Extending Logical Attack Graphs for Efficient Vulnerability Analysis

Diptikalyan Saha
Motorola India Research Lab
Bangalore, India
diptikalyan@motorola.com

ABSTRACT

Attack graph illustrates all possible multi-stage, multi-host attacks in an enterprise network and is essential for vulnerability analysis tools. Recently, researchers have addressed the problem of scalable generation of attack graph by logical formulation of vulnerability analysis in an existing framework called MulVAL. In this paper, we take a step further to make attack graph-based vulnerability analysis useful and practical for real networks. Firstly, we extend the MulVAL framework to include more complex security policies existing in advanced operating systems. Secondly, we present an expressive view of the attack graph by including negation in the logical characterization, and we present an algorithm to generate it. Finally, we present an incremental algorithm which efficiently recomputes the attack graph in response to the changes in the inputs of the vulnerability analysis framework. This is particularly useful for mutation or "what-if" analysis, where network administrators want to view the effect of network or host parameter changes to the attack graph before pushing the changes on the network. Preliminary experiments demonstrate the effectiveness of our algorithms.

Categories and Subject Descriptors

C.2.0 [**General**]: Security and Protection; D.1 [**Programming Techniques**]: Logic Programming

General Terms

Security, Management

Keywords

Attack Graphs, Incremental Analysis, Logic Programming

1. INTRODUCTION

Attack graph represents all possible ways to achieve multi-host, multi-stage attack in a network. Typically, this is useful for enterprise networks where the network administrators want to know the security risks that can result due to the vulnerabilities present in the hosts. Scanning of each host can produce the vulnerabilities present in the network. However, patching or removing all vulnerable software is not always feasible for network administrators, and is not enough to remove the effects of combining multiple vulnerabilities. Thus, it is important to design automatic tools that can generate all possible ways to exploit the network and different ways of counter possibility of such attacks. For the last decade, a lot of research has been done in generation and analysis of attack graphs ([10]). The main goal here are scalable generation, visualization, and analysis of attack graphs, and use of attack graphs for security flaw detection.

Inspite of the vast research in the area of attack graph generation and analysis, there has been a lack of evidence of end-to-end frameworks and formal approaches for vulnerability analysis. Recently, Ou et. al [16] presented a scalable end-to-end framework called MulVAL for vulnerability analysis using a logic-based approach. The use of logical framework makes the implementation less error prone. The logical semantics also facilitates other analysis on attack graphs.

The primary focus of previous research of attack graph had been to develop scalable algorithms for generation of concise yet useful view of all possible multi-state, multi-host attacks. Initially, researchers had focused on generating full attack graph which is exponential in size to the number of configuration parameters, where each state represents boolean combination of all the configurations. The work by Sheyner et. al. [29] uses formal verification techniques to generate such attack graphs. Later solutions [5] focused on different representations of attack graphs, which is based on the assumption of monotonicity in attacker's behavior. Various other kinds of attack graphs are proposed for analyzing network security [14, 15, 13, 37] based on abstraction. Recent work has introduced logical attack graph ([18]) by extending the logical framework of [16] which represents "why the attack can happen", instead of Sheyner's representation which essentially represents "how an attack can happen". The causality relation between system configuration information and an attacker's potential privileges is captured in the logical attack graph.

Logical attack graphs and previous solutions seem to be adequate for one-time generation of attack graphs. In practice, the network and host configurations of an enterprise-wide network change very frequently, as current networks are more dynamic in nature. The current solutions seem to be inadequate for the following requirements:

- How to efficiently re-generate attack graphs in view of changing network/host parameters.
- How the attack graph analysis information changes in response to changes in the network/host parameters.
- How to efficiently perform *mutation analysis* on attack graphs which show the effect of changing network/host parameters, without actually changing those in real network/host.

Our Contributions. The current MulVAL framework lacks in associating complex security policies existing in advanced operating systems with that of vulnerability analysis. In this paper, we try

CCS'08, October 27–31, 2008, Alexandria, Virginia, USA.
ACM 978-1-59593-810-7/08/10.

to bridge the gap by extending the rules of MulVAL to include the rules for security policies of advanced operating systems such as Windows XP®and SELinux™. This helps administrators to identify access control vulnerabilities in the system.

Moreover, logical attack graph cannot provide proper causality reasoning when rules describing the causality relation contain *negation*. For instance, if a privilege of an attacker negatively depends on another privilege of the attacker, logical attack graph can represent that the first privilege is attainable by the attacker as second privilege cannot be attained. However, it cannot give any reason as to why the second privilege cannot be attained.

As a practical example of such a scenario, we consider buffer overflow attack. Buffer overflow attack typically results in termination of the attacked service, thereby, preventing other use of such a service. Let an attack (say A) is dependent on running of that service (say S). The scenario is represented using the following rules: (i) A is feasible if service S is not terminated, (ii) the service S is terminated if buffer overflow attack is possible on the service, and (iii) additional rules describing why buffer overflow attack can be possible on a service. Logical attack graph represents the above case by showing that the attack A is possible if the service S is not terminated. However, it cannot explain the non-termination of service S.

The above problem is also evident when MulVAL rules are augmented with security policies of advanced operating systems which contain negation in rules. We address this problem in this paper, by presenting a novel algorithm to generate logical attack graphs which includes causal reasoning for negatively dependent derived predicates in rules.

Finally, we address the problem of maintaining attack graph in response to the changes in the inputs of vulnerability analysis framework. One way to approach the above problems is to discard the attack graph or analysis result and recompute the attack graph from scratch, followed by the analysis on the new data. The above process is known as *from-scratch* evaluation technique. In contrast, we take an *incremental* approach where attack graphs are incrementally maintained in response to the changes in the input parameters. The main aim of incremental computation is to compute the changes to the result to update the attach graph. Our main goal is to implement mutation analysis on attack graph in which the administrator can view the effect of changing the network/host configuration on attack graphs without actually making changes in the network. This analysis is extremely helpful for answering "what if" questions of the administrators.

The rest of the paper is organized as follows. The deductive framework of vulnerability analysis is presented in Section 2. Based on the framework, the attack graph is generated using an online technique. The attack graph generation algorithm along with its difference from MulVAL's algorithm is presented in Section 3. The incremental algorithm to maintain results of logical deduction is presented in Section 4. The effectiveness of incremental algorithm is demonstrated in Section 5. The relation between our work and other works in the areas of incremental computation and vulnerability analysis is described in Section 6. We conclude in Section 7 with future work.

2. DEDUCTIVE FORMULATION OF VULNERABILITY ANALYSIS

Our work is based on the deductive framework called MulVAL[16], an end-to-end framework and reasoning system that conducts multi-host, multistage vulnerability analysis on a network. MulVAL uses logic programs as the modeling language for the elements in the

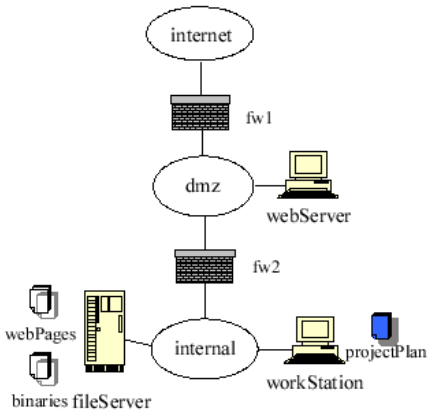

Figure 1: Example Network

analysis. A logic program ([11]) is a sequence of facts and rules. A *fact* has the form *"r(args)"*. A *rule* is of the form

$$r(args) : -r_1(args_1), \ldots, r_n(args_n)$$

and corresponds to a logical implications $r_1(args_1) \wedge \ldots r_n(args_n) \Rightarrow r(args)$. Arguments may contain variables which must start with upper case letters, and literals which start with lower-case letters. The right hand side of ":-" sign (which can be read as 'if') is called the body and the left hand side is called the head. The body consists of one relation or conjunction of one or more relations or negation of relations. Each relation in the body is also called subgoal. Various elements of MulVAL modeling like network and host configuration, vulnerability specification, exploit rules, and privilege model are described using the following example taken from [16]. Note that it is possible to leverage existing vulnerability database and scanning tools by expressing their output in Datalog [35]. Moreover, we augment MulVAL models using more comprehensive security policy language, discussed later.

Illustrative Example. The topology of an example network (taken from [16]) is given in Figure 1. The host access control list along with other facts are specified in Figure 2.

These facts are resolved against a set of MulVAL rules when a query is asked to the Prolog engine. Each rule pertains to a clause of a *derived* predicate. The top level query is made to *policyViolation(Adversary,Access,Resource)*, which is resolved against the following rule:

```
policyViolation(P,Access,Data):-
     access(P,Access,Data),    % Rule
     not allow(P,Access,Data). % Fact

access(P,Access,Data):-
   dataBind(Data, H, Path),     % Fact
   accessFile(P,H,Access,Path). % Rule
```

The top-down query evaluation generates a query (also called subgoal or call) *access(Adversary,Access,Resource)*, which in turn is resolved against the clauses for the predicate access/3 (Predicate/Arity). In XSB, predicates can be marked either as tabled or non-tabled. In our encoding of MulVAL, all derived predicates are marked tabled. A call to a tabled predicate is stored in a table called *call table*.

At a high level, top-down tabled evaluation [33] of logic programs is performed by recording subgoal (referred to as *calls*) and their provable instances (referred to as *answers*) in a table. Clause

```
hacl(internet, webserver, tcp, 80).
hacl(webserver, fileserver, rpc,100003).
hacl(webserver, fileserver, rpc, 100005).
hacl(fileserver,_AnyHost, _AnyProtocol, _AnyPort).
hacl(workstation, _AnyHost,_AnyProtocol, _AnyPort).
hacl(H, H, _AnyProtocol, _AnyPort).

networkSvcInfo(fileserver, mountd, rpc, 100005, root).
networkSvcInfo(webserver, httpd, tcp , 80 , apache).
nfsExportInfo(fileserver, 'export', read, workstation).
nfsExportInfo(fileserver, 'export', write, workstation).
nfsExportInfo(fileserver, 'export', read, webserver).
nfsExportInfo(fileserver, 'export', write, webserver).

nfsMounted(webserver, 'share', fileserver, 'export', read).
nfsMounted(webserver, 'share', fileserver, 'export', write).
nfsMounted(workstation, 'share', fileserver, 'export', read).
nfsMounted(workstation, 'share', fileserver, 'export', write).

vulExists(fileserver, cve20030252, mountd).
vulExists(webserver, cve20020392, httpd).

vulProperty(cve20030252, remoteExploit, privEsc).
vulProperty(cve20020392, remoteExploit, privEsc).

hasAccount(emp, workstation, emplAccount).
hasAccount(sysAdmin, webserver, root).
hasAccount(sysAdmin, fileserver, root).
hasAccount(sysAdmin, workstation, root).

located(attacker, internet).
malicious(attacker).

fileAccessInfo(_,root,read,_).
fileAccessInfo(_,root,write,_).

allow(Anyone,read,webPages).
allow(user,AnyAccess,projectPlan).
allow(sysAdmin,AnyAccess,Data).

dataBind(webpages, fileserver, 'export').
dataBind(projectPlan, workstation, 'home').
```

Figure 2: Example facts

resolution, which is the basic mechanism for program evaluation, proceeds as follows. For non-tabled predicates, the subgoal is resolved against program clauses. For tabled predicates, if the subgoal or its variant[1] is already present in the table, it is resolved against the answers present in the table for that call; otherwise, the subgoal is entered into the table and its answers, computed by resolving the subgoal against the program clause, are also entered into the table.

The resolution of the call to *access/3* predicate (shown above) generates the call *accessFile(Adversary,fileserver,Access,'export')* after resolving with the fact *dataBind(webpages,fileserver,'export')*. The call to *accessFile/4* predicate is then resolved against the following rules:

```
/* Principal P can access files on a NFS server if the files on the server are
mounted at a client and he can access the files on the client side */

accessFile(P,Svr,Access,SvrPath) :-
   nfsMounted(Cl,ClPath,Svr,SvrPath,Access),
   accessFile(P,Cl,Access,ClPath).
  /* NFS shell */
  accessFile(P,Server,Access,Path):-
     execCode(P,Client,root),
     nfsExportInfo(Server,Path,Access,Client),
     hacl(Client,Server,rpc,100003).

  /* Principal P can access files on a NFS client if the files on the server are
  mounted at the client and he can access the files on the server side */

  accessFile(P, Cl, Access, ClPath) :-
     nfsMounted(Cl, ClPath, Svr, SvrPath, read),
     accessFile(P, Svr, Access, SvrPath).
```

The entire resolution is process is not shown here. However, the tabled evaluation generates three answers to the call to *policyViolation/3* viz. *policyViolation(attacker,read,projectPlan)*, *policyViolation(attacker,write,projectPlan)*, and *policyViolation(attacker,write,webPages)*. Tabled evaluation memoizes these calls and their answers. The primary advantage is that the tabled evaluation does not loop (as in ordinary Prolog evaluation) for most of logic programs including Datalog [35].

In the context of vulnerability analysis, memoization facilitates use of existing answers in the table when administrator issues another query. Thus, subsequent queries can be evaluated quickly. Moreover, tabled evaluation is demand-driven [24], which means the analysis only uses the facts that are needed to answer the query of the administrator.

Extended Security Policies. The security policies in MulVAL is modeled using the fact predicate *allow/3*. Each policy is represented as a fact of the form *allow(Principal,Permission,Resource)*. For example, *allow(user,read,projectPlan)* means that *user* can *read* the *projectPlan*.

However, most modern access control model derives the *allow/3* predicate based on other constraints. The two most familiar access control models are Discretionary Access Control (DAC) and Mandatory Access Control (MAC). Windows XP®uses DAC whereas, SELinux™ [4] uses MAC. Thus we extend the security policies of MulVAL to handle security policies of advanced operating systems. Based on these policies, we make *allow/3* as a derived predicate. We also include several rules for vulnerabilities caused by the inconsistent access control rules such as giving write-permission by a non-admin user to a resource which is also executed by the admin user. The main advantage of including host access control policies is to make MulVAL rules take into consideration of access control based vulnerabilities along with software vulnerabilities. This helps network administrator to fix access control vulnerabilities too.

Our encoding of security policies are motivated from [12, 28, 7, 30]. [12] provides rules for Windows XP®security policies and is shown in Figure 3. Due to space limitations we do not discuss the details of the security policies, instead we refer the readers to [12].

Note that our logical encoding of rules uses negation in the rules. We make sure that the negation used in our rules are *safe*, and calls to negated subgoal should be totally *ground* i.e. does not have any un-instantiated variable. Moreover, our encoding ensures use of only *stratified negation*. A logic program with negation is called stratified if there is no cycle involving negation in the predicate dependency graph. In other words, the entire program can be divided into multiple strata of definite (no negation) logic programs, and negation is allowed only at juncture of different strata.

3. ATTACK GRAPH GENERATION

In the example presented in the last section, an attacker can first get access to the webserver by exploiting the remote-exploit vulner-

[1]Two terms are variant to each other if one can be converted to other by renaming its variables.

```
allow(Principal,read,Resource):-
   get_sid(Principal,Sid),
   read(Sid,Resource).
allow(Principal,write,Resource):-
   get_sid(Principal,Sid),
   write(Sid,Resource).
allow(Principal,execute,Resource):-
   get_sid(Principal,Sid),
   execute(Sid,Resource).
read(Sid,RSRC) :-
   processTokenuser(Token,Sid),
   accesscheck(Token,RSRC,r).
write(Sid,RSRC) :-
   processTokenuser(Token,Sid),
   accesscheck(Token,RSRC,w).
execute(Sid,RSRC) :-
   processTokenuser(Token,Sid),
   accesscheck(Token,RSRC,e).
```

```
accesscheck(Token,RSRC,T) :-
   nulldacl(RSRC),
   accesstype(T).
accesscheck(Token,RSRC,T) :-
   firstpass(Token,RSRC,T),
   secondpass(Token,RSRC,T).
allowace(Token,RSRC,T,I):-
   ace(RSRC,I,allow,Sid,T),
   hasenabledSid(Token,Sid).
denyace(Token,RSRC,T,I) :-
   ace(RSRC,I,deny,Sid,T),
   hasenabledSid(Token,Sid).
denyace(Token,RSRC,T,I) :-
   ace(RSRC,I,deny,Sid,T),
   hasdenyonlySid(Token,Sid).
denyace(Token,RSRC,T,I) :-
   denyace(Token,RSRC,T,D),
   numaces(RSRC,NUM), I is D+1, I < NUM.
```

```
firstpass(Token,RSRC,T) :-
   allowace(Token,RSRC,T,I),
   not denyace(Token,RSRC,T,I).
secondpass(Token,RSRC,T) :-
   norestrSids(Token),
   firstpass(Token,RSRC,T).
secondpass(Token,RSRC,T) :-
   restrallowace(Token,RSRC,T,I),
   not restrdenyace(Token,RSRC,T,I).
sestrallowace(Token,RSRC,T,I) :-
   ace(RSRC,I,allow,Sid,T),
   hasrestrSid(Token,Sid).
sestrdenyace(Token,RSRC,T,I) :-
   ace(RSRC,I,deny,Sid,T),
   hasrestrSid(Token,Sid).
sestrdenyace(Token,RSRC,T,I) :-
   restrdenyace(Token,RSRC,T,D),
   numaces(RSRC,NUM), I is D+1,I < NUM.
```

Figure 3: Netra's [13] encoding of Windows XP®security policies

ability (cve20020392) present in httpd running in the webserver. The attacker can run code in the webserver with apache permission. The attacker can now write to the fileserver's export directory using NFS protocol. Attacker can also exploit the vulnerability cve2030252 in mountd in the fileserver to get root access in fileserver, and subsequently can write to the fileserver. Attacker can write to the webpages as webpages are bind to the fileserver's export directory. As workstation's share directory is mounted at fileserver's export directory, attacker can write to workstation's share directory. Now attacker can install a trojan horse to get the root permission and subsequently read and write permission to workstation's home directory which is bound to the projectPlan. Thus a query to policyViolation/3 will return that the attacker can write and read to the projectPlan, and write to the webpages, though they were not allowed by the security policies defined by allow/3 predicate.

The aim of attack graph is to visually represent the above scenario. Each node of logical attack graph of MulVAL corresponds to a logical statement. The nodes do not represent the entire state of the network (as in Sheyner's attack graph), but each node represents one boolean variable which is logical conclusion of the facts given. The edge relation in logical attack graph represents causality relation between system configurations and attackers potential privileges. In this paper we follow the characterization of logical attack graph but remove its drawback to properly represent 'the reason of attack' in presence of negation in the logical encoding of vulnerability analysis.

Logical attack graph in [18] is generated in two steps. Firstly, each MulVAL rule is modified to add a last subgoal in the body, which asserts the reason for which the rule is satisfied. One instance of such transformation is presented below:

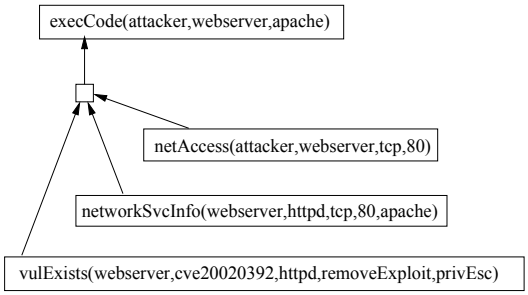

Figure 4: One Step of Logical Attack Graph

```
/* remote exploit of a server program */
execCode(Attacker, H, Perm) :-
   malicious(Attacker),
   vulExists(H, _, Software, remoteExploit, privEsc),
   networkSvcInfo(H, Software, Protocol, Port, Perm),
   netAccess(Attacker, H, Protocol, Port).

execCode(Attacker, H, Perm) :-
   malicious(Attacker),
   vulExists(H, VulID, Software, remoteExploit, privEsc),
   networkSvcInfo(H, Software, Protocol, Port, Perm),
   netAccess(Attacker, H, Protocol, Port),
   assert(because(
   'remote exploit of a server program',
   [networkService(Host,Program,Protocol,Port,User),
   vulExists(H, VulID, Software, remoteExploit, privEsc),
   netAccess(Attacker,H,Protocol,Port)])).
```

Once the transformed rule is executed and satisfied, the last subgoal asserts the provable instance of all the subgoals present in the body. Each such assert produces a part of the attack graph shown in Figure 4. In the second step, after evaluation of the query, the entire logical attack graph is built by composing these parts.

Note that the disadvantage of the above approach is that the assert-subgoal is executed if all the other subgoals in the body are satisfied. However, consider the below rule along with the rules in Figure 3.

```
/* remote exploit of a server program */
policyViolation(P,Access,Data):-
     access(P,Access,Data),    % Rule
     not allow(P,Access,Data). % Fact
     assert(because(
          'policy violation',
  [policyViolation(P,Access,Data),
  not allow(P,Access,Data)])).
```

If a call to *allow/3* predicate fails, only then its parent call of predicate *policyViolation/3* is true. Moreover, an *allow/3* failed clause will not execute the assert subgoal occurring at the last subgoal in its body. Thus logical attack graph fails to provide information on why the *allow/3* predicate has failed. Showing only that the call to *allow/3* fails and not showing why it failed gives incomplete information to the administrator, as the administrator can take action on its security policies to allow some of the overly constrained security policies.

Our Approach. A call to a subgoal fails if all rules whose head subgoal matches (unifies) with the call fail. Each such rule fails if one of its body subgoal fails. For instance, consider a call to *allow(attacker,write,webpages)*, which fails when it matches the second rule of *allow/3*, and the call to *write(attacker_sid,webpages)* fails. The failed resolution procedure is shown in Figure 5.

In our approach, the dependency between failed head subgoal call and the failed body subgoal call is captured and shown in the

Figure 5: Failed Resolution

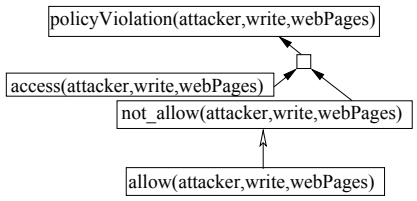

Figure 6: Attack Graph part

attack graph, i.e. in attack graph, we only show the dependencies between the failed calls [f] as shown by the arrows in Figure 5.

Note that the assert subgoal when executed creates the dependencies between provable instances of each subgoal in the body of a rule with that of provable instance of the head subgoal. In other words, it captures the dependency between the answers.

The XSB engine is changed such that the dependency between answers and dependency between calls are created at the time of query processing. No extra assert statements are required to create steps of logical conclusion. Moreover, the second step of logical attack graph creation is not needed as the entire data structure is built at the time of query evaluation. We now formally define our logical attack graph.

DEFINITION 1 (LOGICAL ATTACK GRAPH). *A graph* $G = (V \times E)$ *is a logical attack graph where,*

- $V \subseteq N_s \cup N_a \cup N_c \cup N_f$ *where* N_s, N_a, N_c, *and* N_f *are called support nodes, answer nodes, call nodes, and fact nodes respectively.*
- $E = E_{supports} \cup E_{uses\_of} \cup E_{neg} \cup E_{call}$ *where* $E_{supports} \subseteq (N_s \times N_a)$, $E_{uses\_of} \subseteq ((N_a \cup N_f) \times N_s)$, $E_{neg} \subseteq (N_c \times N_a)$, *and* $E_{call} \subseteq (N_c \times N_c)$.

The nodes N_a and N_c denote the set of answers (provable instances of a call to derived predicates) nodes and call nodes [2], respectively. Each answer node is created because of satisfiable substitution of atleast one clause defining the derived predicate. For each such satisfiable substitution of the body of the clause a support node is created which constitutes the set N_s along with $E_{supports}$ and E_{uses_of} edges. The dependency edges between answers and supports constitute the answer dependency graph or support graph.

We also maintain the call dependency structure where dependency edge exists between a call corresponding to head predicate of a clause and a call generated in the clause body. In our example, a call dependency exists between *policyViolation(Adversary,Access,Resource)* and *access(Adversary,Access,Resource)*. However, this dependency is not exposed as a part of attack graph. Our logical attack graph exposes call dependency that is rooted at the negated subgoals as shown in Figure 5.

We use a simple program transformation in XSB compiler such that each clause does not have a mix of positive and negated subgoals. Below is an example where *not_allow/3* is the newly introduced tabled predicate.

```
policyViolation(P,Access,Data):-
     access(P,Access,Data),
     not allow(P,Access,Data).
⇒
policyViolation(P,Access,Data):-
     access(P,Access,Data),
     not_allow(P,Access,Data).

:- table not_allow/3.
not_allow(P,Access,Data):-
     tnot(allow(P,Access,Data)).
```

The transformation results in an attack graph which separates the answer dependency edges and call dependency edges by a E_{neg} edge, as shown in Figure 6 using unfilled arrow. The part of the attack graph below *allow(attacker,write,webPages)* is already shown in Figure 5.

The main motivation of extending logical attack graph of MulVAL is to provide precise information to the network administrator on why a subgoal is failed. Consider the following vulnerability due to access control rules along with the rules in Figure 3:

```
% Principal P gets root access to Host
% if P can write to a resource RSRC in Host
% which is executed by root
execCode(P,Host,root):- installed(H,RSRC),
                        allow(P,write,RSRC),
                        allow(root,execute,RSRC).
```

The user P can write to the resource RSRC if in the call to firstpass/3, the call to denyace/4 fails. Logical attack graph of MulVAL can only provide the information that the call to denyace/4 fails. However, in this paper, logical attack graph can precisely inform that the call to denyace/4 has failed whether due to absence of deny access control entry (ACE) in access control list (ACL), or allow ACE comes before deny ACE in ACL. This helps administrator to take appropriate action. Note that our algorithm for generation of attack graph also provides the administrators a way to query to know why an attack is not possible in the network.

Acyclic Attack Graph. The attack graph defined above is not acyclic, as the deduction process itself can have cycles. Consider the following rules:

```
% root subsumes other privileges
execCode(P, Host, Perm) :-
    execCode(P, Host, root).
%% Trojan horse installation
execCode(Attacker, H, root) :-
    accessFile(Attacker, H, write, _Path).
%% Permission
accessFile(P, H, Access, Path) :-
    execCode(P, H, Usr),
    fileAccessInfo(H, Usr, Access, Path).
```

In this example, the call *execCode(p1,h1,root)* will create a loop which will indicate that *execCode(p1,hi,root)* is true because *execCode(p1,hi,root)* is true. We use the notion of derivation length of answers to remove cycles in attack graph. The derivation length (*dl*) of different nodes is defined below:

$answer.dl = \{support.dl |\ support$ is the first support of $answer\}$
$support.dl = max\{ans.dl \mid (ans, support) \in E_{uses\_of}\} + 1$
$fact.dl = 0$

[2]Hereafter we do not distinguish between answer and answer nodes. Although in the context of logical deduction `answer` is appropriate and in the context of attack graph `answer node` is appropriate.

Based on the above definition, we can deem a support s of an answer a as *acyclic* if $s.dl \leq a.dl$. Note that any support of an answer which is cyclically dependent on the answer itself will definitely have derivation length more than answer's derivation length. Using this definition, we can identify the acyclic supports during the query processing. No extra pass is required to identify such supports. However, it is possible that there are supports which have higher derivation length than their supported answer, and still do not cyclically dependent on their supported answer. Due to the above reason, our visual display using uDrawGraph of attack graph shows only the acyclic supports and makes non-acyclic supports hidden (using "hidden" node attribute in uDrawGraph graph format), which can be seen by selecting the answer node and using appropriate menu option.

4. INCREMENTAL ALGORITHM

In this section, we describe a novel incremental algorithm to maintain attack graph in response to the changes in the different inputs to the vulnerability analysis framework. Changes in the input to our framework can be modeled as insertion and deletion of the facts and rules corresponding to the inputs. Any update to the input can be modeled as the deletion of the old facts/rules followed by the insertion of the new facts/rules. Note that tabled evaluation in XSB keeps all the calls and answers generated via the query, and in our instrumented XSB we keep dependencies between calls and answers while evaluating a query, and we have shown in Section 3 that an attack graph is a part of dependencies between calls and answers. Thus, the problem of maintaining attack graph amounts to maintaining the tabled calls and answers and their dependencies in response to insertion and deletion of facts and rules. In this paper, we only describe the incremental algorithm which responds to changes in facts, though our implementation handles both changes in facts and rules.

A non-incremental strategy can incorporate any changes in the rules and facts by abolishing all memoized calls, answers, and various dependency structures, and re-issuing the same query which will again generate all information with respect to changed facts and rules. However, such re-computation is often wasteful as typically in network the changes occur are small which typically results in small changes to the already computed information. Therefore, the entire memoized information and dependency structure need not be recomputed. The aim of our incremental algorithm is to identify the parts of the existing computed information that need to be changed, and recompute them. Most incremental algorithms overapproximate the information that needs to be changed, and thus the efficiency of the incremental computation depends on confining this overapproximation.

Algorithm. Our algorithm has two phases: (i) In the first phase (called invalidation phase) it identifies some of the existing calls as *affected* by marking them with the nature of possible effect. These calls are potential candidates for re-evaluation. (ii) The second phase (called re-evaluation phase) re-evaluates (a subset of) the affected calls and propagates the effect of such re-evaluation.

Invalidation Phase. XSB's tabled evaluation memoizes each call and their provable instances (called answers) in answer table associated with each call. The net effect of addition and deletion of some facts and rules is addition and deletion of some answers. The aim of the invalidation phase is to identify the calls whose answers can be potentially deleted, or new answers can be potentially added to them. These calls are potential candidates for re-evaluation in the second phase. Note that not all calls need not be re-evaluated, as only some calls can be actually affected because of insertion and deletion of answers. Invalidation algorithm, presented in the Figure 7 identifies an overapproximate of such calls along with its type of effect which can take four values {DEL, INS, BOTH, NO} based on whether it is possible to only delete, only insert, both insert and delete, or no changes of their answers, respectively. The algorithm explores the dependencies between calls to know which calls can be affected. The direct call dependencies are captured using two sets call.outedge.positive and call.outedge.negative.

```
1 invalidate(Call,Type)
2 if Type==INS
3    Call.falsecount++
4 oldtype=Call.type
5 Call.type=compose(oldtype,TYPE)
6 if(oldtype!=Call.type)
7     propagate_invalid(Call,TYPE)
8
9 propagate_invalid(C,TYPE)
10 enque(affectedq,C)
11 if(C is neg_call)
12    if (C has an answer and TYPE=DEL)
13                  or
14       (C has no answer and TYPE=INS)
15       check_and_propagate(C,TYPE)
16       for C' = C.outedge.negative
17          if TYPE !=DEL
18             C'.falsecount++;
19          oldtype = C'.type;
20          C'.type = compose(oldtype,!TYPE)
21          if(oldtype != C'.type)
22             propagate_invalid(C',!TYPE)
23 else
24 if C is a not call
25    if (C has an answer and TYPE=DEL)
26                      or
27       (C has no answer and TYPE=INS)
28       check_and_propagate(C,TYPE)
29 else
30    check_and_propagate(C,TYPE)
31
32 check_and_propagate(C, TYPE)
33  for all C' in C.outedge.positive
34       if type=INS
35             C'.falsecount++
36       oldtype = C'.type;
37       C'.type = compose(oldtype,TYPE)
38       if(oldtype != C'.type)
39             propagate_invalid(C',TYPE)
40 /* comments:
41 !(INS)=DEL; !(DEL)=INS
42 compose(NO,X)=X, compose(INS,DEL)=BOTH,
43 compose(DEL,INS)=BOTH, compose(BOTH,X)= BOTH
44 */
```

Figure 7: Invalidation Algorithm

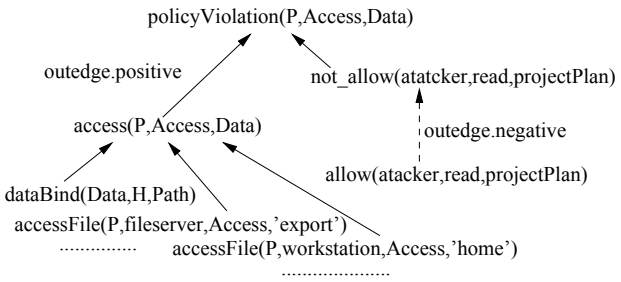

Figure 8: Example Call Dependency

Consider the partial (the call dependencies from *accessFile/4* and call to other *not_allow* calls are not shown in the Figure) call dependencies shown in the Figure 8 corresponding to the transformed *policyViolation/3* predicate. The normal arrows represent the posi-

tive call dependencies, and the dotted arrow represents the negative call dependency which signifies presence of negated subgoal. In this case, the call to *not_allow/3* is called *not call*, and the call which has atleast one negative outgoing edge is called *neg call*, in this case the call *allow(attacker,read,projectPlan)*.

The invalidation phase propagates the invalidation type from a call to its successor calls in the call dependency graph following the positive call and negative call dependencies. Consider addition of a new fact *dataBind(docs,fileserver,'home')*. The invalidation phase first identifies the call *dataBind(Data,H,Path)* and invokes the function *invalidate(dataBind(Data,H,Path),INS)*. The function marks the type of calls *dataBind(Data,H,Path)*, *access(P,Access,Data)*, *policyViolation(P,Access,Data)* as *INS* as answers can be potentially added to these calls. The algorithm also maintains a counter with each call, called *falsecount*, which tracks the number of its immediate predecessor calls that are invalidated with INS. If re-evaluation of a call does not add any answer to the call the *falsecount* of its immediate successor calls is decreased by 1. If a *falsecount* of a call reaches zero it is not re-evaluated as it is not possible to add answers to that call. For example, in this case the addition do not add any answer to the call *access(P,Access,Data)* and thus the call *policyViolation(P,Access,Data)* need not be not re-evaluated. Note that invalidation phase does not update *falsecount* for deletion mark propagation. This is because our algorithm finds whether an answer to be deleted (except for not call) using dependencies among answers, and not using re-evaluation of calls.

In Figure 7, the Function *check_and_propagate* (Lines 33-39) propagates the INS/DEL mark to the positively supported calls. The Function *propagate_invalid* checks whether a call is *not* or *neg*, and propagates the invalidation type accordingly. An INS mark to a *neg* call is propagated as DEL mark to the corresponding negatively supported *not* call. Note that *not* and *neg* calls are grounded and they can have atmost one answer. Thus, an INS mark is not propagated to successors of a *neg/not* call if the *neg/not* call already has one answer, and similar logic holds for DEL mark as shown in the Function *propagate_invalid*.

Re-evaluation Phase. The aim of the re-evaluation phase is to compute the answers deleted from and inserted into the tables. The algorithm essentially interleaves two independent algorithms which perform insertion and deletion of answers for definite logic programs (programs without negation). Note that both the algorithms are discussed in [25, 26] in the context of definite logic programs. Our contribution is to judiciously interleave those algorithms for logic programs with stratified negation. We summarize incremental insertion and deletion algorithms followed by our algorithm which extends them for logic programs with stratified negation.

Incremental Insertion. Our incremental evaluation algorithm to handle *addition* of facts is based on program transformation [25]. This difference-rules based technique is relatively straightforward, and has been studied extensively in the literature (e.g. [20]). For every relation r defined in the program, we derive a new relation, called its *delta* relation (denoted by δ_r), which captures changes to the relation due to the addition of facts. For every rule of the form $r :- r_1, r_2, \ldots r_n$ in the original program, we add rules of the form $\gamma_r :- (r_1; \delta_{r_1}), \ldots, (r_{i-1}; \delta_{r_{i-1}}), \delta_{r_i}, r_{i+1}, \ldots, r_n$ for each $i \in [1, n]$, where δ_r is defined as $\gamma_r - r$.

Incremental Deletion. In short, deletion algorithm called DRed has two phases. In the deletion phase (Function *delete*) all the answers that can be potentially deleted are marked. This is done by propagating the deletion mark from the deleted fact to supports containing the fact. An answer is marked deleted if all its acyclic supports are marked deleted. The second phase known as rederivation phase (Function *rederive*) unmarks all answers which have atleast one unmarked supports and unmarks all supports if all its constituent answers are unmarked and propagating the effect of rederivation to answers and supports. Due to space limitations we do not provide the code of DRed here, instead we refer the readers to [26].

```
1  re—evaluation phase()
2    // affectedq contains the calls
3    // whose type is changed in the first phase
4
5    DRed()
6    re—evaluate()
7    garbage_collect()
8
9  DRed()
10   delete()
11   rederive()
12
13 re—evaluate()
14   while((c=next_invalid_call())!=NULL){
15     call(δc);
16   }
17
18 next_invalid_call()
19   again: C=deque(affectedq)
20   if(C.TYPE!=DEL) and C.falsecount>0
21      return C
22   else
23      if(C is a neg call and C.type!=INS)
24        if still has an unmarked answer
25           propagate_valid(c.outedge.negative)
26      goto again:
27
28  propagate_valid(C)
29    for all C' in C.outedge.positive
30      if C'.falsecount>0
31        C'.falsecount—
32        if(C'.falsecount==0)
33          propagate_valid(C')
34
35  answer_check_insert(Answer)
36   C=Answer.call;
37   if (Answer ∈ answer_table(C))
38     if Answer is marked deleted
39       remove mark from Answer
40       rederive_answer(Answer)
41       if C is a neg call
42          C'=C.outedge.negative
43          C'.falsecount=0;
44          propagate_valid(C')
45  else
46    //new answer is inserted in C
47     if C is neg call
48      C'=C.outedge.negative
49      mark_answer(C'.answer)
50      rederive()
51
52  check_complete_insert(C)
53    if(C.type!=NO)
54       if no new answer has been inserted in C
55          propagate_valid(C)
```

Figure 9: Re-evaluation Algorithm

The Re-evaluation algorithm re-evaluates each affected call which amounts to calling of its corresponding delta predicate call and inserting answers generated in delta predicate's table to answer table of the call. The algorithm of re-evaluation phase is shown in Figure 9. The re-evaluation is done in topological order of calls in the call dependencies. A call is re-evaluated only if it INS/BOTH marked and has the *falsecount* greater than zero (Lines 20-21). After each re-evaluation of each call is completed the re-evaluation algorithm checks (Function *check_complete_insert(C)* in Figure 9) whether the call has any new answer or not. In case the re-evaluation of the call has not generated any new answer in the

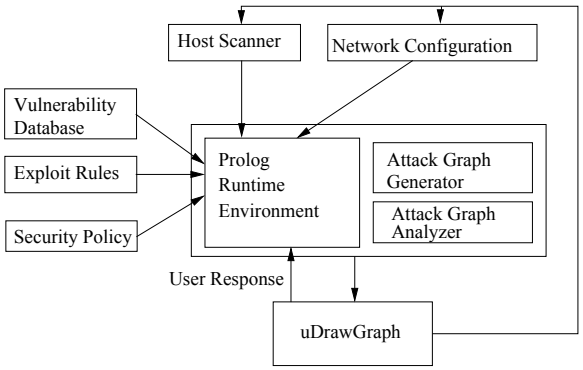

Figure 10: Framework

call, the function *propagate_valid* is called to nullify the effect of invalidation phase. This is already explained while describing the significance of *falsecount* attribute.

While re-evaluation of a call C (i.e. execution of call δ_C) if an answer $Answer$ of the call γ_C is generated, it is first checked whether the answer already existed in the answer table of C. If the answer is already present in the table and *marked deleted*, then re-evaluation of C has found another way of deriving the answer. Subsequently, the effect of its *deletion* is nullified by Function *rederive_answer(Answer)* which rederives a single answer and propagates the effect of the rederivation to other answers. In case C is a *neg* call whose deleted answer is inserted again, the effect of potential deletion of answer in *neg* call, and subsequently insertion in its *not* call done in the invalidation phase needs to be nullified (Lines 40-43). In case the answer a is a new answer of a *neg* call C (Lines 45-49), the corresponding answer in not call needs to be deleted and effect of such deletion is propagated using two functions *mark_answer* and *rederive*.

Rederivation phase eagerly propagates the effect of deletion of answers before processing any calls for re-evaluation. Thus, before re-evaluation of a *neg* call whose type includes DEL and which has an answer (Lines 23–25), if the algorithm finds that its answer is not marked deleted, then the answer is not going to be deleted. Subsequently the effect of DEL mark in the *neg* call done in the invalidation phase is nullified using a function *propagate_valid(C)* which is a reverse function of Function *propagate_invalid(C,TYPE)* in Figure 7.

The incremental algorithm presented here not only improves the time to recompute the attach graph, but is used to generate the difference between previous and present version of the attack graph. The changes are shown in the udrawGraph gui as a hidden graph which can be seen using appropriate menu option.

Note that the associated data structures are built while the re-evaluation of call. Any already existing answer or call dependencies that can arise due to re-evaluation are eliminated. The complexity of attack graph generation algorithm is lesser or equal to non-incremental algorithm whose complexity matches with that of MulVAL.

5. EXPERIMENTS

Our framework is shown in Figure 10. The framework is very similar to MulVAL framework[16]. MulVAL uses XSB Prolog environment [39] to execute its rules. Our implementation uses a modified version of XSB to serve the same purpose. The differences of these two frameworks are mainly the extended security policies input to the system and the interaction of attack graph and

Full	Partitioned	Ring	Star	Tree
9.1	23.6	40.0	60.0	60.3

(a)

Full	Partitioned	Ring	Star	Tree
0.1	0.7	0.1	0.1	0.1

(b)

Table 1: % of From-Scratch Time for Single Host Addition (a) and Deletion (b)

analysis engine. For Windows XP®, we use the rules provided by Netra [12] and for SELinux™ [4] we use the rules provided by PAL[28]. Attack graph is shown to the user in the uDrawGraph environment [34]. uDrawGraph is a graph viewing software which has various abstraction function to hide/view/zoom graphs or part of it which is exposed to the user for easy navigation and view of attack graphs. It also takes graph input as in Prolog term format which is suitable to generate in Prolog environment. It exposes hooks which can be used to define user-defined function on the events. We use its API to present customized menu functionality for various analysis on attack graphs. We have used these features to expose interactive functionality to the attack graph. User can select facts nodes and delete/undelete it and see the effects on the attack graph. Based on user options, the changes to the graphical environment can effect the actual network and host, or can temporarily affect the facts existed in the Prolog environment without affecting the actual configuration. The user of the system can see the effect first and then decide to push the changes to the actual network.

In this section, we demonstrate the effectiveness of our algorithm. Performance measurements were taken on a PC with 1.6GHz Pentium®processor with 512MB of memory running Linux™. We have modified XSB [39] version 2.7.1 to include our algorithms.

The selection of benchmarks used in this paper is motivated from [18]. The benchmarks contain network configuration, machine configuration, and vulnerability information simulated for a variety of network sizes, and topologies. The tuples specify the allowed network traffic among machines in the network. Attacker's machine is located on the internet as the shown in our previous example. The vulnerabilities are set of *vulExists* and *vulProperty* tuples such that the same vulnerability exists on each of the simulated machines, and each vulnerability is a remote exploit of a service program at a unique protocol and port number. The policy states that the attacker is not allowed for root privilege in any of the machine in the network.

Five network topologies are simulated by generating the *hacl/4* facts. The "Fully Connected" (abbreviated as Full) network topology simulates network accessibility of all protocols and ports between every pair of machines. The "star" topology has one centralized machine that has two way accessibility to all protocols and ports of all machines. The non-centralized machines, among which are the attacking machines, have no direct access to any other machines. The "ring" topology has one machine of the ring connected to the internet, and all the other machines on the ring connected only to its two immediate neighbors with two way access to all protocols and ports. The "partitioned" (abbreviated as part) topology was simulated as equal sized fully connected networks connected to each other only by one pair of machines, one from each connected network. The "Tree" topology creates a network of tree shape, where Attacker's machine is connected to the root of the tree. All hosts have 2 software and 2 vulnerabilities each.

Attach Graph Generation. Attack graph generation timings for various network topologies and number of hosts are presented in

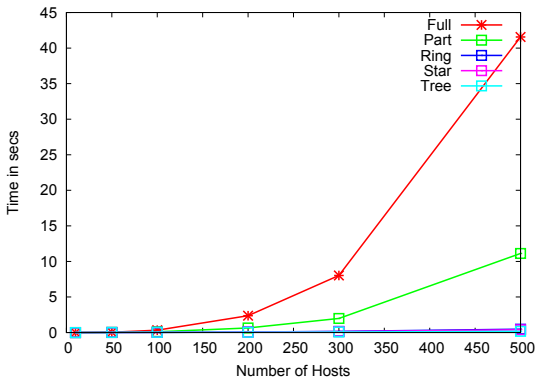

Figure 11: Attack Graph generation times

No. Links Added	Full	Partitioned	Ring	Star	Tree
1	8.0	9.4	68.6	53.7	59.8
10	8.0	10.1	70.4	52.6	61.0
50	9.8	9.8	72.8	54.4	63.9
100	9.9	9.8	73.2	53.4	58.6

(a)

No. Links Deleted	Full	Partitioned	Ring	Star	Tree
1	0.4	0.2	0.2	0.4	0.1
10	0.4	0.2	0.2	0.4	0.1
50	0.4	0.3	0.3	0.4	0.3
100	0.6	0.3	0.3	0.5	0.3

(b)

Table 2: % of From-Scratch Time for Links Addition (a) and Deletion (b)

Figure 11. The worst case is for fully connected network topologies. Note that our query time includes the attack graph generation time whereas [18] requires another step to generate the attack graph from each step. Thus, our algorithm for generation attack graph is empirically faster than the logical attack graph generation algorithm presented in [18].

Incremental Host Addition and Deletion. We experimented with adding and deleting one host into/from the network such that the entire structure of the network remains as it is. That is, when one host is added to full connected network then all links are added to all other host and for all ports. For example, for full connected graph we generate all the facts for 200 and 201 hosts, and take the difference of them. Note that we are adding here hacl/3, vulExists/3, and networkSvcInfo/5 facts. Once the query is performed on network of 200 hosts, all the difference facts are added using incremental addition builtin and the incremental time is noted. This is compared against the time required to perform query on 201 hosts. The percentage value of comparisons is presented in Table 1(a). Note that our algorithm takes 9-60% time for inserting single host on different topologies. The reason is that insertion is based on difference rule evaluation, which involves the join operation on relations. The difference rule based insertion algorithm do not take any advantage of fine grained dependency structure existed between answers which is explored in our deletion algorithm. Inspite of this, our insertion result takes less than 10% of time for 'full' topology for which the non-incremental attack graph generation time is the most.

In contrast, our incremental algorithm performs excellent in case of single host deletion (Table 1(b)). The incremental algorithm takes less than 1% of from-scratch time to delete single host from the network. The excellent incremental deletion time is due to exploiting fine grained dependency structure (i.e. answer dependency) and use of acyclic supports to stop propagation of changes. Incremental deletion algorithm does not involve any rule evaluation, and thereby considerable faster than incremental addition. Note that this example demonstrates ability of incremental algorithms to handle addition and deletion of multiple facts.

Incremental Link Addition and Deletion. We have performed experiments similar to the last one for random link addition and deletion in the network. This is done inserting or deleting *hacl/4* random facts for various number of links (1, 10, 50, 100). The result is shown in Table 2. We obtain the similar result as in case of single host addition/deletion.

In most of the other experiments which affect the attack graph or query computation like configuration changes of network, changes in vulnerability information into vulnerability, and inserting vulnerability into software in host, we have seen similar result as discussed before. In another experiment where we have just introduced a new vulnerability in the vulnerability database without including any software which has that vulnerability, the incremental addition time is almost nil, as no call is invalidated in the first invalidation phase.

Above experiments demonstrate that the incremental algorithm takes lesser time than from-scratch algorithm. Note that for incremental deletion of facts, our algorithm is more effective than handling addition. This is particularly useful for mutation analysis when administrators typically want to delete facts from the attack graph view in uDrawGraph to see its effect. Our framework also incorporates the analysis on the attack graphs namely the minimum cost attack removal strategy and minimum effort prediction of attackers by encoding them as logic programs. The result of such analysis typically boils down to removal of some configuration from the system which results into removal of some facts. Our incremental algorithm is, therefore, useful for mutation analysis on other analysis result.

6. RELATED WORK

In this section, we compare different aspects of our work with works in the areas of vulnerability analysis, and incremental algorithms.

Vulnerability Analysis using Attack Graph. MIT-Lincoin laboratory has produced an excellent review of the attack graph papers in [10]. Some of the earlier works by Sheyner, Ritchey, Amman, Wing, Jha, Jajodia have been well described there, and are not repeated here.

In a recent news article ([1]) Jajodia et al. described a system called CAULDRON (Combinatorial Analysis Utilizing Logical Dependencies Residing on Networks). CAULDRON can analyze and visualize vulnerabilities and attack paths, encouraging "what-if analysis". However, the efficiency problem in "what-if" analysis addressed here is not addressed in their work. CAULDRON utilizes hierarchical graph visualization technique described in [14, 13]. They presented a number of techniques to collapse various parts of the attack graph and transforming the graphs into adjacency matrix [15]. Noel et al. [37] proposed a symbolic equation simplifier to produce recommendations from the attack graphs. Wang et. al. [36] have used an attack graph variant called Queue graph for intrusion alert correlation.

Swarup et. al. have presented a framework in [32] for rule based vulnerability analysis. Their attack graph characterization is same as that of Sheyner's scenario graph and exponential in the size of the configuration. Their work do not present any experimental results. Wang et. al. [38] also discusses a relational model for representation and generation of attack graph. They use database query to perform analysis and generate attack graph using different set of

queries thereafter. In relational databases query evaluation is typically performed in a set-at-a-time basis (for example the join operation is performed on two sets). In that context generating attack graph is real challenging as attack graph captures the dependency between tuples. Infact, different set of queries are required for generation of attack graph for different analysis query to database. In contrast, XSB's top down evaluation is tuple-at-a-time [24] where it is easy to capture dependencies between tuples. They use the term *interactive analysis* which denotes the ability for administrator to define new queries and to evaluate them. They do not consider any incremental algorithm to re-evaluate each query. Moreover, note that generating and running new queries is extremely efficient in XSB's environment as the new queries can be resolved against the existing memoized information in the tables.

Skybox View^{TM} [2] is a commercial tool that performs attack graph analysis. The company's patent [6] describes their algorithm. Based on the patent, we believe that the company may have built a variant of a host-compromised graph, and may report only the shortest attack paths to a target.

MIT-Lincoin laboratory has produced an excellent review of the attack graph papers in [10]. They followed by two works ([21, 9]) on the attack graph. They present a tool called NetSPA which has the same motivation of MulVAL, creating scalable tool which can generate attack graph for thousands of host. However, their tool does not have any incremental capability. It would be an interesting problem to include incremental capability in their tool.

Closest to our formalism of attack graph is the Logical Attack Graph developed by Ou et. al. [22]. The basis of their work is a framework called MulVAL ([19, 17]) which uses Datalog [35] encoding of exploit rules and attack propagation rules, and evaluate the rules in logic programming system (they use XSB [39]) with respect to the factbase which contains network and host configuration, and vulnerability information. Similar logical techniques in the domain of system security was also explored in [22]. The comparison with MulVAL's algorithm to generate the attack graph is already presented in Section 3.

Naldurg et. al. ([12]) have presented a framework called Netra for access control analysis for detecting information flow vulnerability. Their framework is also based on Datalog which generates proof of every answer. Their exploit rules defining only information flow vulnerabilities differ from that of MulVAL as it does not take host or network configuration as input. In other words, their rules are directed towards finding individual vulnerabilities and not the effect of combining multiple exploits resulting multi-stage attacks. Notably their rules contain negation and their attack graph shows negative edges for the fact-subgoal that are not satisfied. Note that it is also possible to show such negative edges in MulVAL based logical attack graph using asserts. However, if the negated subgoal is a derived predicate it does not give relevant information to the user on why the derived goal had failed. Section 3 of this paper solves this representation problem. None of the above work including Netra do not incorporate incremental capability.

Singh et. al. in [30] have used a spreadsheet like framework called Deductive Spreadsheet to represent various security policies, and MulVAL rules. However, the main aim there was to perform query and thus attack graph is not built as a result of the analysis. Even though Deductive Spreadsheet is capable of incremental analysis using the algorithm presented in [27] which just explores call dependencies, the algorithm presented in this paper is more efficient than [27] as it exploits the answer dependencies for deletion propagation. [30] does not present any performance result on incremental maintenance of queries of MulVAL.

Incremental Algorithms. Incremental algorithms have been applied to various fields of research, viz. AI, View Maintenance, Program Analysis, Model Checking, Functional and Logic Programming. The main focus of those algorithms is to efficiently recompute the output in response to the change in the output. However, we do not discuss those algorithms which do not record the evaluation process as proofs to guide the incremental evaluation.

Among the works on materialized view maintenance, our deletion algorithm has similar two phases deletion and rederivation as in [8]. However, the deletion algorithm used here improves over [8] by using acyclic supports to restrict deletion propagation and unmarked supports for fast rederivation.

The product graph generated during the model checking process is used by incremental model checking algorithm of [31]. The graph is similar to attack graph; though, is only applicable for definite logic programs. Incremental attribute grammar evaluation [23] generates an acyclic dependency graph to record the functional dependencies among attribute in the non-circular attribute grammar. Another instance of an acyclic dependency graph is the *augmented dependency graph* [3] which records dependencies between input and output values in the execution of pure functional programs.

In the context of tabled logic program evaluation in XSB, various incremental algorithms are already developed by Saha et. al [25, 27]. [25] can handle only definite logic programs and use only answer dependency graph (or support graph) for deletion, and difference rules for addition. The algorithm presented in [27] can handle stratified negation by only exploiting call dependency to re-evaluate calls to update tables. However, the call graph based algorithm does not present great incremental efficiency[27]. The algorithm presented in this paper takes advantage of having both support graph and call graph. Note that among all the applications where incremental tabled evaluation is applied, attack graph is the first application where support and call graph have served the dual purpose of data representation and incremental efficiency. *The novelty of our algorithm is to judiciously interleave the support graph based incremental deletion algorithm and difference rule based insertion algorithm using topological order existed in call dependencies to handle stratified negation.* Note that to handle incremental algorithms for stratified negation it is usually required to execute addition and deletion algorithms for each strata before propagating effect of lower stratum to higher stratum [8]. An important characteristic of our algorithm is that it does not explicitly require the strata information from predicate dependency graph and uses just call dependencies for interleaving insertion and deletion.

7. CONCLUSION

In this paper, we have presented a logical framework for attack graph generation and maintenance. We have extended the MulVAL framework to include complex security policies. We have extended the concept of logical attack graph to include justification for why a negated subgoal has failed. Finally, we have presented an efficient algorithm for maintaining attack graph in response to changes in the network, and perform mutation analysis. Future work includes evaluation of the algorithm for real enterprise network, and an attack graph compression algorithm.

8. ACKNOWLEDGEMENT

We thank Dr. Subir Saha for supporting this work. We also thank anonymous reviewers, C. Manjari, and Anu Singh for their invaluable comments.

9. REFERENCES

[1] www.physorg.com/news124982803.html.

[2] www.skyboxsecurity.com.

[3] U. A. Acar, G. E. Blelloch, and R. Harper. Adaptive functional programming. In *ACM POPL*, volume 37, pages 247–259, New York, NY, USA, 2002. ACM Press.

[4] The National Security Agency. Security Enhanced Linux™.

[5] Paul Ammann, Duminda Wijesekera, and Saket Kaushik. Scalable, graph-based network vulnerability analysis. In *CCS '02*, pages 217–224, New York, NY, USA, 2002. ACM Press.

[6] G. Cohen et. al. System and method for risk detection and analysis in a computer network united states patent 6,952,779, october 2005.

[7] Sudhakar Govindavajhala and Andrew Appel. A Windows access control demystified. Tech. rep., princeton university, 2006.

[8] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *SIGMOD*, pages 157–166, 1993.

[9] Ingols K., Lippmann R., and Piwowarski K. Practical attack graph generation for network defense. In *Computer Security Applications Conference*, 2006.

[10] R. Lippmann and K. Ingols. An annotated review of past papers on attack graphs. Technical report, MIT Lincoln Laboratory, USA, March 2005.

[11] J. W. Lloyd. *Foundations of Logic Programming*. Springer, 1984.

[12] Prasad Naldurg, Stefan Schwoon, Sriram Rajamani, and John Lambert. Netra:: seeing through access control. In *FMSE '06: Proceedings of the fourth ACM workshop on Formal methods in security*, pages 55–66, New York, NY, USA, 2006. ACM.

[13] Steven Noel, Michael Jacobs, Pramod Kalapa, and Sushil Jajodia. Multiple coordinated views for network attack graphs. In *VizSEC*, page 12, 2005.

[14] Steven Noel and Sushil Jajodia. Managing attack graph complexity through visual hierarchical aggregation. In *VizSEC*, pages 109–118, 2004.

[15] Steven Noel and Sushil Jajodia. Understanding complex network attack graphs through clustered adjacency matrices. In *ACSAC*, pages 160–169, 2005.

[16] X. Ou, S. Govindavajhala, and A. W. Appel. MulVAL: A logic-based network security analyzer. In *14th USENIX Security Symposium*. Society for Industrial and Applied Mathematics, 2005.

[17] Xinming Ou. *A Logic-Programming Approach to Network Security Analysis*. PhD thesis, Department of Computer Science, Princeton University, USA, November 2005.

[18] Xinming Ou, Wayne F. Boyer, and Miles A. McQueen. A scalable approach to attack graph generation. In *CCS '06*, pages 336–345, New York, NY, USA, 2006. ACM Press.

[19] Xinming Ou, Sudhakar Govindavajhala, and Andrew W. Appel. Mulval: a logic-based network security analyzer. In *SSYM'05: Proceedings of the 14th conference on USENIX Security Symposium*, pages 8–8, Berkeley, CA, USA, 2005. USENIX Association.

[20] R. Paige and S. Koenig. Finite differencing of computable expressions. *TOPLAS*, 4(3):402–454, 1982.

[21] Lippmann R., Ingols K., Scott C., Piwowarski K., Kratkiewicz K., and Cunningham R. Validating and restoring defense in depth using attack graphs. In *MILCOM*, 2006.

[22] C. R. Ramakrishnan and R. Sekar. Model-based analysis of configuration vulnerabilities. *Journal of Computer Security (JCS)*, 10(1 / 2):189–209, 2002.

[23] T. Reps, T. Teitelbaum, and A. Demers. Incremental context-dependent analysis for language-based editors. *ACM Trans. Program. Lang. Syst.*, 5(3):449–477, 1983.

[24] K. Sagonas, Terrace Swift, and D. S. Warren. XSB as an efficient deductive database engine. In *ACM SIGMOD*, pages 442–453. ACM, 1994.

[25] D. Saha and C. R. Ramakrishnan. Incremental evaluation of tabled logic programs. In *International Conference on Logic Programming*, volume 2916 of *LNCS*, pages 389–406, 2003.

[26] D. Saha and C. R. Ramakrishnan. Incremental and demand-driven points-to analysis using logic programming. In *Principles and Practice of Declarative Programming*. ACM Press, 2005.

[27] D. Saha and C. R. Ramakrishnan. Incremental evaluation of tabled prolog: Beyond pure logic programs. In *Practical Aspects of Declarative Languages*, volume 3819 of *LNCS*, pages 215–229, Charleston, South Carolina, Jan 2006.

[28] Beata Sarna-Starosta and Scott D. Stoller. Policy analysis for security-enhanced linux. In *Proceedings of the 2004 Workshop on Issues in the Theory of Security (WITS)*, pages 1–12, April 2004. Available at http://www.cs.sunysb.edu/~stoller/WITS2004.html.

[29] Oleg Sheyner, Somesh Jha, and Jeannette M. Wing. Automated generation and analysis of attack graphs. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2002.

[30] Anu Singh, C. R. Ramakrishnan, I. V. Ramakrishnan, Scott Stoller, and David S. Warren. Security policy analysis using deductive spreadsheets. In *5th ACM Workshop on Formal Methods in Security Engineering (FMSE)*, Alexandria, Virginia, Nov 2007.

[31] O. V. Sokolsky and S. A. Smolka. Incremental model checking in the modal mu-calculus. In *CAV*, volume 818 of *LNCS*, pages 351–363, 1994.

[32] Vipin Swarup, Sushil Jajodia, and Joseph Pamula. Rule-based topological vulnerability analysis. In *MMM-ACNS*, pages 23–37, 2005.

[33] H. Tamaki and T. Sato. OLDT resolution with tabulation. In *International Conference on Logic Programming*, pages 84–98, 1986.

[34] uDraw(Graph). Available at `http://www.informatik.uni-bremen.de/uDrawGraph/en/uDrawGraph/uDrawGraph.html`.

[35] J.D. Ullman. *Principles of Database and Knowledge-base Systems, Volume II*. Computer Science Press, 1989.

[36] Lingyu Wang, Anyi Liu, and Sushil Jajodia. Using attack graphs for correlating, hypothesizing, and predicting intrusion alerts. *Computer Communications*, 29(15):2917–2933, 2006.

[37] Lingyu Wang, Steven Noel, and Sushil Jajodia. Minimum-cost network hardening using attack graphs. *Comput. Commun.*, 29(18):3812–3824, 2006.

[38] Lingyu Wang, Chao Yao, Anoop Singhal, and Sushil Jajodia. Interactive analysis of attack graphs using relational queries. In *DBSec*, pages 119–132, 2006.

[39] XSB. The XSB logic programming system. Available at `http://xsb.sourceforge.net`.