Efficient Generation of Exploit Dependency Graph by Customized Attack Modeling Technique

Ishan Chokshi, Nirnay Ghosh, Soumya K Ghosh
School of Information Technology
Indian Institute of Technology
Kharagpur - 721302
Email: ishanchokshi@gmail.com, nirnay.ghosh@gmail.com, skg@iitkgp.ac.in

***Abstract*—A major challenge in today's network is to maintain a secure interconnected world of computing where confidentiality, integrity, availability of information and resources are restored. Traditionally, security is enforced by access control and authentication. However, these security best practices do not take operating system, or network service-based vulnerabilities into account. With the evolution of sophisticated hacking tools, attackers exploit these vulnerabilities and gain legitimate access to network resources, bypassing the access control and authentication policies. Exploit dependency graph models service or application-based attacks and depicts all possible multi-host multi-step attack scenarios that an attacker can launch to penetrate into a network. An important step in the generation of exploit dependency graph is to characterize exploits in terms of a set of precondition and postcondition. Most of the reported works have generated exploit dependency graphs using proprietary vulnerability databases not available in the public domain. This work proposes a customized exploit dependency graph generation through modeling of exploits from open-source databases. Analysis of the developed algorithm shows considerable improvement in terms of time and space complexity in comparison to the reported works.**

***Index Terms*—Attack graph, Exploit, Vulnerability, Access policy, Domain, Fact**

I. Introduction

Cyber attacks have become prominent with the growth of Internet and Web technology. Internet provides seamless access to remote servers, while glitches in Web-based programming have introduced back-doors into these servers. Therefore, using these technologies, attackers from one part of world can compromise servers located at geographically dispersed locations. Even though the critical resources in a network are well-protected, vulnerabilities existing in other hosts from which the critical assets are reachable, can be used as pivot to launch attacks. Therefore, a network administrator has to analyze the security requirements of the network such a way, that it becomes sufficiently secure as well as operational.

Port scanners such as Nessus [1], Retina [2], Nmap [3], CyberCop [4] detect vulnerabilities and even suggest probable patches corresponding them. These scanning tools are useful as far as detecting vulnerabilities local to a system but do not identify all conditions for a complete attack, or how different vulnerabilities existing in different systems are correlated to produce multi-stage attacks. Usually, a vulnerability existing in a particular version of an application has a corresponding *exploit*. An exploit is a sequence of commands which takes advantage of a bug, glitch or vulnerability in order to cause unintended or unanticipated behavior [5]. It gives unauthorized access to a system, which enables a hacker to execute arbitrary code (known as *payload*), for launching the actual attack. The attacks may include *installing an agent, privilege escalation, denial-of-service*, etc.

Traditional network securing approach (e.g. penetration testing) can leave out some complex attack paths generated by interaction and interdependencies of vulnerabilities existing on different hosts. To evaluate the security of any network and get a holistic and a complete picture, it is necessary that the network administrator takes these correlated attacks into account. A tool that depicts a succinct representation of different attack scenarios which jeopardize the security of a network is the *exploit dependency graph* [1]. It is a bipartite graph with two disjoint sets of nodes viz. *exploits* and *conditions*. Each exploit requires a set of conditions (known as preconditions) to be satisfied simultaneously for execution. Launching of an exploit produces one of more conditions (known as postconditions) which becomes the precondition for subsequent exploits. Therefore, an attack scenario is a logical succession of exploits and conditions which begins from a set of *initial conditions* and terminates at the *goal condition*. Multiple attack scenarios can be combined without loss of generality to generate the exploit dependency graph. In the present work, an approach to model this precondition-postcondition sets for an exploit has been proposed. Customized algorithm has been developed to logically combine these exploit descriptions for efficient generation of exploit dependency graph. Therefore, *two* distinct contribution of the present work are as follows:

1) *Exploit modeling*: an exploit modeling technique based on open source exploit framework has been proposed. A series of exploits when logically combined, generates an attack scenario.
2) *Exploit dependency graph generation*: customized algo-

[1]http://www.nesssus.org
[2]http://www.eeye.com/html/products/Retina
[3]http://www.insecure.org/nmap/index.html
[4]http://www.nai.com
[5]http://en.wikipedia.org/wiki/Exploit

rithm has been developed to generate exploit dependency graph. Novel heuristics have been proposed to make the graph generation approach efficient compared to previously reported works.

The organization of the rest of the paper is as follows. Section II presents a formal description of the exploit dependency graph model along with a short review of the related works. In section III, modeling of exploits using data from open source database has been proposed. Section IV presents customized exploit dependency graph generation algorithm. Complexity analysis of the proposed algorithm along with comparison with previous works have also been presented in this section. Finally, a conclusion has been given in section V by identifying the scope of the future work.

II. Exploit Dependency Graph Model

An exploit dependency graph (will be called attack graph interchangeably) consists of a number of *attack paths* (or, attack scenarios), each of which is a logical succession of *exploits* and *conditions*. Conditions in an attack graph represent different attributes of network objects, viz. hosts, network devices, connectivity, etc. and includes the following:

- Platform, architecture, operating system versions of different hosts
- Privilege levels in different hosts
- Availability of vulnerable versions of applications
- Network and transport level connectivity among different hosts

To generate attack graph, a set of *initial conditions* and *goal conditions* are required. Initial conditions refer to those network states which are available by default. Goal conditions are the ones which are to be achieved to compromise a network. Each exploit consists of two sets of conditions:

1) *Precondition*: A *precondition* set of an exploit consists of those conditions which requires to be satisfied *conjunctively* to trigger the same. This implies fulfillment of all conditions to activate an exploit, else the execution of the later fails.
2) *Postcondition*: A *postcondition* set contains those conditions which are generated *disjunctively* after execution of an exploit. These conditions form a subset of *precondition* set for a subsequent exploit.

With the above notions of exploit-dependency graph, the formal definition is given as [2]:

Definition 1: (Exploit-dependency graph). Given a set of exploits E, a set of security conditions C, a relation **require** $R_r \subseteq C \times E$, and a relation **imply** $R_i \subseteq E \times C$, an attack graph $\mathbf{AG}$ is an acyclic directed graph $AG(E \cup C, R_r \cup R_i)$, where $E \cup C$ is the vertex set and $R_r \cup R_i$ is the edge set.

As evident from Definition 1, it can be comprehended that an exploit-dependency graph is a *bipartite* graph which has two disjoint set of vertices, namely, *exploit* and *condition*. The edges are also of two types: (i) *require* edge captures the conjunctive nature of the conditions to activate an exploit, and (ii) *imply* edge identifies those conditions which are yielded after an exploit is successfully executed.

A fairly good number of research works have been done on modeling of *attack graphs*. It has been formally defined in [3] [4] [5] [6] by modeling the preconditions and postconditions for any exploit. Beside this,a network has been defined from various perspectives (physical topology, reachability, availability of services etc.) in order to facilitate formal ways of describing devices such as hosts, routers, switches, firewalls etc. They also describes the attacks, attackers, hosts in the network. In [7], the authors have proposed a new type of attack graph called *multi-prerequisite graphs* (MP graphs) in which state nodes represent attacker's level of access on each host. In [8], a concept of *privilege graph* has been introduced where the nodes represents the a set of privileges for a user or a group of users, and the edges represent vulnerabilities that govern the state transitions. In [9], Wang *et al.* proposed a *Multi-Stage Finite State Machine Model (M-FSM)* to analyze security relationships among network components. Initially, the model represents each attack as an *atom FSM (aFSM)*. Each *aFSM* consists of a single transition and two states viz. the *preconditions*, and the *postconditions*. These *aFSM*s are combined to depict *M-FSM*, modeling vulnerable operations, and possible exploits. In [10], the authors present an attack graph modeling methodology to capture *attack attributes*, *network components*, and *vulnerability specifications*. It presents a goal-oriented approach that works directly with the protected target, assigning each exploit to the target in a way that can identify the weakest node that needs to take measures. Tidwell *et al.* [11] presented an enhanced attack tree model of Internet attacks and system as well as an attack specification languages based on *BackusNaur Form (BNF)* grammars which facilitates modeling, notification, and visualization of distributed attacks. In [12], a scalable, bidirectional-based search strategy to generate attack graphs has been presented. It models the target network in *four* levels: *network service, host system, security system, the host's accessibility*, which facilitates reduction in space complexity. Also, the assumption of monotonicity in the proposed bidirectional-based search strategy reduces the time for generation. In [13], a framework for multi-stage network attack analysis has been proposed that comprises of modeling substrates for network elements, system vulnerabilities and attacker capabilities. Vulnerability interactions can produce sequences of events, called attack chains. This attack chaining is an iterative process that has four steps: (i) precondition evaluation, (ii) exposure creation, (iii) postcondition evaluation, and (iv) creation of an augmented network state.

III. Proposed Exploit Modeling Technique

Modeling of exploits in terms of preconditions and postconditions is the linchpin in generation of exploit dependency graph. In most of the reported works, the authors have not clearly specified the complete set of conditions to characterize an exploit. However, in some papers, the following conditions have been depicted as preconditions for any exploit:

1) availability of vulnerable version of the service/application
2) connectivity with the target host
3) privilege requirement on target host
4) existence of the corresponding vulnerability

The set of postconditions is limited to the following:

1) privilege level gained
2) service/vulnerability disabled

In real-world scenario, executing an exploit requires additional system-level constraints to be satisfied. Moreover, the effect generated by executing an exploit is dependent on the type of *payload* used and is not restricted to privilege escalation only.

For the present work, exploits available from an open source database, *Metasploit* [6], have been considered. The Metasploit framework is a platform for writing, testing, and using exploit code. The primary users of the framework are professionals performing penetration testing, shell code development, and vulnerability research. It has exploits for checking the vulnerability of different platforms like *aix, bsdl, FreeBSD, hpux, Linux, Solaris, Windows* etc. However, documentation of the exploits is grossly insufficient and the exploits have been presented as *ruby* codes instead of an open documentation which is easily-readable and usable. A customized database for the metasploit-based exploits have been developed by parsing the necessary information from the codes and populating the corresponding database fields. The information retrieved from the *ruby* code for a particular exploit are used to model the preconditions essential for its exploitation and the postconditions it generates after its execution. The information obtained from the *Metasploit* database for any exploit are as follows:

- *Name*: name of the vulnerability.
- *Reference*: unique identifier for the vulnerability viz. *CVE-ID* [7], *Bugtraq-ID* [8], *OSVDB-ID* [9] etc.
- *Privileged*: Whether or not the exploit module requires or grants privileged access.
- *Platform*: platform on which the exploit works, for e.g. Windows, Linux, etc.
- *Architecture*: architecture required on the target host for exploit to be executed, for e.g. i386, x64_86, etc.
- *Targets*: version/edition of application or operating system which is vulnerable.
- *Registered_options*: depicting the IP address of the remote server and the port number on which the vulnerable service is listening.
- *Description*: information about the effect of exploit execution given in natural language.

Therefore, as evident from the attribute list presented above, modeling of an exploit requires a generic precondition set to contain the following attributes: *(i) vulnerability, (ii) proper version of the vulnerable application, (iii) platform on which the application is running, (iv) architecture, version, and edition of the operating system, (v) privilege level required to exploit the vulnerability, (vi) port number to which the service is bind, (vii) network layer connectivity between attacker and the target*.

The postcondition or effect generated by an exploit can be obtained by analyzing its *Description* attribute. The *Description* attribute of any exploit is composed in natural language and does not have any standard notation schemes. Moreover, for some exploits, it does not provide any conclusive idea about what effect it will generate after execution. To address this problem, descriptions related to a particular exploit is also obtained from two other sources: (i) *OSVDB (Open Source Vulnerability Database)*, and (ii) *Bugtraq*. Finally, a set of *keywords* are formed from the descriptions obtained from the three sources. These *keywords* give a notion about the possible effect the exploit may generate on successful execution. Some of the *keywords* characterizing exploit effects are: *"arbitrary_code_execution", "service_crash", "privilege_escalation", "installing_agent"* etc.

The exploit modeling using the information obtained from *Metasploit* is done by following *XML* standards which is parsed by the customized exploit dependency graph generation algorithm to extract the necessary conditions. A *domain.xml* file has been manually encoded to model all the exploits acquired from the open source framework.

IV. Algorithm to Generate Exploit Dependency Graph

Algorithm 1 takes a list of exploits and a list of hosts as input and generates exploit dependency graph in the form of adjacency list. *Exploit* is a data structure that contains mainly *three* other structures of type *condition*:

1) *Configuration condition*: specifies configuration parameters such as platform, operating system, version, service, vulnerability which are essential for the execution of exploit
2) *Connectivity condition*: specifies connectivity parameters which are required for an attacker to execute exploit. Connectivity parameters are *tcp* and *ip* level reachabilities, representing whether *tcp* and *ip* traffic are allowed or not.
3) *Effect*: post conditions of exploit

hostList is a list of hosts, each of type *host* data structure. The *host* data structure specifies configuration of host, services running, vulnerabilities, connectivity with other hosts, etc. The proposed algorithm follows a backward chaining approach in which it starts from the *goal* state and finds paths that terminate at the *initial conditions*. Stack is used to facilitate backtracking and contains elements of type *node*. *Node* structure is of type $< node\_type, host, condition/exploit >$. $Node\_type$ can be either *exploit* or *condition*.

The algorithm computes two heuristics before searching for paths.

1) *Distance*: it is distance, in number of hops, of each hosts from the goal host.

[6]http://www.metasploit.com/
[7]http://cve.mitre.org/
[8]http://www.securityfocus.com/bid/
[9]http://osvdb.org/

2) *Rank*: it is distance, in terms of the number of hops, of each hosts from a dummy host *Internet*.

Any host which is connected to *Internet* signifies that the host can be reached from outside of any enterprise network. Both the measure can be computed by applying *depth first search* on network topology. The data structures used by the algorithm are:

- *stack*: used to backtrack and find other paths
- *queue*: to store the path being explored.

Some of the methods which have been invoked in Algorithm 1 have been discussed below:

1) *getTOS(stack)*: returns node present at top of the stack without removing it from the stack.
2) *backtrack(stack, path)*: pops elements out from stack as well as from path; starts with a new path from appropriate node.
3) *savePath(currentNode)*: sets adjacency link between sequential nodes in the path and also sets the flag for each node to indicate that a path exists from that node
4) *findRequiredExploit(currentNode)*: this method searches all the exploits, on executing which required condition are generated. First, it searches for all the vulnerabilities that exists on host. For each vulnerability, it checks whether an exploit is available or not. If the exploit is available, it determines if the effect of the exploit is matching with the one specified in $currentNode$ data structure. All such exploits are put in a list and returned.
5) *findExploitPostcondition(currentNode)*: this method returns a list of conditions required to execute the current exploit from the hosts connected to the one containing the exploit.

A test network has been used to demonstrate the efficacy of the proposed algorithm. The network consists of *three* subnets and the number of hosts in the subnets are *two*, *three*, and *three* respectively. As a single scanning cannot acquire information from all the subnets, there are *Scan servers* to perform and gather subnet-specific scans. The scan result shows a total of 19 exploitable vulnerabilities to be existing in the network.

The implementation of Algorithm 1 has been done using *Java* programming language. The specification files *domain.xml* and *fact.xml* are manually encoded and given as input to the customized algorithm. The generated exploit dependency graph is given in Figure 1. As observed in Figure 1, the condition nodes have *green*, and the exploit nodes have *red* boundaries. As evident from the Figure 1, the exploit dependency graph consists of *two* types of nodes: (i) exploit and (ii) condition. Some of the instances of the nodes are as follows:

- *Exploit nodes*: $apache\_mod\_rewrite\_ldap(H11)$, $phpmyadmin\_config(H11), gld\_postfix(H12)$, $webstar\_ftp\_user(H32), bea\_weblogic\_jsessionid(H23)$, $freesshd\_key\_exchange(H34), goodtech\_telnet(H31)$, etc.
- *Condition nodes*: $root(H11), root(H12), root(H33)$, $root(H31)$, etc.

```
Input: exploitList, hostList
Output: Exploit Dependency Graph
form a goal condition $G(H)$;
initialize $stack$;
initialize $path$;
form a starting node $startNode$;
$push(stack, G(H))$;
while Stack is not empty do
    $currentNode \leftarrow getTOS(stack)$;
    if $currentNode \in path$ then
        $backtrack(stack, path)$;
        continue;
    end
    if path exists from $currentNode$ then
        $path.add(currentNode)$;
        $savePath(currentNode)$;
        $backtrack(stack, path)$;
        continue;
    end
    if $currentNode$ is of type condition then
        $exploitList \leftarrow$
        $findRequiredExploit(currentNode)$;
        if $exploitList == \phi$ then
            $backtrack(stack, path)$;
            continue;
        end
        for each $exploit \in exploitList$ do
            $push(stack, exploit)$;
        end
    end
    else
        $conditionList\ C \leftarrow$
        $findExploitPostcondition(currentNode)$;
        if $C == \phi$ then
            $backtrack(stack, path)$;
            continue;
        end
        if $startNode \in C$ then
            remove $startNode$ from $C$;
            $path.add(startNode)$;
            $savePath(path)$;
            $path.remove(startNode)$;
            if $C == \phi$ then
                $backtrack(stack, path)$;
                continue;
            end
        end
        $heuristicalSort(C)$;
        for each $condition \in C$ do
            if $condition \in path$ then
                remove $condition$ from $C$;
            end
            else
                $push(stack, condition)$;
            end
        end
    end
end
```

Algorithm 1: Exploit Dependency Graph Generation

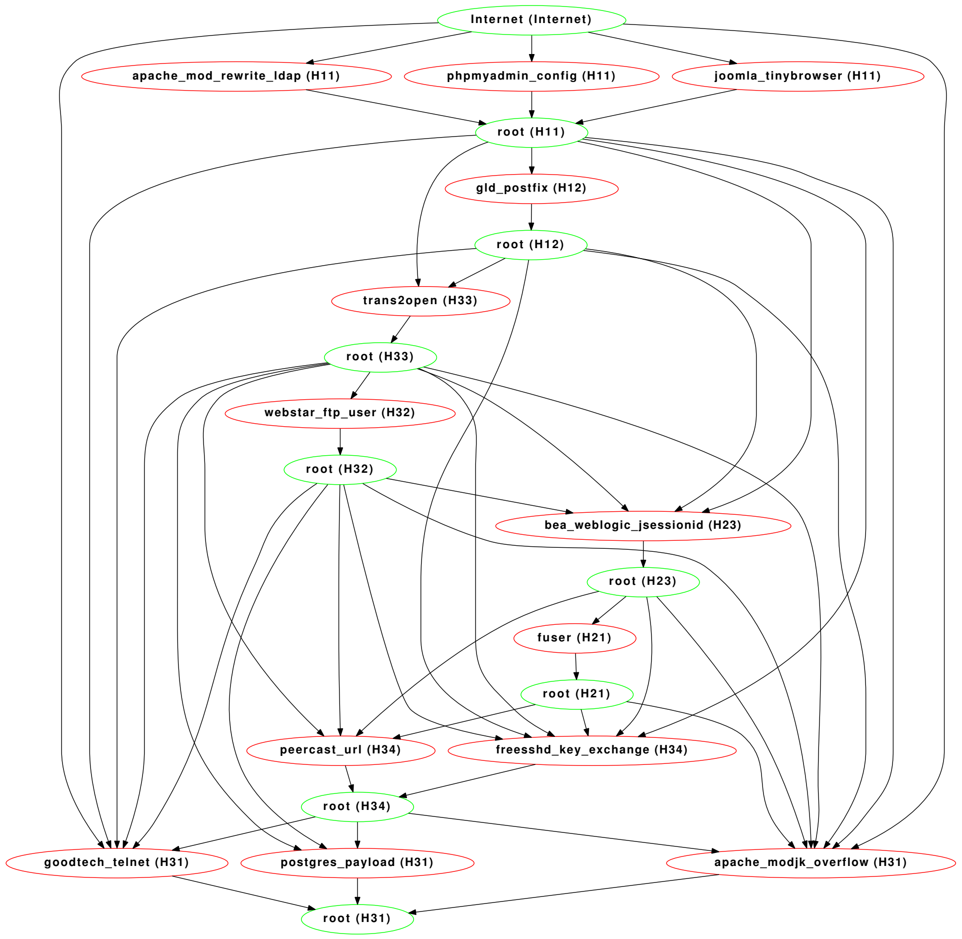

Figure 1. Exploit Dependency Graph

Beside the privilege conditions, other conditions viz. configuration and connectivity, have not been depicted in the generated graph to avoid visibility and comprehension problems. The *goal condition* is to acquire *root* privilege on a particular host $H31$. As evident from the exploit dependency graph, multiple attack paths terminate at the *goal condition*. Such attack paths are guaranteed to remain undetected by port scanners as they are concerned with local vulnerabilities. However, exploit dependency graph takes into consideration all the vulnerabilities in the network, logically combines them by modeling their preconditions and postconditions, and generates complex multi-stage, multi-host attacks, which have become predominant in present time. It enables the administrator to have an overall view on the possible ways a critical asset in the network can be compromised. Such a holistic view facilitates prioritization of network hardening measures.

A. Analysis of Exploit Dependency Graph Generation Algorithm

Exploit dependency graph generation algorithm (refer to Algorithm 1) starts from the specified *goal condition* and backtracks till it reaches the *initial condition* to generate a particular attack path (scenario). Any attack path which does not reach the *initial condition* is discarded. The size of the *state space* is prevented from growing exponentially by using the method *heuristicalSort*. The method *heuristicalSort* sorts the conditions based on the *rank* and *distance* of the hosts (mentioned in Section IV). Hosts which have higher *rank* (and hence more *distance* from dummy host *Internet*) are explored first. If there is a tie then the hosts which have less *distance* from goal host is preferred. If tie is not resolved, any of the nodes can be preferred. Because of this heuristics, the algorithm finds the *longest* path first and hence cover maximum nodes. When the first successful search is complete, more number of nodes will be marked to indicate that a path exists from them. Thus, for subsequent searches if any of these nodes are visited again, then it will be considered as successful search.

The algorithm is guaranteed to find multiple attack paths, as for some exploit node e, it searches for the hosts through which the exploit can be executed. These hosts can be found by connectivity of network. Once this list is available, algorithm finds one best host H, based on computed heuristics and push other nodes in *stack* so that they can be processed later. Next step is to find desired conditions to be satisfied on host H, which could be *root privilege on H*, and finding the exploits that can satisfy this condition. The algorithm chooses an exploit e_1 from possible exploit lists and pushes other exploits in stack so that they can be processed later. Process continues for exploit e_1. This way, once a path to dummy host *Internet* is found, algorithm backtracks and takes a new node from stack.

As the algorithm starts from the *goal condition* and terminates only after reaching the *initial condition*, it can be claimed that even if there exists one and only one attack path, the algorithm guarantees to find the same. This proves the *completeness* of the proposed algorithm.

1) Time Complexity: According to Algorithm 1, whenever a path is found, all the nodes constituting the path are marked with a *flag* to indicate that a path exists from them. Now, in any further path exploration, if search reaches to any of those "flagged" nodes, the algorithm knows that paths exists there onwards. Hence that exploration is called *successful* and new path is saved. This way, every exploit and every condition is explored only once. Thus, a rough estimate of time complexity is given as: $T(n, e) = O(n * T(condition) + e * T(exploit))$ where, n = total number of hosts, e = total number of exploits, $T(condition)$ = time Complexity for exploring a condition node and $T(exploit)$ = Time Complexity for exploring an exploit node.

A condition node is related with a single host. Exploring a condition node is about finding all exploits that can be executed on that host and which produces the condition as its effect. In the worst, if it is assumed that a condition is generated by e number of exploits, then the worst case time complexity for exploring a condition node is given by $T(condition) = O(e)$.

Similarly, exploring an exploit node is about finding all preconditions or all the host from which exploit can be executed. In worst case, there may be a completely connected topology where all the hosts are allowed to access all the services running on all other host. Thus for each exploit there will be n conditions. Also a check is made to ensure that if a condition node is already in the path than it is not pushed into stack again and hence avoid looping. Nevertheless, the complexity will be $T(exploit) = O(n)$.

Thus, the total time complexity is $T(n, e) = O(n * O(e) + e * O(n)) = O(ne)$ Therefore, unless the number of exploits is exponential to the size of the network size, the proposed algorithm yields the exploit dependency graph in polynomial time. However, in realistic scenario, for a large network, with stringent access policy and vulnerability assessment strategy, availability of exponential number of exploits is infeasible.

2) Space Complexity: In the worst case, if a completely connected topology is considered, such that every host is allowed to access every service from all the other hosts, then every exploit can be exploited from all the other hosts. Thus at any time the maximum number of nodes that can be pushed into the stack goes to: $n + (n - 1) + (n - 2) + \ldots + 1 + e = n(n + 1)/2 + e = O(n^2 + e)$. Therefore, similar to time complexity, the space requirement is also polynomial to the number of hosts in the network.

B. Comparison with Related Works

Literature survey on attack graphs show that researchers have used both custom algorithms [14] [11] [13] [15] as well as formal methods [16] [3] [17] [11] [18] [10] to generate attack graphs. Formal methods typically involve representation of attacks, networks, vulnerabilities, and conductivities in some formal language and providing them as input to the model checkers. This, in turn, generates attack paths as counter-examples to show that a security condition is breached. Attack graphs with one single goal as well as those with multiple goals have been generated for network security assessment. The present work aims at finding attack paths between a specified set of *initial condition* and a *goal condition*. The results of research works which deal with generation of attack graphs that considers single goal are presented in Table I. These results have been partially obtained from [19].

V. Conclusion

Security has become a major concern with the proliferation of Internet and Web-based technology. Sophisticated cyber attacks combine vulnerabilities existing on different hosts and use them as pivots to compromise actual targets. Attack graph (exploit dependency graph) models such correlated multi-stage attack scenarios, giving an administrator a holistic idea of a network's security strengths. Most of the works reported in the literature have generated exploit-dependency graphs from proprietary databases without considering all the conditions necessary for exploit modeling. In the present work, modeling the exploits from open source database has been attempted and a generic set of preconditions and postconditions necessary for exploit modeling has been identified. Customized algorithm has been developed which utilizes the proposed modeling technique for efficient exploit dependency graph generation. Complexity analysis of the proposed algorithm shows considerable improvement in comparison to previously reported works. Formulation of attack graph-based security metric for quantification of risk will be taken up as future work.

References

[1] S. Noel, S. Jajodia, B. O'Berry, and M. Jacobs, "Efficient minimum-cost network hardening via exploit dependency graph," in *Proceedings of* 19^{th} *Annual Computer Security Applications Conference (ACSAC 2003)*, 2003, pp. 86–95.

[2] L. Wang, S. Noel, and S. Jajodia, "Minimum cost-network hardening using attack graphs," *Computer Communications, 29(18)*, pp. 3812–3824, November 2006.

[3] S. Templeton and K. Levitt, "A requires/provides model for computer attacks," in *Proceedings of the 2000 Workshop on New Security Paradigms*. ACM Press, 18-21 September 2001, pp. 31–38.

[4] F. Cuppens and R. Ortalo, "Lambda: A language to model a database for detection of attacks," in *Recent Advances in Intrusion Detection*, ser. Lecture Notes in Computer Science, H. Debar, L. M, and S. Wu, Eds. Springer Berlin / Heidelberg, 2000, vol. 1907, pp. 197–216, 10.1007/3-540-39945-3_13. [Online]. Available: http://dx.doi.org/10.1007/3-540-39945-3_13

[5] F. Cuppens and A. Miege, "Alert correlation in a cooperative intrusion detection framework," in *Proceedings of 2002 IEEE Symposium on Security and Privacy*, 2002, pp. 202–215.

Table I
COMPARATIVE STUDY WITH RELATED WORKS

Paper	**Approach**	**Results**	**Remarks**
Ammann, 2002 [20]	Custom algorithm has been developed. A small test network with 3 hosts/6 vulnerabilities has been taken.	The algorithm grows at $O(n^6)$ with the size of the network.	Finds shortest path which can be reached to the goal. Scales to only hundreds of nodes.
Dawkins, 2004 [13]	Formal method to generate attack chaining trees.	Shows poor scaling results.	Generates full attack graph and finds out a "minimum cut set" where the goal cannot be reached if any single vulnerability is removed.
Jajodia, 2003 [21] [1]	Customized algorithm to automatically generate attack graphs. A test network comprising of 3 hosts/4 vulnerabilities has been taken	Base computation grows as n^6.	Computes attack graph using vulnerability and reachability information from *Nessus* and makes recommendations to prevent access to critical resources.
Ritchey, 2000 [16]	Modeling of network hosts, connectivities, attacker's point of view, and exploits using *SMV* model checker. A network consisting of 4 hosts has been used for case study.	Poor	Scalability problem as the size of the state space increases. Modeling of hosts, vulnerabilities, and exploits are done using arrays. This prevents dynamic addition and also their sizes have direct impact on the state space.
Sheyner, 2002 [17]	Uses *NuSMV* model checker to automatically generate attack graphs. The proposed methodology has been tested against a network with 3 hosts/4 vulnerabilities.	Poor	Generates attack graph with the test network in 5 seconds. But for a network with 5 hosts/8 vulnerabilities, it takes 2 hours to generate the graph.
Swiler, 2001 [22]	Proof-of-concept attack graph generation tool. A test network with 2 hosts/5 vulnerabilities have been taken for case study.	Poor	Builds a full attack graph first and then finds out the shortest paths to specified goals by assigning some weights on the edges.
Proposed Work	Uses a customized back-chaining algorithm to generate attack path from manually coded input specification files.	The time complexity is :$O(ne)$ and the space complexity is: $O(n^2e)$ for n hosts and e exploits.	A heuristic based on *rank* and *distance* has been adopted to avoid exploration of the subgraph which has been generated earlier. Backtracking scheme also ensures completeness of the algorithm.

[6] V. Gorodetski and I. Kotenko, "Attacks against computer network: Formal grammar-based framework and simulation tool," in *Recent Advances in Intrusion Detection*, ser. Lecture Notes in Computer Science, A. Wespi, G. Vigna, and L. Deri, Eds. Springer Berlin / Heidelberg, 2002, vol. 2516, pp. 219–238, 10.1007/3-540-36084-0_12. [Online]. Available: http://dx.doi.org/10.1007/3-540-36084-0_12

[7] K. Ingols, R. Lippmann, and K. Piwowarski, "Practical attack graph generation for network defense," in *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC '06)*, December 2006, pp. 121–130.

[8] M. Dacier, Y. Deswarte, and M. Kaaniche, "Quantitative assessment of operational security: Models and tools," in *Proceedings of the LAAS Research Report 96493*, May 1996.

[9] Y. M. Wang, Z. L. Liu, X. Y. Cheng, and K. J. Zhang, "An analysis approach for multi-stage network attacks," in *Proceedings of the 4th International Conference on Machine Learning and Cybernetics (ICMLC)*, 18-21 August 2005, pp. 3949–3954.

[10] X. Liu, C. Fang, D. Xiao, and H. Xu, "A goal-oriented approach for modeling and analyzing attack graph," in *Information Science and Applications (ICISA), 2010 International Conference on*, April 2010, pp. 1–8.

[11] T. Tidwell, R. Larson, K.Fitch, and J. Hale, "Modelling internet attacks," in *Proceedings of the Second Annual IEEE SMC Information Assurance Workshop*. IEEE Press, June 2001, pp. 54–59.

[12] J. Ma, Y. Wang, J. Sun, and X. Hu, "A scalable, bidirectional-based search strategy to generate attack graphs," in *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*, July 2010, pp. 2976–2981.

[13] J. Dawkins and J. Hale, "A systematic approach to multi-stage network attack analysis," in *Proceedings of the Second IEEE Internation Information Assurance Workshop (IWIA '04)*. IEEE Computer Society, 2004, pp. 48–56.

[14] C. Phillips and L. P. Swiler, "A graph-based system for network-vulnerability analysis," in *Proceedings of the Workshop on New Security Paradigms (NSPW)*, 22-26 September 1998, pp. 71–79.

[15] R. Ortalo, Y. Deswarte, and M. Kanniche, "Experimenting with quantitative evaluation tools for monitoring operational security," in *IEEE Transactions on Software Engineering, 25(5)*, October 1999, pp. 633–650.

[16] R. W. Ritchey and P. Ammann, "Using model checking to analyze network vulnerabilities," in *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, May 2000, pp. 156–165.

[17] O. Sheynar, S. Jha, J. M. Wing, R. P. Lippmann, and J. Haines, "Automated generation and analysis of attack graphs," in *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, 2002, pp. 273–284.

[18] F. Chen, J. Su, and Y. Zhang, "A scalable approach to full attack graphs generation," in *Engineering Secure Software and Systems*, ser. Lecture Notes in Computer Science, F. Massacci, J. Redwine, SamuelT., and N. Zannone, Eds. Springer Berlin Heidelberg, 2009, vol. 5429, pp. 150–163.

[19] R. P. Lippmann and I. W. Ingols, "An annotated review of past papers on attack graphs," Lincoln Laboratory, Massachussets Institute of Technology, USA, Tech. Rep. ESC-TR-2005-054, 31 March 2005.

[20] P. Ammann, D. Wijesekera, and S. Kaushik, "Scalable, graph-based network vulnerability analysis," in *Proceedings of CCS 2002: 9^{th} ACM Conference on Computer and Communications Security*. ACM Press, 2002, pp. 217–224.

[21] S. Jajodia, S. Noel, and B. O'Berry, "Topological analysis of network attack vulnerability," in *Managing Cyber Threats: Issues, Approaches and Challenges*, vol. V. Springer US, 2005, pp. 247–266.

[22] L. P. Swiler, C. Phillips, D. Ellis, and S. Chakerian, "Computer-attack graph generation tool," in *Proceedings of the 2nd DARPA Information Survivability Conference & Exposition (DISCEX II)*, vol. II. IEEE Computer Society, 2001, pp. 307–321.