A Scalable Approach to Attack Graph Generation

Xinming Ou *
Purdue University
xou@cerias.purdue.edu

Wayne F. Boyer
Idaho National Laboratory
Wayne.Boyer@inl.gov

Miles A. McQueen
Idaho National Laboratory
Miles.McQueen@inl.gov

ABSTRACT

Attack graphs are important tools for analyzing security vulnerabilities in enterprise networks. Previous work on attack graphs has not provided an account of the scalability of the graph generating process, and there is often a lack of logical formalism in the representation of attack graphs, which results in the attack graph being difficult to use and understand by human beings. Pioneer work by Sheyner, *et al.* is the first attack-graph tool based on formal logical techniques, namely model-checking. However, when applied to moderate-sized networks, Sheyner's tool encountered a significant exponential explosion problem. This paper describes a new approach to represent and generate attack graphs. We propose logical attack graphs, which directly illustrate logical dependencies among attack goals and configuration information. A logical attack graph always has size polynomial to the network being analyzed. Our attack graph generation tool builds upon MulVAL, a network security analyzer based on logical programming. We demonstrate how to produce a derivation trace in the MulVAL logic-programming engine, and how to use the trace to generate a logical attack graph in quadratic time. We show experimental evidence that our logical attack graph generation algorithm is very efficient. We have generated logical attack graphs for fully connected networks of 1000 machines using a Pentium 4 CPU with 1GB of RAM.

Categories and Subject Descriptors

C.2.0 [**Computer-Communication Networks**]: General; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

General Terms

Security, Management

*As of August 14, 2006, Xinming Ou's affiliation is Kansas State University. This work was conducted when he was a post-doctoral research associate at Purdue University. We would like to thank Purdue University and CERIAS for supporting his work.

CCS'06, October 30–November 3, 2006, Alexandria, Virginia, USA.

Keywords

Attack graphs, enterprise network security, logic-programming

1. INTRODUCTION

When analyzing the security of an enterprise network, it is important to consider multi-stage, multi-host attacks. A determined attacker is not likely to stop at the machine he first compromises, but can be expected to try to penetrate deeper into the network by jumping from one machine to another. For this reason, configuring an enterprise network securely is a daunting task for human beings. There are many potential interactions among multiple hosts and components in a network, such that the configuration of one machine will affect the security of others in the network. It is therefore important to design automatic tools that can analyze the configuration of an enterprise network and find potential security vulnerabilities. Such a tool will not be very useful if it cannot inform a system administrator with detailed information about the discovered problems. In particular, an *attack graph* that illustrates all possible multi-stage, multi-host attack paths is crucial for a system administrator to understand the nature of the threats and decide upon appropriate countermeasures.

Various kinds of attack graphs have been proposed for analyzing network security [8, 11, 1, 5, 2, 13]. Although some of them addressed the scalability problem [1, 5], none of the works has shown solid evidence that the graph generation tool can scale to an enterprise network with realistic sizes. In practice it is desirable to compute attack graphs for enterprise networks with 1000 to 10,000 hosts. Lippmann and Ingols gave a good overview on various attack graph tools in the past [3]. It shows that "although research has made significant progress in the past few years, no system has analyzed networks with more than 20 hosts, and computation for most approaches scales poorly and would be impractical for networks with more than even a few hundred hosts."

Besides the scalability problem, many of the existing attack graph tools adopt an ad-hoc way to represent input information and output graph data structures. The graph generation tools often required various auxiliary inputs in custom-designed data format, and the resulting attack graphs are often hard to comprehend and use by a human. These have made those attack graph tools difficult to use in practice.

The work by Sheyner, *et al.* [11] is the first formal treatment of attack graphs. Sheyner uses model checking to compute multi-stage, multi-host attack paths in a network. The state of the network is formally modeled as a collection

of Boolean variables, representing configuration parameters and attacker's privileges. Attacker's actions are modeled as state-transition relations. The security property of the network is specified as a temporal formula, which can be automatically checked against the model by a model checker. Unlike a traditional model checker, which only outputs one counter example when the temporal formula is not satisfied, Sheyner's tool can output all counter examples in the form of a *scenario graph*. In the case of network security, the scenario graph is an attack graph illustrating all the multi-stage, multi-host attack paths that can potentially break a network's security property.

A formal, logic-based approach to attack graph generation, like the one by Sheyner, is advantageous compared to ad-hoc graph generation methods. Using mature logical techniques is less error-prone than custom-designed algorithms, especially for the complex problem of security analysis. A clear logical semantics for attack graphs also makes it easier to conduct further analysis based on the graph data structure. However, a logic-based attack graph tool must scale well with the size of the network to make it feasible to use in practice. When we tried to apply Sheyner's tool to analyze real world networks we found that the graph generation time and graph size were prohibitively large. For example, a network of only 10 hosts with 5 vulnerabilities per host takes about 15 minutes to generate and results in a graph of 10 million edges.

We observe that in Sheyner's attack graph, every node is a collection of Boolean variables encoding the *entire* network state at an attack stage. While the number of variables is polynomial in the size of the network, the possible number of states is exponential. Even though not all the states are reachable in the search, the potential state explosion associated with a model-checking based methodology makes it impractical except for small networks with very few vulnerabilities.

In this paper, we propose a new logic-based approach for representing and generating attack graphs. In our representation, a node in the graph is a logical statement. This logical statement does not encode the entire state of the network, but only some aspect of it. In some sense it can be viewed as one Boolean variable in the nodes of Sheyner's graph. The edges in the graph specify the causality relations between network configurations and an attacker's potential privileges. Intuitively, Sheyner's attack graph illustrates snapshots of attack steps, or "how the attack can happen", whereas our attack graph illustrates causes of the attacks, or "why the attack can happen". To differentiate these two kinds of attack graphs, we call the former "scenario attack graphs", and the latter "logical attack graphs".

One advantage of a logical attack graph is that it clearly specifies the causality relations between system configuration information and an attacker's potential privileges. In a scenario attack graph, one would have to delve into the Boolean variables and follow several steps upstream to identify what really causes the adverse situation that enables an attacker's action at a stage. In a logical attack graph, such causality would be specific as graph edges. From a logical attack graph, it is also possible to enumerate all possible attack scenarios by depth-first traversing. Most importantly, the size of a logical attack graph is always polynomial in the size of the network, whereas in the worst case a scenario attack graph's size could be exponential.

In the past people have proposed "exploit dependency graph" as a way to represent computer attacks [5, 2]. In an exploit dependency graph, the pre- and post-conditions for exploits are encoded as graph nodes and edges. Semantically this is equivalent to our logical attack graph. Our contribution is to formally represent such dependency relations in the form of logic, and generate attack graphs through automatic logic deduction, as opposed to a custom-designed graph search algorithm. We believe the logic-based approach has the advantage of better clarity and trustworthiness.

Our logical attack graphs require that the cause of an attacker's potential privileges be expressible as a propositional formula in terms of network configuration parameters. Attack conditions that cannot be expressed in propositional formulas cannot be captured by logical attack graphs. In our experience, we have not found any situations where this becomes a problem. After all, all computer attacks have a cause, and unsurprisingly those causes are almost always rooted in misconfigurations. For a security tool that aims at finding such misconfigurations, the semantics of logical attack graphs exactly suits that goal.

1.1 Security analysis based on logic programming

The work presented in this paper is based on the MulVAL project [6]. MulVAL is a reasoning system for automatically identifying security vulnerabilities in enterprise networks. The key idea behind MulVAL is that most configuration information can be represented as Datalog[1] tuples, and most attack techniques and OS security semantics can be specified using Datalog rules. Following is a Datalog rule for the remote exploit of a privilege-escalation vulnerability in a service program.

```
execCode(Attacker, Host, User) :-
  networkService(Host, Program,
                 Protocol, Port, User),
  vulExists(Host, VulID, Program,
            remoteExploit, privEscalation),
  netAccess(Attacker, Host, Protocol, Port).
```

Figure 1: An interaction rule for remote exploit.

Capitalized identifiers are variables in Datalog and can be instantiated with any concrete term during evaluation. This is a generic rule specifying the pre- and post-condition for this attack: if `Program` is running as `User` on `Host` as a service listening on `Protocol` and `Port`, it contains a remotely exploitable vulnerability whose impact is privilege escalation, and the attacker can access the service through the network, then the attacker can execute arbitrary code on the machine as `User`. Logically, "`:-`" can be replaced with a reversed "implies" connector. The rule can be viewed as a logical formula $\forall\Phi.Rule$, where Φ contains all the Prolog variables appearing in the rule.

Predicates such as `vulExists` and `networkService` are "primitive" and they represent configuration information reported by host and network scanners. Predicates such as `execCode` and `netAccess` are "derived" and they are computed from the configuration information by iteratively applying the interaction rules on the input. The architecture

[1]A syntactic subset of Prolog

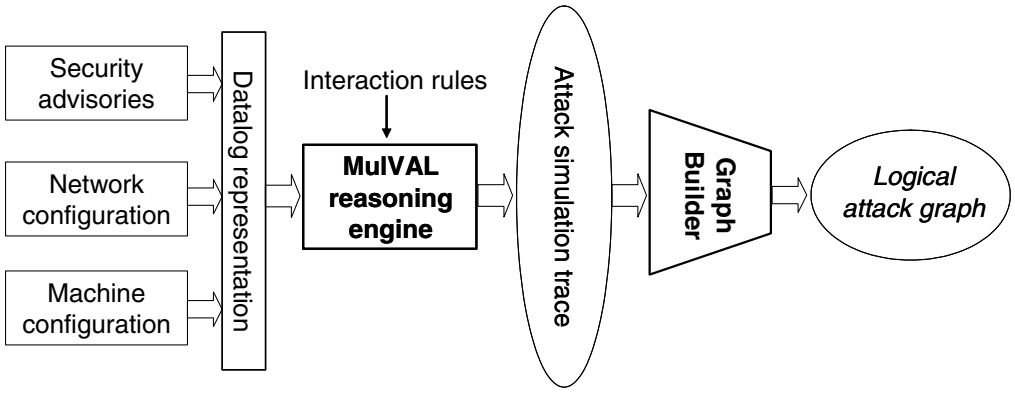

Figure 2: Logical Attack Graph Generator

of the Logical Attack Graph tool is illustrated in Figure 2. The MulVAL reasoning engine uses XSB [9], a Prolog system developed by StonyBrook, to evaluate the Datalog interaction rules on input facts. We modified the MulVAL engine so that the trace of the evaluation is recorded and sent to the graph builder, where the logical attack graph is output.

Our logical attack graph is in essence a derivation graph for Datalog programs. XSB, like other Prolog systems, does not provide functionalities for generating proofs of a successful query. The justifier program for XSB [7] can record execution trace for interactively retrieving reasons for a successful Prolog query, but does not provide the capability of constructing derivation graphs that illustrate all possible derivations. We use a similar approach to XSB's justifier program to record derivation steps when evaluating the MulVAL Datalog program. Those derivation steps are called *attack simulation trace*, which contains sufficient information to construct a logical attack graph.

In the following sections, we will review some of the related works. We will then give a formal definition of logical attack graphs, describe the algorithms for constructing them, and analyze the complexity of the algorithms. We have implemented the algorithm in a combination of Prolog and C++. Experimental results show that our logical attack graph tool is very efficient and can handle networks with thousands of machines.

2. RELATED WORK

Sheyner *et al.* uses model checking techniques to compute attack graphs [11]. We encountered significant scalability problems in applying this tool. One reason for the blow up is that there are many duplicate attack paths in the graph that differ only in the order in which independent attack steps are attempted. Partial-order reduction can remove such duplicate paths, but it has not been shown that the technique can significantly improve the scalability for attack graphs. Even after removing such duplicate paths, the resulting graphs could still be exponential. We also find it is hard to decode the meaning of the Boolean values in a node, and logical correlation among nodes is not always obvious.

Philips and Swiler developed a tool for generating attack graphs [8, 12] in 1998. Like the model checking approach, the nodes in their attack graphs represent the state of the network in the form of a collection of variables, and the edges represent an attacker's actions that change the state. Instead of using a model checker, Philips and Swiler developed a customized search engine to conduct the analysis. Like Sheyner's work, this state-based attack graph representation has inherent exponential problems, and such explosion was indeed reported by the authors. They hence used a technique similar to partial-order reduction to eliminate the duplicate attack paths that contributed to the explosion, but it is not clear from the paper how effective this method has been and no performance data was given.

Ammann, *et al.* also noticed the scalability problem in the model checking-based attack graph tool, and proposed a graph search-based algorithm, which was then used in the Topological Vulnerability Analysis tool [2]. They pointed out that for most computer attacks, one can assume the *monotonicity property*, where an attacker does not decrease his ability by launching attacks, and hence does not need to relinquish privileges he already gained. Under this assumption, an attacker's privileges always increase during the analysis. Since there are only a polynomial number of privileges an attacker can gain, the analysis algorithm will terminate in polynomial time. Our logical attack graph gives another perspective for this monotonicity property. We observe that most attacks, whether monotonic or non-monotonic, have rooted causes in configuration information. Thus, at an appropriate level all those attacks' preconditions can be specified using propositional formulas on configuration information. In some sense non-monotonic attacks can be treated as monotonic if one ignores the low-level details on how the attack can happen. For this reason simple Datalog rules can capture almost all kinds of attack conditions in a network. Ammann gave a theoretical upper bound for their algorithm as $O(|A|^2 \cdot |E|)$, where $|A|$ is the number of "attributes" (describing attack pre- and post-conditions) and $|E|$ is the number of "exploits". The paper stated that typically an exploit involves two hosts, yielding a quadratic number of concrete exploits. The paper did not discuss the number of attributes in terms of network size. We believe the attributes should include host connectivity information, thus the number of attributes is also quadratic in the number of hosts. This will give us an $O(N^6)$ complexity. It is certainly a very conservative estimation of the algorithm's complexity, but we could not find experimental data showing the performance of the proposed algorithm on large configuration settings. In this paper we will show an algorithm that has $O(N^2)$ complexity

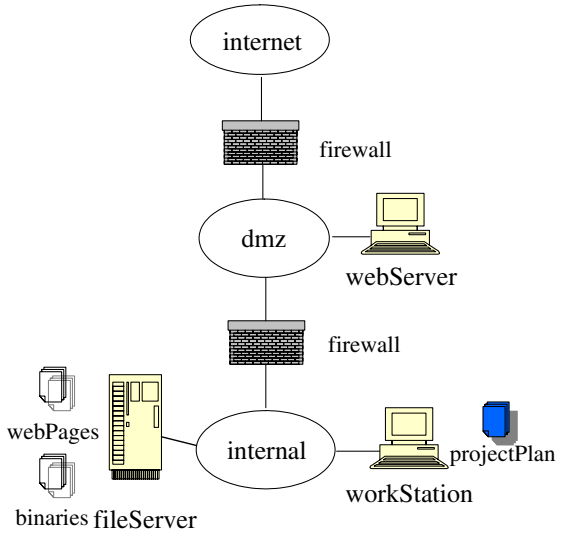

Figure 3: Example

under the assumption of constant table look-up time, and experimental results that demonstrate the worst-case running time of our graph generator grows between $O(N^2)$ and $O(N^3)$ for networks from a handful to a thousand machines.

Noel and Jajodia gave a good review on various representations of attack graphs in their work on using aggregation techniques to manage attack graph complexity [4]. Their work finally chose the "exploit-dependency graph". An exploit-dependency graph can be viewed as a logical attack graph and the two can be translated back and forth. However, we believe an attack-graph that explicitly uses predicates and logical connectives to represent security correlations in a network is better suited for rigorous security analysis and hardening. Noel's work focuses on aggregation techniques for exploit dependency graphs, and does not describe how the attack graphs can be built from configuration information or the scalability of the graph building process.

Schneier introduced the general idea of *attack trees* for representing security threats [10]. The logical attack graph presented in this paper is a special case of Schneier's attack tree. We apply the idea to the specific problem of enterprise network security, formally define the semantics of the attack graph in this context, and describe algorithms for automatically computing attack graphs from network and machine configuration information. We also show experimental evidence on the scalability of our approach.

Our logical attack graph toolkit was based on the MulVAL toolkit. The original MulVAL work [6] did not have the ability to compute complete attack graphs. Rather, separate attack paths can be output by using Prolog's meta-programming techniques. Even for a polynomial attack simulation process, the number of unique attack paths could be exponential in the worst case. And we indeed experienced such exponential blow-up when applying MulVAL's meta-programming based attack path generator to analyze a real network with 20 machines. The logical attack graph tool described in this paper has the ability of generating complete attack graphs for networks with thousands of machines.

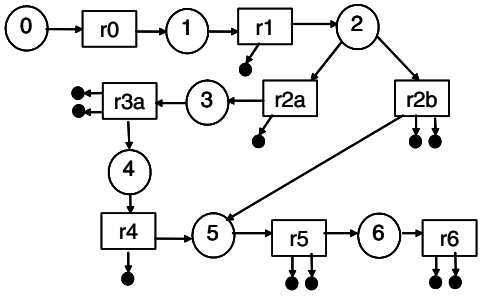

Figure 4: An example logical attack graph

3. LOGICAL ATTACK GRAPH

The example network in Figure 3 is directly borrowed from the MuLVAL paper [6]. Suppose the following potential attack paths are discovered after analyzing the configuration.

An attacker first compromises `webServer` by remotely exploiting vulnerability `CVE-2002-0392` to get local access on the server. Since `webServer` is allowed to access `fileServer` through the NFS protocol, he can then try to modify data on the file server. There are two ways to achieve this. If there are vulnerabilities in the NFS service daemons, he can try to exploit them and get local access on the machine; or if the NFS export table is not set up appropriately, he can modify files on the server through the NFS protocol by using programs like NFS Shell[2]. Once he can modify files on the file server, the attacker can install a Trojan-horse program in the executable binaries on `fileServer` that are mounted by machine `workStation`. The attacker can now wait for an innocent user on `workStation` to execute it and obtain control on the machine.

The logical attack graph corresponding to the above scenarios is illustrated in Figure 4. A logical attack graph is a directed graph and can be represented in the form of a tree with possible cross links between nodes. Figure 5 shows the tree representation of the same attack graph. There are two kinds of nodes in the graph: a *derivation node* and a *fact node*. A derivation node is represented as a rectangle and a fact node is represented as a circle. There are also two kinds of fact nodes: a primitive fact node (represented as a solid small circle), and a derived fact node (represented as a circle with a number in it).

Every fact node in a logical graph is labeled with a logical statement in the form of a predicate applied to its arguments. The root node is the attack goal; in the example it is `execCode(attacker,workStation,root)`, meaning "the attacker can execute arbitrary code as user `root` on machine `workStation`". Every derivation node is labeled with an interaction rule that is used for the derivation step. In the tree representation, every internal node is started with a node number in a bracket, followed by the node's label. A leaf node does not have a node number and is led by an empty square bracket.

The edges in the graph represent the "depends on" relation. A fact node is dependent on one or more derivation nodes, each of which represents an application of an inter-

[2]Downloadable from http://www.deter.com/unix/software/nfsshell.c

```
<0>|--execCode(attacker,workStation,root)
   <r0>Rule5: Trojan horse installation
     <1>|--accessFile(attacker,workStation,write,/usr/local/share)
        <r1>Rule14: NFS semantics
             []-nfsMounted(workStation,/usr/local/share,fileServer,/export,read)
          <2>||--accessFile(attacker,fileServer,write,/export)
             <r2a>Rule10: execCode implies file access
                  []-fileSystemACL(fileServer,root,write,/export)
               <3>|--execCode(attacker,fileServer,root)
                  <r3>Rule3: remote exploit of a server program
                       []-networkServiceInfo(fileServer,mountd,rpc,100005,root)
                       []-vulExists(fileServer,CVE-2003-0252,mountd,
                                    remoteExploit,privEscalation)
                    <4>|--netAccess(attacker,fileServer,rpc,100005)
                       <r4>Rule6: multi-hop access
                            []-hacl(webServer,fileServer,rpc,100005)
                         <5>|--execCode(attacker,webServer,apache)
                            <r5>Rule3: remote exploit of a server program
                                 []-networkServiceInfo(webServer,httpd,tcp,80,apache)
                                 []-vulExists(webServer,CAN-2002-0392,httpd,
                                              remoteExploit,privEscalation)
                               <6>|--netAccess(attacker,webServer,tcp,80)
                                  <r6>Rule7: direct network access
                                       []-hacl(internet,webServer,tcp,80)
                                       []-located(attacker,internet)
             <r2b>Rule15: NFS shell
                  []-hacl(webServer,fileServer,rpc,100003)
                  []-nfsExportInfo(fileServer,/export,write,webServer)
                  |--execCode(attacker,webServer,apache)==> <5>
```

Figure 5: An example logical attack graph, tree representation

action rule that yields the fact; A derivation node is dependent on one or more fact nodes, which together satisfy the preconditions of the rule. Thus a logical attack graph is a bipartite directed graph. The derivation nodes serve as a medium between a fact and its "reasons", i.e., how the fact becomes true. Since a fact may have different ways to become true, the derivation nodes directed from a fact node form a disjunction. A derivation node represents a successful application of an interaction rule, where all its preconditions are satisfied by its children. Thus the fact nodes directed from a derivation node form a conjunction.

For example, node 2 has two derivation nodes as its children: r2a and r2b (note that the tree representation uses `||` to signify that a fact node has more than one derivation). That is, there are two ways the attacker can modify files on `fileServer`. One way is to get root on file server by exploiting bug `CVE-2003-0252` in the `mountd` program, and the other is to use the `NFS Shell` program. Both depend on the condition that an attacker already gained some access on `webServer` (node 5). In the tree representation, there is a cross link (`==> <5>`) pointing to node 5 in the second derivation branch.

A logical attack graph can be viewed as a derivation graph for a successful Datalog query. There may be many different ways to derive a fact in Datalog (corresponding to multiple paths to break into a network), thus we explicitly introduced the derivation node to represent one possible derivation step. Logically, a derivation node is an "and" node, where all its children are the arguments of a conjunction that derives the node; a derived fact node is an "or" node, where all its children represent different ways to derive them. A primitive fact node is a leaf node in the graph. It represents a piece of configuration information. Following is the formal definition of our logical attack graph.

Definition 1. $(N_r, N_p, N_d, E, \mathcal{L}, \mathcal{G})$ is a logical attack graph, where N_r, N_p and N_f are three sets of disjoint nodes in the graph, $E \subset (N_r \times (N_p \cup N_d)) \cup (N_d \times N_r)$, $\mathcal{L}$ is a mapping from a node to its label, and $\mathcal{G} \in N_d$ is the attacker's goal.

N_r, N_p and N_d are the sets of derivation nodes, primitive fact nodes, and derived fact nodes, respectively. A fact is primitive if it comes from the input to the MulVAL reasoning engine. A derived fact is the result of applying interaction rules iteratively on the input facts. The edges in a logical attack graph can only go from a derived fact node to a derivation node, or from a derivation node to a fact node. The labeling function maps a fact node to the fact it represents, and a derivation node to the rule that is used for the derivation. Formally, the semantics of a logical attack graph is defined as follows.

PROPERTY 1. *For every derivation node R, let P be R's parent node and C be the set of R's child nodes, then $(\wedge\mathcal{L}(C)) \Rightarrow \mathcal{L}(P)$ is an instantiation of interaction rule $\mathcal{L}(R)$.*

Here $\wedge$ is the conjunction operator. For example, the derivation node `r3` is an application of the interaction rule shown in Figure 1. The free variables in the rule have all been instantiated with ground terms.

4. ALGORITHMS

We modified the MulVAL reasoning engine so that besides a "yes" or "no" answer, a Datalog query also records an attack simulation trace as a side effect of the evaluation. This is achieved by a source-to-source translation of MulVAL interaction rules. For example, for the interaction rule shown in Figure 1, it will be translated into the following form:

```
execCode(Attacker, Host, User) :-
  networkService(Host, Program,
                 Protocol, Port, User),
  vulExists(Host, VulID, Program,
            remoteExploit, privEscalation),
  netAccess(Attacker, Host, Protocol, Port),
  assert_trace(because(
 'remote exploit of a server program',
   execCode(Attacker, Host, User),
     [networkService(Host, Program,
                     Protocol, Port, User),
      vulExists(Host, VulID, Program,
                remoteExploit, privEscalation),
      netAccess(Attacker, Host,
                Protocol, Port)])).
```

In addition to the sub-goals in the original rule, a new sub-goal is added which calls the function `assert_trace`. When the evaluation of the rule succeeds, this function will record the successful derivation into a trace file. In essence this method for recording execution traces is similar to the one used by the "justifier" program in XSB [7]. The attack simulation trace has the following format.

Definition 2. Attack simulation trace.

TraceStep	: : =	**because**(*interactionRule*, *Fact*, *Conjunct*)
Fact	: : =	*predicate*(*list of constant*)
Conjunct	: : =	*[list of Fact]*

interactionRule is a string uniquely associated with a MulVAL interaction rule. A list is represented as a series of items separated by commas. The semantic meaning of a trace step is "$Conjunct \Rightarrow Fact$ is an instantiation of *interactionRule*". It records the reason why a goal is true during Datalog evaluation.

In order to compute attack graphs that contain all possible attack paths, the logic engine must traverse all possible derivation paths and record trace steps in the process. The XSB logic engine used in MulVAL is a Prolog system that supports tabled execution [15]. Tabling is a form of memoization that can both save computation time and resolve cycles in the derivation. The added `assert_trace` predicate will be called whenever XSB successfully satisfies all the preconditions of an interaction rule. Under tabled execution, *all* possible answers to a query will be computed. Thus the logic engine will have traversed all possible derivation paths before returning.

A logical attack graph can be constructed from the trace step information. The algorithm is depicted in Figure 6. In simple words, every *TraceStep* term becomes a derivation node in the attack graph. The *Fact* field in the trace step becomes the node's parent and the *Conjunct* field becomes its children. The maximum number of iterations for the inner loop at line 7 is the same as the largest number of pre-conditions among all the interaction rules, which is constant for a fixed interaction rule set. Thus, if the look-up operation on line 4 and line 8 is constant time, the graph building algorithm takes time linear in the number of trace steps.

4.1 Loops in attack graphs

Even though the interaction rules contain cycles, the XSB logic engine can avoid entering an infinite loop through tabling. However, the resulting trace steps can still contain loops

Input: set $\mathcal{T}$ containing all the *TraceStep* terms, attacker's goal $\mathcal{G}$
Output: logical attack graph $(N_r, N_p, N_d, E, \mathcal{L}, \mathcal{G})$.

1. $N_r, N_p, N_d, E, \mathcal{L} \leftarrow \emptyset$
2. For each $t \in \mathcal{T}$ {
   let $t =$ **because**(*interactionRule*, *Fact*, *Conjunct*)
3. Create a derivation node r
   $N_r \leftarrow N_r \cup \{r\}$
   $\mathcal{L} \leftarrow \mathcal{L} \cup \{r \rightarrow interactionRule\}$
4. Look up $n \in N_d$ such that $\mathcal{L}(n) = Fact$,
5. If such n does not exist {
   create a new fact node n
   $\mathcal{L} \leftarrow \mathcal{L} \cup \{n \rightarrow Fact\}$
   $N_d \leftarrow N_d \cup \{n\}$
   }
6. $E \leftarrow E \cup \{(n, r)\}$
7. For each fact f in *Conjunct* {
8. Look up fact node $c \in (N_p \cup N_d)$ such that $\mathcal{L}(c) = f$,
9. If such c does not exist {
   create a new fact node c
   $\mathcal{L} \leftarrow \mathcal{L} \cup \{c \rightarrow f\}$
   If f is primitive { $N_p \leftarrow N_p \cup \{c\}$ }
   else $\{N_d \leftarrow N_d \cup \{c\}$ }
   }
10. $E \leftarrow E \cup \{(r, c)\}$
   }
}

Figure 6: Graph building algorithm

from those cyclic rules. For example, the following two interaction rules form a cycle.

```
Rule 'execCode implies file access':
accessFile(Attacker, Host, write, Path) :-
               execCode(Attacker, Host, root).

Rule 'Trojan horse installation':
execCode(Attacker, Host, root) :-
   accessFile(Attacker, Host, write, Path).
```

An attacker can write files on a machine if he can execute arbitrary code on the machine as root; conversely, if an attacker can modify files on a machine, he can install a Trojan horse on it and potentially become root as well. A standard Prolog system will enter an infinite loop when encountering such rules, even though the program has a well defined meaning under the least fixed point semantics. A tabled Prolog system such as XSB will not loop. However, if both rules' preconditions are satisfied, both of them will be evaluated, in which case the output trace file will contain trace steps that form a loop. For example,

```
because('execCode implies file access',
     accessFile(attacker,workStation,
                write,/usr/local/share),
     [execCode(attacker, workStation, root)]).

because('Trojan horse installation',
     execCode(attacker, workStation, root),
     [accessFile(attacker,workStation,
                 write,/usr/local/share)]).
```

Such loops often times render useless information in the attack graph. For example, the above two trace steps would introduce a loop in the attack graph of Figure 4. We show

the loop in Figure 7 (only relevant derived fact nodes are shown; the derivation nodes and primitive fact nodes are ignored).

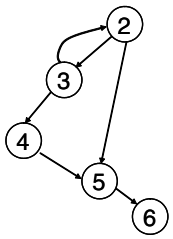

Figure 7: A loop in the example

Obviously, the back edge from node 3 to node 2 is meaningless, because the reason node 2 can be true is that node 3 is true in the first place. Such back edges should be eliminated from the graph. One may tend to think that a standard directed graph DFS algorithm that removes all the back edges will give us a correct DAG that represents all meaningful attack paths. However, this is not the case. Consider the two attack graphs in Figure 8. When starting from node 1, both (2, 3) and (3, 2) could be back edges, depending on the order in which the DFS algorithm traverses node 1's two child nodes. In case (a), only edge (3, 2) should be removed. If edge (2, 3) were removed, the attack path (1, 2, 3, 4) would be lost. Note that path (1, 3, 2) is not a valid derivation because it does not end in a leaf node. For case (b), neither (2, 3) nor (3, 2) should be removed, because then either attack path (1, 2, 3, 4) or (1, 3, 2, 5) would be lost.

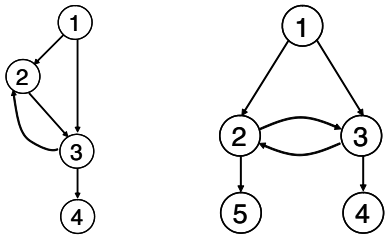

Figure 8: Two more loop examples

In general, an edge in an attack graph does not provide any useful information if it does not contribute to any valid logical derivation for attack goals. We call such edges "useless edges". To determine if an edge (u, v) is useless, we can remove u from the graph and test if v can still be derived in the graph. This can be done by a DFS search from v. If so the edge is not useless because there is a derivation path for v that does not involve u; otherwise the edge is useless and should be removed. The algorithm for finding all useless edges in an attack graph is at most quadratic in the size of the graph. We leave the implementation of this algorithm for future work. We also note that loops are not unique phenomena for logical attack graphs; they exist in other attack graph works as well, although we have not seen the problem addressed in the past.

5. COMPLEXITY ANALYSIS

The process of computing a logical attack graph consists of two stages. The first stage computes attack simulation traces through Datalog evaluation in XSB; the second stage builds attack graph data structures using the algorithm in Figure 6.

5.1 Complexity of computing attack trace

The generation of attack trace only introduces a constant-time overhead for every successful Datalog derivation. So the complexity of the first stage is the same as the complexity of evaluating the MulVAL Datalog program in XSB. The complexity of evaluating a fixed Datalog program against variable size inputs depends on the particular details of the program. The XSB documentation has some discussion on how to determine the complexity of evaluating a tabled Datalog program in XSB [14]. To make it easy to understand, let's consider the following Datalog interaction rule in MulVAL:

```
netAccess(Attacker, H2, Protocol, Port) :-
        execCode(Attacker, H1, _User),
        hacl(H1, H2, Protocol, Port).
```

The meaning of the rule is: if an attacker can become a local user on machine `H1`, and the network allows `H1` to access `H2` through `Protocol` and `Port`, then the attacker can access `H2` through the protocol and port. This rule illustrates multi-hop network access in a network: an attacker can use a machine he controls as a stepping stone to compromise other machines.

When XSB evaluates this rule, it will first compute all possible machines an attacker can execute arbitrary code on (the first sub-goal), and then it will exhaustively search all `H1` and `H2`'s between which network access is possible (the second sub-goal). When all these tuples are computed, the goal predicate `netAccess` will be computed by matching the results of the two sub-goals. Pattern matching in XSB is very efficient due to the use of hash tables and tries. So the time spent is dominated by the number of intermediate tuples that need to be computed. The intermediate computation may of course invoke other interaction rules. In XSB's tabled execution, an invocation will compute *all* results of that goal, thus they can be reused later. One can think that all the rules are evaluated simultaneously in parallel with all possible instantiation of variables in their bodies. Each rule's evaluation time is determined by the number of different instantiations it needs to try. For a fixed Datalog program, the total running time is dominated by the rules that has the maximum number of different instantiations for the variables in its body.

THEOREM 1. *Evaluating MulVAL interaction rules against configuration tuples representing N hosts takes $O(N^2)$ derivation steps.*

PROOF. In MulVAL, the rule that has the most number of different body-variable instantiations happens to be the one we have shown above. In this rule two variables, `H1` and `H2`, can be instantiated with every possible machine in the network; the other variables in the rule are not affected by the size of the network. Thus there are $O(N^2)$ possible instantiations for this rule. □

If the pattern matching in XSB is constant time, every derivation step needs constant time to finish and the overall running time for MulVAL Datalog evaluation will be quadratic. In our experiments we have seen a slightly higher growth than quadratic in the worst test case, due to increased time in pattern matching for large inputs.

Since every trace step was produced by one derivation step in Datalog evaluation, we have the following:

COROLLARY 1. *The number of trace-step terms produced in attack simulation is* $O(N^2)$.

5.2 Complexity of graph building

THEOREM 2. *The logical attack graph for a network with* N *machines has a size at most* $O(N^2)$.

PROOF. There is a one-to-one correspondence between *TraceStep* terms and derivation nodes. Let D be the number of trace steps, then there are D derivation nodes in the graph. If the maximum number of preconditions for an interaction rule is m, the number of edges in the graph is at most mD, and the maximum number of fact nodes is $mD+1$. By Corollary 1 we know D is $O(N^2)$, and so is $mD+1$. □

THEOREM 3. *The graph building algorithm in Figure 6 takes* $O(\delta N^2)$ *time to complete, where* N *is the number of hosts in the network, and* δ *is the maximum time spent in table look up at line 4 and 8 of the algorithm.*

PROOF. The loop in the graph building algorithm in Figure 6 goes through all the *TraceStep* terms. By Corollary 1, we know there are $O(N^2)$ such terms. In each iteration, the algorithm creates a derivation node for the *TraceStep* term and makes links from its parent and to its children. Every operation is constant time except for table look-up at line 4 and 8. □

Thus the time needed to construct the graph data structure is quadratic in the number of hosts, given a constant-time lookup table to store graph nodes. For our implementation we simply used the "map" container in C++'s standard library, which has $log(n)$ look up time. From Theorem 2 we know that the table size is $O(N^2)$. So $\delta = log(N^2)$ and the graph generation running time will be $O(N^2 log(N))$.

The next section shows performance data from our experiments.

6. EXPERIMENTAL RESULTS

Our Logical Attack Graph Generator was tested in the following environment. The CPU was a Pentium 4 3.2 GHz with 1GB of RAM, the operating system was Microsoft Windows XP Professional Version 2002 Service Pack 2, with XSB version 2.7.1.

The network configuration, machine configuration and vulnerability information were simulated for a variety of network sizes, topologies and vulnerability densities. The network configuration was simulated by the creation of a set of `hacl(hostname1,hostname2,protocol,port)` Datalog tuples as input to the MulVAl reasoning engine. Those tuples specify the allowed network traffic among machines in the network, including the attacker machine located on the Internet. The vulnerabilities were simulated by the creation of a set of `vulExists` and `vulProperty` Datalog tuples such that the same vulnerabilities exist on each of the simulated machines, and each vulnerability is a remote exploit of a service program running at a unique protocol and port number.

The "fully-connected" network topology simulation specified network accessibility of all protocols and ports between every pair of machines. The "star" topology was simulated to consist of one centralized machine (not the target machine) that has two-way accessibility of all protocols and ports to every other machine. The non-centralized machines, among which are the attacking and target machines, have no direct network access to any other machine. The "ring" topology was simulated with one machine (not the target 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 of all protocols and ports. The "partitioned" topology was simulated as two approximately equal sized fully-connected networks connected to each other only by one pair of machines (neither is the target machine), one on each sub-network. The only connection to the Internet is through a third machine located on the subnet that does not contain the target machine.

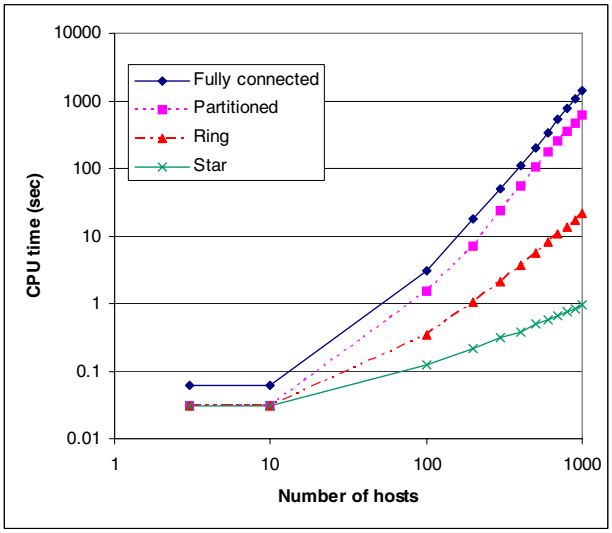

Figure 9: Graph generation CPU usage as a function of network size for several network topologies.

Figure 9 shows the graph generation CPU time for each of the simulated analysis problems of various sizes and topologies. The worst case is for a fully connected network. In this case the asymptotic CPU time is between $O(n^2)$ and $O(n^3)$, where n is the number of hosts. In the discussion of Section 4, we noted that ideally the complexity is $O(n^2)$, if table look-up takes constant time. However, our implementation uses the simple "map" template in C++ standard library and its look-up time depends on the size of the table. We believe after we replace it with a custom-designed hash table implementation the graph generation time will be near quadratic even for the worst case.

Figure 10 shows the memory usage as a function of network size for the same four network topologies. The worst case here is again a fully connected network, which has an asymptotic memory usage slightly lower than $O(n^2)$. In the

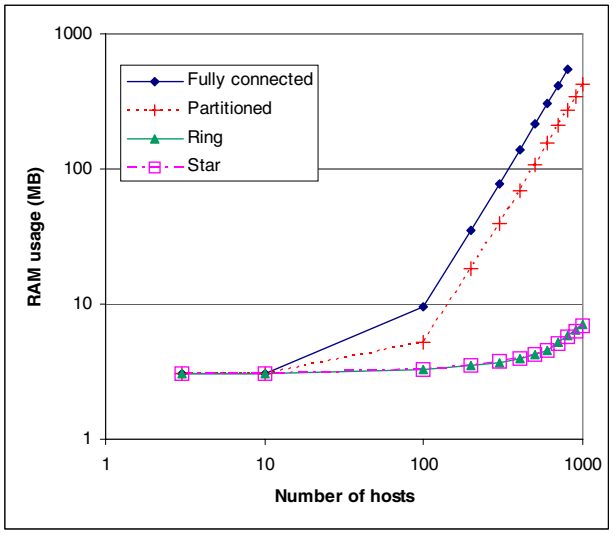

Figure 10: Graph generation memory usage as a function of network size for several network topologies.

two biggest cases (1000 host for fully-connected and partitioned network), we almost exhausted the 1GB memory on the test machine. The memory usage for the "star" and "ring" topology are not identical, although the difference is not visible on logarithmic scale.

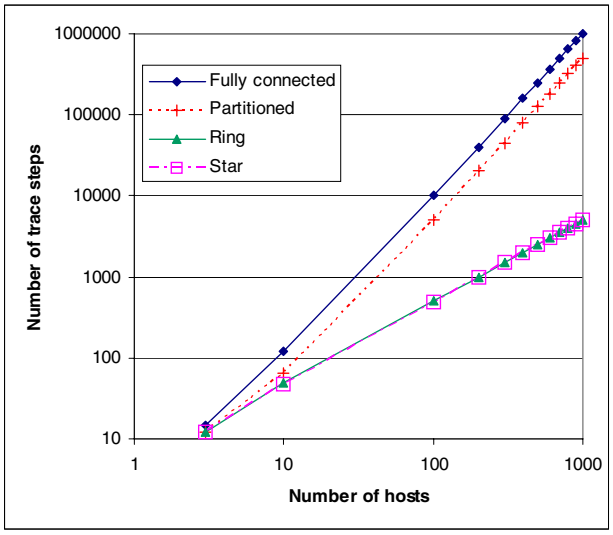

Figure 11: Number of trace-steps as a function of network size for various network topologies.

In Figure 11 the number of attack simulation trace steps, which is the input to the graph builder, is shown for the same set of test cases. For the worst case scenario, the number of trace steps is a quadratic function of the number of hosts. This verifies that Datalog evaluation in MulVAL reasoning engine takes $O(n^2)$ derivation steps to complete (Theorem 1).

Figure 12 shows that the number of derived fact nodes in the attack graph grows linearly with the size of the network.

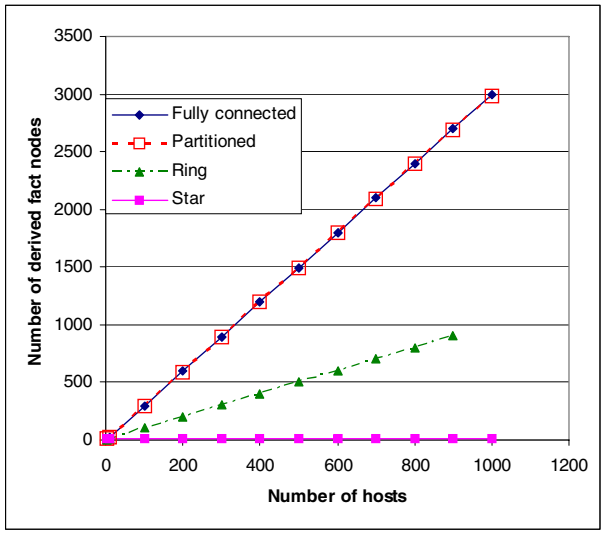

Figure 12: Number of derived fact nodes as a function of network size for various network topologies.

An interesting case is the one for the "star" topology, where the graph nodes remain constant regardless of the network size. This is because in that topology, the only attack path is from the attack machine to the hub, and then from the hub to the target machine.

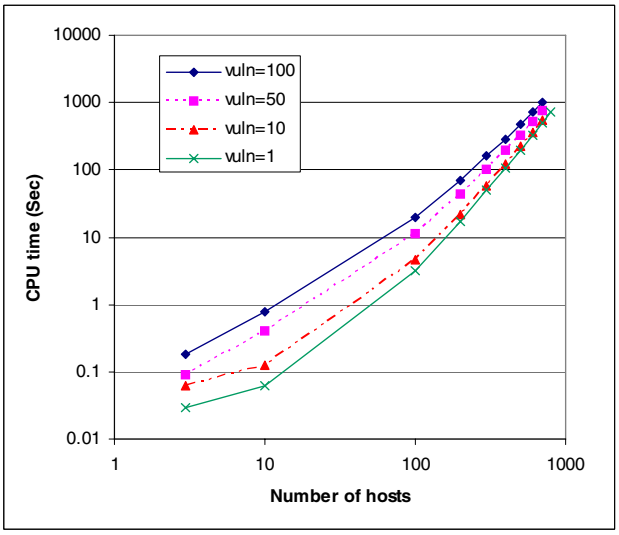

Figure 13: Graph generation CPU time for a fully connected network and number of vulnerabilities per host varying from 1 to 100.

In Figure 13 the attack graph generation CPU time is shown as a function of the network size for a fully connected network and for the number of vulnerabilities per host varied from 1 to 100. It shows that vulnerability density has a bigger impact when the network size is small. As the network size grows the CPU time is dominated by the number of machines, and thus vulnerability density has a less visible impact.

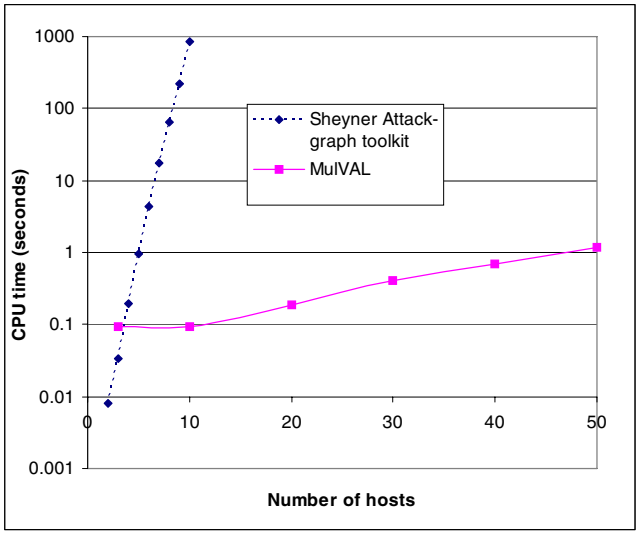

Figure 14: Graph generation CPU time compared to Sheyner attack graph toolkit. Fully connected network and 5 vulnerabilities per host.

Our graph builder was directly compared to the Sheyner attack graph toolkit by running both tools with equivalent input data. The Sheyner attack graph toolkit was tested on a Pentium III-M CPU, 256MB RAM, Fedora Core 1 LINUX operating system. Figure 14 is a comparison of graph builder CPU time for the case of a fully connected network and 5 vulnerabilities per host (note that only the Y axis is on logarithmic scale in this chart). From the diagram it is clear that the running time for Sheyner's tool grows exponentially. The growth trend for MulVAL is not obvious in this diagram because the running time is too short. But the difference between the two tools is obvious.

7. CONCLUSIONS

We have proposed a new approach to represent and generate attack graphs. Logical attack graphs directly illustrate logical dependencies among attack goals and configuration information and therefore have the significant advantage of improved clarity in guiding the user to an understanding of the causality relationship between system configuration and a successful attack. Our logical attack graph approach has dramatically improved scalability compared to previous approaches. We have shown that a logical attack graph has size polynomial to the network being analyzed. Our attack graph generation tool builds upon MulVAL, a network security analyzer based on logical programming. We have demonstrated how to produce a derivation trace in the MulVAL logic-programming engine, and how to use the trace to generate a logical attack graph in quadratic time. We have shown experimental evidence that our logical attack graph generation algorithm is very efficient.

8. REFERENCES

[1] P. Ammann, D. Wijesekera, and S. Kaushik. Scalable, graph-based network vulnerability analysis. In *Proceedings of 9th ACM Conference on Computer and Communications Security*, Washington, DC, November 2002.

[2] S. Jajodia, S. Noel, and B. O'Berry. Topological analysis of network attack vulnerability. In V. Kumar, J. Srivastava, and A. Lazarevic, editors, *Managing Cyber Threats: Issues, Approaches and Challanges*, chapter 5. Kluwer Academic Publisher, 2003.

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

[4] S. Noel and S. Jajodia. Managing attack graph complexity through visual hierarchical aggregation. In *VizSEC/DMSEC '04: Proceedings of the 2004 ACM workshop on Visualization and data mining for computer security*, pages 109–118, New York, NY, USA, 2004. ACM Press.

[5] S. Noel, S. Jajodia, B. O'Berry, and M. Jacobs. Efficient minimum-cost network hardening via exploit dependency graphs. In *19th Annual Computer Security Applications Conference (ACSAC)*, December 2003.

[6] X. Ou, S. Govindavajhala, and A. W. Appel. MulVAL: A logic-based network security analyzer. In *14th USENIX Security Symposium*, Baltimore, MD, USA, August 2005.

[7] G. Pemmasani, H.-F. Guo, Y. Dong, C. Ramakrishnan, and I. Ramakrishnan. Online justification for tabled logic programs. In *The 7th International Symposium on Functional and Logic Programming*, April 2004.

[8] C. Phillips and L. P. Swiler. A graph-based system for network-vulnerability analysis. In *NSPW '98: Proceedings of the 1998 workshop on New security paradigms*, pages 71–79. ACM Press, 1998.

[9] P. Rao, K. F. Sagonas, T. Swift, D. S. Warren, and J. Freire. XSB: A system for efficiently computing well-founded semantics. In *Proceedings of the 4th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR'97)*, pages 2–17, Dagstuhl, Germany, July 1997. Springer Verlag.

[10] B. Schneier. *Secrets & Lies: Digital Security in a Networked World*, chapter 21. John Wiley & Sons, 2000.

[11] O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J. M. Wing. Automated generation and analysis of attack graphs. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 254–265, 2002.

[12] L. P. Swiler, C. Phillips, D. Ellis, and S. Chakerian. Computer-attack graph generation tool. In *DARPA Information Survivability Conference and Exposition (DISCEX II'01)*, volume 2, June 2001.

[13] T. Tidwell, R. Larson, K. Fitch, and J. Hale. Modeling Internet attacks. In *Proceedings of the 2001 IEEE Workshop on Information Assurance and Security*, West Point, NY, June 2001.

[14] D. S. Warren. *On the Complexity of Tabled Datalog Programs*. Department of Computer Science, SUNY @ Stony Brook, Stony Brook, NY 11794-4400, U.S.A., July 1999.

[15] D. S. Warren. *Programming in Tabled Prolog*. Department of Computer Science SUNY @ Stony Brook, July 1999.