An Efficient Method for Scheduling Massive Vulnerability Scanning Plug-ins

Yulong Wang, Nan Li
State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, Beijing, China
Email: {wyl, linanster}@bupt.edu.cn

***Abstract*—More and more security vulnerabilities were found in network softwares nowadays, making network security assessment one of the most important tasks for IT administrators. Vulnerability scanner is the key application for fulfilling such tasks. However, large numbers of vulnerabilities result in even larger number of vulnerability plug-ins including common plug-ins and specific plug-ins, which may involve complex dependencies. Therefore, how to schedule such large number of plug-ins in an efficient manner is a key problem for improving the performance of vulnerability scanners. We analyze the current algorithms and find that they doesn't take the dependencies into consideration or doesn't handle it properly, which would waste a considerable CPU time for scanning. This paper proposes an efficient plug-in scheduling algorithm based on DAG graph. We formalize plug-in scheduling as a tree-like topological sorting problem using DAG theory, in which multi-thread is treated as task lines and all plug-ins are deployed on the task lines. Each task line is occupied by the plug-ins for a period of executing time and waiting time. By constructing the DAG graph of all plug-ins and computing their "height" value, sorting the plug-ins and aligning them to a linked list for scheduling, we solve the plug-in dependency problem properly, therefore eliminate the possibilities that non-ready plug-ins being scheduled to execute. We carry out experiments to validate the effectiveness of our algorithm.**

***Index Terms*—security vulnerability, plug-in scheduling, plug-in dependency, topological sorting**

I. Introduction

A. Security Issue

INTERNET changes information usage mode and brings up communication revolution era. No doubt that it is opening up unlimited prospects for communication applications [1]–[3]. However at the same time, due to the openness of Internet, it is facing more and more internal and external security threats. As a result, network attacks, information theft and other security issues have become increasingly prominent.

Manuscript received March 19, 2013; revised April 27, 2013; accepted May 13, 2013. © 2005 IEEE.
This work was supported in part by Youth Scientific Research and Innovation Plan of Beijing University of Posts and Telecommunications(GrantNo. 2013RC1101), the Disciplinary Joint Construction Project of the Beijing Municipal Commission of Education, the Innovative Research Groups of the National Natural Science Foundation of China (GrantNo. 61121061), the Independent Research Project for the Base (GrantNo. N2012002) and the Important National Science & Technology Specific Projects:Next-Generation Broadband Wireless Mobile Communications Network (GrantNo. 2010ZX03004-001-01).

According to 2012 Global Security Report [4] released by Trustwave, a leading provider of on-demand data security and payment card industry compliance management solutions for businesses and organizations throughout the world, network security situation is quite not optimistic and Internet is facing a wide variety of network security issues compared with past years, ranging from the theft of personally identifiable information to sensitive government documents or credit card data,etc. Cyber criminals target many diverse organizations.

Except for Trustwave, many other security organizations and institutions have issued safety warnings on Internet. For example, CWE/SANS [5] tops the web application security on the 20 Global Security Risk Ranking List. The report of Network Monitoring for Web-Based Threats [6] release by Computer Emergency Response Team (CERT) also points out that web-based vulnerabilities have made the web into a wonderfully powerful yet very dangerous place. Thus, security issues have become increasingly important.

B. Vulnerability Scanning Plug-in Scheduling Issue

Vulnerability scanning is a technology for identifying the possible vulnerabilities in the target network using remote detection. As new vulnerabilities are found from time to time, a vulnerability scanner needs to frequently update their vulnerability knowledge base to improve their scanning capability. In order to update on the fly, most vulnerability scanners adopt plug-in mechanism [7]. Network security vulnerabilities have increased to a large number and continued to grow rapidly. Take Nessus [8] and OpenVAS [9] for example, up to April of 2013 the numbers of plug-in in their libraries have been updated to more than 53000 and 3000 respectively. What's more, plug-ins may depend on other plug-ins. So, there may exist complex relationships between the plug-ins. Therefore, how to design a plug-in scheduling algorithm, which make the scanner to schedule plug-ins just in time so as to reduce the delay introduced by dependency restrictions, is a very valuable problem to solve.

C. The Structure of the Paper

This paper firstly analyzes exiting scanning plug-in algorithms, and find out that their poor performance is caused by the plug-in dependency restrictions. Secondly,

doi:10.4304/jsw.8.11.2761-2769

using DAG graph the paper convert the plug-in scheduling problem to a task scheduling model which implicitly contains the plug-in dependency restrictions. Afterwards, the paper proposes an efficient plug-in scheduling algorithm based on the model. In the end, through experiments the algorithm is verified to be able to solve the plug-in dependency problem properly and improve the performance of vulnerability scanning.

The main contributions of the paper are summarized as follows:

1) Proposes an efficient plug-in scheduling algorithm. Compared with existing vulnerability scanning plug-in scheduling algorithms, the proposed algorithm solves the plug-in dependency problem, which is the key point for improving vulnerability scanning performance.
2) Construct the dependent task scheduling model, and successfully convert the plug-in schedule problem to the model, which greatly simplifies the analysis of plug-in scheduling problem.
3) Make a profound comparative research of exiting plug-in scheduling algorithms and summarizes their advantages and disadvantages, which is helpful for designing a better plug-in scheduling algorithm.

II. PREPARATORY WORK

A. Plug-in Scheduling Requirement

In order to avoid unnecessary scanning operation, some plug-in may reuse the scanning result of other plug-ins. We call these plug-ins dependent plug-ins. Some plug-ins never need to reuse the scanning result of other plug-ins. We call these plug-ins independent plug-ins.

The executing sequences of plug-ins can be classified into two types: sequential and concurrent. Sequential execution means there is a strict order to execute since dependent plug-ins are involved. For example, plug-in p_a depends on plug-in p_b, then the two plug-ins should execute in a fixed order. In other words, p_a's executing prerequisite is the present of the scanning result from p_b. If p_b is not completely over, p_a should not be scheduled since it cannot execute. The plug-in dependencies requires that corresponding plug-ins must be executed in a particular order. Concurrent execution means there is no such dependency restriction on plug-ins, so no need to determine the proper order to schedule each plug-in. For example, plug-in p_a and p_b doesn't depend on each other, so they can be scheduled in any order without affecting the scanning performance. In other words, p_a has no executing prerequisite of the scanning result of p_b, if p_b is not completely over while p_a is scheduled, p_a can execute, and vice versa. In the later case, scanners can use multi-thread to realize concurrent executing of plug-ins for improving the throughput of vulnerability scanning.

Vulnerability scanning plug-ins may have complex dependent relationships. As Fig.1 shows, plug-in p_4 depends on p_7 and p_8, p_5 depends on p_7, p_6 depends on p_8. With regard to running sequence, p_4 and p_5 is able to execute only after the execution of p_7, p_6 is able to execute only when the execution of p_8 is over. On the contrast, p_4, p_5 and p_6 have no dependent relationships with one another, so these three plug-ins are able to execute simultaneously once p_7 and p_8 complete their scanning.

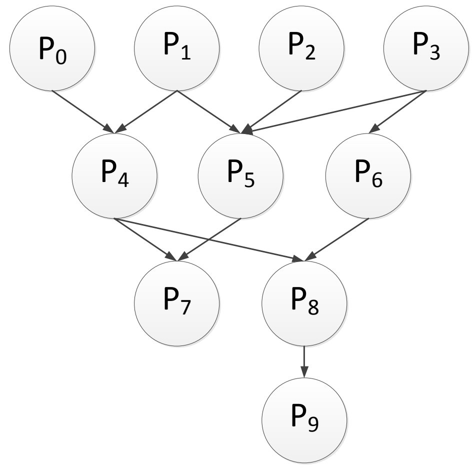

Fig. 1. The dependent relationships among vulnerability scanning plug-ins

B. Problem Description

Based on the above analysis on the requirement, we formalize the plug-in scheduling problem as follows: Given a set of plug-ins $P = \{p_1, p_2, \dots, p_n\}$. The maximum number of concurrent threads is denoted as m ($m \ll n$). Let $c_i(k)$ represent the i_{th} plug-in's pure executing time[1] on the k_{th} task line. Since packet latency doesn't affect scheduling decision and packet delay variation is hard to predict, we ignore the time spent on packet latency, thus $c_i(k)$ could be regard as unchanged for each plug-in. Let $w_i(k)$ represent the i_{th} plug-in's waiting time on the k_{th} task line, which starts just after the $(i-1)_{th}$ plug-in's execution is over and ends before the i_{th} plug-in's beginning of execution. Then, $w_i(k)$ must be one of the three values. If $w_i(k)$ is equal to zero, then the i_{th} plug-in doesn't need to wait. If $w_i(k)$ is equal to the timeout of the thread, it means the i_{th} plug-in should wait for its dependent plug-ins to complete scanning till the timeout of the thread, which would certainly lead to the failure of the $(i-1)_{th}$ plug-in's execution since there is no time for it to execute. If $w_i(k)$ is greater than zero but less than the timeout of the thread, the $(i-1)_{th}$ plug-in would wait for a while before getting executed in the end.

The relationship between different times is shown in

[1]Not include the waiting time for the completion of plug-ins it depends on.

the Eq. 1.

$$\begin{aligned} T_1 &= c_1(1) + w_2(1) + c_2(1) + \cdots + \\ &\quad w_{k1}(1) + c_{k1}(1) \\ T_2 &= c_1(2) + w_2(2) + c_2(2) + \cdots + \\ &\quad w_{k2}(2) + c_{k2}(2) \\ &\dots \\ T_m &= c_1(m) + w_2(m) + c_2(m) + \cdots + \\ &\quad w_{km}(m) + c_{km}(m) \end{aligned} \tag{1}$$

in which, T_m represents the total time consumed on the m_{th} thread (i.e. the m_{th} task line). $w_{km}(m)$ represents the waiting time of $(k_m)_{th}$ plug-in on the m_{th} task line. $c_{km}(m)$ represents the $(k_m)_{th}$ plug-in execution time on the m_{th} task line. What an efficient plug-in scheduling algorithm needs to do is assigning all of the plug-ins to the m task lines so that the maximum of T_m is as small as possible.

The main goal of the paper is designing such an algorithm based on the above abstract model.

III. Related Work

Some plug-in scheduling algorithms had been studied in recent years, among them Obtain-Wait algorithm and Greed algorithm are two representative ones [10].

The plug-in scheduling mechanism in Nessus and OpenVAS are based on the Obtain-Wait algorithm. Obtain-Wait algorithm's main idea is that plug-ins are randomly scheduled at the very beginning. If the current plug-in's prerequisite is met, it is executed instantly; otherwise it will wait until it obtains the scanning result returned from its dependent plug-ins or fails due to thread timeout. Whenever there is a free thread, the scanner will randomly select a waiting plug-in and put it on the task line. Though it is easy to implement, Obtain-Wait algorithm is not efficient since in its eyes all of the plug-ins are of the same kind and should be treated equally. But they are not, some plug-ins should be scheduled before others. An improved algorithm called "Obtain-Wait algorithm based on risk level" introduces risk level of plug-ins for deciding the scheduling order of plug-ins, which may improve the scanning result if the scanning process is interrupted since more severe plug-ins have a higher possible to execute. However, the performance of the scanning remains the same because of the dependent relations between plug-ins. If a plug-in is assigned to a thread but the plug-in's precedent plug-in hasn't finished its scanning, then the thread has to sleep and cannot serve for other plug-ins even when the plug-ins' execution prerequisites are met. In the worst case, all of the threads are assigned such plug-ins waiting for their precedent plug-ins to complete but those precedent plug-ins are waiting to be assigned to idle threads, the vulnerability scanner would enter into a deadlock state. Even with a deadlock resolving mechanism(e.g. deprive some scheduled plug-ins' threads and assign them to other plug-ins), the total throughput of the scanner is still low.

The Greed algorithm uses dependent degree to further improve the performance of vulnerability scanning. The dependent degree includes:

1) un-executed dependent degree (num_deps): the number of un-executed plug-ins on which the current plug-in depends.
2) un-executed depended degree (num_deps_B): the number of un-executed plug-ins that the current plug-in being depended on.

The algorithm's basic idea is: select plug-ins whose num_deps is zero, and from them pick the plug-ins with larger num_deps_B to run. That is, the larger the plug-in's num_deps_B is, the higher priority it should be scheduled when the plug-in's num_deps is zero. In this way, Greed algorithm partially solved the plug-in dependent problem and improved the scanning performance. However, the values of num_deps and num_deps_B need to be updated once a dependent plug-in finishes its execution. Note that the plug-in library contains a huge number of plug-ins, so the update itself is quite a heavy resource consumption task that would easily become the bottleneck of the performance of a vulnerability scanner.

Therefore, we need a pre-scanning mechanism to improve the scheduling efficiency of vulnerability plug-ins which is the major work of this paper.

IV. DAG-based Scheduling Model

A. DAG Graph

In the theory of parallel computing, the task scheduling problem is usually modeled using a DAG graph [11] [12] [13], where nodes represent tasks, directed lines represent the dependency relationship between the nodes they connected, nodes' weight represent time consumption of the nodes, the weight of directed lines represents the communication time consumption. Take the diagram in Fig. 2 for example, c_0 represents a task, the directed line from c_0 to c_1 means that c_1 is followed by c_0 (i.e. c_1 depends on c_0), t_0 represents task c_0's time consumption and e_{01} refers to the time consumption of the communication between c_0 and c_1. From the perspective of plug-in scheduling, the goal of scheduling is to reduce the total task time to as short as possible. In other words, under the premise of plug-ins' dependency restriction, the designed algorithm should make use of overlapping communication and computing to shorten the machine's free time.

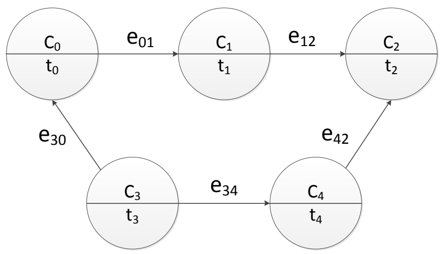

Fig. 2. Directed Acyclic Graph Diagram

On account of DAG's applicability and convenience, there are many studies applying it to solve the problem of network security [14]. In the field of network security, it is usually called attack graph [15]. However, we use DAG in another way in that we focus on the plug-ins' dependency relationship and the concurrent mechanism, and use DAG to abstract the plug-in scheduling problem as a model called Dependent Plug-ins Scheduling Model, from which we build our efficient plug-ins scheduling algorithm.

B. Dependent Plug-ins Scheduling Model

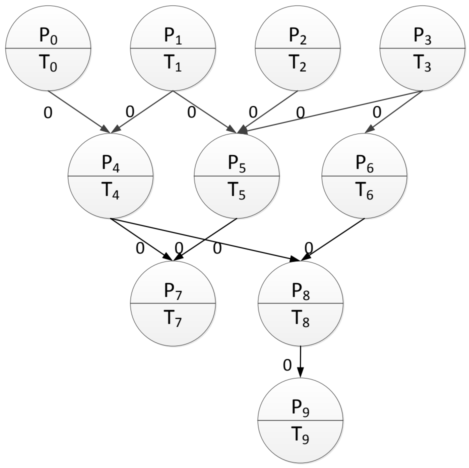

Fig. 3. The Dependent Plug-ins Scheduling Model

As shown on Fig. 3, the dependent plug-ins scheduling model depicts a set of vulnerability scanning plug-ins and the dependent relationships among them. Compared with DAG graph in the parallel computing theory, the task nodes are replaced by plug-ins, the task dependent relationships are interpreted as plug-in dependent relationships, the task time consumption is replaced by the plug-in execution time ,and the communication time consumption is ignored[2]. In this way, we abstract the plug-in scheduling problem as dependent task schedule model by means of DAG graph.

The restriction of dependent plug-ins in the dependent plug-ins scheduling model is very similar to the restriction of dependent tasks in DAG graph. If and only if all of its dependent plug-ins are executed, can a plug-in be executed. Otherwise, the thread running the plug-in could not progress and will hang up waiting for the completion of the dependent plug-ins' execution till the thread times out. What should be pointed out here is that the dependency relationships between all the plug-ins are directed and it's impossible to produce a circle. Therefore, applying the dependent plug-ins scheduling model to solving the plug-in scheduling problem is feasible. Two typical methods are usually used in DAG graph: Topological Sort and Critical Path. In order to utilize the topological sort method in DAG graph, we convert the plug-in scheduling problem to a plug-in sort problem based on the dependent plug-ins scheduling model.

[2]Communication time consumption between different threads is so small compared with plug-in execution time that it can be ignored safely without affecting the plug-in scheduling result.

V. PLUG-IN SCHEDULING ALGORITHM

A. *The* Height *in DAG*

In order to determine the scheduling order under the dependency relationships, we introduce the concept *Height* to indicate the position of each plug-in in the dependent plug-ins scheduling model. The calculation of *Height* (denoted as H) is shown in Eq. 2 .

$$H_i = \begin{cases} 1 & Prev(p_i) = \oslash \\ 1 + max(H(p_j)) & p_j \in Prev(p_i) \end{cases} \quad (2)$$

Applying Eq. 2 to Fig. 3, we obtain Fig. 4.

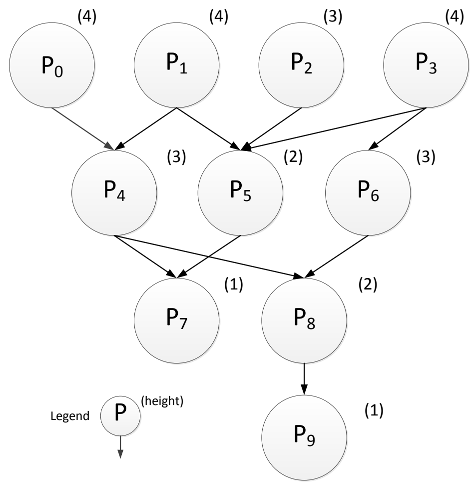

Fig. 4. Example indicating *Height*

Take p_9 for example. The precedent node of p_9 is null, so H_9 is equal to 1. Take p_4 for another example. Because p_4 has two direct precedent nodes p_7 and p_8 whose *Height* is equal to 1 and 2 respectively, H_4 is equal to 1 plus $Max\{H_7, H_8\}$ that is 3. The rest can be calculated in the same manner, finally we can obtain all the *Height* values in the dependent plug-ins scheduling model.

From the definition of *Height*, it is clear that the plug-ins whose *Height* are equal to 1, such as p_7 and p_9 in Fig. 4, have no dependent plug-ins so they can be executed directly at the very beginning. More common situation is that there are two plug-ins p_x and p_y, whose *Height* conform to $H_x > H_y$ and $H_x \neq 1, H_y \neq 1$. $H_x > H_y$ means the *Height* of p_x is larger than that of p_y. On the condition of $H_x > H_y$, we set the rule that P_y should have a higher priority to be scheduled than P_x. We can prove that all of a plug-in's dependent ones would have been scheduled before the moment when the plug-in itself is to be executed.

Theorem 1: Suppose that there is a plug-in $\hat{p}$ whose *Height* is $\hat{h}$, and its direct dependent plug-ins are

$p_1, p_2, \cdots, p_n$ whose *Height* are $H_1, H_2, \cdots, H_n$ respectively. Set the rule that *if* $H_x > H_y$, p_y *must be scheduled before* p_x. If followed by the rule above, when the plug-in $\hat{p}$ is to be executed, all of its dependent plug-ins would have been done.

Proof: According to Eq. 2, $H_i = 1 + max(H(p_j))$ when $p_j \in Prev(p_i)$. Thus, $\hat{h} > H_i$, $H_i \in \{H_1, H_2, \cdots, H_n\}$. Therefore, $\hat{h}$ is always larger than any *Height* of its dependent plug-ins. According to the rule "*if* $H_x > H_y$, p_y *must be scheduled before* p_x", $p_1, p_2, \cdots, p_n$ would be scheduled earlier than the plug-in $\hat{p}$.

■

The concept *Height* is helpful for handling the plug-ins' dependency problem, and its calculation is straightforward. However, when dealing with some peculiar plug-in dependencies, it doesn't work well. Fig.5 is such an example.

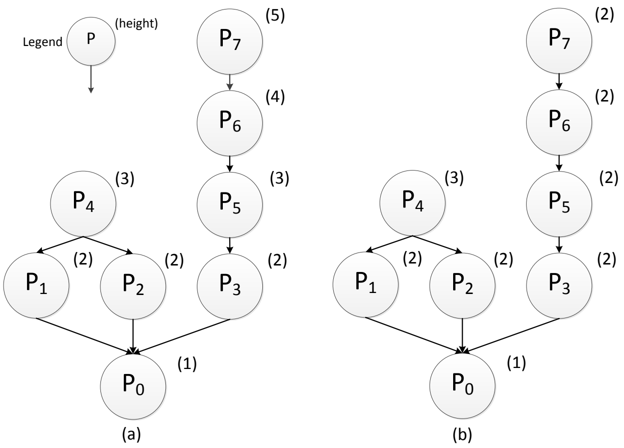

Fig. 5. A remediation of *Height* calculation

According to Eq. 2, the *Height* values is calculated as in Fig. 5 (a). Because p_4's *Height* value is smaller than that of p_6 and p_7, p_4 will be scheduled earlier them. However, we notice that the dependency relationships between p_5, p_6 and p_7 is linear and the plug-ins can run one after another, thus these three plug-ins can be seen as one plug-ins logically. Therefore, in this situation, p_4 is not necessary to be scheduled earlier than p_6 and p_7. Since p_4 and p_5 have the same *Height* value, the scheduling sequence between them would be determined by other factors (such as $Time$ defined in section V-B). But once p_5 is selected to be scheduled, it's better to schedule p_5, p_6 and p_7 as a whole(i.e. assigning to a thread one by one) than let p_4 compete with p_6 and p_7 because the scheduling cost(e.g. searching for the candidate plug-ins and comparing of $height$) can be saved. Note that scheduling in this way would not break the plug-in dependency restrictions.

Therefore, we remedy the calculation method of *Height* as follow: once the *Height* value of one plug-in is determined, if its succeed node has only one dependent plug-in (i.e. the plug-in itself), assign the succeed node the same *Height* value as that of the plug-in. Using this improved method of *Height* calculation, we obtain the new *Height* values as shown in Fig.5 (b). It can be seen that p_5, p_6 and p_7 are set to the same *Height* value so the vulnerability scanner would schedule them as a whole. In the general situation, whenever there is an idle thread, the vulnerability scanner would put a plug-in on that thread. However, in order to use the improved *Height* in Fig.5 (b), the vulnerability scanner should put the plug-ins with linear dependencies(e.g. p_5, p_6 and p_7) on the same thread. For example, after p_5 is scheduled to a thread named $thread_A$, p_6 should wait for $thread_A$ although p_6 has the same *Height* value and there is an idle thread $thread_B$ available. This is because p_6 has to wait for the completion of p_5. Putting p_5 on $thread_B$ would not speed up the scanning process and may downgrade the scanning performance since $thread_B$ could serve other ready-to-run plug-ins with the same *height* with p_6. P_7 should be treated in the same way since it is also a part of the logical one compound plug-in composed of p_5, p_6 and p_7.

B. The Time in DAG

The *Height* value in DAG helps to solve the plug-ins dependency relationship problem by ruling that the plug-in must be scheduled after its dependent ones. However, among the plug-ins with the same *Height* value, there should be a more specific scheduling mechanism.

In order to determine the scheduling priority of the plug-ins with same *Height* value, we introduce the concept *Time* to indicate each plug-in's time consumption level in the dependent plug-ins scheduling model. Since plug-ins would send different numbers of scanning packets and perform different kinds of operations on the received responses, their duration of execution would be different. The time consumed by a plug-in is determined largely by its complexity and the network situation. Since all of the plug-ins in a scanning job would run under the same network situation, we only consider the complexity of plug-ins as the metric for measuring the plug-ins' *Time* in DAG. Since vulnerability scanning is a batch job from the perspective of plug-ins, the plug-ins with higher time consumption level should be scheduled if the *Height* values are the same so as to obtain a higher throughput and shorter scanning time. In this way, the vulnerability scanner could minimize $T_{max} = Max\{T_1, T_2, \cdots, T_m\}$.

As shown in Fig.6, when p_0 finishes its scanning, the vulnerability scanner needs to pick the next plug-in from p_1, p_2 and p_3 whose $height$ are all equal to 2. It seems rational to pick p_1 since its $Time$ value is the largest of the three candidate plug-ins. However, for the same consideration when introducing $height$, plug-ins p_3, p_5, p_6 and p_7 should be seen as a compound plug-in $p_{3,5,6,7}$ whose $Time$ value is the sum of that of the four plug-ins. Since the $Time$ of $p_{3,5,6,7}$ is larger than p_1 and p_2, $p_{3,5,6,7}$ should be scheduled. p_3, as the leading plug-ins of $p_{3,5,6,7}$, should be put on the thread to run.

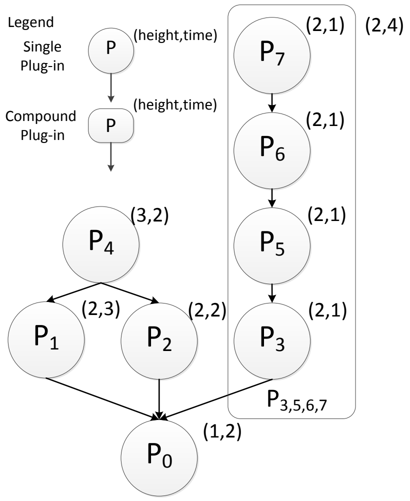

Fig. 6. The plug-in DAG with $Time$ and $Height$

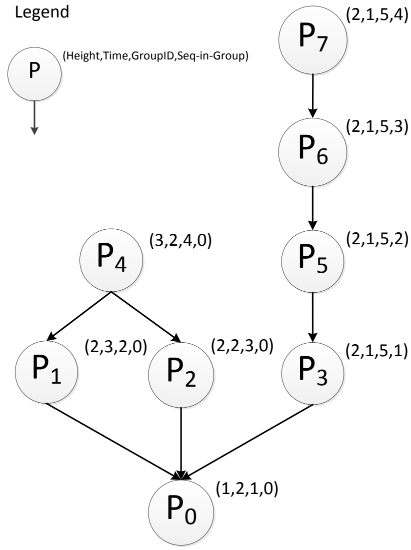

Fig. 7. The plug-in DAG with $GroupID$ and $Seq-in-Group$

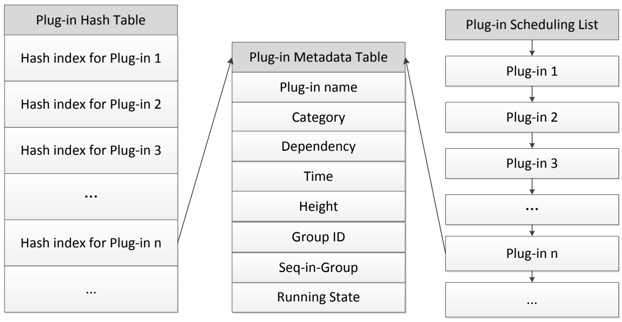

Fig. 8. Auxiliary Structures for Sorting Plug-ins

C. The Group ID and Seq-in-Group in DAG

In order to guarantee that plug-ins composing compound plug-ins are scheduled as a whole and their scheduling conforms to the dependency relationships between them, we introduce the concept of $GroupID$ and $Seq-in-Group$. Each compound plug-in is assigned a unique $GroupID$. Each plug-in composing the compound plug-in is assigned a $Seq-in-Group$ reflecting its scheduling sequence within the group according its dependency. Plug-ins that doesn't belong to any compound plug-ins is itself a group and is assigned a unique $GroupID$ and a zero $Seq-in-Group$. Once a group of plug-ins finish their scanning, the vulnerability scanner will first select a group according to the $height$ and $time$ metric defined above, then pick the candidate plug-ins with the smallest $Seq-in-Group$. If the finished plug-in belongs to a group that doesn't complete its scanning, the the vulnerability scanner would just pick the next plug-in in the group and put it on the same thread serving the finished plug-in. Fig.7 shows an example.

D. Algorithm Description

Based on the above concepts, the paper proposes an improved and efficient plug-in scheduling algorithm using topological sorting of DAG graph, which is described in Algorithm 1.

In order to speed up the sorting of plug-ins, we design some auxiliary data structures include plug-in hash table, plug-in metadata table and plug-in scheduling list, as shown in Fig.8.

With plug-in hash table, the scanner can pin-point plug-ins by their name in constant time. The plug-in metadata table stores information for scheduling plug-ins, among them $plug-invname$, $category$ and $dependency$ can be obtained from plug-ins' definition file(e.g. nvt in Nessus and OpenVAS). The $Time$ value can be set using the time out value of the plug-in or user-defined value. The rest three variables ($Height$, $GroupID$ and $Seq-in-Group$) are set by Algorithm 1. In this way, the scanner only needs to construct a proper plug-in scheduling list so as to determine the plug-in running sequence.

TABLE I
SORTING PRINCIPLE

1) The plug-in with smaller *Height* value wins out.
2) If the *Height* values are the same, the plug-in with larger *Time* value wins out.
3) If the *Time* values are the same, randomly select a set of plug-ins with the same *GroupID* value.
4) If the *GroupID* values are the same, the plug-in with smaller *Seg-in-Group* wins out.

E. Algorithm Analysis

The scheduling sequence of plug-ins is determined by four factors including *Height*, *Time*, *GroupID* and *Seq-in-Group*. The *Height* values reflect the plug-in dependency relationships. Because the time wasted for waiting unexecuted dependent plug-ins is the major factor affecting the scanner's plug-in scheduling efficiency, *Height* is the primary scheduling decision factor. The remaining three factors are used to improve the throughput of plug-in execution.

Therefore, for each plug-in in the set $P = \{p_1, p_2, \ldots, p_n\}$, the proposed algorithm generates a 4-

Algorithm 1 Plug-in Scheduling Algorithm

Input:
An unsorted plug-in set $P = \{p_1, p_2, \ldots, p_n\}$.

Output:
Scheduled plug-ins

Traverse the plug-in set *P*, fill the plug-ins' information in *plug-in hash table* and initialize *plug-in scheduling list* to a null list.
Traverse the *plug-in hash table* and construct a corresponding DAG graph *G* according to the dependencies in *plug-in metadata table*.
Initialize four variables $\tilde{H}, \tilde{T}, \tilde{G}$ and $\tilde{S}$ by setting their value to *0*.
while $G \neq \oslash$ **do**
  $\tilde{H}$++, set $\tilde{G}$ = *0*.
  Let *V* denotes the set of vertexes in *G*.
  for all $v \in V$ **do**
    Create a node in *plug-in scheduling list* for v.
    Set *Time* of v to $\tilde{T}$.
    Set *Height* of v to $\tilde{H}$.
    Set *GroupID* of v to $\tilde{G}$.
    if *in-degree* of *next(v)* is not equal to 1 **then**
      Set *Seg-in-Group* of v to 0.
    else
      Set $\tilde{S}$ to 1.
      Set *Seg-in-Group* of v to $\tilde{S}$, *w* = *next(v)* and create a new variable $\hat{t}$.
      while *in-degree* of *next(w)* is equal to 1 **do**
        Read w's *Time* value from *plug-in meta table* to $\hat{t}$.
        $\tilde{S}++$; $\tilde{T} = \tilde{T} + \hat{t}$.
        Set *Height* of w to $\tilde{H}$.
        Set *GroupID* of w to $\tilde{G}$.
        Set *Seg-in-Group* of w to $\tilde{S}$.
        $w = next(w)$; delete $prev(w)$ and w's *out-edge* in *G*.
      end while
      Reset all the *Time* value of plug-ins whose *GroupID* is $\tilde{G}$ to $\tilde{T}$.
    end if
    Delete v from *G*.
    $\tilde{G}++$.
  end for
end while
Quick sort *plug-in scheduling list* using the four-tuple *(Height,Time,GroupID,Seq-in-Group)* followed according to the principle defined in Table I.
Schedule the plug-ins using the sorted *plug-in scheduling list* according to the principle defined in TableII.
return Scheduled plug-ins.

TABLE II
SCHEDULING PRINCIPLE

1) Schedule the plug-ins according to the sorted *plug-in scheduling list* one by one.
2) Check the current plug-in's *Seg-in-Group* value
   - if its *Seg-in-Group* value is equal to zero, fetch it and execute it.
   - if its *Seg-in-Group* value is equal to nonzero, fetch it and all of its following plug-ins with the same *GroupID*, and put them on a selected thread's buffer.
3) Whenever there comes up an idle thread, check the thread's buffer
   - if it is non-empty, take the first plug-in in the buffer and execute it.
   - if it is empty, schedule the next plug-in in the *plug-in scheduling list*.

tuple <Height,Time,GroupID,Seq-in-Group>. Then, the algorithm sorts the plug-ins according to the above rules and put the sorted plug-ins in a scheduling list. All that remains to do is fetching the plug-ins one after another from the scheduling list and put them on idle threads to run. Note that plug-ins belong to the same group should be put on the same thread to avoid unnecessary waiting.

Back to Eq.1, the maximum of T_m would be smallest if the plug-ins are scheduled by our algorithm because the algorithm can ensure that when a plug-in is to be scheduled all of its dependent plug-ins have already been scheduled[3]. $w_{km}(m)$ is zero when the $(k_m)_{th}$ plug-in is a single plug-in or is the first plug-in of a compound plug-in. The waiting time for dependent plug-ins that are not belonging to the same group is saved. Furthermore, by utilizing the *Time*, *GroupID* and *Seq-in-Group* in the 4-tuple, the plug-in scheduling process works in a batch job way. Therefore, the time of each task line time tends to be average and $T_{total} = Max\{T_1, T_2, \cdots, T_m\}$ would be minimized.

In summary, the proposed algorithm converts the plug-in scheduling problem into topological sorting problem based on DAG theory. Thus, the algorithm can achieve the effect of topological sorting by putting the plug-in's dependent plug-ins in front of itself, then schedule the sorted plug-ins one after another. In this way, the algorithm reduces the situation that some running plug-ins have to stop and wait for their dependent ones, which is the main reason restricting plug-in scheduling efficiency and the performance of a vulnerability scanner.

VI. EVALUATION

We carry out experiments for evaluating the performance and scalability of the proposed algorithm and compare it with that of Obtain-Wait algorithm and Greedy algorithm. The experiment environment is: the vulnerability scanner implementing the scheduling algorithms runs on Fedora14 with 1G memory in VMWare player, whose

[3]Note that this does not mean all of its dependent plug-in have finished executing, so the waiting time may not be zero.

host operating system is Windows XP on a 4G memory, Intel Core i3 Dual CPU 2.4G frequency box.

We measure the time consumed for running the same set of plug-ins using the three algorithm to compare their performance for scheduling. The result is shown in Table III. It can be seen that when the number of plug-ins is small the performances of the three algorithm are roughly the same. This is because the dependencies between plug-ins are less if not none. Nearly no unnecessary waiting time is wasted in such situations. When the number of plug-ins increases, the difference between the performance of these three algorithm becomes apparent. Our algorithm saved more than half of the scanning time compared to the other two algorithms.

We also conduct the experiment by increasing the number of the plug-ins in the set from 10 to 1000 in order to observe the trend of the vulnerability scanning time under different scheduling algorithms. The result is shown in Fig.9. It can be seen that the increasing speed of the scanning time of our algorithm is much lower than that of the other two algorithms. Thus our algorithm is more scalable than the existing one.

The experiments on performance and scalability of the algorithms verifies the efficiency of the proposed vulnerability scanning plug-in scheduling algorithm. Therefore, our algorithm is more suitable for large-scale vulnerability scanners such as those that will be deployed in public cloud and serves large numbers of customers.

TABLE III
SCANNING TIME UNDER DIFFERENT SCHEDULING ALGORITHMS

Algorithm \ Time(s)	10	50	100	500	1000
Obtain-Wait Algorithm	3	6	37	408	1010
Greed Algorithm	3	6	28	310	703
Algorithm based on DAG	3	5	11	100	251

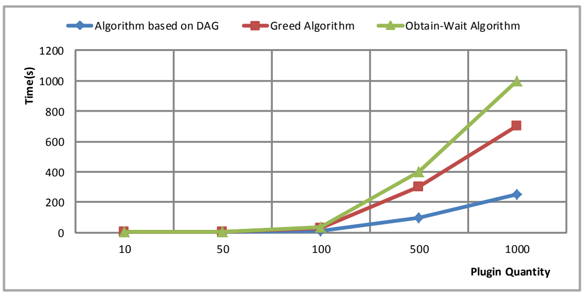

Fig. 9. The Trend of Scanning Time Under Different Scheduling Algorithms

VII. CONCLUSION

Existing vulnerability scanning plug-in algorithms have a low performance. Obtain-Wait algorithm doesn't carry out pre-treatment and schedules plug-ins randomly. Thus it wastes much time on waiting for plug-ins being depended on. Although Greed algorithm carries out pre-treatment and the dependency relationship problem is solved to some extent, it still wastes some unnecessary waiting time. The scheduling algorithm based on DAG graph proposed by the paper solves the dependency relationship problem by guaranteeing that any plug-in will be scheduled before all of its dependent plug-ins have been scheduled. Thus it avoids unnecessary waiting situations. The experiments show that out algorithm is the most efficient one among the existing algorithms.

Our next step work would focus on vulnerability scanning as a service in public cloud. We would design the methods for properly splitting a vulnerability scanning task and integrating the scanning results so that the scanning task can be carried out using map/reduce computing model so as to enjoy various benefits provided by cloud computing.

ACKNOWLEDGMENT

We thank professors and colleagues for numerous discussions concerning this work, State Key Laboratory of Networking and Switching Technology for assistance, and the reviewers for their detailed comments.

REFERENCES

[1] M. Song, T. Yang, and Y. qing Song, "A web survey program based on computer technology and its application to evaluation model about youth self-organizations in china," *Journal of Computers*, vol. 6, pp. 1812–1818, 2011.
[2] J. Tan, X. Chen, and M. Du, "An internet traffic identification approach based on ga and pso-svm," *Journal of Computers*, vol. 7, pp. 19–29, 2012.
[3] Z. Chen, H. Wang, Y. Liu, F. Bu, and Z. Wei, "A context-aware routing protocol on internet of things based on sea computing model," *Journal of Computers*, vol. 7, pp. 96–105, 2012.
[4] S. B. Brown, "2012 Global Security Report," Trustwave's Spider Labs," Technical Report, 2011.
[5] MITRE. (2013, April) Common vulnerabilities and exposures. [Online]. Available: http://cve.mitre.org/
[6] M. Heckathorn, "Network Monitoring for Web-Based Threats," Carnegie Mellon University, SEI Administrative Agent," Technical Report, Feb. 2011.
[7] C. Yuanda, L. Xianfeng, and X. Jingfeng, "Research of Plug in Technology on Vulnerability Scanner," *Microcomputer Development*, vol. 15, no. 9, pp. 72–74, 2005.
[8] Tenable. (2013, April) Nessus vulnerability scanner. [Online]. Available: http://www.tenable.com/products/nessus
[9] OpenVAS. (2013, April) About openvas nvt feed. [Online]. Available: http://www.openvas.org/openvas-nvt-feed.html
[10] L. Junjie, Z. Bing, and Z. Peng, "Nessus Plugin Scheduling Algorithm Research," *Network Security Technology and Application*, vol. 3, no. 4, pp. 80–82, 2009.
[11] Z. Zhongping and L. Xinyuan, "Static Heuristic Task Scheduling Algorithm in the Grid," *Journal of Computer Research and Development*, vol. 45, no. 21-25, pp. 21–25, 2008.
[12] H. Shuixia, Z. Guosun, and T. Yiming, "A Method of Heterogeneous and Reconfigurable Task Partitioning Based on DAG," *Journal of Tongji University (Natural Science)*, vol. 39, pp. 1693–1698, 2011.
[13] Z. Qian, N. Wei-wei, X. Chang-zhen, and L. Hong, "A Scheduling Algorithm of Related Tasks Based on DAG Graph in Grid," *Journal of Chinese Computer Systems*, vol. 33, no. 5, pp. 971–975, 2012.

[14] P. Ammann, D. Wijesekera, and S. Kaushik, “Scalable, graph-based network vulnerability analysis,” in *Proceedings of the 9th ACM conference on Computer and communications security*, ser. CCS ’02. New York, NY, USA: ACM, 2002, pp. 217–224. [Online]. Available: http://doi.acm.org/10.1145/586110.586140
[15] C. Feng, Z. Yi, S. JinShu, and H. WenBao, “Two Formal Analyses of Attack Graphs,” vol. 21, pp. 838–848, 2010.

Yulong Wang received his Ph.D. degree in computer science from the Beijing University of Posts and Telecommunications, China, in 2010. He is currently a lecturer at the Beijing University of Posts and Telecommunications. His research interests include network security and cloud computing.

Nan Li is currently a MS candidate at the Beijing University of Posts and Telecommunications, China. His research interests include network security and next generation network.