Attack Grammar: A New Approach to Modeling and Analyzing Network Attack Sequences

Yinqian Zhang, Xun Fan, Yijun Wang, Zhi Xue
School of Information Security Engineering, Shanghai Jiao Tong University
Email: {jeffreyzhang, fancystar, ericwyj, zxue}@sjtu.edu.cn

Abstract

Attack graphs have been used to show multiple attack paths in large scale networks. They have been proved to be useful utilities for network hardening and penetration testing. However, the basic concept of using graphs to represent attack paths has limitations. In this paper, we propose a new approach, the attack grammar, to model and analyze network attack sequences. Attack grammars are superior in the following areas: First, attack grammars express the interdependency of vulnerabilities better than attack graphs. They are especially suitable for the IDS alerts correlation. Second, the attack grammar can serve as a compact representation of attack graphs and can be converted to the latter easily. Third, the attack grammar is a context-free grammar. Its logical formality makes it better comprehended and more easily analyzed. Finally, the algorithmic complexity of our attack grammar approach is quartic with respect to the number of host clusters, and analyses based on the attack grammar have a run time linear to the length of the grammar, which is quadratic to the number of host clusters.

1. Introduction

Attack graphs are useful tools to visualize multi-step network attacks in graphs. They can be applied in both offensive applications (penetration testing) and defensive scenarios (network hardening) [18]. Though ramifications exist in attack graph representations, generations and analyses, the core idea remains the same: "an attack graph shows the way an attacker can compromise a network or host [1]."

Though improved gradually over the years, the studies of attack graphs still have a number of disadvantages that limit their usage. One fundamental disadvantage of attack graphs is the complexity in visualization [18]. As the number of hosts and vulnerabilities increases, the complexity of attack graphs boosts rapidly, preventing the administrator from understanding the graph and extracting remedies manually. Moreover, attack graphs contain useless information as well. For example, full graphs that "illustrate every order in which attackers can compromise the hosts in the network" [1] are redundant by nature. Predictive graphs [3] and scenario graphs [14] also contain redundancies which make the analysis of attack graphs harder. The multiple-prerequisite graph (MP graph) proposed in [1] is quite succinct. But it is only a middle-level attack graph which means that in order to obtain every potential attack path in the network from the MP graph, one need to explore the graph first. Another disadvantage of attack graphs lies in their limited application of IDS alerts correlation. Attack graphs are constructed based on the interdependency of vulnerabilities, but when it comes to IDS alerts correlation, which needs analyzing the interdependency of vulnerabilities, one has to extract information from attack graphs and reconstruct the interdependency relationships.

In this paper we propose a new technique to complement or replace attack graphs, the attack grammar. The attack grammar is superior to the attack graph in many ways. First of all, it is more suitable in applications like IDS alerts correlation than attack graphs. IDS alerts or the corresponding vulnerabilities, can be regarded as letters in the attack grammar of the underlying network. To correlate alerts, one simply needs to run syntax checking algorithms on the attack grammar. False positives and false negatives can be filtered out by syntax checking as well. However, the attack graph is less competitive in such fields. Secondly, the attack grammar avoids the visualization problem of the attack graph. It can be deemed as a compact representation of the attack graph, because it can be converted to various types of attack graphs easily. In this paper, the conversion from attack grammars to both middle-level attack graphs (expressing attack paths implicitly) and low-level attack graphs (expressing attack paths explicitly) are discussed. Thirdly, the attack grammar is essentially a context-free grammar. It is a formal approach and therefore is easier to understand, analyze and extend. Furthermore, the approach proposed in this paper is scalable with respect to the number of host clusters in the network. Algorithmic complexity in the attack grammar generation is proved to be quartic. And once the grammar is fixed, various analyses based on the attack grammar can be finished in time $O(N^2)$, where N is the number of host clusters.

The remainder of this paper is organized as follows: In section 2, we mainly deal with definition and construction of the network attack model. Section 3 presents algorithms for construction and simplification of attack grammars. Network attack sequences analyses based on attack grammars are presented in section 4. A prototype tool is described in section 5 together with a simple example. Scalability of the algorithm and results of a simulation test are also discussed in section 5. Section 6 sums up previous works related to this paper. Finally in section 7 we summarize our work and discuss future research areas.

2. Network attack model

2.1 Model Components

2.1.1. Hosts: In our model, a host can be any vulnerable entity in a network. Following Ingols et al [1], each host has multiple interfaces, and each interface links to a logical subnet. We also assume that each interface has multiple open ports and each port has zero or multiple vulnerabilities, which belong to local or remote accessible services and software.

2.1.2. Reachability: The term "reachability" relates to the network connectivity between computers and ports. Following Ritchey and Ammann [11], reachability is calculated from a specific host to a specific open port on another host in the network, and the reachability is invariable in the model thereafter.

2.1.3. Vulnerabilities: We use the term "vulnerability" to refer to both vulnerabilities and exposures [2]. Therefore, effects of vulnerabilities in this paper are not limited to obtaining system access but also include obtaining system information. In our model, we use objects to model the preconditions and effects of vulnerabilities. An object uses multi-dimension vectors to describe network situations in order to quantify network conditions.

Definition 2.1: An object is a data structure storing attackers' capability and privilege. An object is a combination of multi-dimension vectors denoted as *obj* $(d_1, d_2, d_3, \ldots, d_i, \ldots)$, where d_i is the *i*th most significant dimension of the object.

Definition 2.2: Comparison of objects is defined as follows: If $d_{1k} > d_{2k}$ holds for certain k and $d_{1i} = d_{2i}$ holds for each $i<k$, $obj_1\ (d_{11}, d_{12}, \ldots, d_{1i}, \ldots)$ is greater than $obj_2(d_{21}, d_{22}, \ldots, d_{2i}, \ldots)$, or obj_1 contains obj_2.

An example of an object is demonstrated in figure 1. In this example, the "*other*" option in the system privilege dimension means the attacker only has privileges in the context of certain application but not in the entire system.

Definition 2.3: A vulnerability v is defined as a 4-tuple *(id, obj_p, obj_e, locality),* where *id* is the identification of the underlying vulnerability, *obj_p* and *obj_e* are the precondition object and effect object of the vulnerability, and *locality* indicates where the vulnerability can be exploited.

Object = {Dim1;Dim2;Dim3}
Dim1: System privilege = {*root, user, other*}
Dim2: Database privilege = {*sa, dbowner, user, none*}
Dim3: Web privilege = {*admin, write, read, none*}

Figure 1. An example of an object

2.1.4. Host Clusters: The number of hosts in the network is a major factor of algorithm complexity. In order to reduce the run time of our program, we define that a host cluster is an aggregation of hosts that share the same reachability property and have similar vulnerabilities.

2.1.5. The Attacker Status: The attacker status describes attackers' privilege and capability on his current focusing host. Privilege and capability is not limited to possessing certain privilege level on the host such as *root* or *user*. It also includes the knowledge that the attacker has acquired to fulfill a future attack. For example, knowledge of a port scan result may be an attacker status on that host. The attacker status is modeled by the object defined in definition 2.1.

2.1.6. Intrusion Starting Point and Targets: The starting point of the intrusion can be anywhere. But at first we assume that the starting point is a host outside the protected network. In our model, we allow multiple targets. The targets of an intrusion may be *root* or other privileges on one or more target hosts.

2.2 Pushdown Automata Model

In this paper we model network attacks with pushdown automata (PDA) [26].

Definition 2.4: An attack PDA M is a 7-tuple $(S, \sum, \Gamma, \delta, q_0, Z_0, F)$, and the meanings of the seven components are as follows. S is a finite set of states, representing host clusters in the network attack scenario. Σ is a finite set of input symbols, which represents exploits or the corresponding vulnerabilities. Γ is a finite set of stack alphabets. δ is the set of transition functions $(\Sigma \times S \times \Gamma) \rightarrow (S \times \Gamma)$. q_0 is the starting state, which stands for attackers' starting point. Z_0 is the initial symbol in the stack, which is only useful in defining language "accepted by empty stack". F is a subset of S which represents final states.

There are two types of states in our model: final states and non-final states. A non-final state corresponds to a host cluster defined in section 2.1.4. The initial state is a non-final state, which stands for the host that the attacker starts his intrusion. We define a sole final state in the PDA model. Only those non-final states representing a target host cluster are connected to the final state with a ε-transition.

Exploitations of vulnerabilities can be modeled as input symbols in our PDA model. A string of input symbols can be considered as a sequence of attacks (exploitations) used by the attacker in temporal order. Note that we use one input symbol to represent exactly one type of vulnerabilities. When we examine whether a string can be accepted by the PDA, we can tell whether the corresponding series of vulnerabilities can lead the attacker to his final target.

The attacker status on a host, which is discussed in section 2.1.5, is modeled with stack alphabets. In our model, we use objects as the stack alphabets and every possible status on each host is assigned a stack alphabet.

A transition function describes the process of executing an exploit to obtain certain privileges on the target host. It also indicates that the attacker is "moving" his intention of intrusion from the current host to the next. The original stack alphabet denotes the privilege and capability an attacker must possess in order to exploit the vulnerability and the new stack alphabet represents the attacker status on the new host. Comparison of objects is needed to determine whether the precondition is met. Moreover, transition functions can, graphically, link one state to another, which stands for a remote attack, and they can also link one state back to itself, which represents a local privilege escalation. Reachability property of network hosts can be implicitly included in PDA model because only reachable hosts can be linked by transition functions.

It is important to note that our PDA model is defined as accepting strings by final state. However, it can be easily converted to a PDA accepting strings by empty stack [26].

All the model components needed for PDA construction can be obtained using approaches proposed in previous researches such as [1]. The algorithm for adding transition functions to the PDA model is listed in figure 2.

```
Algorithm
    for each state $p$, $p \in S - F$
        for each vulnerability $v$ on $p$
            for each state $q$, $q \in S - F$
                if $q$ can reach $v.port$ on $p$
                    for each stack alphabet $X$
                        if $X$ contains $v.obj\_p$
                            Define $X_e = v.obj\_e$
                            Add $(q, v, X) \rightarrow (p, X_eX)$ to $\delta$
                    if $q$ is the start state $q_0$
                        Add $(q_0, v, Z_0) \rightarrow (p, X_eZ_0)$ to $\delta$
        if $p$ contains $host$ in attack targets $(host, obj)$
            for each $X$ contains $obj$ and $X \in \Gamma$
                Add $(p, \varepsilon, X) \rightarrow (q_F, X)$ to $\delta$
    for each $X \in \Gamma$
        Add $(q_F, \varepsilon, X) \rightarrow (q_F, \varepsilon)$ to $\delta$
```

Figure 2. Algorithm for adding transition functions

3. Attack grammar

3.1 Grammar Construction

The attack grammar in this paper is proposed in the form of a context-free grammar. It can be converted from PDA without changing the expressive power.

Definition 3.1: An attack grammar is a 4-tuple (V, T, R, S), where V is a set of variables, T is a set of terminal variables, R is a set of productions, and S is the start symbol.

We can construct a context-free grammar from the PDA model using approaches described in [26]. Particularly for attack grammars, for each transition function $\delta(q,a,X) = (p,YZ)$, we add a production $[qXr] \rightarrow a[pYk][kZr]$ to the set R for every state r and k. The variable $[qXr]$ denotes the process of moving from state q to state r and popping out the stack alphabet X from the top out of the stack. Therefore the meaning of the production $[qXr] \rightarrow a[pYk][kZr]$ is that: In order to move from state q to state r and pop out the stack alphabet X at the same time, we can first read an input symbol a and then move from state p to state k, popping out stack alphabet Y, and then move from state k to state r, popping out stack alphabet Z.

3.2. Grammar Simplification

3.2.1. Eliminating ε-Production: The ε-productions in attack grammars are generated by the ε-transitions in the PDA model. Eliminating ε-productions is critical in simplification of the attack grammar. In fact, the process of eliminating ε-productions actually generates more productions in the grammar. The algorithm for eliminating ε-productions first marks all "nullable" variables, which produce ε in zero or multiple steps. Proof exists that the algorithm in figure 3 is capable of marking all "nullable" variables in the grammar [26].

```
Algorithm
    The context-free grammar $G = (V, T, R, S)$
    The new constructed grammar $G_1 = (V, T, R_1, S)$
    Initiate Queue $M$ and Add $\varepsilon$ to $M$
    while new variables are added to $M$
        for each production $A \rightarrow \alpha$
            if each variable $X$ in $\alpha$ is in $M$
                Mark $A$ as nullable
                Add $A$ to $M$
    for each production $A \rightarrow aX_1X_2$
        if $X_1$ is nullable
            Add $A \rightarrow aX_2$ and $A \rightarrow aX_1X_2$ into $R_1$
        else if $X_2$ is nullable
            Add $A \rightarrow aX_1$ and $A \rightarrow aX_1X_2$ into $R_1$
        else if $X_1$ and $X_2$ are both nullable
            Add $A \rightarrow aX_2$, $A \rightarrow aX_1$, $A \rightarrow aX_1X_2$, into $R_1$
            if $a$ is not $\varepsilon$
                Add $A \rightarrow a$ into $R_1$
        else
            Add $A \rightarrow aX_1X_2$ into $R_1$
    for each production $A \rightarrow aX_1$
        if $X_1$ is nullable
            Add $A \rightarrow aX_1$ into $R_1$
            if $a$ is not $\varepsilon$
                Add $A \rightarrow a$ into $R_1$
        else
            Add $A \rightarrow aX_1$ into $R_1$
    for each production $A \rightarrow a$
        if $a$ is not $\varepsilon$
            Add $A \rightarrow a$ into $R_1$
```

Figure 3. Algorithms for finding and eliminating nullable variables

3.2.2. Eliminating Meaningless Variables: Recall that in our PDA construction algorithm in figure 2, stack alphabet is only popped out when the attacker reaches the final state. It is apparently that only the non-terminal variables with the form *[pYr]*, where *r* is the final state, have logical meaning in attack grammars. Eliminating meaningless non-terminal variables simplifies the attack grammar significantly and the meaning of the non-terminal variables is clearer thereafter: Each non-terminal variable stands for the process of moving from a certain attacker status on certain host (cluster) to the final state. Therefore the total number of different non-terminal variables is no more than MN, where M is the number of host clusters and N is the number of different attacker statuses. Moreover, we notice that only productions of the form $A \rightarrow aB$ or $A \rightarrow a$ are left in the attack grammar after simplification.

3.2.3. Eliminating Non-Generating Variables: A variable is generating if it can generates certain strings. All terminal variables are generating by definition. In attack grammars, we only need to focus on analysis of generating variables, because a non-generating variable implies that it cannot be the parsing tree of any substring of the language and hence no attack sequences can be "generated" using this variable. Algorithms for finding and eliminating all the generating variables can be found in [26], in which related proofs are also provided.

3.2.4. Eliminating Non-Reachable Variables: A variable T is defined as reachable if T can be produced from the initial variable S in zero or multiple steps. A non-reachable terminal variable implies that when assuming the attacker starts his intrusion from the outside network, the exploit corresponding to the terminal variable may not be useful in the network attack scenario. As we will see in section 4, non-reachable variables are still meaningful in our application. However, a grammar without non-reachable variables can help us focus on those exploitations that are usable by outside attackers. Algorithms for finding reachable variables can be found in [26].

3.3. Discussion

In our grammar-based approach to analyzing network attack sequences, using certain type of automata and grammar to model the scenario and using strings accepted by the grammar to represent network attack sequences are the core idea. But why use context-free grammar and PDA? All the previous approaches ([1][14][25]), no matter what the attack graph looks like, use a finite state automaton to model network attack scenarios, because they do not have any kind of memory system to memorize the attack histories. The fact is that using finite state automata to model this kind of scenario is weak due to its memoryless property and only Turning machine is sufficient to solve such problems. PDA has one stack memory which can be accessed only on the top. Therefore it is insufficient in expressive power as well.

We use PDA and context-free grammar in our research is a tradeoff. On the one hand, using PDA to model network attack components is better than previous finite states methods using {host, privilege} pairs to represent states, because it allows us to model more complex attacker statuses. Also, since in PDA model, each state (except for the final state) represents a host cluster, administrators can comprehend and examine the network model more easily. On the other hand, it avoids the complexity of dealing with those more sophisticated automata and hence facilitates the development of a practical approach.

4. Grammar Analysis

4.1. Constructing Attack Graphs

The attack grammar can be regarded as a compact representation of attack graphs. In this section, we propose approaches to converting the attack grammar into different types of attack graphs.

4.1.1. Middle-level Graphs: Some of the previous works [1] represent attack paths using middle-level graphs, in which paths are implied in the graph rather than shown directly. MP graph is of this kind. To construct a middle-level graph like MP graph, every non-terminal variable in the attack grammar should first be converted into a node. If the relationship between two non-terminal variables *A* and *B* is indicated by a production *A->aB*, then an edge should be added to connect the two corresponding nodes. The graph constructed in this way is a variation of MP graph.

4.1.2. Acyclic Full Attack Graphs: One can also use **a**ttack grammars to generate all potential attack paths, and hence build a full graph. Intuitively, the collection of attack paths is just the set of all strings that can be generated by the attack grammar. Moreover, in reality attackers seldom exploit the same vulnerability on the same host more than once, nor do they exploit different vulnerabilities on the same host to gain a privilege that has already been obtained. Therefore, the generated full attack graph must eliminate all cyclic paths. Exploring the attack grammar with a variation of DFS method can derive acyclic attack paths exhaustively.

```
Algorithm
    The context-free grammar $G = (V, T, R, S)$
    Initialize level = 0;
    push [S, $Z_0$, level] into stack1
    while stack1 not empty
        pop from stack1 non-terminal variable A, terminal
        variable a, corresponding level n
        if A is not marked
            mark A
            clear stack to level n (unmark variables)
            push a to stack2
            for each production A->bB
                push [B, b, n+1] to stack1
            for each production A->b
                push c to stack2
                record the attack path
                pop from stack2
```

Figure 4. Algorithm for constructing attack graphs

4.2. Security Evaluation

Various security evaluations can be derived from attack grammars. Using attack grammars together with their simplification algorithms, the administrators can answer the following questions.

(1) Is the attacker able to compromise the target hosts from the outside network?

It is notable that if the initial variable *S* is not marked as generating, the attacker cannot accomplish the intrusion from outside network. And on the other hand, *S* is generating means there exists a path which leads the attacker to the target.

(2) Is the attacker able to compromise the target hosts from inside network? If so, which hosts could be the starting point?

Recall that the non-terminal variables correspond to certain attacker status on certain host. All generating non-terminal variables constitute the collection of hosts from which the attacker can initiate the intrusion and reach the final targets.

(3) Which type of vulnerabilities does the administrator need to fix in order to secure the network?

All reachable terminal variables are the actual exploits that the attacker needs to fulfill his intrusion from his initial starting point. Non-reachable terminal variables may be used when attackers' initial point is inside the enterprise network. Therefore, when the network hardening policy is to be made based on the "outside threats only" assumption, an administrator needs only eliminate those vulnerabilities corresponding to the reachable terminal variables. Temporarily, the problem of the minimal critical set of non-terminal variables has not been addressed yet. But it is likely that approaches used in attack graphs [12] are also suitable for attack grammars.

(4) Can a given attack sequence be used by the attacker to fulfill the intrusion?

Determining whether or not an attack sequence can be used for the attacker to achieve his final goal is equal to determining whether a string of letters can be generated by the attack grammar. The Cocke-Younger-Kasami (CYK) algorithm [8] is adopted in our research. The result of the algorithm can tell the administrators whether the attack sequence can lead the attacker to his final target.

4.3. IDS Alerts Correlation

An important application of the attack grammar is IDS alerts correlation. IDSs which produce large volume of alerts may generate many false positives (false alert) and false negatives (non-detected attack). It is very difficult for administrators to manage the inundant elementary security alerts together with these false alarms. The attack grammar can help solve this problem. We follow the approaches in [7] to use a vulnerability-centric method in IDS alert correlation. In this approach we match alerts with exploits directly.

A variation of CYK algorithm is developed to correlate alerts. Suppose the string to be correlated is $w = a_1a_2a_3a_4$, and we construct the table as shown in figure 5 from the bottom up to the top, line by line. Each X_{ij} represents a set of non-terminal variables. X_{11} contains all the non-terminal variables that have prefix a_1, and X_{22} contains all the non-terminal variables that have prefix a_2, and so on. In the second line, we fill X_{12} with all the non-terminal variables A which is in production $A \to a_1 B$, where $B \in X_{22}$. The construction goes on till the top-left of the table. The non-terminal variables appearing in X_{14} represent our correlated scenario. That is equal to say that the input string is a prefix of the strings that are derived from the non-terminal variables in X_{14}. The algorithm is shown in figure 6. We note that the construction of the table can be also conducted alone diagonals: X_{11}, X_{22}, X_{12}, X_{33}, X_{23}, X_{13},…In this way, we can use the algorithm to incrementally correlate each new arrival alert with those already correlated.

$$\begin{array}{|llll}
X_{14} & & & \\
X_{13} & X_{24} & & \\
X_{12} & X_{23} & X_{34} & \\
X_{11} & X_{22} & X_{33} & X_{44} \\
\hline
\end{array}$$

Figure 5. Table constructed by CYK algorithm

```
Algorithm
Input: The attack sequence a1, a2, a3, … an.
Output: Non-terminal variables of attack grammar in D[n][0]
    The context-free grammar G = (V, T, R, S)
    Initialize 2-dimension array of non-terminal variable sets D[n][n]
    for i=1~n
        for each production A->a in R
            if a = a[i]
                add A to D[0][i]
    for i=1~n
        for j=1-n-i
            for each production A->aB where A∈D[i-1][j]
                if a = a[i] and B∈D[i-1][j+1]
                    add A to D[i][j]
```

Figure 6. Algorithm for checking membership of grammars

In order to correlate IDS alerts, first of all, an attack grammar has to be generated off line. Then the grammar is loaded to a correlation engine. As those established studies, traffic data should be collected using distributed IDS agents and only alerts that match one of the reachable terminal variables in our attack grammar are stored in memory in temporal order. If the sequence is a prefix of an attack sequence which leads the attacker to his final targets, the engine will output one or more non-terminal variables representing a scenario that the attacker is trying to move from the corresponding hosts to the target.

However, due to false positives and false negatives, correlation may not always succeed as expected. Alerts correlation using attack grammars can also serve to ruling out these false alarms. Firstly, we can ignore those alarms which do not match to any of the terminal variables. The rest false positives can be ruled out by correlation. False negatives may result in missing letters in strings. Therefore, we might only be able to obtain segments of the entire attack sequences. But the algorithm in figure 6 is sufficient to discover attack sequence segments.

Another problem of alert correlation would be scenario collision, where actions by multiple attackers fire the alarm simultaneously. To solve this problem, a multi-level correlation engine should be designed. If an alert is discarded by the top level engine, it is stored in the second level for another correlation. Future work may refine this method.

The techniques of syntax checking in compilers are enlightening to the process of alerts correlation. But the difference is that correlation of alerts runs on the fly and is a much more critical application. We believe that techniques in error handling in compilers can be borrowed to strengthen our study in the future.

5. Practical test

We have developed a prototype tool to verify our method. The tool has four components: Data Collection Component, PDA Construction Component, Grammar Construction and Simplification Component, and Grammar Analysis Component. The components are shown in figure 7.

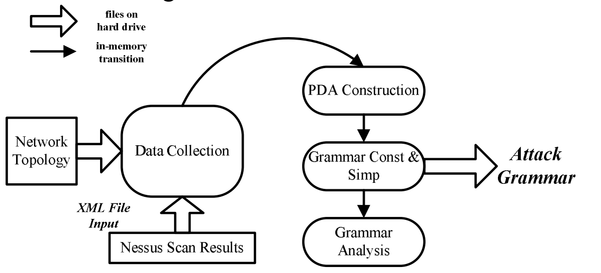

Figure 7. Components in the prototype tool

5.1. A simple example

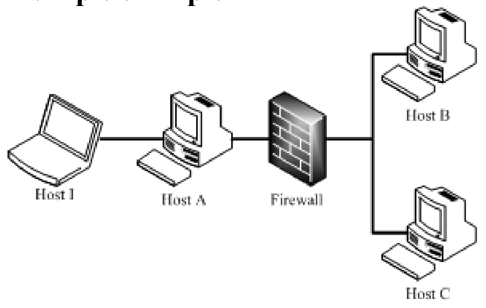

Figure 8. Topology of an example network

The example network in figure 8 comprises 4 hosts and a firewall. The attacker starts his intrusion from host *I*, which locates outside the protected network. The firewall only allows connection between host *A* and the protected network, where host *B* and host *C* are located. The goal of the attacker is to obtain the root privilege on host *C*. Host *A* has one vulnerability *a*, host *B* has two vulnerabilities *b* and *c*, and host *C* has two vulnerabilities *d* and *e*. The specifications of these vulnerabilities are listed in table 1.

Table 1 **Vulnerabilities**

Vuls	pre	effect	locality	port
A	*root*	*user*	*remote*	*1000*
B	*root*	*user*	*remote*	*1001*
C	*user*	*root*	*remote*	*1002*
D	*root*	*root*	*local network*	*1003*
E	*root*	*user*	*remote*	*1004*

Table 2 **Reachability**

Host	Host.port
I	*A.1000*
A	*B.all, C.all*
B	*A.all, C.all*
C	*A.all, B.all*

The automatically generated PDA model is $P = (S, \sum, \Gamma, \delta, q_I, Z_0, \{q_{Final}\})$, where $S = \{q_I, q_A, q_B, q_C, q_{Final}\}$, $\sum = \{a, b, c, d, e\}$, $\Gamma = \{R, U, Z\}$. Here $R = \{root\}$, $U = \{user\}$. The transition functions of the PDA model are shown in the figure 9. To better demonstrate the computation of our PDA, we take the computation of PDA with input *acd* as an example. The computation process is demonstrated in figure 10. The attack grammar can be constructed from the PDA automatically. The simplified attack grammar is shown in figure 11. Attack paths can be generated from attack grammars automatically. We use simple graph to represent the acyclic attack paths, as is shown in figure 12. Note there are cyclic attack paths that are not shown in this graph.

Our tools automatically list all the generating non-terminal variables: $[q_IZq_F]$, $[q_IRq_F]$, $[q_AUq_F]$, $[q_ARq_F]$, $[q_BRq_F]$, $[q_CUq_F]$, $[q_CRq_F]$, and *S*. Since the initial variable *S* is generating in this example, the network is not safe and there are attack paths that lead the attacker to the target. The result also indicates that the attacker can initiate the intrusion from the host with corresponding status listed above. The reachable terminal variables in this example are *a, c, d*, and *e*. That means the vulnerability *b* is useless in the intrusion and the priority of fixing the corresponding bug is lower than that of the others.

We also provide a simple example for IDS alerts correlation. For example, if the attack sequence is *cd*, we input it to our correlation engine and the table is constructed in figure 13.

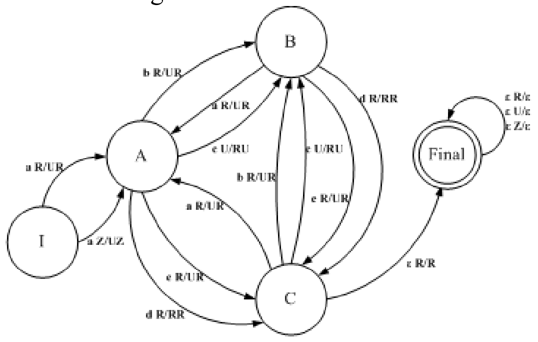

Figure 9**. PDA model of the example**

$$
\begin{aligned}
& [q_I, acd, Z] \\
\mapsto\ & [q_A, cd, U] \\
\mapsto\ & [q_B, d, RU] \\
\mapsto\ & [q_C, \varepsilon, RRU] \\
\mapsto\ & [q_{Final}, \varepsilon, RRU] \\
\mapsto\ & [q_{Final}, \varepsilon, RU] \\
\mapsto\ & [q_{Final}, \varepsilon, U] \\
\mapsto\ & [q_{Final}, \varepsilon, Z] \\
\mapsto\ & [q_{Final}, \varepsilon, \varepsilon]
\end{aligned}
$$

Figure 10**. A sample computation of PDA**

$$
\begin{aligned}
& S \rightarrow [q_I Z q_F] \\
& [q_I Z q_F] \rightarrow a[q_A U q_F] \\
& [q_A U q_F] \rightarrow c[q_B R q_F] \\
& [q_B R q_F] \rightarrow e[q_C U q_F] \mid d[q_C R q_F] \mid a[q_A U q_F] \mid d \\
& [q_C U q_F] \rightarrow c[q_B R q_F] \\
& [q_C R q_F] \rightarrow a[q_A U q_F]
\end{aligned}
$$

Figure 11**. Attack grammar of the example**

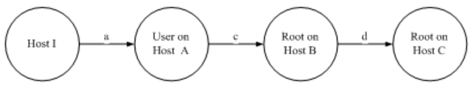

Figure 12**. A sample attack graph converted from the attack grammar**

$$
\begin{array}{|ll}
\begin{Bmatrix} [q_A U q_F] \\ [q_C U q_F] \end{Bmatrix} & \\
\begin{Bmatrix} [q_A U q_F] \\ [q_C U q_F] \end{Bmatrix} & \{[q_B R q_F]\} \\
\hline
\end{array}
$$

Figure 13**. Table constructed during correlation**

5.2. Scalability Analysis

Algorithm for constructing transition function set in PDA model shown in figure 2 is the most timeconsuming part in the PDA construction algorithm. Assuming the underlying network has M host clusters, and V different vulnerabilities, N different attacker statuses (stack alphabets) and the time needed for adding one transition function to the set δ is $O(1)$, thus, the overall computational complexity in constructing the PDA model is $O(M^2VN)$.

The upper bound of the algorithm for converting the PDA model into the attack grammar can be obtained by observing the time consumed by adding new context-free grammar productions to set R. Since the maximum number of transition functions in the set δ is about M^2VN, the overall complexity of the algorithm for adding production to R is about $O(M^4VN)$.

The algorithm for eliminating ε-productions is combinational. However, due to the special structure of our PDA model and the attack grammar, the complexity of the algorithm shown in figure 3 can be polynomial. The original algorithm is to generate 2^m production for every $A \rightarrow X_1X_2X_3X_4...X_k$, in which m variables are nullable. Luckily, the maximum number of k in $A \rightarrow X_1X_2X_3X_4...X_k$ is 2 in our simplified attack grammar, and therefore the upper bound of the algorithm is linear to the number of productions in the attack grammar, with a constant of 4.

Actually, since only non-terminal variables with the form $[qXq_F]$ are meaningful, the number of non-terminal variables can be no more than MN. Moreover, because every production complies with the Greibach Normal Form [8] with its maximum non-terminal variables on the right-hand side no more than 1, the total number of productions in the simplified attack grammar, or the length of the grammar, should be less than M^2VN^2+MNV, which is $O(M^2VN^2)$. The attack grammar is greatly simplified.

Using special data structures specified in [26], the complexity of algorithms for finding all generating variables and reachable variables can be reduced to linear to the length of the grammar, namely $O(M^2VN^2)$.

The algorithm in figure 4 is essentially a variation of DFS if we consider non-terminal variables as nodes and productions as edges. Thus the run time is approximately $O(n)$ where n is the length of the grammar.

If special data structure is used in the implementation of the algorithm in figure 6 to check whether A belongs to $D[i][j]$ in $O(1)$ time, and if the input sequence has the length of W, the overall complexity of constructing the table should be $O(W^2)$. However, if the table is constructed along the diagonals the complexity can be $O(W)$ for each new arriving alerts. In scenarios of alerts correlation where W is relatively small and neglectable compared with the length of the grammar n, the complexity should be $O(n)$.

To sum up, the computational complexity of the construction of the attack grammar is at most quartic to the number of host clusters and that of the grammar simplification and the grammar analyses is linear to the length of the attack grammar, which is quadratic to the number of host clusters.

5.3. Simulation test results

Temporarily, Data Collection Component of our tool can not derive reachability from rules of firewalls and routers yet and part of the input must be done manually. To test the scalability of our methods, a network generator is developed for simulation. Subnets, hosts, vulnerabilities and firewall rules are all generated randomly but a configuration check is performed manually to make sure the network configuration is reasonable.

The simulation test is performed on a Windows XP platform with 3.0GHz Intel Pentium 4 processor and 512MB main memory. The results of simulation test are shown in table 3. The field test results substantiated the scalability analysis results.

Table 3 Simulation Test Results

	Test 1	**Test 2**	**Test 3**
Num of subnets	**3**	**5**	**8**
Num of Host clusters	**15**	**50**	**80**
Num of vulnerabilities	**20**	**20**	**20**
Num of stack alphabets	**3**	**3**	**3**
Vulnerabilities per host	**0~5**	**0~5**	**0~5**
Reachability rate	**43.2%**	**36.0%**	**30.0%**
Num of targets	**1**	**1**	**2**
Time for construction	**0.031s**	**5.45s**	**43.28s**
Num of Productions after eliminating nullable	**20208**	**2150619**	**11659221**
Time for simplification	**0.001s**	**0.031s**	**0.25s**
Num of Productions after simplification	**19**	**380**	**735**

6. Related work

Previous researches can be roughly divided into two groups: graph-based approaches and logic-based approaches. But some researches can fit into both categories.

Swiler et al [9, 10] developed one of the first graph-based formalisms for analyzing network vulnerabilities.

Similar to [10], the NETSpa system [16] also constructs the full attack graph which makes it rather impractical in analyzing large network because full graphs grow combinatorially. A brilliant algorithm was proposed by Ammann et al [17] which introduced a monotonicity assumption [18] and used to develop a polynomial algorithm with the complexity of proximately $O(N^6)$. The monotonicity assumption is also identified in Jajodia et al [21], which described the Topological Vulnerability Analysis tool whose scalability is $O(N^6)$ with respect to the number of hosts. Ammann et al [19] presented an $O(N^3)$ algorithm to determine worst-case paths to all compromised hosts. But the expense was that only suboptimal recommendation can be provided to the administrator. Lippmann et al [3] proposed a host-compromised graph called predictive graph with the construction complexity about $O(N^3)$, but only shortest attack path to a target is shown.

Most of the early logic-based approaches are derived from model checking. Ritchey and Ammann [11] proposed the use of model checkers to generate attack paths for known exploits. Later work such as Jha et al [12] and Sheyner et al [14, 15] extended the use of model checking to analyze attack graphs on complicated networks. Model checking does not assume monotonicity and therefore is poor in scalability. MulVAL [20] proposed a monotonic, logic-based approach which produces counterexamples for a given security policy. They represented vulnerability information in Prolog and results imply the complexity is between $O(N^2)$ and $O(N^3)$, but only separate attack paths can be outputted. X. Ou et al [13] extends their previous research on MulVAL by providing an efficient graph generating algorithm. The complexity is proved to be $O(N^2 log N)$. However, removing loops in the generated graph would have a run time about $O(N^4)$, which is indicated but not addressed in the paper.

Many other researches devoted themselves to network attack modeling. Templeton and Levitt [22] first proposed the "requires/provides" model to describe exploits and this approach has been used by many other researchers in order to chain these exploits together. Languages such as LAMBDA [23] focus on description of detailed attacks, but great amount of detailed attack specifications are needed to input manually therefore the approach was actually impractical. Computation of reachability problem is addressed in Ritchey et al [24].

One of the most important applications of the attack grammar is to correlate IDS alerts. Most previous researches on IDS alerts correlation are based on dependencies among events only, such as Cuppens et al [27] and Cheung et al [4]. In [27], the approach is to define logical rules through pre-conditions and post-conditions. In [4], the approach uses a bottom-up methodology to recognize attack scenario, and the drawback is the difficulty of designing detailed attack models or patterns for every attack. Moreover, due to lacking of global view of network attacks, no high level attack scenario is recognized. Ning et al [6] uses attack graphs to represent relationships among IDS events. The graph is constructed when the events occur. The approach proposed in [7] is one of the few researches that use attack graphs to filter alerts. Attack graphs are generated without fixed starting point and targets. The actual start and end of an intrusion are indicated by alerts.

7. Conclusion and future work

In this paper, we have proposed a grammar-based approach to modeling and analyzing multi-step network attack sequences. Methods for modeling attacks with PDA are proposed and algorithms for constructing, simplifying and analyzing the attack grammars are demonstrated. The underlying purpose of the new approach is to conquer the limitations of using graphs in previous researches. However, since the algorithm complexity for converting attack grammars to attack graphs is merely linear to the length of the grammar, it is also possible and instructive to construct attack graphs using our method. One possible future work might be finishing our tool and trying out the attack grammar on realistic networks. Another potential work is to combine our approach to existing IDS applications for alerts correlation.

References

[1] K. Ingols, R. Lippmann and K. piwowarski, "Practical Attack Graph Generation for Network Defense", 22nd Annual Computer Security Applications Conference, December, 2006.

[2] Common vulnerabilities and exposures dictionary. http://cve.mitre.org/.

[3] R. P. Lippmann et al, "Validating and restoring defense in depth using attack graphs", in Proceedings of MILCOM 2006, Washington, DC.

[4] S. Cheung, U. Lindqvist, et al, "Modeling multistep cyber attacks for scenario recognition", in Proceedings of the Third DARPA Information Survivability Conference and Exposition (DISCEX III), pages 284–292, 2003.

[5] S. Noel, S. Jajodia, "Correlating Intrusion Events and Building Attack Scenarios through Attack Graph Distances," Proceedings of the 20th Annual Computer Security Application Conference (ACSAC04).

[6] P. Ning, D. Xu, C. Healey, R. St. Amant, "Building Attack Scenarios through Integration of Complementary Alert Correlation Methods", in Proceedings of the 11th

Annual Network and Distributed System Security Symposium, February, 2004.
[7] Wang, L., Liu, A., Jajodia, S., "Using attack graphs for correlating, hypothesizing, and predicting intrusion alerts", Computer communications, vol. 29, 2006.
[8] Thomas A. S., Languages and machines, an introduction to the theory of computer science.
[9] C. Phillips and L. Swiler, "A graph-based system for network-vulnerability analysis", In Proceedings of the New Security Paradigms Workshop, pages 71–79, Charlottesville, VA, 1998.
[10] L. Swiler, C. Phillips, D. Ellis, and S. Chakerian, "Computer-attack graph generation tool", in Proceedings DISCEX '01: DARPA Information Survivability Conference & Exposition II, pages 307–321, June 2001.
[11] R. W. Ritchey, P. Ammann. "Using model checking to analyze network vulnerabilities", in Proceedings of the 2000 IEEE Symposium on Security and Privacy (Oakland 2000), pages 156–165, Oakland, CA, May 2000.
[12] S. Jha, O. Sheyner, and J. Wing, "Two formal analyses of attack graphs", in Proceedings of the 2002 Computer Security Foundations Workshop, pages 49–63, Nova Scotia, June 2002.
[13] Xinming Ou, Wayne F. Boyer and Miles A. McQueen, "A scalable approach to attack graph generation", in Proceedings of the 13th ACM conference on computer and communications security, Alexandria, Virginia, 2006.
[14] O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J. Wing, "Automated generation and analysis of attack graphs", in Proceedings of the 2002 IEEE Symposium on Security and Privacy (Oakland 2002), Oakland, CA, May 2002.
[15] O. Sheyner and J. Wing, "Tools for generating and analyzing attack graphs", in Proceedings of International Symposium on Formal Methods for Components and Objects, Lecture Notes in Computer Science, 2005.
[16] M. Artz, "NETspa, a network security planning architecture", Master's thesis, Massachusetts Institute of Technology, 2002.
[17] 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, pages 217–224, ACM Press, 2002.
[18] S. Noel and S. Jajodia, "Managing attack graph complexity through visual hierarchical aggregation", in Proceedings of the 2004 ACM workshop on Visualization and data mining for computer security, pages 109–118, New York, NY, USA, 2004. ACM press.
[19] P. Ammann, J. Pamula, R. Ritchey, and J. Street, "A host-based approach to network attack chaining analysis", in ACSAC'05: Proceedings of the 21st Annual Computer Security Applications Conference, pages 72–84, IEEE Computer Society, 2005.
[20] X. Ou, S. Govindavajhala, and A. Appel, "MulVAL: A logic-based network security analyzer", in Proceedings of the 14th USENIX Security Symposium, pages 113–128, 2005.
[21] S. Jajodia, S. Noel, and B. O'Berry, *Topological Analysis of Network Attack Vulnerability*, chapter 5, Kluwer Academic Publisher, 2003.
[22] S. Templeton and K. Levitt, "A requires/provides model for computer attacks", in Proceedings of the New Security Paradigms Workshop, Cork, Ireland, September 2000.
[23] F. Cuppens and R. Ortalo, "LAMBDA: A Language to Model a Database for Detection of Attacks", Recent Advances in Intrusion Detection (RAID) 2000, Lecture Notes in Computer Science 1907, H. Debar, L. Me, and F. Wu, Eds., Berlin: Springer Verlag, 2001.
[24] R. Ritchey, B. O'Berry, and S. Noel, "Representing TCP/IP connectivity for topological analysis of network security", in Proceedings of the 18th Annual Computer Security Applications Conference, Las Vegas, NV, 2002.
[25] O, Sheyner, "Scenario Graphs and Attack Graphs", PhD Thesis, Carnegie Mellon University, April 2004.
[26] Hopcroft, Motwani, Ullman, *Introduction to Automata Theory, Languages, and Computation (2Ed Aw 2001).*
[27] F. Cuppens, A. Miege, "Alert Correlation in a Cooperative Intrusion Detection Framework", in Proceedings of the 2002 IEEE Symposium on Security and Privacy, May 2002.