Journal of Computer Security 16 (2008) 419–437
DOI 10.3233/JCS-2008-0327
IOS Press

# Implementing interactive analysis of attack graphs using relational databases

Lingyu Wang [a,*], Chao Yao [b], Anoop Singhal [c] and Sushil Jajodia [d]
[a] *Concordia Institute for Information Systems Engineering, Concordia University, Montreal, QC, H3G 1M8, Canada*
*E-mail: wang@ciise.concordia.ca*
[b] *Center for Secure Information Systems, George Mason University, Fairfax, VA 22030-4444, USA*
*E-mail: cyao@gmu.edu*
[c] *Computer Security Division, National Institute of Standards and Technology, Gaithersburg, MD 20899, USA*
*E-mail: anoop.singhal@nist.gov*
[d] *Center for Secure Information Systems, George Mason University, Fairfax, VA 22030-4444, USA*
*E-mail: jajodia@gmu.edu*

An attack graph models the causal relationships between vulnerabilities. Attack graphs have important applications in protecting critical resources in networks against sophisticated multi-step intrusions. Currently, analyses of attack graphs largely depend on proprietary implementations of specialized algorithms. However, developing and implementing algorithms causes a delay to the availability of new analyses. The delay is usually unacceptable due to rapidly-changing needs in defending against network intrusions. An administrator may want to revise an analysis as soon as its outcome is observed. Such an *interactive* analysis, similar to that in decision support systems, is desirable but difficult with current approaches based on proprietary implementations of algorithms. This paper addresses the above issue through a relational approach. Specifically, we devise a relational model for representing necessary inputs, such as network configurations and domain knowledge, and we generate attack graphs from these inputs as relational views. We show that typical analyses can be supported through different type of searches in an attack graph, and these searches can be realized as relational queries. Our approach eliminates the needs for implementing algorithms, because an analysis is now simply a relational query. The interactive analysis of attack graphs becomes possible, since relational queries can be dynamically constructed and revised at run time. As a side effect, experimental results show that the mature optimization techniques in relational databases can transparently improve the performance of the analysis.

## 1. Introduction

*Attack graph* is an important concept in defending against attackers who employ multiple attack steps to gain privileges while evading detection. An *attack graph* enumerates sequences of attack steps that may allow attackers to compromise critical resources. By providing the *context* of attacks, an attack graph can reveal threats in

[*]Corresponding author: Tel.: +1 514 848 2424 5662, Fax: +1 514 848 3171.

a more meaningful way compared to isolated vulnerabilities. Attack graphs can help to harden a network at the least cost through finding critical vulnerabilities whose removal can prevent potential attacks [9,18]. Attack graphs are used to monitor and predict intrusions for real-time attack responses [16,17]. Attack graphs may also be used as a basis for designing network security metrics [19].

Presently, the analysis of attack graphs typically depends on proprietary implementations of specialized algorithms. Standard graph-related algorithms are not directly applicable due to unique characteristics of attack graphs. An algorithm must be developed and implemented before the corresponding analysis become available. However, such a delay is usually unacceptable in defending against network intrusions because the needs for analyzing attack graphs can change rapidly due to constantly changing threats and network configurations. An administrator may want to modify an analysis immediately after the analysis result is observed. Such an *interactive analysis*, similar to that in decision support systems, is difficult if at all possible with current approaches of implementing specialized algorithms.

This paper describes a relational approach to attack graph analysis. First, we model necessary inputs including network configurations and domain knowledge as relational tables. We then generate attack graphs using relational queries, which can either be materialized as relations or left as view definitions. The latter case is especially suitable for large networks where materializing the complete attack graph is prohibitive. Second, we show through examples that typical analyses of attack graphs can be supported through different type of searches in the attack graph, and these searches can be realized as relational queries.

Our approach eliminates the needs for implementing specialized algorithms, because the analysis of attack graphs is now simply relational queries. The approach enables the interactive analysis of attack graphs, since administrators can dynamically construct or revise queries based on the outcome of previous analyses. As a generalization of typical analyses, the searches in attack graphs studied in this paper provide templates for writing queries for a broader range of analyses. As a side-benefit, our experimental results show that the performance of an analysis can usually be transparently improved by the mature optimization techniques found in most relational databases.

## 1.1. Related work

Attack graph is a model of knowledge about the inter-dependency between vulnerabilities and the connectivity in network. By regarding a collection of security-related conditions as a state and each exploit as a transition between these states, an attack graph can be generated by searching forwards from the initial state or backwards from a given goal state. Model checking was first used to decide whether a goal state is reachable from the initial state [10,11] and later used to enumerate all possible sequences of attacks connecting the two states [6,12]. The number of attack sequences is potentially exponential, leading to prohibitive complexity. Hence, a more compact representation was proposed based on the *monotonicity assumption* (that is, an attacker never relinquishes an obtained capability) [2]. The new representation keeps exactly one vertex for each exploit or condition, leading to attack graphs of polynomial size.

The analysis of attack graphs has played important roles in various aspects of defending against network intrusions [6,8,9,12,16,19]. In contrast to data mining-based approaches [13–15], attack graph-based analysis can usually provide results with more certainty due to the domain knowledge encoded in an attack graph. The *minimal critical attack set* analysis finds a minimal cut set in the attack graph whose removal prevents any attacker from reaching the goal state [6,12]. However, the attacks in a minimal critical attack set are not necessarily independent, and a consequence cannot be removed without first removing its causes. This observation leads to the *minimum-cost hardening* analysis, which finds a minimal set of security conditions that can be disabled at will [9].

Finding the minimum set of attacks leading to a given goal is computationally infeasible, whereas a minimal set can be found in polynomial time [2,6,12]. All attacks involved in at least one such minimal set of attacks, namely, *relevant attacks*, can also be enumerated [2]. The minimal number of attacks required for reaching given goals can be found in polynomial time (although the exact set of attacks cannot be) [2]. Finally, in *exploit-centric alert correlation* [8,16], an attack graph is used to correlate isolated intrusion alerts into attack scenario and to predict possible future attacks.

The afore-mentioned analysis of attack graphs is largely based on proprietary algorithms. However, as mentioned earlier, this may delay a new analysis and make interactive analysis impossible. To our best knowledge, our study is the first effort towards removing this limitation and enabling interactive analysis of attack graphs. On the other hand, decision support systems have long been used for interactive analysis of business data. However, an analyst using decision support systems is usually interested in generalized data and statistical patterns, which is different from the analysis of attack graphs. Recently, Agrawal et al. explore database technologies for implementing the P3P standard [1], which shares a similar motivation with our work.

The preliminary results of this paper have appeared in [20]. The examples discussed there are generalized in this paper as different type of searches in an attack graph, and queries written for these searches can thus be used as templates in writing queries for other analyses. The rest of the paper is organized as follows. Section 2 proposes a relational model for representing the attack graph. Section 3 then discusses how typical analyses can be generalized as different searches in attack graphs and how these searches can be written as relational queries. Section 5 describes our implementation of the proposed methods. Finally, Section 6 concludes the paper.

## 2. A relational model for representing attack graphs

We study the representation of attack graphs in relational model. First, Section 2.1 reviews basic concepts of attack graphs. Section 2.2 then proposes a relational model for representing attack graphs.

### 2.1. Attack graph

We define *attack graph* as a directed graph having two type of vertices, *exploits* and *security conditions* (or simply *conditions*). An exploit is a triple $(h_\mathrm{s}, h_\mathrm{d}, v)$, where $h_\mathrm{s}$ and $h_\mathrm{d}$ represent two connected hosts and $v$ a vulnerability on the destination host $h_\mathrm{d}$. A security condition is a pair $(h, c)$, indicating the host $h$ satisfies a condition $c$ relevant to one or more exploits. Notice that $h_\mathrm{s}$, $h_\mathrm{d}$ and $v$ are abstract notations (for example, $h_\mathrm{s}$ and $h_\mathrm{d}$ can be host names, IP addresses, and so on, and $v$ can be the name of a vulnerability). Moreover, both exploits and conditions may occasionally involve more or less hosts. For example, an exploit may require an intermediate host as stepping stone between the source and destination hosts, and a condition may apply to a pair of hosts. It is our belief that the methods proposed in this paper can be extended to such cases.

Corresponding to the inter-dependency between exploits and conditions, there are two types of edges in an attack graph. First, the *require* relation is a directed edge pointing from a condition to an exploit, which means the exploit cannot be executed unless the condition is satisfied. For example, an exploit $(h_\mathrm{s}, h_\mathrm{d}, v)$ requires following two conditions, that is the existence of the vulnerability $v$ on $h_\mathrm{d}$ and the connectivity between $h_\mathrm{s}$ and $h_\mathrm{d}$. Second, the *imply* relation pointing from an exploit to a condition means executing the exploit will satisfy the condition. Notice that there is no edge directly connecting two exploits (or two conditions). These concepts are illustrated in Example 1 and formally characterized in Definition 1.

**Example 1.** The left-hand side of Fig. 1 shows a simple attack graph as our running example. The attack graph indicates that by exploiting a buffer overflow vulnerability in the Sadmind service (Nessus ID 11841), an attacker can gain the privilege of using a remote machine. The right-hand side shows a simplified version where $x$ denotes the existence of the vulnerability, $y$ the user privilege, and $A$ the exploitation of that vulnerability. The attack graph shows an attacker having user privilege on host 3 can exploit the vulnerability on hosts 1 and 2 and obtain user privilege on the hosts. Notice that after an attacker has obtained user privilege on host 1, he/she can then exploit host 2 from either host 3 or host 1.

**Definition 1.** Given a set of exploits $E$, a set of conditions $C$, a *require* relation $R_\mathrm{r} \subseteq C \times E$, and an *imply* relation $R_\mathrm{i} \subseteq E \times C$, an *attack graph* $G$ is the directed graph $G(E \cup C, R_\mathrm{r} \cup R_\mathrm{i})$ ($E \cup C$ is the vertex set and $R_\mathrm{r} \cup R_\mathrm{i}$ the edge set).

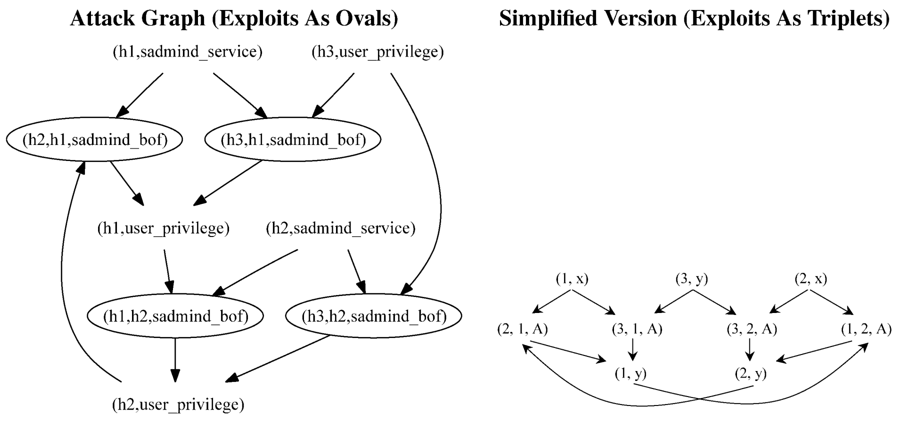

Fig. 1. An example of attack graph.

The two types of edges in an attack graph have different semantics. The require relation is regarded as *conjunctive*, whereas the imply relation is *disjunctive*. More specifically, an exploit cannot be realized until *all* of its required conditions have been satisfied (different variations of an exploit that require different sets of conditions should be regarded as different exploits), but a condition can be satisfied by one of the realized exploits that imply that condition. Another important perspective is that conditions can be divided into *initial* conditions (those not implied by any exploit) and *intermediate* conditions. The main reason for such a distinction is that initial conditions can be independently disabled to harden a network, whereas intermediate conditions cannot be without first removing the exploits implying them.

### 2.2. *A relational model for attack graphs*

Instead of modeling an attack graph, we model necessary inputs required for generating the attack graph. The attack graph then becomes the result of relational queries over these inputs. Such a result may be materialized or simply kept as the definition of relational views. This flexibility is important in cases where materializing the complete attack graph is prohibitive. The inputs we model include *network configuration* and *domain knowledge*. Here network configuration refers to the network connectivity, and that which host has which vulnerabilities. Domain knowledge refers to the interdependency between different type of vulnerabilities and conditions. These are illustrated in Example 2.

**Example 2.** To generate the attack graph in Example 1, we need the network configuration and domain knowledge shown in Fig. 2. The left-hand side shows the connectivity between three hosts. Initially, hosts 1 and 2 satisfy the condition $x$ and host 3 satisfies $y$. The right-hand side says that an attacker can exploit the vulnerability $A$ on the destination (denoted by the symbol $D$) host, if it satisfies $x$ and the

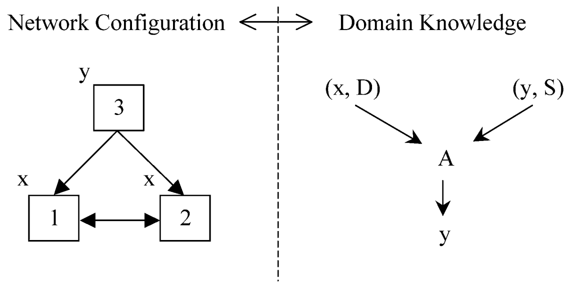

Fig. 2. An example of network configuration and domain knowledge.

source host satisfies $y$ at the same time. This exploitation will then satisfy $y$ on the destination host.

We assume the domain knowledge required for generating attack graphs is available from tools like the Topological Vulnerability Analysis (TVA) system, which covers more than 37,000 vulnerabilities taken from 24 information sources including X-Force, Bugtraq, CVE, CERT, Nessus and Snort [5]. On the other hand, we assume the configuration information including vulnerabilities and connectivity can be obtained using available tools, such as the Nessus scanner [4].

The schemata of our model are given in Definition 2. The *connectivity relation* represents the connectivity from the source host $H_\text{s}$ to the destination host $H_\text{d}$. The *condition relation* indicates that a host $H$ has an initial condition $C$. The condition-vulnerability dependency relation indicates that a condition $C$ is required for exploiting a vulnerability $V$ on the destination host. The attribute $F$ indicates whether the condition $C$ belongs to the source ($S$) or the destination ($D$) host. The vulnerability–condition dependency relation indicates a condition $C$ is satisfied by exploiting a vulnerability $V$.

The next three relations and the condition relation together represent the complete attack graph. Those relations may or may not need to be materialized. The vertices are conditions (that is, the relation $HC$) and exploits (that is, the relation $EX$), and the edges interconnecting them are represented by relations $CE$ and $EC$. Each relation has a composite key composed of all attributes in that relation. Example 3 shows the relational model of Example 2. We do not show an example for the complete attack graph but shall provide partial examples in later sections.

**Definition 2.** Define the following relational schemata:

- *Connectivity* $HH = (H_\text{s}, H_\text{d})$,
- *Condition* $HC = (H, C)$,
- *Condition–vulnerability dependency* $CV = (C, F, V)$,
- *Vulnerability–condition dependency* $VC = (V, C)$,
- *Exploit* $EX = (H_\text{s}, H_\text{d}, V)$,

Table 1

Representing network configuration and domain knowledge in relational model

| $hh(HH)$ | | $hc(HC)$ | | $cv(CV)$ | | | $vc(VC)$ | |
|---|---|---|---|---|---|---|---|---|
| $H_s$ | $H_d$ | $H$ | $C$ | $C$ | $F$ | $V$ | $V$ | $C$ |
| 1 | 2 | 3 | $y$ | $x$ | $D$ | $A$ | $A$ | $y$ |
| 2 | 1 | 1 | $x$ | $y$ | $S$ | $A$ | | |
| 3 | 1 | 2 | $x$ | | | | | |
| 3 | 2 | | | | | | | |

- *Condition–exploit* $CE = (H, C, H_s, H_d, V)$,
- *Exploit–condition* $EC = (H_s, H_d, V, H, C)$.

**Example 3.** To represent the network configuration and domain knowledge in Example 2, Table 1 shows a set of relational schemata according to Definition 2.

The design of those schemata will be justified in later sections when we discuss the generation and analysis of attack graphs using relational queries. Notice that for clarity purposes we deliberately keep redundancy in the schema. In practice, one may want to normalize the relations to avoid redundancy, such as adding an additional primary key to both $HC$ and $EX$, and replacing $CE$ and $EC$ with these keys. Also, the schema needs to be extended by adding more attributes, when exploits or conditions involve more than two hosts. However, we leave out these straightforward extensions for simplicity of discussions.

## 3. Searching attack graphs

Instead of only providing examples to a few analyses, this section first generalizes typical analyses into generic searches in attack graphs (notice that such searches are intended for any attack graphs, instead of being peculiar to special cases discussed before). The next section will then discuss how to write relational queries for these searches. This approach allows the proposed queries to be used as templates in writing queries for a broader range of analyses. First, based on the study of several analyses that have previously appeared in the literature, Definition 3 classifies searches in attack graphs along four diagonal dimensions.

**Definition 3.** We say a search in an attack graph is:

- *forward* if it follows the directed edges in an attack graph; conversely, it is *backward* if it follows the edges in their reversed direction,
- *logical* if it takes into accounts the conjunctive nature of the require relation; conversely, it is *casual* if it ignores that property,

- *positive* if it starts from satisfied conditions; conversely, it is *negative* if it starts from disabled conditions,
- *duplicate-eliminating* if it visits each vertex at most once; conversely, it is *duplicate-preserving* if it visits vertices multiple times.

First, Example 4 illustrates the generation of an attack graph. This attack graph is an example of a forward, logical, positive and duplicate-eliminating search. First, it is forward since it follows the directed edges in their original direction. Second, it is a logical search because it visits an exploit (for example, $(3, 1, A)$) only if the required conditions are *all* satisfied (for example, $(1, x)$ and $(3, y)$). Third, it is a positive search since it starts from the satisfied initial conditions (that is, $(1, x)$, $(3, y)$ and $(2, x)$). Finally, it eliminates duplicates. For example, although the exploit $(2, 1, A)$ implies the condition $(1, y)$, no duplicate copy of that condition is inserted (as described earlier, the attack graph has no duplicate vertices as a result of the monotonic assumption).

**Example 4.** Given the network configuration and domain knowledge in Example 2, the attack graph in Fig. 1 can be generated by an iterative procedure as follows. Initially, the attack graph only includes the three initial conditions $(1, x)$, $(3, y)$, $(2, x)$ as vertices. First, domain knowledge implies that the conditions $(1, x)$ and $(3, y)$ jointly imply the exploit $(3, 1, A)$, and $(2, x)$ and $(3, y)$ jointly imply $(3, 2, A)$. Second, the two conditions $(1, y)$ and $(2, y)$ are satisfied. Next, we repeat the above two steps with the two new conditions and insert four more edges between $(1, y)$, $(2, y)$ and the two exploits. The process then terminates because no new conditions are inserted in the second iteration.

Example 5 illustrates the vulnerability-centric alert correlation and prediction method (before analyzing the attack graph, an alert actually needs to be mapped to the corresponding exploit, which is omitted here for simplicity) [16]. Unlike the generation of an attack graph, this analysis will ignore the conjunctive nature of the require relation. The reason lies in that the relationship between alerts is usually regarded as *casual* instead of logical [3,7]. Such a conservative approach is more appropriate here, because alerts are often missed by intrusion detection systems. This analysis is thus an example of a casual, positive and duplicate-eliminating search. It is backward in alert correlation since the objective of correlation is to find those alerts that prepare for the current one; it is forward in alert prediction, as the objective is to enumerate attacks as possible consequences of the current one.

**Example 5.** In Fig. 1, suppose the current alert maps to the exploit $(2, 1, A)$. The backward search will first reach conditions $(1, x)$ and $(2, y)$ and then follows $(2, y)$ to $(3, 2, A)$ and $(1, 2, A)$ to find a previous correlated alert if there is any, or to make a hypothesis for a missing alert, otherwise. The search continues from $(1, 2, A)$ to $(1, y)$ and $(2, x)$, then from $(1, y)$ to $(3, 1, A)$ (the branch to $(2, 1, A)$ is a loop and hence ignored) and consequently to $(1, x)$ and $(3, y)$. The search stops when it reaches only initial conditions or if a loop is encountered.

Examples 6 and 7 illustrate two different analyses, namely, enumerating sequences of *relevant* exploits (that is, exploits appearing in at least one sequence of attacks leading to given goal conditions [2]) and finding a *network hardening* solution (that is, given goal conditions represented as a logic formula of initial conditions [9]), respectively. The two analyses, however, share a similar backward, logical, positive and duplicate-preserving search in the attack graph. The key difference of this search from previous ones is that it may need to visit a vertex for multiple times. The reason lies in that duplicate appearances of exploits and conditions must be kept in both analyses. Preserving duplicates is necessary in enumerating sequences of relevant exploits, because different sequences may share common exploits or conditions. For network hardening, the resultant logic formula produced by the analysis clearly contains duplicates.

An additional issue caused by preserving duplicates is the avoidance of loops in the analysis. Loops in the attack graph are naturally avoided in a duplicate-eliminating analysis, since such an analysis never needs to revisit a vertex. However, for duplicate-preserving searches, loops must avoided through maintaining a predecessor list for each vertex [18], as in a standard breadth-first search (BFS) (note that none of these searches is a standard BFS).

**Example 6.** In Fig. 1, we start from a given goal condition $(1, y)$ and search backwards in the attack graph. First, the two exploits $(3, 1, A)$ and $(2, 1, A)$ are reached. The former branch ends at initial conditions, and the latter leads to one initial condition $(1, x)$ and an intermediate condition $(2, y)$. The condition $(2, y)$ then leads to $(3, 2, A)$ and $(1, 2, A)$. The former ends at initial conditions, and the latter leads to a loop back to $(1, y)$. The relevant exploits with respect to the goal condition $(1, y)$ are thus $(2, 1, A)$, $(3, 1, A)$ and $(3, 2, A)$ (the exploit $(1, 2, A)$ is not relevant because it can never be realized before satisfying the goal $(1, y)$ itself).

**Example 7.** With a similar search, we can transform the goal condition $(1, y)$ into a logic formula of initial conditions as follows (by regarding the exploits and conditions as Boolean variables). In the fourth line, the value *FALSE* replaces the second appearance of the goal condition $(1, y)$, because it is a predecessor of $(1, 2, A)$, indicating a loop. The final result says that if any of the two conditions $(1, x)$ and $(3, y)$ is disabled, then the goal can no longer be satisfied.

$$\begin{aligned}
(1, y) &\equiv (3, 1, A) \vee (2, 1, A) \\
&\equiv (1, x) \wedge (3, y) \vee (1, x) \wedge (2, y) \\
&\equiv (1, x) \wedge (3, y) \vee (1, x) \wedge ((3, 2, A) \vee (1, 2, A)) \\
&\equiv (1, x) \wedge (3, y) \vee (1, x) \wedge ((3, y) \wedge (2, x) \vee (2, x) \wedge \mathit{FALSE}) \\
&\equiv (1, x) \wedge (3, y).
\end{aligned}$$

Example 8 illustrates how to compute the effect of disabling a given collection of initial conditions. This analysis is useful in many applications. For example, it can be used to determine the potential effect of introducing a new security measure (effectively some initial conditions will be disabled). It can be used to decided whether the goal condition is reachable with only stealthy attacks [12]. It can also be used to update the attack graph when the network configuration has changed and some initial conditions are no longer satisfied (on other hand, adding a new initial condition can be easily handled with more iterations in the generation of attack graphs). Notice that in these applications we can certainly recompute the attack graph from scratches with the disabled initial conditions removed. However, this is not desired if the attack graph is much larger than the collection of conditions to be disabled. Instead, we should incrementally update the attack graph by computing the effect of disabled conditions.

Example 8 is a forward, logical, negative, duplicate-eliminating search. The example shows that such a negative search is quite different from the previous positive ones. The previous searches are all *unidirectional* in the sense that the edges are only followed in one direction (either forwards or backwards). However, the above forward search actually needs special backward steps. For example, after it reaches the condition $(1, y)$ from the exploit $(2, 1, A)$, it must go back to see whether other exploits also imply the condition $(1, y)$ (in this case, the exploit $(3, 1, A)$ does so), since the imply relation is disjunctive.

**Example 8.** In Fig. 1, suppose the condition $(2, x)$ is disabled. Then the exploits $(1, 2, A)$ and $(3, 2, A)$ can no longer be realized. Then the condition $(2, y)$ becomes unsatisfiable, because it can only be implied by the above two exploits. Finally, the exploit $(2, 1, A)$ cannot not longer be realized. However, the condition $(1, y)$ is still satisfiable, due to another exploit $(3, 1, A)$.

More searches are possible by combining the four dimensions stated in Definition 3. These searches may also need to be combined in an analysis. For example, deriving a minimal collection of exploits leading to given goal conditions is based on a backward, logical, positive and duplicate-eliminating search. However, to decide whether a set of exploits is minimal with respect to satisfying a condition, we must test whether deleting an exploit from the set can still satisfy the condition [2]. Such an algorithm thus requires one iteration of a forward, logical, negative and duplicate-eliminating search. It is our belief that the different type of searches discussed here can cover many useful analyses of attack graphs.

## 4. Analyzing attack graphs with relational queries

The previous section has illustrated how typical analyses may be supported using different searches in an attack graph. This section shows how to realize the searches in attack graphs as relational queries based on the relational model given in Section 2.2. Using these queries as templates will greatly ease writing queries for new analyses, since the search is the core component of each analysis. Realizing the search usually comprises a major part of efforts in realizing the analysis. Additional steps required by an analysis are usually straightforward compared to the search. For example, for the network hardening analysis described earlier, we only need to add logic connectives *AND* and *OR* to concatenate the partial results collected during the search [9,18].

## *4.1. The generation of attack graphs*

We start with the generation of attack graphs, which can be regarded as a special analysis. As described in the previous section, this analysis is an example of a forward, logical, positive and duplicate-eliminating search. The key challenge in realizing this search using relational queries lies in its logical aspect. It must take into account the conjunctive nature of the require relation, that is an exploit cannot be realized unless *all* of its required conditions are satisfied. In contrast, the disjunctive imply relation can be more easily realized using a join operation, since a condition can be satisfied by any *one* of the realized exploits. This is illustrated in Example 9.

**Example 9.** Considering the two relations $vc$ and $cv$ in Table 1. Suppose an exploit $(1, 2, A)$ is realized, then an equijoin between this tuple and $vc$ followed by a projection on attributes $H_\mathrm{d}$ and $c$ yields the condition $(2, y)$. That is, the realized exploit $(1, 2, A)$ causes the condition $(2, y)$ to be satisfied. On the other hand, a similar join between a satisfied condition $(1, x)$ with the relation $cv$ yields a wrong result, that is the exploit $A$ can be realized on the host 1 (this also requires the condition $y$ to be satisfied on the source host).

We deal with this issue with two set-difference operations as follows (similar to the division operation in relational algebra). Intuitively, we first subtract (that is, set difference) the satisfied conditions from the conditions required by all possible exploits. The result includes all the unsatisfied but required conditions, from which we can derive the exploits that cannot be realized. Then we subtract the unrealizable exploits from all possible exploits to derive those that can indeed be realized. In Example 9, the first set difference will give the unsatisfied condition $(3, y)$ and $(2, y)$, which tells us that none of the exploits $(3, 1, A)$ and $(2, 1, A)$ can be realized. The second set difference then yields the final result as an empty set.

For simplicity of presentation, we shall use relational algebra and psuedo codes. The queries can be easily implemented using standard SQL, whereas the looping statements are supported in most procedural extensions, such as PL/SQL. Figure 3 gives a procedure for generating attack graphs in a way similar to that in Example 4. In the procedure, $Q_1$ and $Q_2$ are temporary relations (we shall use subscripts in numbers to denote temporary relations). In Line 3, the Cartesian product $hh \times \Pi_V(vc)$

```
Procedure Generating_Attack_Graph
Input: Relations $hh(HH)$, $hc(HC)$, $cv(CV)$, $vc(VC)$
Output: Attack graph represented by $Q_e(EX)$, $Q_c(HC)$, $Q_{ce}(CE)$, $Q_{ec}(EC)$
Method:
    1. Let $Q_c = hc$ and $Q_e(EX)$, $Q_{ce}(CE)$, $Q_{ec}(EC)$ be empty relations
    2. Do
    3.     Let $Q_1 = \sigma_{H_s=H \vee H_d=H}(hh \times \Pi_V(vc) \times Q_c)$
    4.     Let $Q_2 = \Pi_{H_s,H_d,V,H_d,C}(hh \times \sigma_{F=D}(cv)) \cup \Pi_{H_s,H_d,V,H_s,C}(hh \times \sigma_{F=S}(cv)) - Q_1$
    5.     Let $Q_e = (\Pi_{H_s,H_d,V}(hh \times cv) - \Pi_{H_s,H_d,V}(Q_2)) \cup Q_e$
    6.     Let $Q_{ce} = \Pi_{H_d,C,H_s,H_d,V}(Q_e \times \sigma_{F=D}(cv)) \cup \Pi_{H_s,C,H_s,H_d,V}(Q_e \times \sigma_{F=S}(cv)) \cup Q_{ce}$
    7.     Let $Q_{ec} = \Pi_{H_s,H_d,V,H_d,C}(\sigma_{Q_e.V=vc.V}(Q_e \times vc)) \cup Q_{ec}$
    8.     Let $Q_c = \Pi_{H,C}(Q_{ec}) \cup Q_c$
    9. While $|Q_c|$ is increased in Line 8
```

Fig. 3. Queries for generating attack graphs.

includes all possible exploits, and its join with the satisfied conditions in $Q_c$ describes how these conditions may contribute to the realization of exploits.

In Line 4, the left-hand side of the set difference operator includes all possible exploits and the conditions they require, from which subtracting $Q_1$ yields the unrealizable exploits in $Q_2$. Line 5 adds to $Q_e$ the exploits realized in this iteration through subtracting the unrealizable exploits in $Q_2$ from all possible exploits. Lines 6 and 7 collect the edges from conditions required by a realized exploit to that exploit, and those from an exploit to its implied conditions, respectively. Finally, Line 8 adds to $Q_c$ the conditions satisfied in this iteration.

The correctness of the Procedure *Generating_Attack_Graph* can be easily justified through mathematical induction on the number of required iterations, and is omitted here. The set union operation in the procedure does not keep duplicates, and hence at some point the size of $Q_c$ must stop increasing. The procedure thus always terminates. The total number of iterations is bound by the smallest between the number of all possible exploits and the number of all possible conditions (in the worst case, the exploits form a chain). Example 10 illustrates how this procedure works.

**Example 10.** Table 2 shows how the Procedure *Generating_Attack_Graph* generates the attack graph in Example 1. For simplicity, we only show the first iteration. The relation $Q_1$ are the satisfied conditions and their related (but not necessarily realizable) vulnerabilities. Subtracting those from the conditions required by all possible exploits yields the two unsatisfied conditions and the unrealizable exploits in $Q_2$. Then, subtracting the unrealizable exploits from all possible exploits gives the two realizable exploits in $Q_e$. The exploits then imply the two conditions in $Q_c$. The edges in $Q_{ce}$ and $Q_{ec}$ interconnect the conditions and exploits.

Table 2

An example of one iteration in deriving the complete attack graph

| $Q_1$ | | | | | $Q_2$ | | | | | $Q_e$ | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $H_s$ | $H_d$ | $V$ | $H$ | $C$ | $H_s$ | $H_d$ | $V$ | $H$ | $C$ | $H_s$ | $H_d$ | $V$ |
| 1 | 2 | $A$ | 1 | $x$ | 1 | 2 | $A$ | 1 | $y$ | 3 | 1 | $A$ |
| 1 | 2 | $A$ | 2 | $x$ | 2 | 1 | $A$ | 2 | $y$ | 3 | 2 | $A$ |
| 2 | 1 | $A$ | 1 | $x$ | | | | | | | | |
| 2 | 1 | $A$ | 2 | $x$ | | | | | | | | |
| 3 | 1 | $A$ | 1 | $x$ | | | | | | | | |
| 3 | 1 | $A$ | 3 | $y$ | | | | | | | | |
| 3 | 2 | $A$ | 2 | $x$ | | | | | | | | |
| 3 | 2 | $A$ | 3 | $y$ | | | | | | | | |

| $Q_{ce}$ | | | | | $Q_{ec}$ | | | | | $Q_c$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $H$ | $C$ | $H_s$ | $H_d$ | $V$ | $H_s$ | $H_d$ | $V$ | $H$ | $C$ | $H$ | $C$ |
| 1 | $x$ | 3 | 1 | $A$ | 3 | 1 | $A$ | 1 | $y$ | 1 | $y$ |
| 2 | $x$ | 3 | 2 | $A$ | 3 | 2 | $A$ | 2 | $y$ | 2 | $y$ |
| 3 | $y$ | 3 | 1 | $A$ | | | | | | | |
| 3 | $y$ | 3 | 2 | $A$ | | | | | | | |

### *4.2. Alert correlation and prediction*

In contrast to a logical search, realizing a casual search in relational queries is more straightforward. Figure 4 states a procedure corresponding to the backward, casual, positive and duplicate-eliminating search in alert correlation in Example 5. The forward search for alert prediction can be realized in a similar way and hence is omitted. First, the relation $Q_3$ includes the conditions reachable from the current exploits while ignoring the conjunctive relationship between those conditions. Second, subtracting from $Q_3$ the initial conditions in $hc$ and the previously visited conditions in $Q_5$ (to avoid loops) yields the reachable conditions and consequently the exploits in $Q_4$. The above two steps are repeated until no more conditions are left (that is, all the conditions are in $hc$ or in $Q_5$). The exploits encountered in this process are collected in $Q_A$ as the final result. Loops are avoided in this process because the set union operation does not keep duplicates and the relation $Q_5$ ensures each condition to be visited at most once.

**Example 11.** Table 3 shows the three iterations corresponding to the backward search in Example 5. The first iteration starts from the given exploit $(2, 1, A)$ and reaches two exploits $(1, 2, A)$ and $(3, 2, A)$ through the condition $(2, y)$. The second iteration reaches $(3, 1, A)$ and $(2, 1, A)$ through $(1, y)$. The exploit $(2, 1, A)$ leads to two previously visited conditions (that is, a loop) and the other exploit $(3, 1, A)$ reaches only initial conditions. Consequently, no new exploit appears in $Q_4$ in this iteration and the search terminates.

```
Procedure Alert_Correlation
Input: Relations $hh(HH)$, $hc(HC)$, $cv(CV)$, $vc(VC)$ and a tuple $(h_{\mathrm{s}}, h_{\mathrm{d}}, V)$
Output: A collection of exploits $Q_A$
Method:
    1. Let $Q_3(HC)$, $Q_5$ and $Q_A$ be empty relations and $Q_4(EX) = \{(h_{\mathrm{s}}, h_{\mathrm{d}}, V)\}$
    2. Do
    3.     Let $Q_3 = \Pi_{h_{\mathrm{d}},C}(Q_4 \bowtie \sigma_{F=D}(cv)) \cup \Pi_{h_{\mathrm{s}},C}(Q_4 \bowtie \sigma_{F=S}(cv))$
    4.     Let $Q_4 = \Pi_{H_{\mathrm{s}},H_{\mathrm{d}},V}(\sigma_{H_{\mathrm{d}}=H \wedge Q_3.C=vc.C}(hh \times (Q_3 - hc - Q_5) \times vc))$
    5.     Let $Q_5 = Q_5 \cup Q_3$
    6.     Let $Q_A = Q_A \cup Q_4$
    9. While $Q_4$ is not empty
```

Fig. 4. Queries for alert correlation.

Table 3

An example of analyzing attack graphs for alert correlation and prediction

| First iteration | $Q_3$ | | $Q_4$ | | | $Q_5$ | | $Q_A$ | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | $H$ | $C$ | $H_{\mathrm{s}}$ | $H_{\mathrm{d}}$ | $V$ | $H$ | $C$ | $H_{\mathrm{s}}$ | $H_{\mathrm{d}}$ | $V$ |
| | 1 | $x$ | 1 | 2 | $A$ | 1 | $x$ | 1 | 2 | $A$ |
| | 2 | $y$ | 3 | 2 | $A$ | 2 | $y$ | 3 | 2 | $A$ |
| Second iteration | $Q_3$ | | $Q_4$ | | | $Q_5$ | | $Q_A$ | | |
| | $H$ | $C$ | $H_{\mathrm{s}}$ | $H_{\mathrm{d}}$ | $V$ | $H$ | $C$ | $H_{\mathrm{s}}$ | $H_{\mathrm{d}}$ | $V$ |
| | 1 | $y$ | 3 | 1 | $A$ | 1 | $x$ | 1 | 2 | $A$ |
| | 2 | $x$ | 2 | 1 | $A$ | 2 | $y$ | 3 | 2 | $A$ |
| | 3 | $y$ | | | | 1 | $y$ | 3 | 1 | $A$ |
| | | | | | | 2 | $x$ | 2 | 1 | $A$ |
| | | | | | | 3 | $y$ | | | |
| Third iteration | $Q_3$ | | $Q_4 = \phi$ | | | $Q_5$ | | $Q_A$ | | |
| | $H$ | $C$ | | | | $H$ | $C$ | $H_{\mathrm{s}}$ | $H_{\mathrm{d}}$ | $V$ |
| | 1 | $x$ | | | | 1 | $x$ | 1 | 2 | $A$ |
| | 3 | $y$ | | | | 2 | $y$ | 3 | 2 | $A$ |
| | 2 | $y$ | | | | 1 | $y$ | 3 | 1 | $A$ |
| | | | | | | 2 | $x$ | 2 | 1 | $A$ |
| | | | | | | 3 | $y$ | | | |

### *4.3. Enumerating relevant attacks and network hardening*

Figure 5 states the relational queries for the backward, logical, positive and duplicate-preserving search used to enumerate relevant exploits or to generate the

```
Procedure Relevant_Attack_Network_Hardening
Input: Relations $hh(HH)$, $hc(HC)$, $cv(CV)$, $vc(VC)$ and a non-empty relation $Q_7(HC)$
Method:
    1. Let $Q_6(EX)$ be empty relations
    2. Do
    3.     Let $Q_6 = \Pi_{H_\mathrm{s},H_\mathrm{d},V}(Q_\mathrm{ec} \bowtie (Q_7 - hc))$
    4.     Let $Q_7 = \Pi_{H,C}(Q_\mathrm{ce} \bowtie Q_6)$
    9. While $Q_6$ is not empty
```

Fig. 5. Queries for enumerating relevant attacks and network hardening.

Table 4

An example of enumerating relevant exploits and network hardening

| First iteration | $Q_6$ | | | $Q_7$ | |
|---|---|---|---|---|---|
| | $H_\mathrm{s}$ | $H_\mathrm{d}$ | $V$ | $H$ | $C$ |
| | 3 | 1 | $A$ | 1 | $x$ |
| | 2 | 1 | $A$ | 2 | $y$ |
| | | | | 1 | $x$ |
| | | | | 3 | $y$ |
| Second iteration | $Q_6$ | | | $Q_7$ | |
| | $H_\mathrm{s}$ | $H_\mathrm{d}$ | $V$ | $H$ | $C$ |
| | 3 | 2 | $A$ | 3 | $y$ |
| | | | | 2 | $x$ |

logic formula in network hardening. The two queries at Lines 3 and 4 simply traverse the attack graph. Note that the actual analyses will require additional steps, such as adding the *and* and *or* connectives. Also, a predecessor list should be maintained for each visited vertex to avoid loops. These should be realized outside the procedure. Example 12 illustrates this search.

**Example 12.** Table 4 shows the iterations corresponding to the procedure in Examples 6 and 7. Originally, $Q_7 = \{(1, y)\}$.

## *4.4. Effect of disabling initial conditions*

Figure 6 shows the relational queries for the forward, logical, negative, duplicate-eliminating search used for computing the effect of disabling initial conditions. Line 3 derives unrealizable exploits from disabled conditions through a join.

```
Procedure Disabled_Conditions
Input: Relations $hh(HH)$, $hc(HC)$, $cv(CV)$, $vc(VC)$ and a non-empty relation $Q_{11}(HC) \subseteq hc$
Output: A collection of conditions $Q_c$ and exploits $Q_e$
Method:
    1. Let $Q_8(EX)$, $Q_9(EC)$, $Q_{10}(EC)$, $Q_e$ and $Q_c$ be empty relations
    2. Do
    3.     Let $Q_8 = \Pi_{H_s,H_d,V}(Q_{11} \bowtie Q_{ce})$
    4.     Let $Q_9 = Q_8 \bowtie Q_{ec}$
    5.     Let $Q_{10} = Q_{ec} \bowtie \Pi_{H,C}(Q_9) - Q_9$
    6.     Let $Q_{11} = \Pi_{H,C}(Q_9) - \Pi_{H,C}(Q_{10})$
    7.     Let $Q_e = Q_e \cup Q_8$
    8.     Let $Q_c = Q_c \cup Q_{11}$
    9. While $Q_{11}$ is not empty
```

Fig. 6. Queries for computing the effect of disabled conditions.

Table 5
An example of incremental updates

First iteration

| $Q_8$ | | | $Q_9$ | | | | | $Q_{10} = \phi$ | $Q_{11}$ | |
|---|---|---|---|---|---|---|---|---|---|---|
| $H_s$ | $H_d$ | $V$ | $H_s$ | $H_d$ | $V$ | $H$ | $C$ | | $H$ | $C$ |
| 3 | 2 | $A$ | 3 | 2 | $A$ | 2 | $y$ | | 2 | $y$ |
| 1 | 2 | $A$ | 1 | 2 | $A$ | 2 | $y$ | | | |

Second iteration

| $Q_8$ | | | $Q_9$ | | | | | $Q_{10}$ | | | | | $Q_{11} = \phi$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $H_s$ | $H_d$ | $V$ | $H_s$ | $H_d$ | $V$ | $H$ | $C$ | $H_s$ | $H_d$ | $V$ | $H$ | $C$ | |
| 2 | 1 | $A$ | 2 | 1 | $A$ | 1 | $y$ | 3 | 1 | $A$ | 1 | $y$ | |

Lines 4–6 use two set difference operations to derive the unsatisfied conditions. Here the disjunctive nature of the imply relation must be taken into consideration. That is, a condition will be unsatisfied, if *all* of the exploits implying it are unrealizable (because any such exploit will be sufficient to satisfy the condition). Finally, the results are collected in the two relations $Q_c$ and $Q_e$. Example 13 illustrates this search.

**Example 13.** Table 5 shows the iterations corresponding to the procedure in Example 8 (the results in $Q_c$ and $Q_e$ are omitted for simplicity). Originally, $Q_{11} = \{(2, x)\}$.

## 5. Empirical results

To study the performance of the proposed approach, we implement the analyses discussed in previous sections. The corresponding queries are written in PL/SQL and tested in Oracle 9i in its default settings on a Pentium IV 2 GHz PC with 512 MB RAM. In our preliminary experiments, we test the queries against the attack scenario originally studied in [2]. The results of analyses match those reported in [2], which justifies the correctness of our techniques.

We have two main objectives in testing the performance of our techniques. First, we want to determine whether the running time is practical for interactive analysis of attack graphs. For most decision support systems, the typical delay to a query that is considered as tolerable in interactive analyses is usually in a matter of seconds. Such a short delay is also critical to the analysis of attack graphs, especially when the analysis is used for real-time detection and prevention of intrusions. The second purpose of the experiments is to determine whether the techniques scale well in the size of attack graphs. Although the attack graph may be very large for a large network, an analysis and its result usually only involves a small part of the attack graph. The running time of an analysis thus depend on how efficiently an analysis searches the attack graph. We expect the mature optimization techniques available in most databases can transparently improve the performance and make the analyses more scalable. To test the queries against large attack graphs in a manageable way, we increase the number of vertices in the original attack graph by randomly inserting new hosts with random connectivity and vulnerabilities. We then execute the same set of analyses in the new network and measure the running time of each analysis.

The main results are shown in Fig. 7. All the results have 95% confidence intervals within about 5% of the reported values. The left-hand side shows the running time of generating the attack graph in the size of that attack graph. The attack graph with about 20,000 vertices can be generated in less than seven minutes. The result also shows that our methods scale well in the size of attack graphs. The right-hand

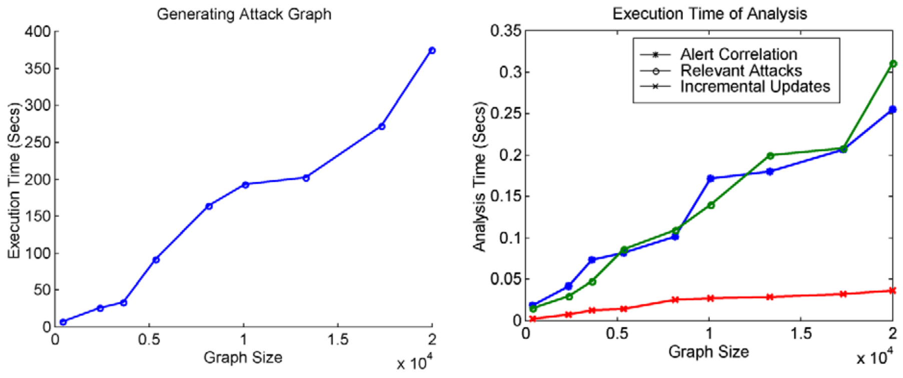

Fig. 7. The performance of analyzing attack graphs using relational queries.

side shows the running time of each analysis in the size of the attack graph. The result shows that all the analyses require less than a second, which clearly meets the requirement of an interactive analysis. The analyses all scale well with the size of the attack graph. This proves our conjecture that the optimization techniques in databases such as indexing can transparently help to keep analyses efficient. A closer look at the result reveals that the increase in running time is mainly caused by larger results. This also explains the fact that the incremental update analysis scales differently from the other two. That is, the effect of disabled initial conditions does not change much when the size of the attack graph increases.

## 6. Conclusion

We have proposed a relational model to support interactive analysis of attack graphs for intrusion detection and prevention. We have shown that the complete attack graph can be represented as relational views. Any analysis of attack graphs are thus relational queries against the views. We showed how to write relational queries for typical analyses previously studied in the literature. This approach made the analysis of attack graphs an interactive process similar to that in the decision support systems. As a side effect, the mature optimization techniques existing in most relational databases also improved the performance of analyses.

## Acknowledgements

This material is based upon work supported by National Institute of Standards and Technology Computer Security Division; by Homeland Security Advanced Research Projects Agency under the contract FA8750-05-C-0212 administered by the Air Force Research Laboratory/Rome; by Army Research Office under grant W911NF-05-1-0374, by Federal Aviation Administration under the contract DTFAWA-04-P-00278/0001, by the National Science Foundation under grants CT-0627493, IIS-0242237 and IIS-0430402, and by Natural Sciences and Engineering Research Council of Canada under Discovery Grant N01035. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsoring organizations.

## References

[1] R. Agrawal, J. Kiernan, R. Srikant and Y. Xu, Implementing P3P using database technology, in: *Proceedings of the 19th International Conference on Data Engineering (ICDE'03)*, Bangalore, India, 2003.

[2] P. Ammann, D. Wijesekera and S. Kaushik, Scalable, graph-based network vulnerability analysis, in: *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS'02)*, ACM Press, Alexandria, VA, USA, 2002, pp. 217–224.

[3] F. Cuppens and A. Miege, Alert correlation in a cooperative intrusion detection framework, in: *Proceedings of the 2002 IEEE Symposium on Security and Privacy (S&P'02)*, IEEE, Oakland, CA, USA, 2002, pp. 187–200.

[4] R. Deraison, Nessus scanner, available at http://www.nessus.org, 1999.

[5] S. Jajodia, S. Noel and B. O'Berry, Topological analysis of network attack vulnerability, in: *Managing Cyber Threats: Issues, Approaches and Challenges*, V. Kumar, J. Srivastava and A. Lazarevic, eds, Kluwer Academic Publisher, Norwell, MA, USA, 2003.

[6] S. Jha, O. Sheyner and J.M. Wing, Two formal analysis of attack graph, in: *Proceedings of the 15th Computer Security Foundation Workshop (CSFW'02)*, Washington, DC, USA, 2002.

[7] P. Ning, Y. Cui and D.S. Reeves, Constructing attack scenarios through correlation of intrusion alerts, in: *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS'02)*, ACM Press, Alexandria, VA, USA, 2002, pp. 245–254.

[8] S. Noel and S. Jajodia, Correlating intrusion events and building attack scenarios through attack graph distance, in: *Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC'04)*, Tucson, AZ, USA, 2004.

[9] S. Noel, S. Jajodia, B. O'Berry and M. Jacobs, Efficient minimum-cost network hardening via exploit dependency graphs, in: *Proceedings of the 19th Annual Computer Security Applications Conference (ACSAC'03)*, Las Vegas, NV, USA, 2003.

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

[11] R. Ritchey and P. Ammann, Using model checking to analyze network vulnerabilities, in: *Proceedings of the 2000 IEEE Symposium on Research on Security and Privacy (S&P'00)*, IEEE, Oakland, CA, USA, 2000, pp. 156–165.

[12] 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 (S&P'02)*, IEEE, Oakland, CA, USA, 2002, pp. 273–284.

[13] A. Singhal, *Application of Data Warehousing and Data Mining Techniques for Intrusion Detection Systems*, Springer, The Netherlands, 2006.

[14] A. Singhal and S. Jajodia, Data mining for intrusion detection, in: *Data Mining and Knowledge Discovery Handbook*, O. Maimon and L. Rokach, eds, Springer, The Netherlands, 2005, pp. 1225–1237.

[15] A. Singhal and S. Jajodia, Data warehousing and data mining techniques for intrusion detection systems, *Journal of Parallel and Distributed Databases (DAPD)* **20**(2) (2006), 149–166.

[16] L. Wang, A. Liu and S. Jajodia, An efficient and unified approach to correlating, hypothesizing, and predicting intrusion alerts, in: *Proceedings of the 10th European Symposium on Research in Computer Security (ESORICS 2005)*, LNCS, Vol. 3679, Springer-Verlag, Milan, Italy, 2005, pp. 247–266.

[17] L. Wang, A. Liu and S. Jajodia, Using attack graphs for correlating, hypothesizing, and predicting intrusion alerts, *Computer Communications* **29**(15) (2006), 2917–2933.

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

[19] L. Wang, A. Singhal and S. Jajodia, Measuring the overall security of network configurations using attack graphs, in: *Proceedings of the 21th IFIP WG 11.3 Working Conference on Data and Applications Security (DBSec 2007)*, Redondo Beach, CA, USA, 2007.

[20] L. Wang, C. Yao, A. Singhal and S. Jajodia, Interactive analysis of attack graphs using relational queries, in: *Proceedings of the 20th IFIP WG 11.3 Working Conference on Data and Applications Security (DBSec 2006)*, LNCS, Vol. 4127, Springer-Verlag, Sophia Antipolis, France, 2006, pp. 119–132.