Available online at www.sciencedirect.com
ScienceDirect
Computer Communications 29 (2006) 2917–2933
computer communications
www.elsevier.com/locate/comcom

Using attack graphs for correlating, hypothesizing, and predicting intrusion alerts

Lingyu Wang *, Anyi Liu, Sushil Jajodia

Center for Secure Information Systems, George Mason University, Fairfax, VA 22030-4444, USA

Available online 25 April 2006

Abstract

To defend against multi-step intrusions in high-speed networks, efficient algorithms are needed to correlate isolated alerts into attack scenarios. Existing correlation methods usually employ an in-memory index for fast searches among received alerts. With finite memory, the index can only be built on a limited number of alerts inside a sliding window. Knowing this fact, an attacker can prevent two attack steps from both falling into the sliding window by either passively delaying the second step or actively injecting bogus alerts between the two steps. In either case, the correlation effort is defeated. In this paper, we first address the above issue with a novel *queue graph* (QG) approach. Instead of searching *all* the received alerts for those that prepare for a new alert, we only search for the latest alert of each type. The correlation between the new alert and other alerts is implicitly represented using the temporal order between alerts. Consequently, our approach can correlate alerts that are arbitrarily far away, and it has a linear (in the number of alert types) time complexity and quadratic memory requirement. Then, we extend the basic QG approach to a unified method to hypothesize missing alerts and to predict future alerts. Finally, we propose a compact representation for the result of alert correlation. Empirical results show that our method can fulfill correlation tasks faster than an IDS can report alerts. Hence, the method is a promising solution for administrators to monitor and predict the progress of intrusions and thus to take appropriate countermeasures in a timely manner.

Keywords: Intrusion detection; Alert correlation; Vulnerability analysis; Intrusion prevention

1. Introduction

A network intrusion that is composed of multiple attacks preparing for each other can infiltrate a well-guarded network. Defending against such *multi-step intrusions* is important but challenging. It is usually impossible to respond to such intrusions based on isolated alerts corresponding to individual attack steps. The reason lies in the well-known impreciseness of Intrusion Detection Systems (IDSs). That is, alerts reported by IDSs are usually filled with false alerts that correspond to either normal traffic or failed attack attempts.

To more effectively defend against multi-step intrusions, isolated alerts need to be correlated into attack scenarios. Alerts can be correlated based on similar attributes (for example, the same destination hosts) and prior knowledge about alert types or corresponding vulnerabilities. Regardless of the different knowledge used to decide whether an alert *prepares for* another, the following *nested loop* procedure is usually adopted. Each new alert searches all the previous alerts for those who prepare for it. For off-line applications with a fixed set of alerts, such as computer forensics, the nested loop approach is a natural choice with reasonably good performance.

However, the defense against multi-step intrusions in large-scale high-speed networks poses a new challenge to the nested loop approach. To correlate potentially intensive alerts, the search in alerts must be implemented using an in-memory index [21]. However, an index built on all the received alerts would quickly exhaust any finite memory. Hence, the index can only be maintained for those alerts that are close enough to the new alert, namely, those inside

* Corresponding author. Tel.: +1 703 993 3931.
E-mail addresses: lwang3@gmu.edu (L. Wang), aliu1@gmu.edu (A. Liu), jajodia@gmu.edu (S. Jajodia).

0140-3664/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.comcom.2006.04.001

a *sliding window*. Unfortunately, an attacker aware of this fact can prevent two attack steps from both falling into the sliding window by either passively delaying the second step or actively injecting bogus alerts between the two steps. In either case, the correlation effort is completely defeated.

In this paper, we first propose a novel *queue graph* approach to remove the above obstacle towards efficient alert correlation. A queue graph only keeps in memory the latest alert matching each of the known exploits (that is, host-bound vulnerabilities). The correlation between a new alert and those in-memory alerts is explicitly recorded, whereas the correlation with other alerts is implicitly represented using the temporal order between alerts. Our approach has a linear time complexity and a quadratic memory requirement (both in the number of known exploits in the given network). Those are both independent of the number of received alerts, meaning the efficiency does not decrease over time. Our approach can correlate two alerts that are separated by arbitrarily many others. It is thus immune to deliberately slowed intrusions and injected bogus attacks.

Our second contribution is a unified method for the correlation, hypothesis, and prediction of intrusion alerts. The method compares *knowledge* encoded in a queue graph with *facts* represented by correlated alerts. An inconsistency between the knowledge and the facts implies potential attacks missed by IDSs, whereas extending the facts in a consistent way with respect to the knowledge indicates potential future attacks. Empirical results indicate that our method can fulfill those tasks even faster than the IDS can report alerts. The proposed techniques thus provide a promising solution for administrators to monitor and predict the progress of intrusions, and thus to take appropriate countermeasures in a timely manner.

Finally, we extend the preliminary results in [42] with a compact representation of the result graph. This compact representation removes all transitive edges and aggregates those alerts that are *indistinguishable* in terms of correlation. Consequently, the result graph contains no redundant information and becomes more perceptible. We modify the QG approach to support the compact representation. Unlike existing methods that take extra efforts to compress a result graph, the modified QG approach is actually more efficient in most cases. This extension is especially important in high-speed networks where potentially intensive attacks can lead to complex result graphs.

The rest of this paper is organized as follows. The next section reviews related work. Section 3 states basic concepts and assumptions. Section 4 proposes the queue graph approach. Section 5 devises a unified method for alert correlation, hypothesis, and prediction. Section 6 proposes a compact representation for the result graph. Section 7 gives empirical results. Finally, Section 8 concludes the paper. All the figures can be found at the end of the paper.

2. Related work

To reconstruct attack scenarios from isolated alerts, some correlation techniques employ prior knowledge about attack strategies [7,9,4,10,38] or alert dependencies [3,20,22]. Some techniques aggregate alerts with similar attributes [2,6,37,41] or statistical patterns [17,29]. Hybrid approaches combine different techniques for better results [22,30,43]. Alert correlation techniques are also used for other purposes than analyzing multi-step intrusions, such as to relate alerts to the same thread of attacks [14]. The privacy issue of alert correlation has recently been investigated [44]. Alert correlation is employed to deal with insider attacks [32,31]. To our best knowledge, the limitation of the nested-loop approach, especially for correlating intensive alerts in high-speed networks, has not been addressed.

Network vulnerability analysis enumerates potential attack sequences between fixed initial conditions and attack goals [26,33,5,34,36,16,1,15,25,45,11]. To avoid potential combinatorial explosion in the number of attack sequences, we adopt a notation of attack graphs similar to that of [1,26]. However, we do not assume fixed initial or goal conditions but let alerts to indicate the actual start and end of an intrusion. Efforts in integrating information from different sources include the formal model *M2D2* [19] and the Bayesian network-based approach [46]. We adopt the vulnerability-centric approach to alert correlation [24] because it can effectively filter out bogus alerts irrelevant to the network. However, the nested loop procedure is still used in [24].

Attack scenarios broken by missed attacks are reassembled by clustering alerts with similar attributes [23], and those caused by incomplete knowledge are pieced together through statistical analyses [30,29]. Instead of repairing a broken scenario afterwards, our method can tolerate and hypothesize missed attacks at the same time of correlation. Real-Time detection of isolated alerts is studied in [18,28]. Some products claim to support real-time analyses of alerts, such as the Tivoli Risk Manager [13]. Designed for a different purpose, the RUSSEL language is similar to our approach in that the analysis of data only requires one-pass of processing [12].

This paper extends the preliminary results reported in [42] as follows. First, we show that a result graph produced by the basic QG approach given in [42] still contains redundant information, such as transitive edges. The transitive edges can be removed to simplify a result graph without losing any information. Moreover, some alerts in a result graph are *indistinguishable* with respect to their relationship with others. Those alerts can thus be aggregated to make the result graph more perceptible. Second, we modify the basic QG approach such that it produces result graphs with all transitive edges removed and indistinguishable alerts aggregated. Unlike existing approaches that take extra efforts to compress result graphs, the modified version of our QG approach not only produces more compact result graphs, but also is more efficient in most cases.

3. Preliminaries

This section reviews relevant concepts and states our assumptions. Section 3.1 introduces *attack graphs* and Section 3.2 discusses *intrusion alerts*. Section 3.3 then indicates the limitations of the *nested loop* approach to motivate our study.

3.1. Attack graph

An *attack graph* represents prior knowledge about a given network in terms of vulnerabilities and connectivity [1,36]. An attack graph is a directed graph having two type of vertices, *exploits* and *security conditions*. Exploits are host-bound vulnerabilities. More precisely, an exploit, denoted as a triple (*vul*, *src*, *dest*), indicates that the vulnerability *vul* exists on the host *dest* and the two hosts *src* and *dest* are connected (*src* and *dest* may refer to the same host in a local exploitation). Security conditions refer to the network states required or implied by exploits, such as privilege levels or trusts. The interdependencies between exploits and security conditions form the edges of an attack graph. An edge from a security condition to an exploit indicates that the exploit cannot be executed until the security condition has been satisfied; an edge from an exploit to a security condition indicates that executing the exploit will satisfy the security condition.

Example 3.1. Fig. 1 depicts part of an attack graph. Security conditions appear as ovals and exploits as rectangles. The edges indicate that the buffer overflow exploit can be executed only if the attacker can access the source host and the destination host has the vulnerable service.

We assume attack graphs can be obtained with existing tools, such as the Topological Vulnerability Analysis (TVA) system [15], which can model 37,000 vulnerabilities taken from 24 information sources including X-Force, Bugtraq, CVE, CERT, Nessus, and Snort. We assume the attack graph can be placed in memory. For a given network, the size of an attack graph can usually be estimated and the required memory can be accordingly allocated.

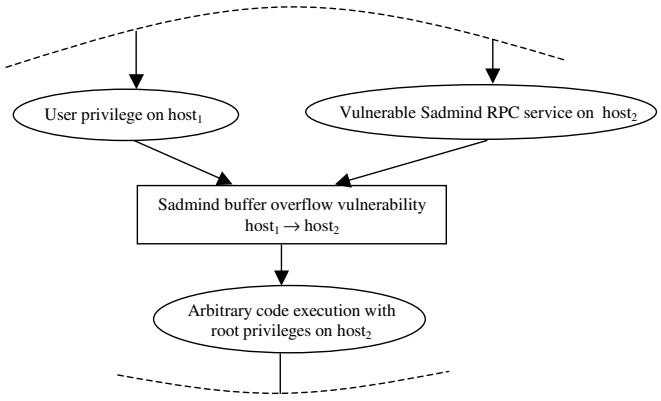


Fig. 1. An example of attack graph.

Unlike in [36,33], we do not assume fixed initial or goal conditions in an attack graph but let alerts to indicate the actual start and end of an intrusion. We do not assume external host addresses to be trustful and use wildcards to match them. This may cause false correlations when multiple attackers concurrently launch similar attacks while they do not intend to cooperate with each other.

To simplify our discussion, we introduce some notations to formally denote attack graphs. Let E be the set of known exploits, and C be the set of relevant security conditions. Denote the *require* and *imply* relationship between exploits and security conditions as the relation $R_r \subseteq C \times E$ and $R_i \subseteq E \times C$, respectively. An attack graph is the directed graph $G(E \cup C, R_r \cup R_i)$. The *prepare-for* relationship between exploits, as captured by many alert correlation methods [3,20], is simply the composite relation $R_i \circ R_r$. We shall denote this composite relation using the symbol $\rightarrow$.

3.2. Intrusion alert

Intrusion alerts reported by IDS sensors in a given network typically contain attributes like the type of events, the address of the source and destination host, the time stamp, and so on. Our discussion does not depend on specific format of alerts, so we simply regard each alert as a relational tuple with a given (usually implicit) schema. For example, with the schema *(event type, source IP, destination IP, time stamp)*, an alert will have the form *(RPC portmap sadmind request UDP, 202.77.162.213, 172.16.115.20, 03/07-08:50:04.74612)*.

A vulnerability-centric correlation approach matches alerts with exploits such that alerts can be correlated using the knowledge encoded in an attack graph [24]. To match alerts with exploits, the event type attributes of alerts need to be mapped to the vulnerability attributes of exploits using domain knowledge, such as the correspondence between Snort identifiers and Nessus identifiers [27]. For simplicity, we denote the matching between alerts and exploits as a function Exp() from the set of alerts A to the set of exploits E (in some cases an event type matches multiple vulnerabilities, which will be handled by creating a copy of alert for each matched exploit).

The vulnerability-centric approach can mitigate the negative impact of disruptive alerts. For example, if the attacker blindly launches some Windows-specific attacks on UNIX machines, then the reported alerts will be ignored by the approach. On the other hand, the approach also has limitations in that relevant alerts do not always match exploits. For example, an ICMP PING matches no vulnerability, but it may signal the probing preparation for following attacks. Such relevant alerts can be identified based on attack graphs and the knowledge about alert types. We extend the concept of exploits to include alert types in the place of vulnerability attributes. Such special *exploits* are added to attack graphs and the function Exp is extended accordingly.

Our methods critically depend on temporal characteristics of alerts, such the order of arrivals and timestamps. In practice, those characteristics are expected to exhibit much uncertainty due to various delays in hosts and network, especially when alerts are collected from multiple sensors at different locations. We address such temporal impreciseness in more details in Section 4.3. We assume the clocks of IDS sensors are loosely synchronized with the correlation engine. This can be achieved in many different ways depending on specific IDS systems. For example, Snort has built-in support of automatic time synchronization through the network time protocol (NTP) [35]. We leave the case where attackers may temper with the clocks as future work.

3.3. The nested loop approach and its limitations

A natural way to correlate alerts is to search all the received alerts for those who prepare for a new alert. This *nested loop* procedure is assumed by many correlation methods. The left side of Fig. 2 shows a sequence of alerts ascending in time, $a_0, a_1, \ldots, a_n$. For each $i = 1, 2, \ldots, n$, the approach searches $a_0, a_1, \ldots, a_{i-1}$ for those a_js that satisfy $\mathrm{Exp}(a_j) \rightarrow \mathrm{Exp}(a_i)$. However, this does not imply that a_i must be compared to every $a_j (0 \leqslant j \leqslant i-1)$, although it comprises a simple implementation of the search. The search for the alerts that prepare for a_i can be optimized with an index on $a_0, a_1, \ldots, a_{i-1}$. After a_i is processed, an entry corresponding to a_i is inserted into the index. By maintaining such an index in memory, the nested loop approach can have a relatively good performance (for example, 65 k alerts can be processed in less than 1 s [21]).

It is not always possible to have enough memory for indexing all the alerts. Hence, a *sliding window* approach comes to the rescue. That is, only the alerts close enough to the new alert are considered for correlation. As illustrated in the right side of Fig. 2, for the alert a_i the search is only performed on $a_{i-k}, a_{i-k+1}, \ldots, a_{i-1}$, where k is a given window size determined by available memory. However, this sliding window approach leads to an unavoidable tradeoff between the performance and the completeness of correlation. On one hand, the performance requires k to be small enough so the index fits in memory. On the other hand, a smaller k means less alerts will be considered for correlation with the new alert, and thus the result may be incomplete as two related alerts may in fact be separated by more than k others.

This tradeoff between the performance and the completeness causes a more serious problem for real-time correlation in high-speed networks, where performance is critical and attacks can be intensive. The problem is exacerbated by those attackers who are aware of the ongoing detection effort. An attacker can employ the following *slow attack* to defeat alert correlation. More specifically, given an arbitrarily large window size k, for any two attacks that raise the correlated alerts a_i and a_j, the attacker can delay the second attack until at least k other alerts have been raised since a_i, so $j - i > k$ meaning a_i and a_j will not be correlated. Instead of passively awaiting, a smarter attacker can actively launch bogus attacks between the two real attack steps, so the condition $j - i > k$ can be satisfied in a shorter time. The attacker can even script bogus attack sequences between the real attack steps, such that a deceived correlation engine will be kept busy in producing bogus attack scenarios, while the real intrusion will be advanced in peace of mind.

4. The queue graphs (QG) approach to correlating alerts

This section proposes a novel Queue Graph (QG) data structure to remove the limitation of a nested loop approach. Section 4.1 makes a key observation. Section 4.2 then introduces the QG data structure. Finally, Section 4.3 deals with alerts with imprecise temporal characteristics.

4.1. Implicit correlation and explicit correlation

The key observation is that the correlation between alerts does not always need to be explicit. In Fig. 3, suppose the first three alerts a_i, a_j, and a_k all match the same exploit $\mathrm{Exp}(a_k)$ (that is, their event types match the same vulnerability and they involve the same source and destination hosts); the alert a_h matches another exploit $\mathrm{Exp}(a_h)$; $\mathrm{Exp}(a_k)$ prepares for $\mathrm{Exp}(a_h)$. Hence, a_i, a_j, and a_k should all be correlated with a_h. However, if the correlation between a_k and a_h is explicitly recorded (shown as a solid line in the figure), then the correlation between a_j and a_h can be kept implicit (shown as a dotted-line). More precisely, the facts $\mathrm{Exp}(a_j) = \mathrm{Exp}(a_k)$ and $\mathrm{Exp}(a_k) \rightarrow \mathrm{Exp}(a_h)$

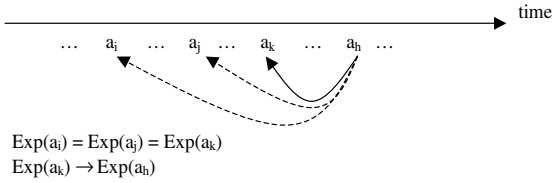


Fig. 3. Implicit and explicit correlation.

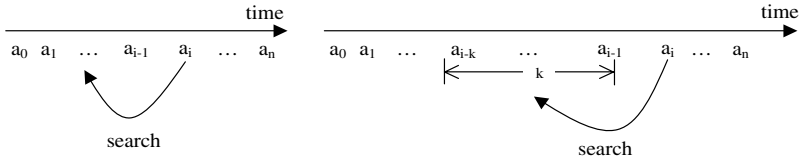


Fig. 2. The nested loop approach with or without a sliding window.

jointly imply $\mathrm{Exp}(a_j) \rightarrow \mathrm{Exp}(a_h)$, and the facts that a_j occurs before a_k and a_k occurs before a_h jointly imply that a_j must also occur before a_h. Similar arguments apply to the correlation between a_i and a_h.

To generalize the above observation, a new alert only needs to be explicitly correlated with the *latest* alert matching each exploit. Its correlation with other alerts matching the same exploit can be kept implicit with the temporal order (for example, a_j occurs before a_k and a_k occurs before a_h) and with the matching between alerts and exploits (for example, a_j and a_k match the same exploit). In the above case, if a_k is indeed the latest alert matching $\mathrm{Exp}(a_k)$, then only the correlation between a_h and a_k needs to be explicit. As we shall show shortly, keeping correlations implicit can significantly reduce the complexity and memory requirement. Intuitively, for each exploit the correlation algorithm only needs to search backward for the first (a_k in the above case) alert matching that exploit. For the nested loop approach, however, the correlation is always explicit. Hence, the approach must unnecessarily search all the received alerts, as discussed in Section 3.3.

4.2. Correlating alerts using queue graphs

We design an in-memory data structure, called *Queue Graph*, to take advantage of the above observation about implicit correlation. A queue graph is an in-memory materialization of the given attack graph with enhanced features (the purpose of the features will be clear in the following sections). Each exploit is realized as a queue of length one, and each security condition as a variable.

The realization of edges is a little more complicated. Starting from each exploit e_i, a breadth-first search (BFS) is performed in the attack graph by following the directed edges. For each edge encountered during the search, a *forward* pointer is created to connect the corresponding queue and variable. Similarly, another search is performed by following the directed edges in their reversed direction, and a *backward* pointer is created for each encountered edge. Later, we shall use the backward edges for correlation purposes and use the forward edges for prediction purposes.

The two collections of pointers are then placed at a separate *layer* tailored to the queue that corresponds to the exploit e_i. The reason for separating pointers into layers is as follows. A BFS always creates a tree (namely, the BFS tree), and hence later another BFS starting from the same queue can follow only the pointers at that layer. This later BFS will then be performed within a *tree* instead of a *graph*, reducing the complexity from quadratic to linear. We first illustrate the concepts in Example 4.1.

Example 4.1. In Fig. 4, from left to right are a given attack graph, the corresponding queues (shown as buckets) and variables (shown as texts), and the (both forward and backward) pointers at different layers. Notice that the layer one pointers do not include those connecting v_2 and Q_3, because a BFS in the attack graph starting from e_1 will reach c_2 only once (either via e_2 or via e_3, but we assume e_2 in this example). The layer one pointers thus form a tree rooted at Q_1.

In Section 3.3, we discussed how a nested loop approach correlates alerts that prepare for each other. As a comparison, we now perform the same correlation using a queue graph (we shall discuss other correlation requirements in Section 5). Intuitively, we let the stream of alerts flow through the queue graph, and at the same time we collect correlation results by searching the queue graph. More specifically, each incoming alert is first matched with an exploit and placed in the corresponding queue. Then, because the length of each queue is one, a non-empty queue must dequeue the current alert before it can enqueue a new alert.

During this process, the results of correlation are collected as a directed graph, namely, the *result graph*. First, each new alert is recorded as a vertex in the result graph. Second, when a new alert forces an old alert to be dequeued, a directed edge between the two alerts is added into the result graph, which records the temporal order between the two alerts and the fact that they both match the same exploit. Third, after each new alert is enqueued, a search starts from the queue and follows two consecutive backward pointers; for each non-empty queue encountered during the search, a directed edge from the alert in that queue

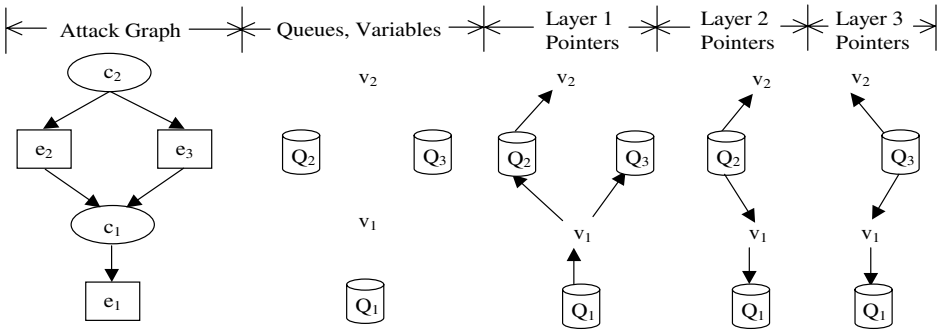


Fig. 4. An example queue graph.

Procedure *QG_Alert_Correlation*
Input: A queue graph Q_g (with n queues and m variables), the initial result graph $G_r(V, E_r \cup E_l)$, and an alert a_{new} satisfying $Exp(a_{new}) = e_i$ for some $1 \leq i \leq n$
Output: The updated result graph $G_r(V, E_r \cup E_l)$
Method:
1. **Insert** a_{new} into V
2. **If** Q_i contains an alert a_{old}
   Insert edge (a_{old}, a_{new}) into E_l
   Dequeue a_{old} from Q_i
3. **Enqueue** a_{new} into Q_i
4. **For** each $Q_j (1 \leq j \leq n)$ satisfying $\langle Q_i, v_k \rangle \in \mathcal{P}_i$ and $\langle v_k, Q_j \rangle \in \mathcal{P}_i$, for some $1 \leq k \leq m$
   If Q_j contains an alert a_j
   Insert (a_j, a_{new}) into E_r
5. **Return** $G_r(V, E_r \cup E_l)$

Fig. 5. A procedure for correlating alerts with queue graphs.

to the new alert is added into the result graph. This is illustrated in Example 4.2.

Example 4.2. Consider correlating the four alerts a_i, a_j, a_k, and a_h in Fig. 3 with the queue graph given in Fig. 4, and suppose $\mathrm{Exp}(a_h) = e_1$, $\mathrm{Exp}(a_k) = e_2$, and no other alerts match e_1 or e_2 besides a_i, a_j, a_k, and a_h. First, when a_i arrives, it is placed in the empty queue Q_2. Then, a_j forces a_i to be dequeued from Q_2, and a directed edge (a_i, a_j) in the result graph records the facts that a_i is before a_j and they both match e_2. Similarly, a_k replaces a_j in Q_2, and a directed edge (a_j, a_k) is recorded. Finally, a_h arrives and occupies Q_1, a search starting from Q_1 and following two layer one backward pointers will find the alert a_k in Q_2. Hence, a directed edge (a_k, a_h) records the only explicit correlation.

Definition 1. Let $G(E \cup C, R_r \cup R_i)$ be an attack graph, where $E = \{e_i \mid 1 \leqslant i \leqslant n\}$, $C = \{c_i \mid 1 \leqslant i \leqslant m\}$, $R_r \subseteq C \times E$, and $R_i \subseteq E \times C$.

- For $k = 1, 2, \ldots, n$,
  [•] use $BFSR(k)$ to denote the set of edges visited by a breadth-first search in $G(E \cup C, R_r \cup R_i)$ starting from e_k, and
  [•] use $BFS(k)$ for the set of edges visited by a breadth-first search in $G(E \cup C, R_r^{-1} \cup R_i^{-1})$ staring from e_k, where R_r^{-1} and R_i^{-1} are the inverse relations.
- The **queue graph** Q_g is a data structure with the following components:
  [•] $\mathcal{Q} = \{Q_i \mid 1 \leqslant i \leqslant n\}$ are n queues of length one,
  [•] $\mathcal{V} = \{v_i \mid 1 \leqslant i \leqslant m\}$ are m variables,
  [•] for each $k = 1, 2, \ldots, n$,
  $\mathcal{P}_k = \{\langle Q_j, v_i \rangle \mid (c_i, e_j) \in BFS(k)\} \cup \{\langle v_i, Q_j \rangle \mid (e_j, c_i) \in BFS(k)\}$ are the layer k backward pointers, and
  $\mathcal{PR}_k = \{\langle v_i, Q_j \rangle \mid (c_i, e_j) \in BFSR(k)\} \cup \{\langle Q_j, v_i \rangle \mid (e_j, c_i) \in BFSR(k)\}$ are the layer k forward pointers.

Definition 1 formally characterizes the queue graph data structure. To rephrase Example 4.1 using those notations, the queue graph has three queues $\mathcal{Q} = \{Q_1, Q_2, Q_3\}$ and two variables $\mathcal{V} = \{v_1, v_2\}$.

The layer-one forward pointers are $\mathcal{PR}_1 = \phi$, and the layer-one backward pointers are $\mathcal{P}_1 = \{\langle Q_1, v_1 \rangle, \langle v_1, Q_2 \rangle, \langle Q_2, v_2 \rangle, \langle v_1, Q_3 \rangle\}$.[1] The layer two pointers include $\mathcal{P}_2 = \{\langle Q_2, v_2 \rangle\}$ and $\mathcal{PR}_2 = \{\langle Q_2, v_1 \rangle, \langle v_1, Q_1 \rangle\}$. The layer three pointers include $\mathcal{P}_3 = \{\langle Q_3, v_2 \rangle\}$ and $\mathcal{PR}_3 = \{\langle Q_3, v_1 \rangle, \langle v_1, Q_1 \rangle\}$.

The process for correlating alerts using a queue graph, as illustrated in Example 4.2, is more precisely stated as the procedure *QG_Alert_Correlation* in Fig. 5 . The result graph G_r has a set of vertices V and two separate sets of edges E_r and E_l. The edges in E_r correspond to the explicit correlations and those in E_l record the temporal order between alerts matching the same exploit. Initially, we set the queues in $\mathcal{Q}$, the sets V, E_r, and E_l as empty. The first step of the procedure inserts the new alert into the result graph. The second step dequeues a non-empty queue and updates the result graph by adding an edge between the old alert and the new alert. The third step enqueues the new alert into the queue graph. The fourth step does correlation by searching for the alerts that need to be explicitly correlated to the new alert.

4.2.1. Complexity analysis

The procedure *QG_Alert_Correlation* demonstrates the advantages of the QG approach over the nested loop approach (some of the features of a queue graph, such as the variables and the forward pointers, are not used by the procedure and will be needed in the next section). First, the time for processing each new alert with the QG approach is linear in $(m + n)$, that is the number of exploits and security conditions in the attack graph. In Procedure *QG_Alert_Correlation*, the fourth step visits at most $(m + n)$ edges, because it searches in a *tree* (that is, the BFS tree rooted at Q_i) by following the layered pointers in $\mathcal{P}_i$; the other steps of the procedure take almost constant time. Hence, the performance of the QG approach does not depend on the number of received alerts, as n and m are relatively stable for a given network. On the other hand, the nested loop approach (without using a sliding window) searches all alerts, and hence the performance keeps decreasing as more and more alerts are received.

[1] We use the notation $\langle a, b \rangle$ for a pointer in a queue graph and (a, b) for an edge in a graph.

Procedure *QG_Alert_Correlation_Hypothesis*
Input: A queue graph $Q_g(\mathcal{Q} \cup \mathcal{V}, \mathcal{P}_i \cup \mathcal{PR}_i)$ with $| \mathcal{Q} | = n$ and $| \mathcal{V} | = m$
The initial result graph $G_r(V, E_r \cup E_l)$
An alert a_{new} enqueued in $Q_i (1 \leq i \leq n)$ with the timestamp t_{new}
Output: The updated result graph $G_r(V, E_r \cup E_l)$
Method:
For each $v_j \in \mathcal{V}$ satisfying $< Q_i, v_j > \in \mathcal{PR}_i$
Update the value of v_j as $TRUE$ and timestamp as t_{new}
Insert a copy of v_j into V, and an edge (a_{new}, p) into E_r
Do a BFS starting from Q_i and following pointers in $\mathcal{P}_i$
For each $v_j \in \mathcal{V}$ that the search is leaving
If v_j has a value $TRUE$ or HYP
Skip searching the queues connected to v_j
If v_j has a value $FALSE$
Update v_j to be HYP and t_{new}
Insert the tuple j, HYP, t_{new} into V
For each $Q_l \in \mathcal{Q}$ that the search reaches
Insert the edge (Q_l, v_j) into E_r
For each $Q_j \in \mathcal{Q}$ that the search is leaving
If Q_j contains a hypothesized alert
Skip searching the variables connected to Q_j
If Q_j is empty
insert into Q_j a hypothesized alert a_{hyp}
Insert a_{hyp} into V
For each $v_l \in \mathcal{V}$ that the search reaches
Insert the edge (v_l, Q_j) into E_r
Return $G_r(V, E_r \cup E_l)$

Fig. 6. A procedure for alert correlation and hypothesis.

Second, the memory usage of the QG approach is roughly $O(n(n+m))$ (n layers, with each layer having maximally $(n+m)$ pointers),[2] and hence does not depend on the number of received alerts, either. In comparison, the nested loop approach without a sliding window needs memory for indexing on all the received alerts. Third, the QG approach is not vulnerable to slowed attacks, which can easily defeat the nested loop approach using a sliding window as described in Section 3.3. In the procedure *QG_Alert_Correlation*, an alert is dequeued (and no longer considered for correlation) only when a new alert matching the same exploit arrives. Hence, if one alert prepares for another, then no matter how many unrelated alerts are injected, the earlier alert will always sit in the queue graph waiting for the later one. In case some temporal constraint states that an alert should not be considered for correlation once it gets *too old*, a timer can be used to periodically dequeue alerts. However, care must be taken in using such temporal constraints because they may render the correlation vulnerable to slowed attacks again.

4.3. Alerts with imprecise temporal characteristics

The correctness of the QG approach critically depends on the correct order of alerts. However, neither the order suggested by timestamps nor the order of arrivals should be trusted, because the temporal characteristics of alerts are typically imprecise. Instead, we adopt the following conservative approach. First, any two alerts whose timestamps have a difference no greater than a given threshold t_{con} are treated as *concurrent*; the *correct* order of concurrent alerts is always the one that allows the alerts to be correlated. Second, for non-concurrent alerts, the correct order is the one suggested by their timestamps, but alerts are allowed to arrive in a different (and incorrect) order. This conservative approach takes into account varying delays in a network and small differences between the clocks of sensors.[3]

The basic QG approach does not work properly if alerts do not arrive in the correct order. Consider an alert a_1 that prepares for another alert a_2 but arrives later then a_2. As described in Section 4.2, the procedure *QG_Alert_Correlation* will only look for those alerts that prepare for a_1, but not those that a_1 prepares for (a_2 in this case). Moreover, if another concurrent alert a_2' matches the same exploit as a_2 does and arrives after a_2 but before a_1. Then, a_2 is already dequeued by the time a_1 arrives, and hence the correlation between a_1 and a_2 will not be discovered.

We address this issue by reordering alerts inside a time window before feeding them into the queue graph.[4] More specifically, assume the varying delays are bounded by a threshold t_{max}. We postpone the processing of an alert a_1 with a timestamp t_1 until t_{max} (the larger one between t_{max} and t_{con}, when concurrent alerts are also considered) time has passed since the time we receive a_1. We reorder the postponed alerts, so they arrive at the correlation engine in the correct order. Then after t_{max} time, any alert a_2 will have a timestamp t_2 satisfying $t_2 > t_1$ (the worst case is when a_1 is not delayed but a_2 is delayed t_{max} time, and the fact a_2 is received t_{max} later than a_1 indicates $t_2 + t_{\text{max}} - t_{\text{max}} > t_1$, and hence $t_2 > t_1$).

The capability of dealing with concurrent alerts and varying delays comes at a cost. The additional delay introduced for reordering alerts causes an undesired decrease in the timelineness of alert correlation. However, if we choose to report results immediately as each alert arrives, then the imprecise temporal characteristics of alerts may cause incorrect and confusing results. Such results may diminish the value of the correlation effort. This reflects the inherent tradeoff between the capability of containing unavoidable uncertainties and the performance of processing alerts.

5. A unified method for alert correlation, hypothesis, and prediction

In this section, we extend the basic QG-based correlation procedure to a unified method for correlating received alerts, hypothesizing missing alerts, and predicting future alerts. Section 5.1 introduces some key concepts. Sections

[2] The correlation only appends to the result graph but does not read from it, and hence the result graph needs not to be in memory.

[3] We assume the clocks are loosely synchronized, as discussed in Section 3.2.

[4] Notice that here a time window is used for reordering alerts and no alert is excluded from correlation, which is different from the time window used by the nested loop approach.

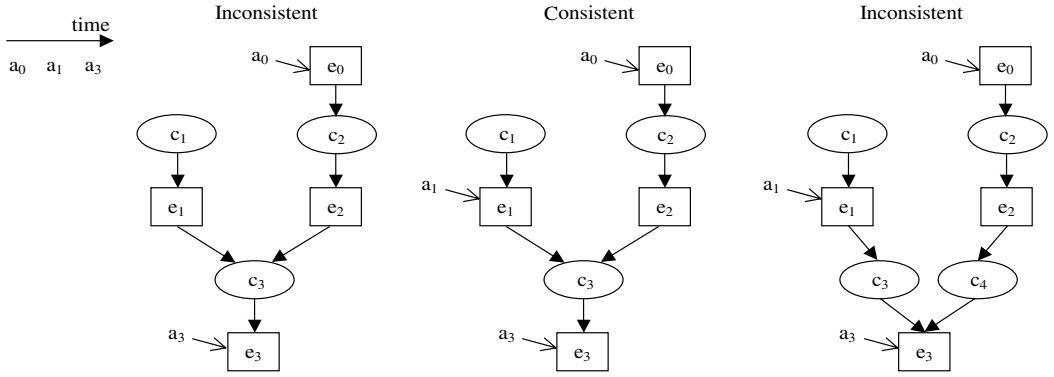


Fig. 7. Examples of consistent and inconsistent alert sequences.

5.2 describes the integration of alert correlation with alert hypothesis. Section 5.3 then discusses alert prediction.

5.1. Consistent and inconsistent alert sequences

The queue graph approach introduced in Section 4 provides unique opportunities to tolerate and hypothesize alerts missed by IDSs, as well as to predict possible consequences. Intuitively, missing alerts cause *inconsistency* between the knowledge (encoded in attack graphs) and the facts (represented by received alerts).[5] By reasoning about such inconsistency, missing alerts can be plausibly hypothesized. On the other hand, by extending the facts in a consistent way with respect to the knowledge, possible consequences of an intrusion can be predicted. To elaborate on those ideas, we first illustrate consistent and inconsistent sequences of alerts in Example 5.1 and Example 5.2.

Example 5.1. The sequence of alerts shown on the left hand side of Fig. 7(that is, a_0, a_3) is inconsistent with respect to the attack graph, because the security condition c_3 is not satisfied before the exploit e_3 is executed (as indicated by the alert a_3).

Example 5.2. In Fig. 7, the sequence a_0, a_1, a_3 is consistent, because executing the exploit e_1 (as indicated by the alert a_1) satisfies the only security condition c_3 that is required by the execution of e_3 (as indicated by a_3). The sequence shown on the right hand side of Fig. 7 is inconsistent, because the security condition c_4 is not satisfied before the execution of e_3.

To generalize the above examples, we say an exploit is *ready* to be executed if all of its required security conditions are satisfied by previous executions of exploits (or initially satisfied security conditions, such as c_1 in Fig. 7). We say a sequence of alerts is *consistent*, if every alert in the sequence matches an exploit that is ready to be executed by the time the alert is received. Example 5.1 depicts an inconsistent alert sequence in which the consecutive executions of exploits is broken by missing alerts. Example 5.2 indicates that the relationship between exploits can be either *disjunctive* (executing e_1 or e_2 makes e_3 ready in the first case) or *conjunctive* (both e_1 and e_2 must be executed to make e_3 ready), and security conditions play an important role in such relationship (the approach in [24] cannot distinguish the two cases in Example 5.2, because it is based on a simplified version of attack graphs with no security conditions).

5.2. Alert correlation and hypothesis

In Section 4.2, the correlation algorithm searches for the alerts that prepare for a new alert by following two consecutive pointers. Such an approach only works for consistent alert sequences. For inconsistent sequences, such as those in Example 5.1 and Example 5.2, the search will stop at empty queues that correspond to missing alerts and the correlation result will be incomplete. A natural question is, *Can we continue to search and hypothesize missing alerts if necessary?* This question motivates us to propose a unified method to correlating received alerts and at the same time making hypotheses of missing alerts.

Intuitively, the approach attempts to explain the occurrence of a new alert by including it in a consistent sequence of alerts (alert correlation) and missing alerts (alert hypothesis). More specifically, a search starts from the queue that contains the new alert; it hypothesizes about a missing alert for each encountered empty queue; it stops at each received alert because it knows that this received alert must have already been explained previously. The search expands its frontier in a breadth-first manner[6] after each hypothesis is made, because the hypothesis itself may also need an explanation. Such attempts continue until a satisfactory explanation for the new alert and all the hypothesized ones is obtained. The explanations of all received alerts collec-

[5] Notice that we assume our vulnerability-centric correlation approach can effectively filter out bogus alerts that do not match any existing vulnerability, and hence attacks cannot fool the system into producing false inconsistency.

[6] Other approaches, such as a DFS, may work as well, but a queue graph organizes its pointers in layered BFS trees to improve performance, and this makes BFS a preferred choice.

tively form the result, that is a graph composed of alerts, hypothesized alerts, and security conditions that are either satisfied or hypothetically satisfied. This is illustrated in Example 5.3.

Example 5.3. Consider again the three cases, from left to right, in Fig. 7 when the alert a_3 is received. For the first case, two missing alerts matching e_1 and e_2 need to be hypothesized and then a_3 can be correlated to a_0 (through one of the hypothesized alerts). For the second case, no alert needs to be hypothesized because the sequence is already consistent, and a_3 needs to be correlated to a_1. For the third case, a_0 needs to be correlated to a_1, and it also needs to be correlated to a_0 through a hypothesized alert matching e_2.

We extend the basic QG approach described in Section 4 by modifying its correlation sub-procedure. The new procedure, as shown in Fig. 6, correlates and hypothesizes alerts as follows. Given a queue graph Q_g with n queues $\mathcal{Q}$ and m variables $\mathcal{V}$. Each variable in $\mathcal{V}$ can now have one of the three values *TRUE*, *FALSE*, and *HYP*, together with a timestamp; those denote a satisfied security condition, an unsatisfied one, a hypothetically satisfied one, and the time of the last update, respectively. Each queue in $\mathcal{Q}$ can contain alerts or hypothesized alerts. The result graph $G_r(V, E_1 \cup E_r)$ is similar to that described in Section 4.2. However, the vertex set V now includes not only alerts but also hypothesized alerts and security conditions.

Suppose a new alert a_{new} with the timestamp t_{new} is received and enqueued in the queue $Q_i (1 \leqslant i \leqslant n)$. First, we start from Q_i and follow the pointers in $\mathcal{PR}_i$ to set each variable $v_j (1 \leqslant j \leqslant m)$ adjacent to Q_i with the value TRUE and the timestamp t_{new}. This step records the security conditions satisfied by a_{new}. Second, we start from Q_i and make a partial BFS by following the pointers in $\mathcal{P}_i$. The BFS is partial, because it stops upon leaving[7] a variable with the value *TRUE* or the value *HYP* (or a queue that contains a hypothesized alert). This step correlates a_{new} to previously received or hypothesized alerts.

The result graph G_r is updated during the above process as follows. First, after we enqueue a_{new} into Q_i and make changes to each v_j adjacent to Q_i, we add a_{new} and v_j (that is, the value and timestamp of v_j) as vertices, and an edge from a_{new} pointing to v_j into the result graph G_r. This step records the fact that the new alert a_{new} satisfies its implied security conditions at time t_{new}. Second, during the partial BFS, we record each hypothesis. Whenever we change the value of a variable v_j from *FALSE* to *HYP*, we record this update in G_r; similarly, whenever we enqueue a hypothesized alert into an empty queue, we record this hypothesized alert in G_r. Third, whenever we leave a variable v and reach a queue Q, we insert into G_r a directed edge from each queue Q to v; similarly, we insert edges from a queue to its connected variables when we leave the queue.

Example 5.4. Consider the left-hand side case of Fig. 7. The first alert a_0 will only cause (the variable corresponding to) the security condition c_2 to be changed from *FALSE* to *TRUE*. The result graph will be updated with the alert a_0 and satisfied security condition c_2 and the directed edge connecting them. When a_3 is received, a search starts from (the queue corresponding to) e_3; it changes c_3 from *FALSE* to *HYP*; it inserts a hypothesized alert a_1 into e_1 and a_2 into e_2, respectively; it stops at c_1 (which is initially set as *TRUE*) and c_2 (which has been set as *TRUE* when a_0 arrived). The result graph will be updated with the alert a_3, the hypothesized alerts a_1 and a_2, the hypothetically satisfied security condition c_3, and the directed edges between them.

5.2.1. Complexity analysis

At first glance, the procedure described above takes quadratic time, because a BFS takes time linear in the number of vertices $(n + m)$ and edges $(n + m)^2$, where n and m is the number of exploits and security conditions in the attack graph, respectively. However, this is not the case. As described in Section 4.2, a queue graph organizes its pointers in separate layers, and each layer is a BFS tree rooted at a queue. Hence, a BFS that starts from a queue and follows the pointers in the corresponding layer will be equivalent to a tree traversal, which takes linear time $(n + m)$. This performance gain seems to be obtained at the price of more memory requirement, because a pointer may appear in more than one layer. However, as described in Section 4.2, the memory requirement is quadratic (that is, $O(n(n + m))$), which is indeed asymptotically the same as that of the original attack graph.

5.3. Attack prediction

In the previous section, we explain the occurrence of a new alert by searching backwards (that is, in the reversed direction of the edges in attack graphs) for correlated (or hypothesized) alerts. Conversely, we can also predict possible consequences of each new alert by searching forwards. A BFS is also preferred in this case, because the predicted security conditions will be discovered in the order of their (shortest) distances to the new alert. This distance roughly indicates how imminent a predicted attack is, based on the alerts received so far (although not pursued in this paper, probability-based prediction techniques, such as [30], can be easily incorporated into the QG data structure).

The procedure of prediction is similar to that of correlation and hypothesis discussed in the previous section, and hence is omitted. The main differences between the two procedures are as follows. After the correlation and hypothesis completes, the prediction starts. It begins at the security conditions satisfied by the new alert and makes a partial BFS in the queue graph by following the pointers

[7] Given that a BFS is implemented through manipulating a separate queue as usual, we shall refer to the enqueues as *reaching* and the dequeues as *leaving* to avoid confusions.

in $\mathscr{PR}_i$ (suppose the new alert is enqueued by Q_i). The search stops at previously received (or hypothesized) alerts and their (hypothetically) satisfied security conditions to avoid repeating the previous prediction.

The result of the prediction process is a sequence of non-empty sets Con_1, $\mathrm{Con}_2, \ldots$, with $\mathrm{Con}_i (1 \leqslant i \leqslant m)$ containing the security conditions that can possibly be satisfied in i steps from now. Unlike in correlation and hypothesis, the prediction process does not reason about the disjunctive and conjunctive relationship between exploits as discussed in Section 5.1. Instead, a security condition c will appear in the set Con_i as long as there exists a path of length $2i$ (the path consists of both security conditions and exploits) from c to some previously satisfied security condition. Hence, the number i provides a lower bound to the number of exploits that must be executed before c can be satisfied.

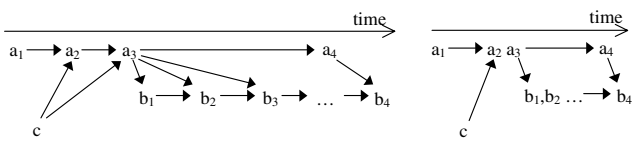
6. Compressing result graphs

In this section, we propose a compact representation for the correlation result graph. First, Section 6.1 shows that the size of a result graph can be reduced without losing information by removing transitive edges and aggregating *indistinguishable* alerts. Then, Section 6.2 explains how the QG approach can produce the compact result graph, and how this will improve the performance.

6.1. A compact representation of result graphs

Avoiding unnecessary searches enables the QG approach to have a constant performance regardless of the number of received alerts. As a side-effect, the approach also reduces the size of result graphs by having less transitive edges. However, the QG approach does not completely remove transitive edges from result graphs, as illustrated in Example 6.1. In practice, brute force attempts of the same attack with different parameters usually lead to a large number of alerts in a short time (the treasure hunt data used in Section 7 is a good example for such brute force attempts). In Example 6.1, if the b_is happen to be such an attack, then a large number of transitive edges will make the result graph less perceptible. It is thus desired to remove such transitive edges.

Example 6.1. The left side of Fig. 8 shows the result graph of correlating a series of alerts using the QG approach. It can be observed that transitive edges, such as (a_1, b_1) and (a_2, b_1), are not present. Because when b_1 arrives, the QG approach will search backwards and inserts the edge (a_3, b_1) into the result graph, and then it immediately stops (on the other hand, the nested loop approach will continue to search in this case). However, the edges (a_3, b_2) and (a_3, b_3) are both transitive edges. When b_2 and b_3 arrives, the QG approach repeats the same search as it does for b_1 and thus the two transitive edges are inserted into the result graph. Similarly, the edge (c, a_3) is also transitive.



$\mathrm{Exp}(a_i) = \mathrm{Exp}(a_j)$, $\mathrm{Exp}(b_i) = \mathrm{Exp}(b_j)$, and $\mathrm{Exp}(c_i) = \mathrm{Exp}(c_j)$ for $1 \leq i, j \leq 4$
$\mathrm{Exp}(a_i) \rightarrow \mathrm{Exp}(b_i)$ and $\mathrm{Exp}(c_i) \rightarrow \mathrm{Exp}(b_i)$

Fig. 8. An example of compressing result graphs.

In addition to transitive edges, some alerts carry redundant information as well. Following the above example, b_1, b_2, and b_3 are *indistinguishable* in terms of alert correlation. That is, any other alert prepares for (or be prepared for by) either all or none of them. The three alerts can thus be *aggregated* as a single vertex in the result graph, with the edges connecting those alerts deleted. Similarly, a_2 and a_3 are also indistinguishable. However, a_1, a_2, a_3 are not indistinguishable, because c prepares for a_2 and a_3 but not a_1. The right side of Fig. 8 shows the compact version of result graph, with transitive edges deleted and indistinguishable alerts aggregated.

6.2. Modifying the QG approach to produce compact result graphs

Existing approaches usually take extra efforts in making the result graph more compact, such as aggregating alerts before correlating them [20]. The additional step increases the performance overhead of alert correlation. We show that our QG approach can be modified to directly produce a compact result graph. We also show that the modified QG approach is actually more efficient.

We first modify the QG approach to avoid inserting transitive edges into the result graph. We let each backward pointer in a queue graph to have one of the two states, *on* and *off*. Initially, all the backward pointers are on. The backward pointers are then switched between the two states as follows. Whenever a directed edge (a_i, a_j) is inserted into E_r, we turn off the backward edges between the corresponding queues Q_i and Q_j. Whenever an alert is enqueued in a queue Q_i, all the backward pointers arriving at Q_i will be turned on. Finally, when we search for older alerts that prepare for a new alert, we follow a backward edge only if it is currently turned on. This process is illustrated in Example 6.2.

Example 6.2. For example, in the left side of Fig. 8 suppose the alerts a_i, b_i, c correspond to the queues Q_a, Q_b, and Q_c, respectively. When the alert b_1 arrives, it searches through the backward pointers from Q_b to Q_a and inserts an edge (a_3, b_1) into E_r. Then according to the above discussion, the backward pointers from Q_b to Q_a will be turned off. Consequently, the alerts b_2 and b_3 will not follow those pointers, and the transitive edges (a_3, b_2) and (a_3, b_3) are avoided. This remains true until the alert a_4 arrives, which turns on all the backward pointers arriving at the queue Q_a. Then later when b_4 arrives, it follows the

backward pointers from Q_b to Q_a and inserts the edge (a_4, b_4).

Alerts are aggregated during the above process as follows. Suppose an alert a_i arrives and the corresponding queue Q_i already contains another alert a_i'. Then a_i is aggregated with a_i' if the following two conditions are true. First, all the backward pointers arriving at Q_i are on. Second, all the backward pointers leaving Q_i are off. The first condition ensures that a_i' does not prepare for any other alerts that arrive between a_i' and a_i, because otherwise a_i and a_i' would not be indistinguishable. The second condition ensures that a_i' and a_i are prepared for by the same collection of alerts, so they are indistinguishable with respect to those alerts. This process is illustrated in Example 6.3.

Example 6.3. Following the above example, a_3 is aggregated with a_2 because the backward pointers from Q_b to Q_a are on and those from Q_a to Q_c have been turned off by the alert a_2. Similarly, b_2 and b_3 are aggregated with b_1, because the backward pointers from Q_b to Q_a have been turned off by b_1. On the other hand, the alert b_4 will not be aggregated, because the backward pointers from Q_b to Q_a must have been turned on by a_4 by the time b_4 arrives.

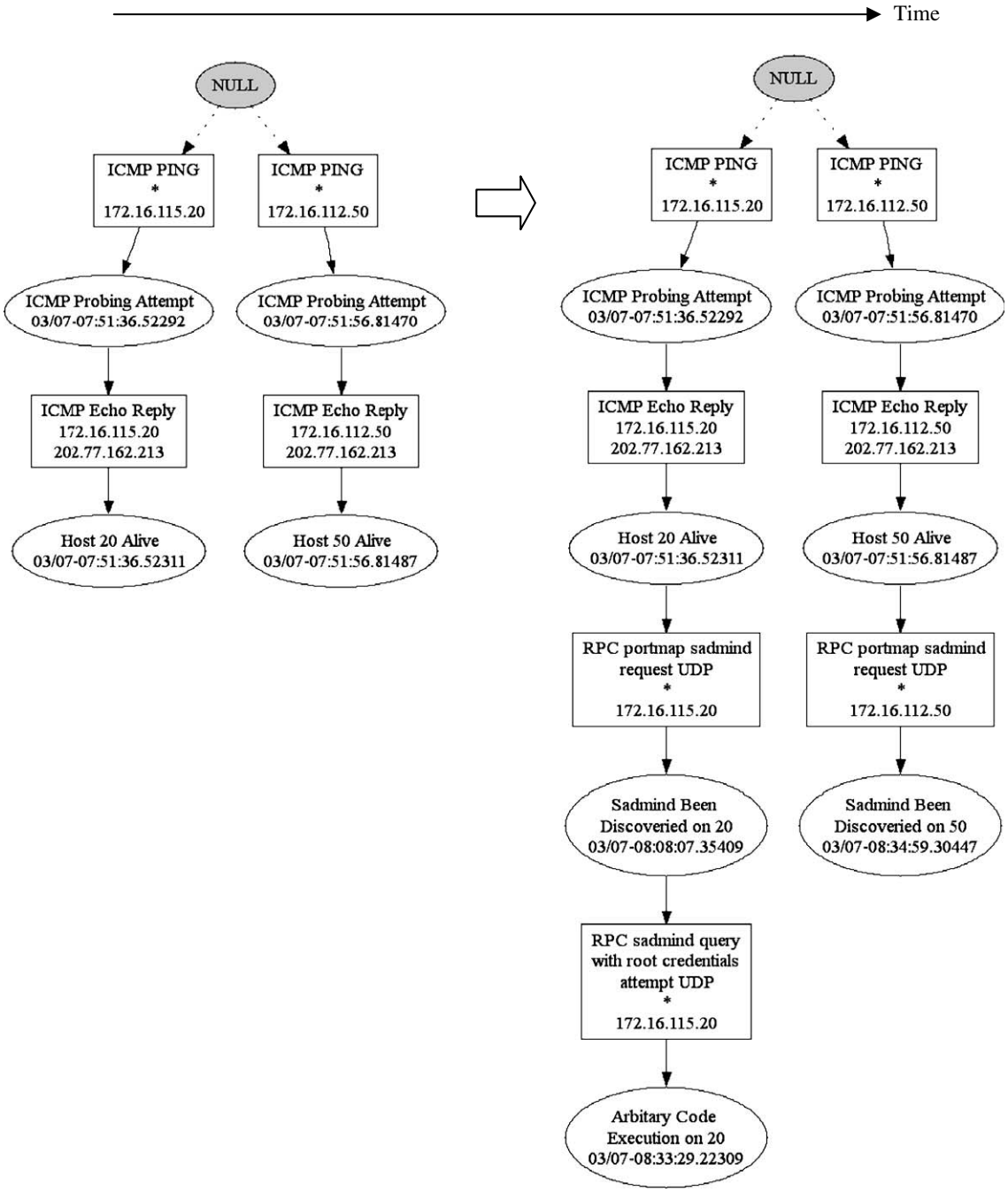


Fig. 9. The evolving result graphs of alert correlation.

The modified QG approach not only produces a more compact result graph, but is also more efficient than the original one in most cases. This is because unnecessary searches corresponding to transitive edges are avoided. In Fig. 8, the alerts a_3, b_2, and b_3 will not lead to a search in the modified approach because the backward pointers have been turned off by earlier alerts. The performance gain can be significant in the case of brute force attempts where a large number of searches can be avoided.

7. Empirical results

In this section, we evaluate the proposed techniques through implementation and empirical results. The correlation engine is implemented in C++ and tested on a Pentium III 860 MHz server with 1 GB RAM running RedHat Linux. We use Snort-2.3.0 [35] to generate isolated alerts, which are directly pipelined into the correlation engine for analyses. We use Tcpreplay 2.3.2 [40] to replay network traffic from a separate machine to the server running the correlation engine.

Two data sets are used for experiments, the Darpa 2000 intrusion detection LLDOS 1.0 by MIT Lincoln Labs [8], and the treasure hunt dataset by the University of California, Santa Barbara [39]. The attack scenario in the Darpa 2000 dataset has been extensively explored before (such as in [20]). Our experiments with the dataset show similar results, validating the correctness of our correlation algorithm. The treasure hunt dataset generates a large amount of alerts (about two million alerts taking about 1.4 GB of disk space, with most of them being brute force attempts of the same attacks), which may render a nested loop-based correlation method infeasible (we found that even running a simple database query over the data will paralyze the system). In contrast, our correlation engine processes alerts with negligible delays (Snort turns out to be the bottleneck).

7.1. Effectiveness

The objective of the first set of experiments is to demonstrate the effectiveness of the proposed algorithms in alert correlation, hypothesis, and prediction. We use the Darpa 2000 dataset for this purpose. The reason we use this dataset is that it has well known attack scenarios, which can be referenced in the included description or previous work, such as [20]. For correlation without hypothesis and prediction, we expect our method to produce exactly the same result as previous work do, with the redundant transitive edges removed in the result graph (given that the domain knowledge encoded in our attack graph exactly matches that used by previous work). Notice that the key contribution of this work is to improve the performance of previous approach and make them immune to slowed attacks. The

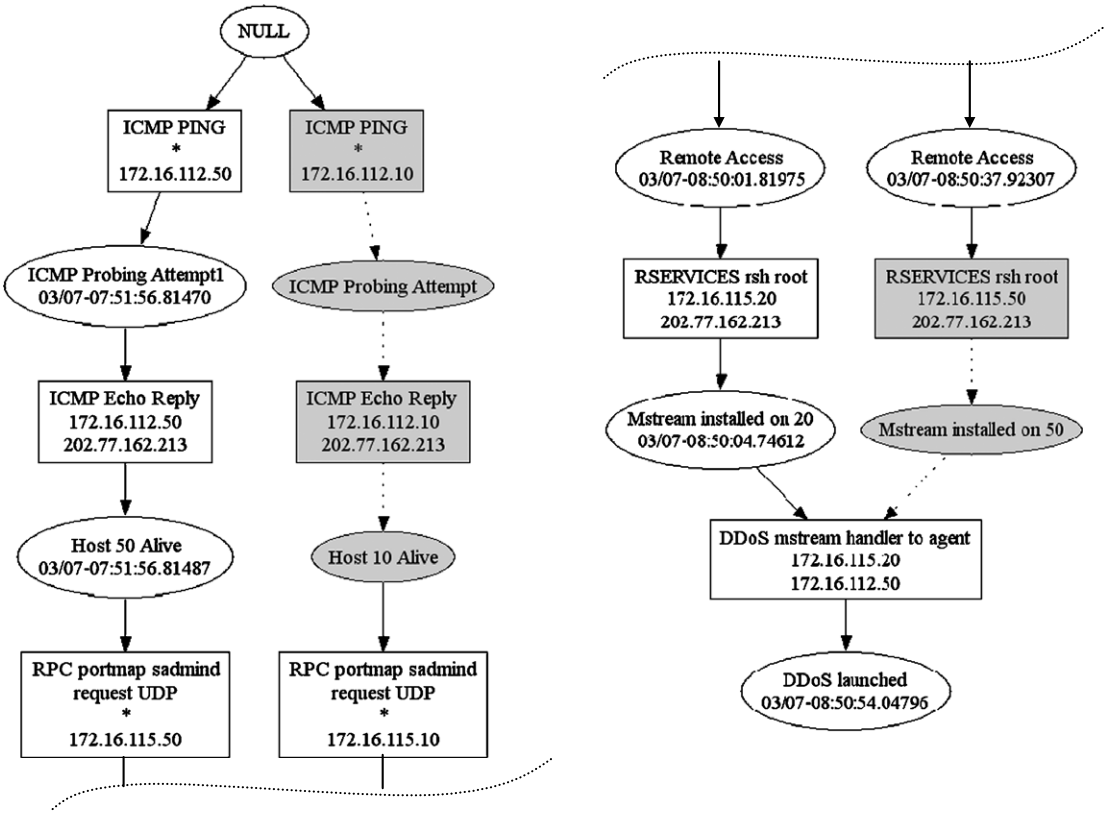


Fig. 10. The hypothesis of missing alerts during correlation.

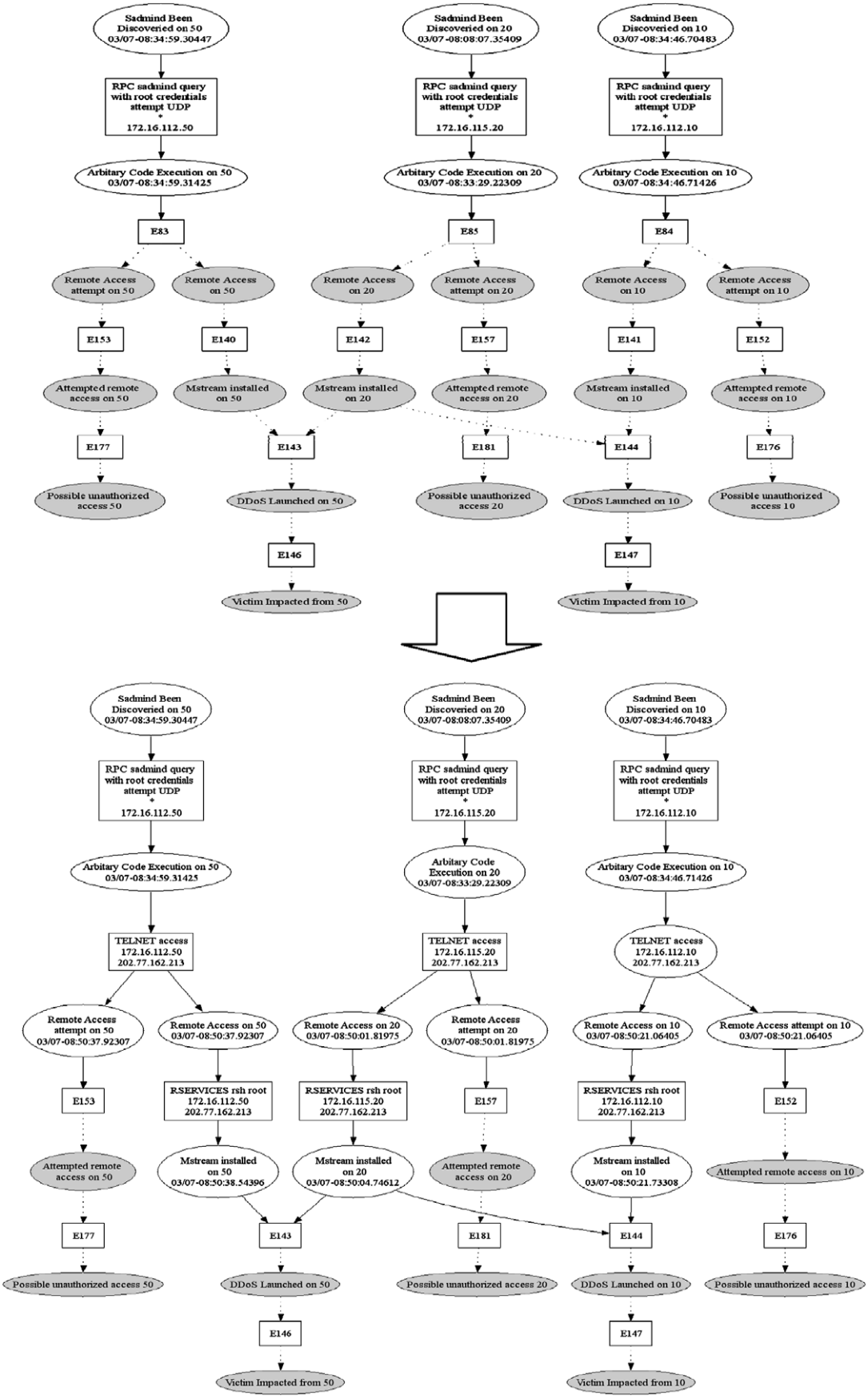


Fig. 11. The prediction of possible consequences during correlation.

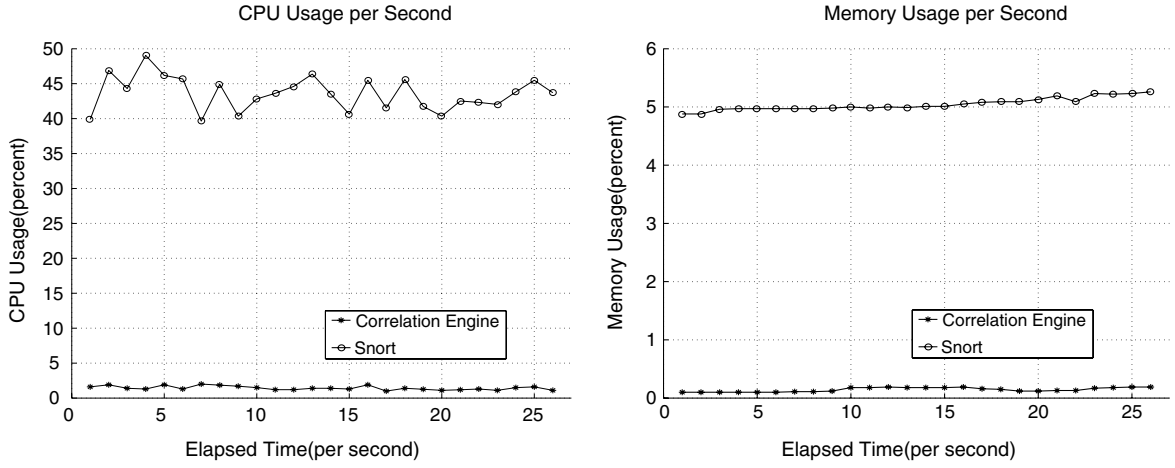
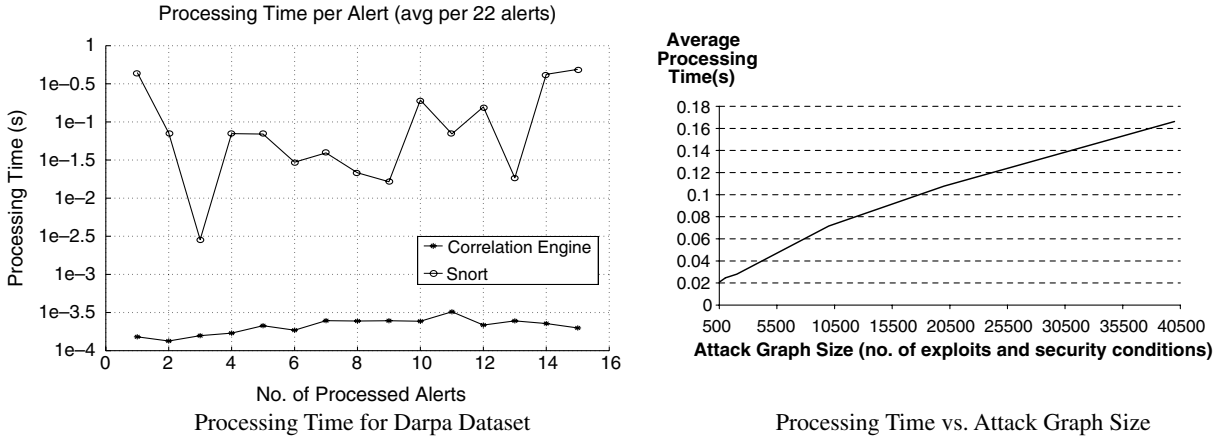


Fig. 12. The CPU and memory usage.



Processing Time for Darpa Dataset

Processing Time vs. Attack Graph Size

Fig. 13. The processing time and its relationship with the size of attack graph.

correlation methodology itself is not different from that found in previous work, and similarly the accuracy of the correlation result also depends on the domain knowledge used for correlation. However, in contrast to the static result graph in previous work, our result *evolves* in time with the continuously arriving alerts, as illustrated in Fig. 9 (due to space limitations, only two partial snapshots of the result graphs are shown). Such a result can more clearly reveal the actual progress of an intrusion.

Fig. 10 shows two results on hypothesizing missing alerts during the correlation. On the left-side of the figure, two consecutive missing alerts (ICMP PING and ICMP Echo Reply) and the corresponding security conditions are hypothesized (shown as shaded) when an alert (RPC portmap sadmind request UDP) is received but its required security condition (Host 10 Alive) has not been satisfied. The right-hand side of the figure shows a conjunctive relationship between alerts, that is a DDoS mstream traffic between two hosts requires the mstream software to be installed on both hosts. We deliberately deleted the RSERVICES rsh alert on one of the host, which is successfully hypothesized (shown as shaded).

Fig. 11 shows a result of alert prediction. In the left figure, some security conditions are predicted to be satisfied by possible upcoming alerts. The predicted security conditions are shown as shaded, and the numbers are placeholders for alerts. The right-hand side figure shows a later snapshot of the result graph, in which some of the predicted security conditions are indeed realized. Notice that here the attack graph exactly (and only) captures the necessary domain knowledge, and hence the prediction result is highly accurate. In practice, both false positives (predicted but not realized) and false negatives (realized but not predicted) may be introduced because of incomplete or inaccurate domain knowledge. Refining our prediction method to reduce such inaccuracy comprises an interesting future direction.

7.2. Performance

The objective of the second set of experiments is to evaluate the performance of the correlation engine. The performance metric includes the resource usage (CPU and memory) and the processing time of each alert. The correlation engine measures its own processing time and compares

the processing time to the delay between receiving two consecutive alerts from Snort. All the results have 95% confidence intervals within about 5% of the reported values. Fig. 12 shows the CPU usage (on the left-hand side) and memory usage (on the right-hand side) over time for the Darpa data set. The correlation engine clearly demands less resources than Snort (on average, the correlation engine's CPU usage and memory usage are both under 10% of Snort's).

The left chart in Fig. 13 shows the processing time per alert (averaged per 22 alerts). Clearly, the correlation engine works faster than Snort in processing the entire data set. The result also proves that the performance does not decrease over time. Indeed, the processing time per alert remains fairly steady. We examine the scalability of the correlation engine in terms of the number of exploits and security conditions. The treasure hunt data set is used for this purpose. The original attack graph only has about one hundred exploits. We increase the size of attack graphs by randomly inserting dummy exploits and security conditions. The inserted exploits increase the complexity of correlation because the correlation engine must search through them. The right chart in Fig. 13 shows that the average processing time scales with the size of attack graphs as expected.

We replay network traffic at relatively high speed (for example, the Darpa data set is replayed in about 26 s while the actual duration of the dataset is several hours). Real-world attacks are usually less intensive, and consequently our correlation engine will exhibit a better performance. However, we are aware that real-world traffic may bring up new challenges that are absent in synthesized data sets. For example, we currently set the time window used to reorder alerts (that is, $t_{\max}$ as discussed in Section 4.3) as one second to deal with identical time stamps of alerts. In a real network, the windows size must be decided based on the actual placement of IDS sensors and the typical network delays. In our future work, we plan to integrate our correlation engine in our TVA tool and test it in real-world network settings.

8. Conclusion

In this paper, we studied the real-time correlation of intrusion alerts. We identified limitations in applying the nested loop-based correlation methods and proposed a novel QG approach to remove this limitation. The method has a linear time complexity and a quadratic memory requirement and can correlate alerts arbitrarily far away. We extend the QG approach to a unified method for the correlation, hypothesis, and prediction of alerts. We also extend the method to produce a compact version of result graphs. Empirical results showed that our correlation engine can process alerts faster than an IDS can report them, making our method a promising solution for an administrator to monitor the progress of intrusions. Our future work includes evaluating the techniques with real-world traffic in live networks.

Acknowledgments

The authors thank the anonymous reviewers for their valuable comments, and Giovanni Vigna for providing the Treasure Hunt dataset. This material is based upon work supported by HSARPA under the contract FA8750-05-C-0212 administered by the Air Force Research Laboratory, by Army Research Office under Grants DAAD19-03-1-0257 and W911NF-05-1-0374, by Federal Aviation Administration under the contract DTFAWA-04-P-00278/0001, and by the National Science Foundation under grants IIS-0242237 and IIS-0430402. 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] P. Ammann, D. Wijesekera, S. Kaushik, Scalable, graph-based network vulnerability analysis, in: Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS'02), 2002, pp. 217–224.
[2] F. Cuppens, Managing alerts in a multi-intrusion detection environment, in: Proceedings of the 17th Annual Computer Security Applications Conference (ACSAC'01), 2001.
[3] F. Cuppens, A. Miege, Alert correlation in a cooperative intrusion detection framework, in: Proceedings of the 2002 IEEE Symposium on Security and Privacy (S& P'02), 2002, pp. 187–200.
[4] F. Cuppens, R. Ortalo, LAMBDA: a language to model a database for detection of attacks, in: Proceedings of the 3rd International Symposium on Recent Advances in Intrusion Detection (RAID'01), 2001, pp. 197–216.
[5] M. Dacier, Towards quantitative evaluation of computer security, Ph.D. Thesis, Institut National Polytechnique de Toulouse, 1994.
[6] O. Dain, R.K. Cunningham, Building scenarios from a heterogeneous alert system, in: Proceedings of the 2001 IEEE Workshop on Information Assurance and Security, 2001.
[7] O. Dain, R.K. Cunningham, Fusing a heterogeneous alert stream into scenarios, in: Proceedings of the ACM Workshop on Data Mining for Security Applications, 2001, pp. 1–13.
[8] 2000 darpa intrusion detection evaluation datasets. http://www.ll.mit.edu/IST/ideval/data/2000/2000_data_index.html, 2000.
[9] H. Debar, A. Wespi, Aggregation and correlation of intrusion-detection alerts, in: Proceedings of the 3rd International Symposium on Recent Advances in Intrusion Detection (RAID'01), 2001, pp. 85–103.
[10] S.T. Eckmann, G. Vigna, R.A. Kemmerer, STATL: an attack language for state-based intrusion detection, Journal of Computer Security 1 (1/2) (2002) 71–104.
[11] D. Farmer, E.H. Spafford, The COPS security checker system, in: USENIX Summer, 1990, pp. 165–170.
[12] N. Habra, B.L. Charlier, A. Mounji, I. Mathieu, ASAX: software architechture and rule-based language for universal audit trail analysis, in: Proceedings of the 2nd European Symposium on Research in Computer Security (ESORICS 1992), 2004, pp. 430–450.
[13] IBM, IBM tivoli risk manager, Available from: <http://www.ibm.-com/software/tivoli/products/risk-mgr/>.
[14] SRI International, Event monitoring enabling responses to anomalous live disturbances (EMERALD). Available from: <http://www.sdl.sri.com/projects/emerald/>.
[15] S. Jajodia, S. Noel, B. O'Berry, Topological analysis of network attack vulnerability, in: V. Kumar, J. Srivastava, A. Lazarevic (Eds.), Managing Cyber Threats: Issues Approaches and Challenges, Kluwer Academic Publisher, 2003.

[16] S. Jha, O. Sheyner, J.M. Wing, Two formal analysis of attack graph, in: Proceedings of the 15th Computer Security Foundation Workshop (CSFW'02), 2002.
[17] Klaus Julisch, Marc Dacier, Mining intrusion detection alarms for actionable knowledge, in: Proceedings of the 8th ACM SIGKDD international conference on Knowledge discovery and data mining, 2002, pp. 366–375.
[18] W. Lee, J.B.D. Cabrera, A. Thomas, N. Balwalli, S. Saluja, Y. Zhang, Performance adaptation in real-time intrusion detection systems, in: Proceedings of The 5th International Symposium on Recent Advances in Intrusion Detection (RAID 2002), 2002.
[19] B. Morin, L. Mé, H. Debar, M. Ducassé, M2D2: a formal data model for IDS alert correlation, in: Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection (RAID'02), 2002, pp. 115–137.
[20] P. Ning, Y. Cui, 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), 2002, pp. 245–254.
[21] P. Ning, D. Xu, Adapting query optimization techniques for efficient intrusion alert correlation, Technical report, NCSU, Department of Computer Science, 2002.
[22] P. Ning, D. Xu, Learning attack strategies from intrusion alerts, in: Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS'03), 2003.
[23] P. Ning, D. Xu, C.G. Healey, R.S. Amant, Building attack scenarios through integration of complementary alert correlation methods, in: Proceedings of the 11th Annual Network and Distributed System Security Symposium (NDSS'04), 2004, pp. 97–111.
[24] S. Noel, 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), 2004.
[25] S. Noel, S. Jajodia, B. O'Berry, M. Jacobs, Efficient minimum-cost network hardening via exploit dependency graphs, in: Proceedings of the 19th Annual Computer Security Applications Conference (ACSAC'03), 2003.
[26] R. Ortalo, Y. Deswarte, M. Kaaniche, Experimenting with quantitative evaluation tools for monitoring operational security, IEEE Trans. Software Eng. 2 (5) (1999) 633–650.
[27] OSSIM, Open source security information management, Available from: <http://www.ossim.net/>.
[28] V. Paxson, Bro: a system for detecting network intruders in real-time, Comput. Networks 3 (23-24) (1999) 2435–2463, 12.
[29] X. Qin, W. Lee, Statistical causality analysis of INFOSEC alert data, in: Proceedings of the 6th International Symposium on Recent Advances in Intrusion Detection (RAID 2003), 2003, pp. 591–627.
[30] X. Qin, W. Lee, Discovering novel attack strategies from INFOSEC alerts, in: Proceedings of the 9th European Symposium on Research in Computer Security (ESORICS 2004), 2004, pp. 439–456.
[31] A.R. Chinchani andIyer, H. Ngo, S. Upadhyay, Towards a theory of insider threat assessment, in: Proceedings of the IEEE International Conference on Dependable Systems and Networks (DSN'05), 2005.
[32] I. Ray, N. Poolsappasit, Using attack trees to identify malicious attacks from authorized insiders, in: Proceedings of the 10th European Symposium on Research in Computer Security (ESORICS'05), 2005.
[33] R. Ritchey, 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), 2000, pp. 156–165.
[34] R. Ritchey, B. O'Berry, S. Noel, Representing TCP/IP connectivity for topological analysis of network security, in: Proceedings of the 18th Annual Computer Security Applications Conference (ACSAC'02), 2002, p. 25.
[35] M. Roesch, Snort – lightweight intrusion detection for networks, in: Proceedings of the 1999 USENIX LISA Conference, 1999, pp. 229–238.
[36] O. Sheyner, J. Haines, S. Jha, R. Lippmann, J.M. Wing, Automated generation and analysis of attack graphs, in: Proceedings of the 2002 IEEE Symposium on Security and Privacy (S& P'02), 2002, pp. 273–284.
[37] S. Staniford, J.A. Hoagland, J.M. McAlerney, Practical automated detection of stealthy portscans, J. Comput. Security 1 (1/2) (2002) 105–136.
[38] S. Templeton, K. Levitt, A requires/provides model for computer attacks, in: Proceedings of the 2000 New Security Paradigms Workshop (NSPW'00), 2000, pp. 31–38.
[39] Treasure hunt datasets, <http://www.cs.ucsb.edu/~vigna/treasure-hunt/index.html/>, 2004.
[40] A. Turner, Tcpreplay: Pcap editing and replay tools for *nix, Available from: <http://tcpreplay.sourceforge.net/>.
[41] A. Valdes, K. Skinner, Probabilistic alert correlation, in: Proceedings of the 4th International Symposium on Recent Advances in Intrusion Detection, 2001, pp. 54–68.
[42] L. Wang, A. Liu, 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), 2005, pp. 247–266.
[43] D. Xu, P. Ning, Alert correlation through triggering events and common resources, in: Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC'04), 2004, pp. 360–369.
[44] D. Xu, P. Ning, Privacy-preserving alert correlation: a concept hierarchy based approach, in: Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC'05), 2005.
[45] D. Zerkle, K. Levitt, Netkuang – a multi-host configuration vulnerability checker, in: Proceedings of the 6th USENIX Unix Security Symposium (USENIX'96), 1996.
[46] Y. Zhai, P. Ning, P. Iyer, D. Reeves, Reasoning about complementary intrusion evidence, in: Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC'04), 2004, pp. 39–48.

Lingyu Wang is a Ph.D. candidate in Information Technology at George Mason University where he is being advised by Sushil Jajodia. Since 2000, he has been a Research Assistant inthe Center for Secure Information Systems (CSIS). His main research interests are information security and privacy. He holds a M.E. from Shanghai Jiao Tong University anda B.E. from Shen Yang Institute of Aeronautic Engineering in China.

Anyi Liu received his B.S. and M.S. degree in Computer Science from Dalian University of Technology in 1997 and 2001, respectively. He is currently a Ph.D. student in Information Technology at George Mason University. His research interests are information security/assurance, intrusion detection and correlation. He is a student member of ACM SIGSAC.

Sushil Jajodia is BDM International Professor of Information Technology and the director of Center for Secure Information Systems at the George Mason University, Fairfax, Virginia. His research interests include information security, temporal databases, and replicated databases. He has authored five books, edited twenty-four books, and published more than 300 technical papers in the refereed journals and conference proceedings. He is the founding editor-in-chief of the Journal of Computer Security and on the editorial boards of ACM Transactions on Information and Systems Security, IEE Proceedings on Information Security, International Journal of Cooperative Information Systems, and International Journal of Information and Computer Security.