code package contains a series of relatively independent functional codes. In each separate code package, hundreds of code classes are included. There may be only a few or a dozen code classes affected by vulnerabilities. Therefore, locating vulnerabilities at the micro code class level enables vulnerability propagation analysis more accurately.

In this section, firstly through the analysis of open source project netty, we attempt to find all the key code classes affected by vulnerabilities in the open source project netty. The basic information of the open source project netty is shown in TABLE III below.

TABLE III. OPEN SOURCE PROJECT NETTY CODE STATISTICS

Item	number
Number of found vulnerabilities	4
Number of code packages	32
Number of code submissions	8984
Number of code classes	2412
Number of released versions (after 4.0.0)	105

The vulnerability description includes not only the product version information affected and the impact information caused by the vulnerability, but also the key information related to the code. The semantic similarity between vulnerability description and submission log and submission code is used to associate vulnerabilities with a code submission.

Association results are shown in Table IV below. In the open source project netty, there are 8, 984 code submissions, 2, 412 code classes and 105 releases. Four CVE vulnerabilities have been identified. 24 classes among 2, 412 code classes are directly affected by vulnerabilities. Evidently, locating code

classes affected by vulnerabilities assists us to analyze vulnerability propagation more accurately.

TABLE IV. OPEN SOURCE PROJECT NETTY VULNERABILITY AND CODE LINK ASSOCIATION RESULTS

CVE-Name	Fix-Commit-ID	Number of code classes directly affecting
CVE-2016-4970	"bc8291c"	1
CVE-2015-2156	"8005554"	18
CVE-2014-3488	"2fa9400"	1
CVE-2014-0193	"93fab1d"	4

At the same time, we can also give the key code classes of open source project netty affected by vulnerabilities.

In the above experiments, indirect dependencies are based primarily on the dependencies of component or project version in Maven project configuration. Normally, a project usually contains dozens or hundreds of different components or jar packages for large open source projects. The code that causes the vulnerability is usually located in a specific java file, that is, in a particular code class.

On the basis of microscopic code class search algorithm based on difficulty of propagation, the expansion nodes in this experiment are sorted according to the difficulty of propagation from low to high. Priority is given to the expansion node with lower difficulty of propagation for each expansion. In this way, it is conducive to first generate a shorter and simpler vulnerability propagation path. Hence, the problem of huge complexity space can be solved and vulnerability propagation path can be identified effectively.

As shown in schematic diagram Fig.2 below, after the vulnerability is accurately located in the code class, microscopic code class search algorithm based on difficulty of propagation is able to identify whether code classes that contain vulnerabilities are introduced in the project.

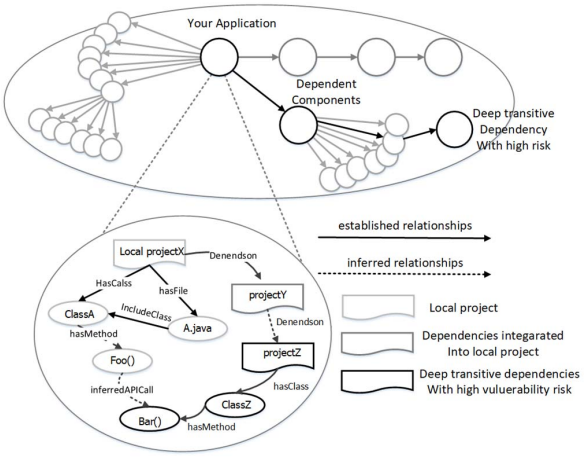

Fig. 2. Schematic diagram of vulnerability propagation impact analysis based on microscopic code class

Let us still take the open source project netty as the experimental case. The experiment results are based on the 4 vulnerabilities and 24 vulnerability code classes in the list of key code classes affected by the vulnerability in the open source project netty. In TABLE V, a list of related class files that indirectly refer to code classes containing vulnerabilities is displayed.

TABLE V. EXAMPLE OF REFERENCE TO CLASSES CONTAINING VULNERABILITIES

Project Name	Indirect references to classes containing vulnerability classes	Vulnerability Code Class
com.twitter.finaglenetty	com.twitter.finagle.http.netty3.SameSiteSupportingCookie	org.jboss.netty.handler.codec.http.DefaultCookie
	com.twitter.finagle.http.netty3.SameSiteSupportingCookie	org.jboss.netty.handler.codec.http.Cookie
	com.twitter.finagle.http.netty3.SameSiteSupportingCookie.compareTo.c	
org.apache.activemq.artemiscommons	org.apache.activemq.transport.netty.NettyWSTransport	io.netty.handler.codec.http.websocketx.ContinuationWebSocketFrame
	org.apache.activemq.transport.netty.NettyWebSocketTransportHandler	
	org.apache.activemq.transport.netty.NettyWebSocketTransportHandler.channelRead0.continuationFrame	
org.apache.hadoop:hadoopmapreduce	org.apache.hadoop.mapred.ShuffleHandler	org.jboss.netty.handler.ssl.SslHandler
org.apache.hadoop:hadoop-mapreduce	org.apache.hadoop.mapred.ShuffleMetrics	
	org.apache.hadoop.mapred.ReduceMapFileCount	
	……	
	org.apache.hadoop.mapred.AttemptPathIdentifier	
	org.apache.hadoop.mapred.LastSocketAddress	org.jboss.netty.handler.codec.http.HttpRequest
	org.apache.hadoop.mapred.ShuffleMetrics	
	org.apache.hadoop.mapred.ReduceMapFileCount	
	……	
	org.apache.hadoop.mapred.AttemptPathIdentifier	
…	……	……

C. *Analysis of Optimal Blocking Analysis of Vulnerability Propagation Path*

In order to measure and analyze the degree of threat to a potential target project, based on the optimal blocking analysis algorithm of propagation path, relevant comparative experiments were carried out in this paper to calculate the critical vulnerability code class nodes in the propagation path[14]. By replacing the code class or updating the version of the code class, the goal of quickly blocking the transmission path and solving the potential risks of the project can be achieved.

Based on the optimal blocking analysis algorithm for vulnerability propagation path, in the following TABLE VI, the number of paths included by the key nodes of the propagation path between vulnerability classes in other projects and netty projects is listed. As shown in the table, node "org.jboss.netty.handler.codec.http.HttpRequest" has the most number of propagation paths, reaching 88. Therefore, by fixing the vulnerability of the node, potential risks from other projects can be minimized.

TABLE VI. EXAMPLES OF KEY NODES BASED ON OPTIMAL BLOCKING ALGORITHM

Vulnerability code class key node	Number of paths through nodes
org.jboss.netty.handler.codec.http.DefaultCookie	1
org.jboss.netty.handler.codec.http.Cookie	2

org.jboss.netty.handler.codec.http.HttpRequestEncoder	2
org.jboss.netty.handler.codec.http.HttpResponseDecoder	2
io.netty.handler.codec.http.websocketx.ContinuationWebSocketFrame	3
org.jboss.netty.handler.codec.http.HttpRequestDecoder	4
org.jboss.netty.handler.codec.http.HttpResponseEncoder	4
org.jboss.netty.handler.codec.http.HttpResponseStatus	17
org.jboss.netty.handler.ssl.SslHandler	27
org.jboss.netty.handler.codec.http.HttpResponse	67
org.jboss.netty.handler.codec.http.HttpRequest	88

Optimal blocking analysis of vulnerability propagation path aims to see an optimal scheme for the above objectives to complete vulnerability repair for target project with minimal blocking operation. The cost of the optimal scheme can measure the extent to which the project is currently affected by vulnerability to a certain extent, that is, risk level.

VI. CONCLUSION

Targeted at the issues such as enormous scale and high complexity of path searching, in the process of further analysis on the impact of vulnerability propagation, the macro construction search algorithm based on lazy strategy and micro code class search algorithm based on propagation difficulty are proposed in this paper. After optimization of the algorithm, the complexity of search space is optimized from exponential level to polynomial level depending on the number of relationships. Furthermore, the optimal blocking concept is proposed to represent the cost of repairing the propagation path of all vulnerabilities, so the calculation of optimal blocking is modeled as the minimum cut problem of network flow. It solves the critical dependencies with risk by maximum flow algorithm, and providing an effective support for repairing the impact of key vulnerabilities to maximize cost-benefit. Based on the experimental analysis of various vulnerability dependence risk types, it indicates that the proposed algorithm can effectively identify software vulnerabilities that may affect a specific project.

ACKNOWLEDGMENT

This work is supported by the National Key Research and Development Program of China (No. 2017YFB1400805)

REFERENCES

[1] Maven Central Repository [EB/OL]. http://central.maven.org/maven2/, 2018-03-01.

[2] Common Vulnerabilities and Exposures (CVE) [EB/OL]. https://cve.mitre.org, 2018-03-01.

[3] Han X, Sun L. Context-Sensitive Inference Rule Discovery: A Graph-Based Method; proceedings of the Proceedings of COLING 2016, the 26th International Conference on Computational Linguistics: Technical Papers, F, 2016 [C].

[4] Plate H, Ponta S E, Sabetta A. Impact assessment for vulnerabilities in open-source software libraries; proceedings of the Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on, F, 2015 [C]. IEEE.

[5] Nguyen V H, Dashevskyi S, Massacci F. An automatic method for assessing the versions affected by a vulnerability [J]. Empirical Software Engineering, 2016,21(6): 2268-2297.

[6] Cadariu M, Bouwers E, Visser J, et al. Tracking known security vulnerabilities in proprietary software systems; proceedings of the Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on, F, 2015 [C]. IEEE.

[7] OWASP Dependency Check [EB/OL]. https://www.owasp.org/index.php/OWASP_Dependency_Check, 2018-03-01.

[8] Wang Q, Wang B, Guo L. Knowledge Base Completion Using Embeddings and Rules; proceedings of the IJCAI, F, 2015 [C].

[9] Kotnis B, Bansal P, Talukdar P P. Knowledge base inference using bridging entities; proceedings of the Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing, F, 2015 [C].

[10] Wang Q, Liu J, Luo Y, et al. Knowledge base completion via coupled path ranking; proceedings of the Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), F, 2016 [C].

[11] Zhou Z-H. Ensemble methods: foundations and algorithms [M]. Chapman and Hall/CRC, 2012.

[12] Zili Shao, Meng Wang, Ying Chen, Chun Xue, Meikang Qiu, Laurence T. Yang, Edwin H. –M. Sha, Real-Time Dynamic Voltage Loop Scheduling for Multi-Core Embedded Systems, IEEE Transactions on Circuits and Systems II, Volume: 54, Issue: 5, May 2007, 445 – 449

[13] Jiayin Li, Meikang Qiu, Jianwei Niu, Wenzhong Gao, Ziliang Zong, Xiao Qin, Feedback Dynamic Algorithms for Preemptable Job Scheduling in Cloud Systems, 2010 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology, 253-259