Chapter 23

INVESTIGATING COMPUTER ATTACKS USING ATTACK TREES

Nayot Poolsapassit and Indrajit Ray

Abstract System log files contain valuable evidence pertaining to computer attacks. However, the log files are often massive, and much of the information they contain is not relevant to the investigation. Furthermore, the files almost always have a flat structure, which limits the ability to query them. Thus, digital forensic investigators find it extremely difficult and time consuming to extract and analyze evidence of attacks from log files. This paper describes an automated attack-tree-based approach for filtering irrelevant information from system log files and conducting systematic investigations of computer attacks.

Keywords: Forensic investigation, computer attacks, attack tree, log file filtering

1. Introduction

Following a large-scale computer attack an investigator (system administrator or law enforcement official) must make a reasoned determination of who launched the attack, when the attack occurred, and what the exact sequence of events was that led to the attack. The system log file, which contains records of all system events, is often the starting point of the investigation. However, extracting and analyzing relevant information from log files is almost always performed manually; these tasks are prone to error and often produce inconclusive results.

There are three major contributing factors. First, a standard model does not exist for log file organization. Log files are usually flat text files (Figure 1), which limits the ability to query them. Second, there is no minimum requirement for information that needs to be stored in a log file; log files are invariably massive, and most of the information they contain is not relevant to the investigation. Finally, there are no established procedures for filtering and retrieving information from log

```
1697 05/04/1998 08:51:19 00:00:06 172.016.113.084 172.016.113.064/28  cp
2525 05/04/1998 09:12:42 00:00:01 172.016.114.158 172.016.114.128/28  nessus
2538 05/04/1998 09:13:21 00:00:07 172.016.114.159 172.016.114.128/28  nessus
2701 05/04/1998 09:16:02 02:42:20 135.013.216.191 135.013.216.10/24   mv
2731 05/04/1998 09:17:09 00:05:00 135.013.216.182 135.013.216.10/24   telnet
3014 05/04/1998 09:23:44 00:00:07 172.016.114.158 172.016.114.128/28  ftp
3028 05/04/1998 09:24:54 00:00:07 172.016.114.159 172.016.114.128/28  cp
3461 05/04/1998 09:37:14 00:00:13 172.016.114.159 172.016.114.128/28  telnet
4598 05/04/1998 10:01:51 00:00:01 196.037.075.158 196.037.075.10/24   finger
4612 05/04/1998 10:02:37 00:00:01 196.037.075.050 196.037.075.10/24   xterm
4834 05/04/1998 10:09:39 00:00:01 172.016.114.158 172.016.114.128/28  rlogin
4489 05/04/1998 10:10:22 00:00:01 195.073.151.050 195.073.151.10/24   smtp
4859 05/04/1998 10:10:33 00:00:01 195.073.151.150 195.073.151.10/24   smtp
4930 05/04/1998 10:11:36 00:00:06 172.016.114.158 172.016.114.128/28  telnet
5014 05/04/1998 10:13:55 00:00:06 172.016.114.148 172.016.114.128/28  bind
5092 05/04/1998 10:14:59 00:00:10 172.016.114.159 172.016.114.128/28  suid
5308 05/04/1998 10:21:38 00:00:01 194.027.251.021 194.027.251.021/24  smtp
5323 05/04/1998 10:23:11 00:00:01 196.037.075.158 196.037.075.5/24    ssh
5456 05/04/1998 10:28:50 00:00:01 194.027.251.021 194.027.251.021/24  ftp
5467 05/04/1998 10:29:26 00:00:01 196.037.075.158 196.037.075.10/24   sftp
5730 05/04/1998 10:36:58 00:00:03 135.008.060.182 135.008.060.10/24   ssh
7270 05/04/1998 11:09:08 00:00:02 135.008.060.182 135.008.060.10/24   mv
8098 05/04/1998 11:33:26 00:00:13 172.016.114.158 172.016.114.128/28  telnet
9057 05/04/1998 11:57:00 00:00:01 172.016.112.207 172.016.112.10/24   smtp
9113 05/04/1998 11:58:26 00:00:08 172.016.114.148 172.016.114.128/28  telnet
9352 05/04/1998 12:48:01 00:00:01 172.016.113.078 172.016.113.64/28   cp
```

Figure 1. Sample system log file.

files other than sequential backward scans starting from the most recent entry. Therefore, investigators typically rely on their experience and intuition to conduct *ad hoc* manual searches of log file entries.

To address this problem, we propose an attack-tree-based approach for filtering log files. The attack tree model captures the different ways a particular system can be attacked based on knowledge about system vulnerabilities and exploits. The filtering approach then selects the records from the log file that are relevant to the attack by matching against the attack tree. Subsequently, SQL queries may be used to extract evidence from the filtered records in an automated manner.

The next two sections present the basic attack tree model, and an augmented model that associates malicious operations with attack trees. Section 4 describes our attack-tree-based approach for filtering log files. The final section, Section 5, presents our concluding remarks.

2. Attack Trees

Attack trees have been proposed [2, 5, 8] as a systematic method for specifying system security based on vulnerabilities. They help organize intrusion and/or misuse scenarios by (i) identifying vulnerabilities and/or weak points in a system, and (ii) analyzing the weak points and dependencies among the vulnerabilities and representing these dependencies in the form of an AND-OR tree.

An attack tree is developed for each system to be defended. The nodes of the tree represent different stages (milestones) of an attack. The root node represents the attacker's ultimate goal, which is usually to cause damage to the system. The interior nodes, including leaf nodes, represent possible system states during the execution of an attack. System states may include the level of compromise (e.g., access to a web page or acquisition of root privileges), alterations to the system configuration (e.g., modification of trust or access control, or escalation of privileges), state changes to specific system components (e.g., placement of a Trojan horse), or other subgoals that lead to the final goal (e.g., the sequence of exploited vulnerabilities). The branches of an attack tree represent change of states caused by one or more actions taken by the attacker.

Changes in state are represented as AND-branches or OR-branches in an attack tree. Each node in an attack tree may be decomposed as:

- A set of events (exploits), all of which must be achieved for the subgoal represented by the node to succeed. These events are combined by an AND branch at the node. An example is a root account compromise, which involves changing the file mode of `/proc/self/` files and executing the `suid` command (CVE-2006-3626).

- A set of events (exploits), any one of which will cause the subgoal represented by the node to succeed. These events are combined by an OR branch at the node. An example is a root compromise resulting from a stack buffer overflow that exploits the `libtiff` library in SUSE v10.0 (CVE-2006-3459) or the SQL injection in Bugzilla v2.16.3 (CVE-2003-1043).

Attack trees are closely related to attack graphs used for vulnerability analysis [1, 3, 4, 6, 9, 10]. The difference lies in the representation of states and actions. Attack graphs model system vulnerabilities in terms of all possible sequences of attack operations. Ritchey and Ammann [7] have observed that scalability is a major shortcoming of this approach. In contrast, attack trees model system vulnerabilities in terms of cause and effect, and the sequential ordering of events does not have to be captured in an attack tree. Therefore, it is much simpler to construct an attack tree than an attack graph. One criticism of attack trees (vis-a-vis attack graphs) is that they cannot model cycles. However, we believe that this criticism is valid only when attack trees are used to represent sequences of operations leading to attacks, not when they are used to represent the dependencies of states that are reached. Another criticism is that attack trees tend to get unwieldy when modeling complex attack scenarios, but the same is true for attack graphs.

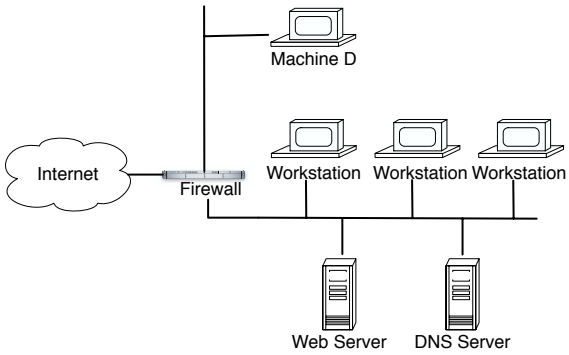

Figure 2. Corporate network configuration.

Figure 2 presents the network configuration of a hypothetical company. We use this network configuration to demonstrate how an attack tree is used to represent system vulnerabilities.

The company has installed a firewall to protect its network from the Internet. The company's web server is located in the de-militarized zone (DMZ). Other machines are on the local area network behind the firewall. The company's system administrator has configured the firewall to block port scans and flooding attacks. The firewall allows incoming connections only via port 25 (`smtp`) and port 80 (`http`).

Assume that a disgruntled employee, John Doe, plans to attack the company's network. He performs a vulnerability scan of network and determines that he needs to obtain root privileges on the web server to achieve his objective.

John Doe discovers that there are two alternative ways for gaining root privileges – his ultimate goal. One is by launching the `FTP/.rhost` attack. In this attack, the `.rhost` file on the web server is overwritten by a `.rhost` file of John Doe's choosing (say the `.rhost` file on his own machine) by exploiting a known vulnerability. This exploit causes the web server to trust John Doe's machine, enabling John Doe to remotely login on the server from his machine without providing a password. Having gained access to the web server, John Doe conducts the well-known `setuid` buffer overflow attack and obtains root privileges.

The second way to attack the web server is via a buffer overflow attack on the local DNS server. John Doe knows that the system administrator uses an old unpatched version of the BIND DNS application program. This enables him to perform the BIND buffer overflow attack on the local DNS server to take control of the machine. Next, he installs a network sniffer on the DNS server to observe sessions across the network.

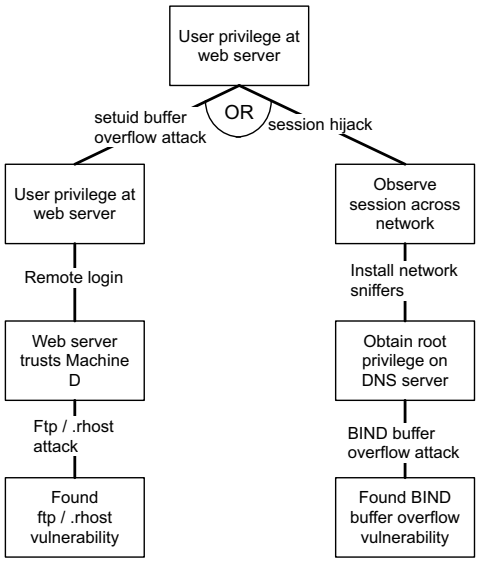

Figure 3. Attack tree for the corporate network.

Eventually, he hijacks the system administrator's `telnet` session to the web server and gains root privileges.

The two attacks are concisely represented in the simple attack tree in Figure 3. In general, an attack tree can be created to capture all the ways a system can be breached (clearly, it would not represent unknown or zero-day attacks). Such an attack tree can greatly simplify log file analysis: it is necessary to search the log file only for those operations that lie in the paths leading to the attack. For example, with reference to the attack tree in Figure 3, if an investigator knows that the root account at the web server was compromised, he needs to examine the log file only for the sequences of operations in the left and right branches of the attack tree. These operations must be in the same temporal order as the nodes going down the tree; any other order is not relevant to the attack. In fact, if the attack-tree-based log file analysis approach does not manifest a sequence of events leading to a specific attack, the attack in question is an unknown or zero-day attack.

3. Augmented Attack Trees

To facilitate the use of attack trees in forensic investigations, we define an "augmented attack tree," which extends the basic attack tree by associating each branch of the tree with a sequence of malicious operations that could have contributed to the attack.

DEFINITION 1 *An atomic event is an ordered pair* $\langle operation,\ target \rangle$.

DEFINITION 2 *An atomic event is an incident if its execution contributes to a system compromise.*

DEFINITION 3 *An augmented attack tree is a rooted labeled tree given by* $AAT = (V,\ E,\ \epsilon,\ Label,\ SIG_{u,v})$, *where*

1. V *is the set of nodes in the tree representing different states of partial compromise or subgoals that an attacker needs to move through in order to fully compromise a system.* $\mathcal{V} \in V$ *is the root node of the tree representing the ultimate goal of the attacker (full system compromise). The set* V *is partitioned into two subsets,* $leaf\_nodes$ *and* $internal\_nodes$, *such that*

   (i) $leaf\_nodes \cup internal\_nodes = V$,

   (ii) $leaf\_nodes \cap internal\_nodes = \phi$, *and*

   (iii) $\mathcal{V} \in internal\_nodes$

2. $E \subseteq V \times V$ *constitutes the set of edges in the attack tree. An edge* $(u, v) \in E$ *defines an "atomic attack" and represents the state transition from a child node* v *to a parent node* u $(u, v \in V)$. *An atomic attack is a sequence of incidents. The edge* (u, v) *is said to be "emergent from"* v *and "incident to"* u.

3. ϵ *is a set of tuples of the form* $\langle v,\ decomposition \rangle$ *such that*

   (i) $v \in internal\_nodes$ *and*

   (ii) $decomposition \in [AND\text{-}decomposition,\ OR\text{-}decomposition]$

4. *Label is the name of the exploit associated with each edge*

5. $SIG_{u,v}$ *is an attack signature (defined below).*

DEFINITION 4 *An incident-choice is a group of related incidents, the occurrence of any one of which can contribute to a state transition in the attack tree.*

DEFINITION 5 *An attack signature* $SIG_{u,v}$ *is a sequence of incident-choices* $\langle incident\text{-}choice_1,\ incident\text{-}choice_2,\ \ldots,\ incident\text{-}choice_n \rangle$ *for which the sequence* $(incident_{i,1},\ incident_{j,2},\ \ldots,\ incident_{m,n})$ *constitutes an atomic attack.*

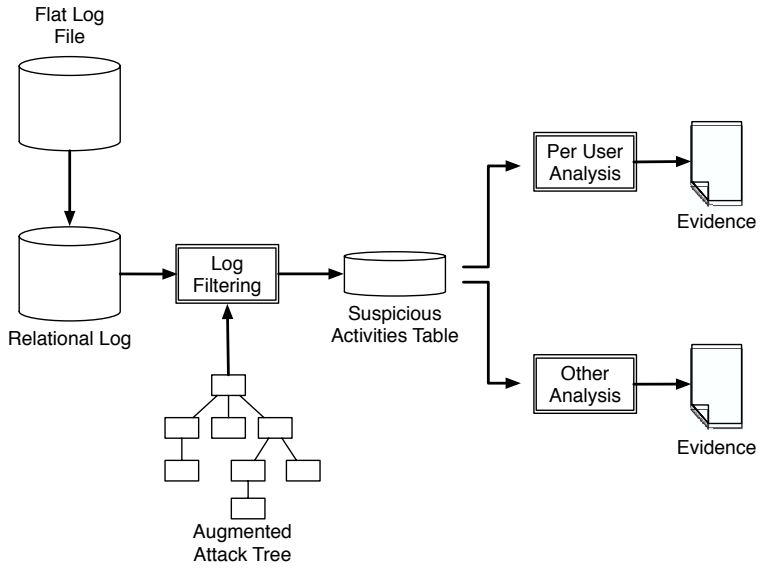

Figure 4. Log file investigation process.

The attack signature corresponding to the attack discussed in Bugtraq:3446 (CVE-1999-1562) – involving the execution of `wuftp` on a target machine (say A) and resulting in a cleartext password disclosure – is represented by:

```
((ftp, A),(debug, A),(open localhost, A), ("user name root", A),
    ("password xxx", A), (quote user root, A),(quote pass root, A))
```

DEFINITION 6 *A node $v \in internal\_nodes$ is an* AND-decomposition *if all the edges incident to the node are connected by the AND operation, or there is exactly one edge incident to the node.*

DEFINITION 7 *A node $v \in internal\_nodes$ is an* OR-decomposition *if all the edges incident to the node are connected by the OR operation.*

For an AND-decomposition node v, every subgoal of v represented by a child of v must be reached in order to reach v. For an OR-decomposition, the goal v is reached if any one of the subgoals is reached. Note that reaching a child goal is a necessary, but not sufficient, condition for reaching the parent goal.

4. Conducting a Forensic Investigation

Figure 4 shows how an augmented attack tree may be used to support a forensic investigation. First, the augmented attack tree is used

to prepare the set of incidents for all the attack signatures. Next, the attack tree is used to filter suspicious activities (operations) from non-suspicious ones. Finally, the suspicious activities are written to a relational database for further investigation.

A database structure with seven fields is used to store the filtered log file: *id*, *time stamp*, *source*, *source-group*, *operation*, *target* and *duration*. The *source* field stores the IP address of the connection originator. The *source-group* field contains the network address of the originator, if available. The *target* field similarly stores the destination address of the network connection. If investigative policies dictate, additional information from the log file may be included in the database.

4.1 Filtering Log Files

The augmented attack tree is first used to generate the set of incidents corresponding to all the attack signatures for the system. Each edge in the attack tree specifies an attack signature. Each attack signature is a collection of several incidents. The union of these incidents covers all the activities that can result in system compromise. The attack being investigated must have resulted from some incidents from this set of incidents. The set of incidents is then used to filter suspicious activities from normal activities.

The log file filtering algorithm (Algorithm 1) sequentially executes SQL queries to extract suspicious activities from the original log file. The results are written to a separate table called the Suspicious-Activities-Table for further investigation. This table has the same schema as the log file, but is significantly smaller.

The algorithm starts at the root node of the attack tree. It traverses every edge incident to the root node. For each edge, the algorithm extracts the attack signature $SIG_{u,v}$ given by the label of the edge. As mentioned earlier, the attack signature is the sequence of steps where an attacker may or may not have a choice of incidents (operation on a particular machine/target) to execute. For each step in the attack signature, the algorithm searches the log file for matching operations. An incident in the table matches the signature if the operation is executed on the particular machine or against the particular target as indicated in the attack signature. Note that only matched incidents that were executed prior to the time that the root node was compromised are suspected. Next, the suspected incidents are recorded into the Suspicious-Activities-Table by the selection procedure.

After the algorithm finishes exploring a particular edge e[u,v], it sets a time threshold for node v by selecting from the earliest incidents in e[u,v].

Algorithm 1 (Log File Filtering)

{**Description:** This algorithm traverses an augmented attack tree in a depth-first manner starting at the root. It examines all the edges under the current node u for suspicious incidents. If any suspicious activity is seen in the log file, it extracts the activity record and stores it in a separate file. When all the nodes have been visited, the algorithm returns the suspicious activity records as potential evidence.}
{**Input:** node u (initial from root), database table System-Log-File-Table}
{**Output:** database table Suspicious-Activities-Table}
BEGIN
if u is a leaf node **then**
  return
else
  for all $v \in Adj[u]$ **do**
    $SIG_{u,v} \leftarrow get\_SIGNATURE(e[u,v])$
    for all $\{incidents\}_i \in SIG_{u,v}$ **do**
      $Record_i \leftarrow SQL\{SELECT\ id, timestamp, source, source\text{-}group, operation, target, duration\ FROM$ System-Log-File-Table $WHERE\ operation, target\ Like\ \{incidents\}_i\ AND\ timestamp < u.timestamp;\}$
      if $Record_i \neq \{\ \}$ **then**
        Insert $Record_i$ into Suspicious-Activities-Table
      end if
    end for
    Set $v.timestamp$ = earliest timestamp of all $Record_i$ from the previous loop
    Recursively call Investigate(v, System-Log-File-Table)
    Mark e[u,v] if all $Record_i$ are not empty AND node v is previously compromised in Investigate(v)
  end for
  if node u has an AND-Decomposition AND all edges e[u,v] incident to u are fully marked **then**
    Mark node u as "Compromised"
  end if
  if node u has an OR-Decomposition AND there exists an e[u,v] incident to u already marked **then**
    Mark node u as "Compromised"
  end if
end if
END

This threshold is assumed to be the time when node v was compromised. Therefore, there is no need to suspect any incident in the subtree(s) under v that executed after this time. Next, the algorithm recursively calls itself to investigate the subtree under v from which the edge e[u,v] emerged. All the subtrees under the node are explored recursively. After all the subtrees under the root node or any intermediate node u have been explored, the algorithm marks an edge e[u,v] if it finds evidence that shows that all the steps in the attack signature $SIG_{u,v}$ have been

executed. If node u has an AND-decomposition, node u is considered compromised when all exploits (represented by edge e[u,v]) incident to u together with the state v from where the exploit emerged are marked. If node u has an OR-decomposition, node u is compromised when any one of its branches together with the state v are marked. Upon completion, the algorithm returns the augmented attack tree (with certain nodes marked as compromised) and the Suspicious-Activities-Table.

4.2 Identifying Likely Attack Sources

The next step is to process the Suspicious-Activities-Table produced by the log file filtering algorithm for candidate sources of the attack. This is accomplished by sorting the table entries by source aggregated by source-group to produce the list of candidate sources. Further investigation of this list can be performed on a per source basis either to reinforce or discard specific sources.

Algorithm 2 implements this task. The output of the algorithm is a table named Evidence-Log(source) where "source" is the identity of the source being investigated. This table has almost the same schema as the Suspicious-Activities-Table; the only difference is that it has an extra column called *exploit*. This field holds the exploit label corresponding to a relevant edge of the attack tree. If the algorithm returns a non-empty table, it supports the suspicion of the suspected source. On the other hand, if the algorithm returns an empty table, no decision can be made about the involvement of the suspected source. This is because of the possibility of zero-day attacks. Therefore, the algorithm should be used very carefully – it only provides evidence of activities that were possibly involved in an attack.

Algorithm 2 is similar to the log file filtering algorithm. The difference is that SQL queries are executed on a per source basis for sources in the Suspicious-Activities-Table. The algorithm marks the suspected records with the corresponding exploit labels. An investigator may use these labels to map the evidence back to exploits in the attack tree.

The Evidence-Log(source) Table holds the activities that are believed to be responsible for an attack on the system. The records are stored in chronological order. Typically, if an internal node is marked by the algorithm, it is almost certain that the suspected activity is responsible for the attack.

5. Conclusions

This paper has two main contributions. The first is an attack-tree-based filtering algorithm that eliminates information from a log file that

Algorithm 2 (Likely Attack Source Identification)

{**Description:** This algorithm takes an augmented attack tree and the Suspicious-Activities-Table generated by Algorithm 1 and filters the table based on a suspected source of attack. The algorithm traverses the augmented attack tree in a depth-first manner starting at the root. It examines all the edges under the current node u for suspicious incidents corresponding to the specific source. If any suspicious activity is seen in the log file, it extracts the activity record and stores it in a separate file. When all the nodes have been visited, the algorithm returns the set of suspicious activities for a specific source.}
{**Input:** node u (initial from root), specific-source, database table Suspicious-Activities-Table}
{**Output:** database table Evidence-Log(specific-source)}
BEGIN
if u is a leaf node **then**
  return
else
  for all $v \in Adj[u]$ **do**
    Sequence $SIG_{u,v} \leftarrow get\_SIGNATURE(e[u,v])$
    for all $\{incidents\}_i \in SIG_{u,v}$ **do**
      $Record_i \leftarrow SQL\{SELECT\ id, timestamp, source, source\text{-}group, operation, target, duration\ FROM$ Suspicious-Activities-Table $WHERE\ source = specific-source\ AND\ operation, target\ Like\ \{incidents\}_i\ AND\ timestamp < u.timestamp;\}$
      if $Record_i \neq \{\ \}$ **then**
        Insert $Record_i$ in Evidence-Log(specific-source)
        Mark Record$_i$ in Evidence-Log(specific-source) with the exploit label from the edge e[u,v]
      end if
    end for
    Set $v.timestamp$ = the earliest timestamp of all $Record_i$ from the previous loop
    Recursively call Investigate(v, specific-source, Suspicious-Activities-Table)
    Mark e[u,v] if all $Record_i$ are not empty AND node v is previously compromised in Investigate(v)
  end for
  if node u has an AND-Decomposition AND all edges e[u,v] incident to u are fully marked **then**
    Mark node u as "Compromised"
  end if
  if node u has an OR-Decomposition AND there exists edge e[u,v] incident to u already marked **then**
    Mark node u as "Compromised"
  end if
end if
END

is not related to the attack being investigated. The second is an additional filtering algorithm that extracts evidence corresponding to a particular source's role in an attack. Both the algorithms produce relational tables that are significantly smaller than the original log file and are, therefore, more manageable from an investigator's point of view. Furthermore, since the tables are relational in nature, they can be used as input to a database engine for rapid processing of evidence.

The approach is limited by its inability to handle unknown or zero-day attacks. This is because it assumes that knowledge exists about how system vulnerabilities can be exploited; this knowledge is, of course, not available for unknown or zero-day attacks. If such attacks are suspected, it is important not to discard the original log file as it may be needed for a future investigation. Another limitation arises from the assumption that the log file records all network events, whereas in reality individual machines maintain their own logs. The approach also assumes that the log file contains accurate information, i.e., the attacker has not tampered with the entries. Our research is currently investigating these issues with the goal of improving the attack-tree-based log file filtering algorithms.

References

[1] P. Ammann, D. Wijesekera and S. Kaushik, Scalable, graph-based network vulnerability analysis, *Proceedings of the Ninth ACM Conference on Computer and Communications Security*, pp. 217–224, 2002.

[2] J. Dawkins, C. Campbell and J. Hale, Modeling network attacks: Extending the attack tree paradigm, *Proceedings of the Workshop on Statistical Machine Learning Techniques in Computer Intrusion Detection*, 2002.

[3] S. Jha, O. Sheyner and J. Wing, Minimization and reliability analysis of attack graphs, Technical Report CMU-CS-02-109, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, 2002.

[4] S. Jha, O. Sheyner and J. Wing, Two formal analyses of attack graphs, *Proceedings of the Computer Security Foundations Workshop*, pp. 45–59, 2002.

[5] A. Moore, R. Ellison and R. Linger, Attack modeling for information survivability, Technical Note CMU/SEI-2001-TN-001, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, 2001.

[6] C. Phillips and L. Swiler, A graph-based system for network vulnerability analysis, *Proceedings of the New Security Paradigms Workshop*, pp. 71–79, 1998.

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

[8] B. Schneier, Attack trees: Modeling security threats, *Dr. Dobb's Journal*, December 1999.

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

[10] L. Swiler, C. Phillips, D. Ellis and S. Chakerian, Computer-attack graph generation tool, *Proceedings of the DARPA Information Survivability Conference and Exposition*, vol. 2, pp. 307–321, 2001.