An Efficient and Unified Approach to Correlating, Hypothesizing, and Predicting Intrusion Alerts*

Lingyu Wang, Anyi Liu, and Sushil Jajodia

Center for Secure Information Systems,
George Mason University,
Fairfax, VA 22030-4444, USA
{lwang3, aliu1, jajodia}@gmu.edu

Abstract. To defend against a multi-step network intrusion, its progress needs to be monitored and predicted in real-time. For this purpose, isolated alerts must be correlated into attack scenarios as soon as the alerts arrive. Such efficient correlation of alerts demands an in-memory index to be built on received alerts. However, the finite memory implies that only a limited number of alerts inside a sliding window can be considered for correlation. 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 invoking 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 explicitly correlating a new alert to *all* the old ones that prepare for it, the approach only correlates the new alert to the latest copy of each type of alerts. The correlation with other alerts is kept implicit using the temporal order between alerts. Consequently, the approach has a quadratic (in the number of alert types) memory requirement, and it can correlate two alerts that are arbitrarily far away (namely, an infinitely large sliding window with a quadratic memory requirement). Our second contribution is a unified method based on the QG approach that can correlate received alerts, hypothesize missing alerts, and predict future alerts all at the same time. Empirical results show that our method can fulfill those tasks faster than an IDS can report alerts. The method is thus a promising solution for administrators to monitor and predict the progress of an intrusion, and thus to take appropriate countermeasures in a timely manner.

1 Introduction

For most well-administrated networks, a realistic intrusion is usually composed of multiple attacks with earlier ones preparing for later ones. Defending against such *multi-step intrusions* is important but challenging. It is usually impossible to respond to such intrusions based on isolated alerts that correspond 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.

* This work was partially supported by the National Science Foundation under grant CCR-0113515, by Air Force Research Laboratory, Rome under the contract F30602-00-2-0512, and by Army Research Office under the grant DAAD19-03-1-0257.

S. De Capitani di Vimercati et al. (Eds.): ESORICS 2005, LNCS 3679, pp. 247–266, 2005.

The defense of multi-step intrusions will be more effective, if the attack scenarios of such intrusions can be reconstructed from isolated alerts. *Alert correlation* techniques achieve this [1] by exploiting either the similarity in alert attributes or the *a priori* knowledge about alert dependencies (related work will be reviewed in the next section). Alert correlation can also be based on the knowledge about a given network, such as network connectivity and the relationship between vulnerabilities. Regardless of the different knowledge used by correlation methods, the following *nested loop* procedure is usually assumed. That is, for each new alert, a search is performed in previously received alerts to find those who *prepare for* the new one. For off-line applications with a fixed set of alerts, such as computer forensics, this approach is a natural choice with reasonably good performance. For example, by maintaining an in-memory index on alerts, 65k alerts can be processed in less than a second [20].

However, the defense against multi-step intrusions poses a new challenge to existing correlation methods that are based on the nested loop approach. A timely defense requires that each new alert be correlated with older ones as soon as the new alert arrives. This performance requirement demands an in-memory index to be maintained on received alerts. An index on all received alerts would exhaust any finite memory when more and more alerts arrive. Hence, the index can only be maintained for those alerts that are close enough to the new alert, namely, those inside a *sliding window*. Unfortunately, an attacker aware of this fact can prevent any two attack steps from both falling into the sliding window. This can be achieved by either passively delaying the second step or actively invoking bogus alerts between the two steps. In either case, the correlation effort is completely defeated.

In this paper, we first remove the above obstacle towards efficient correlation of intrusion alerts. We propose a novel *queue graph* (QG) data structure for this purpose. The QG only keeps in memory the latest alert matching each of the known exploits (that is, host-bound vulnerabilities). The correlation is explicit only between the new alert and these in-memory alerts, while that between the new alert and other older alerts is kept implicit with the temporal order between alerts. We then study a QG-based correlation method that can not only correlate received alerts, but also hypothesize missing alerts and predict possible future alerts all at the same time. Finally, we evaluate the proposed techniques through implementations and empirical results.

The contribution of this work is two-fold. First, the QG-based alert correlation removes the limitation of a nested loop approach. Our approach has a quadratic memory requirement and a linear time complexity (in the number of known exploits in the given network) that are both independent of the number of received alerts. Hence, the efficiency does not decrease over time. Our approach can correlate alerts that are arbitrarily far away. It thus defeats slowed attacks and injected bogus attacks. Second, the unified approach to alert correlation, hypothesis, and prediction provides a promising solution to the defense of multi-step intrusions. Empirical results indicate that our methods can fulfill the tasks even faster than the IDS can report alerts. Hence, the proposed tech-

[1] There are alert correlation techniques used for other purposes, such as correlating multiple victims targeted by the same attacker, but we shall focus on the techniques used for analyzing multi-step intrusions.

niques can help an administrator to monitor and predict the progress of a multi-step intrusion, and thus to take appropriate countermeasures in a timely manner.

The rest of this paper is organized as follows. The next section reviews related work. Section 3 introduces some basic concepts and states our assumptions. Section 4 proposes the QG approach to alert correlation. Section 5 studies a unified method for alert correlation, hypothesis, and prediction. Section 6 evaluates the proposed techniques with implementation and empirical results. Finally, Section 7 concludes the paper and gives future directions.

2 Related Work

To reconstruct attack scenarios from isolated alerts, some alert-based correlation techniques employ the a priori knowledge about known attack strategies [6,8,4,9,35] or alert dependencies [3,19,21]. Other techniques do not depend on such knowledge, but cluster alerts through the similarity in their attributes (such as same sources and destinations) [2,5,34,38] or statistical and temporal patterns [16,28]. Hybrid approaches combine different techniques to obtain better results [21,29]. Alert correlation techniques have also been used for other purposes than the analysis of multi-step intrusions, such as to relate alerts to the same attack thread [13]. In real-time applications, the correlation methods based on a nested loop approach either suffer from performance decreases over time or can be easily defeated by slowed attacks and injected bogus attacks. To our best knowledge, this has not been extensively studied. Our work addresses this important issue and provides a solution.

Network vulnerability analyses enumerate potential attack sequences between fixed initial conditions and attack goals [25,30,31,33,15,1,14,24,40,10]. To avoid potential combinatorial explosion in the number of attack sequences, we adopt a notation of attack graphs similar to that of [1,25]. However, we do not assume fixed initial or goal conditions in an attack graph but base the actual start and end of an intrusion on alerts. Efforts in integrating information from different sources include *M2D2*, a formal model of alerts, vulnerabilities, networks, and security tools [18]. By organizing IDS alerts and the reports of vulnerability scanners (or other monitoring tools such as anti-virus software) into a Bayesian network, the alerts corresponding to successful attacks can be distinguished from others with higher confidence [41]. In another recent approach, alert correlation is based on the shortest distance between exploits in an attack graph [23]. We also adopt such a vulnerability-centric approach, because it can effectively filter out bogus alerts that do not match any exploit in the given network. However, the correlation in [23] still assumes a nested loop approach, and hence has the same limitation in real-time applications.

Broken scenarios caused by missing alerts are reassembled through clustering alerts with similar attributes [22], and those caused by incomplete knowledge are pieced together through statistical analyses [29,28]. Instead of repairing a broken scenario afterwards, our method can tolerate and hypothesize missing alerts at the same time of correlation. This unified approach makes our method more appropriate for real-time applications. Real-Time detection of isolated alerts is studied in [17,27]. Some products claim to have the capability of real-time analyses of alerts, such as the Tivoli Risk Man-

ager [12], although their efficiency and resistance to slowed attacks may need further study. The RUSSEL language used in the ASAX system is similar to our QG approach in that the analysis of data only requires one-pass of data processing, although the RUSSEL language is designed for the generic analysis of audit trails [11].

3 Preliminaries

This section reviews relevant concepts and states our notations and assumptions about those concepts. First, we discuss *attack graph* in Section 3.1. We then address intrusion alerts and alert correlation in Section 3.2. Finally, we address the *nested loop* approach and its limitations in Section 3.3.

3.1 Attack Graph

An *attack graph* represents the *a priori* knowledge about a given network in terms of vulnerabilities and connectivity [1,33]. 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 is a triple $(vul, src, dest)$ that indicates the following facts. 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, and those exploitations that involve more than two hosts are beyond the scope of this paper). Security conditions refer to the network states that are 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 1. Figure 1 depicts part of an attack graph. In the attack graph, security conditions appear as ovals and exploits as rectangles. The edges in the attack graph reflects that the buffer overflow exploit can be executed only if the attacker can access the source host and the vulnerable service exists on the destination host.

We assume attack graphs can be obtained by analyzing the given network with existing tools. For example, the Topological Vulnerability Analysis (TVA) tool reported

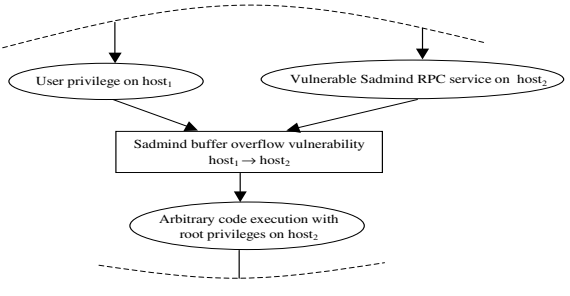


Fig. 1. An Example of Attack Graph

in [14] can model 37,000 vulnerabilities taken from 24 information sources including X-Force, Bugtraq, CVE, CERT, Nessus, and Snort. We assume the attack graph is updated in a timely fashion upon changes in network topology and configuration. We assume that the attack graph of a given network can be placed in memory. Unlike the number of alerts which may increase indefinitely over time, the size of an attack graph is usually relatively stable. The required memory can thus be predetermined and allocated accordingly. We leave the case when the attack graph does not fit in memory as future work. Different from the attack graph in [33,30], we do not assume fixed initial or goal conditions in an attack graph. Instead, the actual start and end of an intrusion are based on alerts reported by IDSs. We do not assume external host addresses can be trusted and thus our attack graphs use wildcards for external addresses. 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 exploits discovered in a subject network, and C be the set of relevant security conditions. Denote the *require* and *imply* relationship between exploits and security conditions as two relations $R_r \subseteq C \times E$ and $R_i \subseteq E \times C$, respectively. Then 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 [3,19], is simply the composite relation $R_e = R_i \circ R_r$.

3.2 Intrusion Alert

Intrusion alerts are suspicious events reported by IDS sensors placed in the given network. Although the alerts reported by different IDSs may vary in format, they 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, and hence we simply regard each alert as a relational tuple of relevant attributes. The schema of the relation will usually be clear from context. For example, with the schema *(event type, source IP, destination IP, time stamp)*, an alert will have the form of *(RPC portmap sadmind request UDP, 202.77.162.213, 172.16.115.20, 03/07-08:50:04.74612)*.

We adopt a vulnerability-centric approach to correlating alerts that is similar to [23]. Roughly speaking, the approach first matches alerts with corresponding exploits and then correlate alerts based on the knowledge encoded in an attack graph. The matching has two parts, that is the mapping from the event type attributes of alerts to the vulnerability attributes of exploits, and the comparison between the addresses of the source and destination hosts. The mapping from event types to vulnerabilities can be established using domain knowledge, such as the correspondence between Snort identifiers and Nessus identifiers available in OSSIM [26]. The comparison between host addresses supports using wildcards in exploits for untrustworthy external addresses. For simplicity, we denote the matching between alerts and exploits as a function f from the set of alerts A to the set of exploits E (more generally, an event type can match multiple vulnerabilities, and one way to handle this is to *duplicate* any alert of that even type such that each copy of the alert matches exactly one exploit).

Using the vulnerability-centric approach can potentially 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 signals the probing preparation for following attacks. Such relevant alerts can be identified based on attack graphs and the knowledge about alert types. We accommodate them by allowing exploits to have alert types in the place of vulnerability attributes. Such special *exploits* are inserted into attack graphs and the function f is extended accordingly.

Our methods critically depend on temporal characteristics of alerts, such as timestamps and the order of arrivals. In practice, those characteristics are expected to exhibit much uncertainty due to various delays in hosts and network, especially when alerts are from multiple sensors placed differently. 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) [32]. 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 previously received alerts for those who prepare for the new alert. Such a *nested loop* approach is assumed by many correlation methods. Suppose we have 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_j's that satisfy $f(a_j)R_e f(a_i)$. However, this does not imply that a_i must be compared to every $a_j (0 \leq j \leq i-1)$, although it comprises a naive implementation of the search. The search can certainly be optimized with standard indexing schemes. More specifically, an index on $a_0, a_1, \ldots, a_{i-1}$ is employed for searching the alerts that may prepare for a_i. After a_i is processed, the index needs to be updated by inserting an entry for a_i. By maintaining such an index in memory, the nested loop approach can have a relatively good performance (for example, 65k alerts can be processed in less than one second [20]).

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. 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. Apparently, an unavoidable tradeoff exists between the performance and completeness of correlation. On one hand, 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 this may cause incomplete result because two related alerts may actually be separated by more than k others.

The tradeoff between performance and completeness causes a more serious problem for real-time correlation, where performance is critical and alerts accumulate in time. The problem can be 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 discussed in the previous section. First, we make a key observation about implicit and explicit correlation in Section 4.1. We then introduce the QG data structure and discuss correlating alerts using QG in Section 4.2. Finally, we address the issue of imprecise temporal characteristics of alerts in Section 4.3.

4.1 Implicit Correlation and Explicit Correlation

The key observation is that the correlation between alerts does not always need to be explicit. In Figure 2, suppose the first three alerts a_i, a_j, and a_k all match the same exploit $f(a_k)$ (that is, their event types match the same vulnerability and the same source and destination hosts are involved); the alert a_h matches another exploit $f(a_h)$; $f(a_k)$ prepares for $f(a_h)$. 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 $f(a_j) = f(a_k)$ and $f(a_k)R_e f(a_h)$ jointly imply $f(a_j)R_e f(a_h)$, and the facts that a_j is before a_k and a_k is before a_h jointly imply that a_j must also be before a_h. Similar arguments apply to the correlation between a_i and a_h.

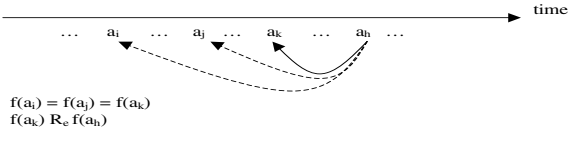


Fig. 2. Implicit and Explicit Correlation

To generalize the above observation, a new alert only needs to be explicitly correlated with the *latest* alert matching each exploit. The correlation with other older alerts matching the same exploit can be kept implicit with the temporal order (for example, a_j is before a_k and a_k is before a_h) and the matching from alerts to 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 $f(a_k)$, then only the correlation between a_h and a_k needs to be explicit [2].

[2] This is analogous to a moving-average or smoothing model, although what is concerned here is not the accumulated effect of one sequence, but the relationship between multiple sequences.

As we shall show shortly, this distinction between implicit and explicit correlation can reduce the complexity and memory requirement of correlation. 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 previous alerts, as discussed in Section 3.3.

4.2 Correlating Alerts Using Queue Graphs

Based on the observation about the implicit and explicit correlation, we design an in-memory data structure, namely, *Queue Graph*. 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 2.

Example 2. In Figure 3, 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

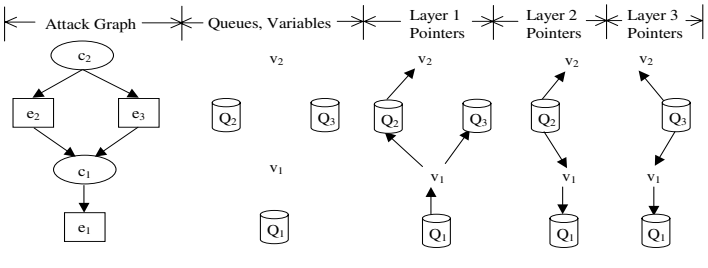


Fig. 3. An Example Queue Graph

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 to the new alert is added into the result graph. This is illustrated in Example 3.

Example 3. Consider correlating the four alerts a_i, a_j, a_k, and a_h in Figure 2 with the queue graph given in Figure 3, and suppose $f(a_h) = e_1$, $f(a_k) = e_2$, 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 $E = \{e_i \mid 1 \leq i \leq n\}$, $C = \{c_i \mid 1 \leq i \leq m\}$, $R_r \subseteq C \times E$, 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:*
  - *$\mathcal{Q} = \{Q_i \mid 1 \leq i \leq n\}$ are n queues of length one,*
  - *$\mathcal{V} = \{v_i \mid 1 \leq i \leq m\}$ are m variables,*
  - *for each $k = 1, 2, \ldots, n$,*
    - *$\mathcal{P}_k = \{< Q_j, v_i > \mid (c_i, e_j) \in BFS(k)\} \cup \{< v_i, Q_j > \mid (e_j, c_i) \in BFS(k)\}$ are the layer k backward pointers, and*
    - *$\mathcal{PR}_k = \{< v_i, Q_j > \mid (c_i, e_j) \in BFSR(k)\} \cup \{< Q_j, v_i > \mid (e_j, c_i) \in BFSR(k)\}$ are the layer k forward pointers.*

Definition 1 formally characterizes the queue graph data structure. To rephrase Example 2 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 backward pointers are $\mathcal{P}_1 = \{< Q_1, v_1 >,$

$< v_1, Q_2 >, < Q_2, v_2 >, < v_1, Q_3 >\}$[3], and the layer one forward pointers are $\mathcal{PR}_1 = \phi$. The layer two pointers include $\mathcal{P}_2 = \{< Q_2, v_2 >\}$ and $\mathcal{PR}_2 = \{< Q_2, v_1 >, < v_1, Q_1 >\}$. The layer three pointers include $\mathcal{P}_3 = \{< Q_3, v_2 >\}$ and $\mathcal{PR}_3 = \{< Q_3, v_1 >, < v_1, Q_1 >\}$.

The process for correlating alerts using a queue graph, as illustrated in Example 3, is more precisely stated as the procedure *QG_Alert_Correlation* in Figure 4. 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.

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 $f(a_{new}) = e_i$ 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 $< Q_i, v_k > \in \mathcal{P}_i$ and $< v_k, Q_j > \in \mathcal{P}_i$, 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. 4. A Procedure for Correlating Alerts with Queue Graphs

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.

Second, the memory usage of the QG approach is roughly $O(n(n + m))$ (n layers, with each layer having maximally $(n + m)$ pointers) [4], and hence does not depend on

[3] We use the notation $< a, b >$ for a pointer in a queue graph and (a, b) for an edge in a graph.

[4] 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.

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 [5].

4.3 Handling 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 [6].

The basic QG approach does not work properly if alerts do not arrive in the correct order. To illustrate, 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, suppose another concurrent alert a_2' matches the same exploit as a_2 does, and it 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 through reordering alerts inside a time window before feeding them into the queue graph. 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 $t_2 + t_{max} - t_{max} > t_1$, 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 certainly 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

[5] 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.

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

correlation effort. This reflects the inherent tradeoff between the capability of containing unavoidable uncertainties and the performance of processing alerts.

5 A Unified Approach to Alert Correlation, Hypothesis, and Prediction

In this section, we extend the basic QG-based correlation procedure to a unified approach to correlating received alerts, hypothesizing missing alerts, and predicting future alerts. Section 5.1 introduces some key concepts. Sections 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). 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 4 and Example 5.

Example 4. The sequence of alerts shown on the left hand side of Figure 5(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. The sequence shown in the middle of Figure 5 (that is, 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 Figure 5 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 Figure 5). We say a sequence of

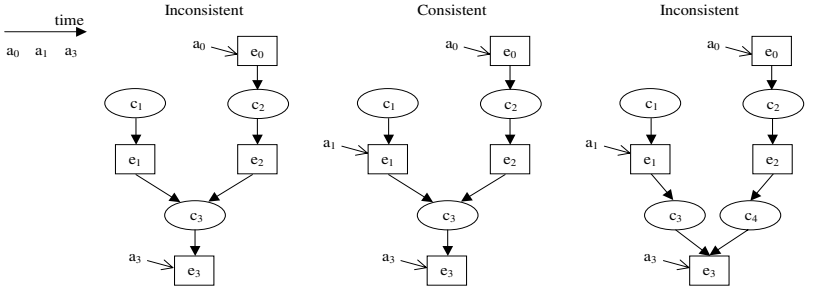


Fig. 5. Examples of Consistent and Inconsistent Alert Sequences

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 4 depicts an inconsistent alert sequence in which the consecutive executions of exploits is broken by missing alerts. Example 5 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 [23] cannot distinguish the two cases in Example 5, 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 4 and Example 5, 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 approach 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 [7] 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 collectively 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 6.

Example 6. Consider again the three cases, from left to right, in Figure 5 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.

More precisely, we extend the basic QG approach described in Section 4 by modifying the fourth step of Procedure *QG_Alert_Correlation*. Due to space limitations, we describe how the modified procedure works but leave out more details (a detailed procedure can be found in [39]). Consider 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*,

[7] 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.

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 $G_r(V, E_l \cup E_r)$ 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 $Q_i (1 \leq i \leq n)$. First, we start from Q_i and follow the pointers in $\mathcal{PR}_i$ to set each variable $v_j (1 \leq j \leq m)$ 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 [8] 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 7. Consider the left-hand side case of Figure 5. 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.

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 Sec-

[8] 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.

tion 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.

The procedure of prediction is similar to that of correlation and hypothesis discussed in the previous section, although they differ in some details. More specifically, 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 in $\mathcal{PR}_i$ (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, $Con_2, \ldots$, with $Con_i (1 \leq i \leq m)$ 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 Empirical Results

This section evaluates the proposed correlation, hypothesis, and prediction techniques through implementation and empirical results. The correlation engine is implemented in C++ and tested on a Pentium III 860MHz server with 1G RAM running RedHat Linux. We use Snort-2.3.0 [32] to generate isolated alerts, which are directly pipelined into the correlation engine for analyses. We use Tcpreplay 2.3.2 [37] to replay network traffic from a separate machine to the server running the correlation engine.

We use two data sets for experiments, the Darpa 2000 intrusion detection LLDOS 1.0 by MIT Lincoln Labs [7], and the treasure hunt dataset by the University of California, Santa Barbara [36]. The attack scenario in the Darpa 2000 dataset has been extensively explored before (such as in [19]). 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 1.4G 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).

Effectiveness. The objective of the first set of experiments is to justify the effectiveness of the proposed algorithms in alert correlation, hypothesis, and prediction. We use the Darpa 2000 dataset for this purpose, as the attack scenario can be easily referenced in both the included description and previous results, such as [19]. Our correlation algorithm produces similar result to that of previous work. However, in contrast to the static result graph seen in those work, our result graph actually *evolves* in time with the continuously arriving alerts. Such a result can more clearly reveal the actual progress of an intrusion (due to space limitations, snapshots of the result graph during real-time correlation can be found in [39]). To save space, only the latest alert matching each exploit is shown in the figures in this section.

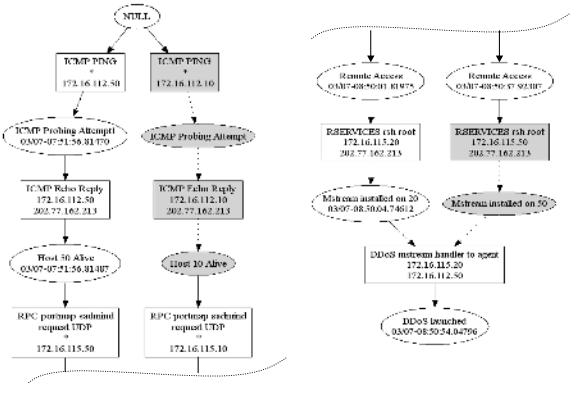


Fig. 6. The Hypothesis of Missing Alerts During Correlation

Figure 6 includes 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).

Figure 7 includes 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.

Performance. The objective of the second set of experiments is to evaluate the real-time performance of the correlation engine. The performance metric includes the resource

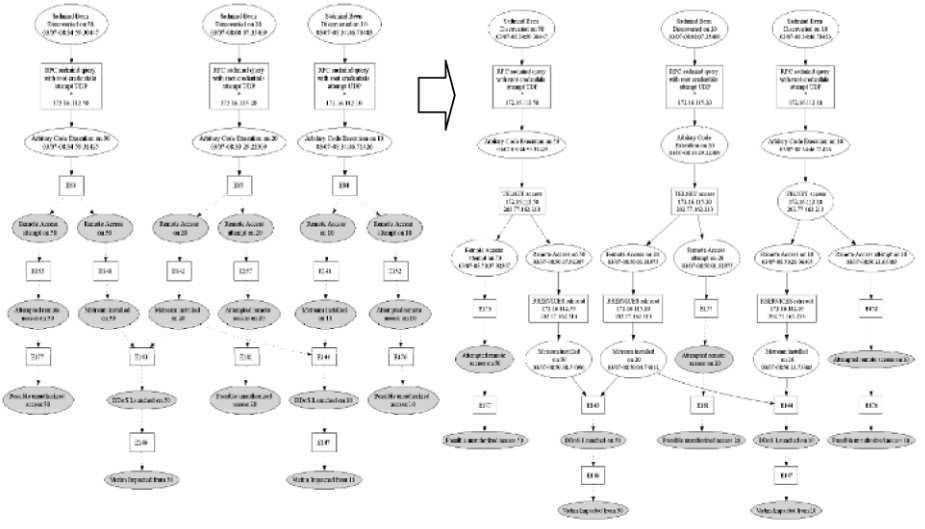


Fig. 7. The Prediction of Possible Consequences During Correlation

usage (CPU and memory) and the processing time of each alert. The correlation engine measures its own processing time and treats the delay between receiving two consecutive alerts as Snort's processing time. All the results have 95% confidence intervals within about 5% of the reported values. Figure 8 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).

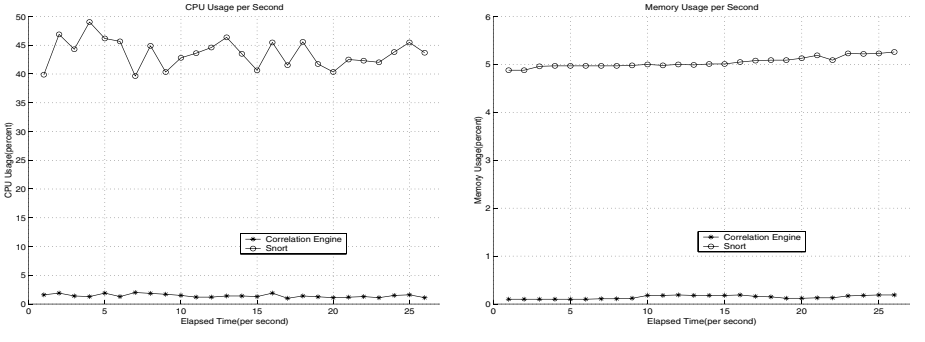
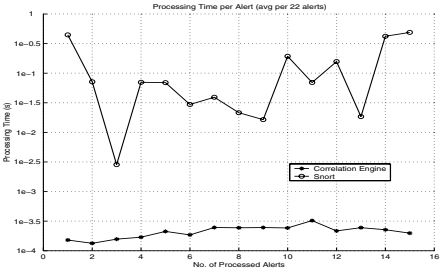


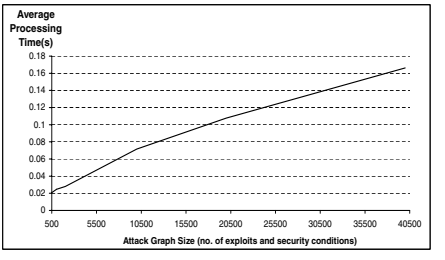
Fig. 8. The CPU and Memory Usage

The left chart in Figure 9 shows the processing time per alert (averaged per 22 alerts). Clearly, the correlation engine takes much less time than Snort throughout the processing of the entire data set. The result also proves that the performance of our

correlation method does not decrease over time. Instead, the time required for correlating each alert remains fairly steady. Next we examine the scalability of the correlation engine in terms of the number of exploits and security conditions. We use the treasure hunt data set for this purpose. The initial attack graph only has about one hundred exploits. We increase the size of attack graphs by randomly inserting dummy exploits and corresponding security conditions. The inserted exploits increase the complexity of correlation because the correlation engine must search through them. The right chart in Figure 9 shows the average processing time as a function of the attack graph size. The result shows that the average time for correlation scales with the size of attack graph as expected.



The Processing Time for Darpa Dataset



The Processing Time vs. Attack Graph Size

Fig. 9. The Processing Time and Its Relationship with the Size of Attack Graph

We replay network traffic at a high speed (for example, the Darpa data set is replayed in about 26 seconds). Real-world traffic is 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. We plan to remove such limitations in our future work.

7 Conclusion

In this paper, we studied the real-time correlation of intrusion alerts. We identified a limitation 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. It can correlate alerts that are arbitrarily far away. Based on the QG method, we proposed a unified method for the correlation, hypothesis, and prediction of alerts. The proposed techniques are implemented and evaluated. 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 and predict the progress of multi-step intrusions. Our future work is to integrate the proposed methods in a prototype system and evaluate it with real-world traffic in live networks.

Acknowledgements. The authors thank Marc Dacier and the anonymous reviewers for their valuable comments, and Giovanni Vigna for providing the Treasure Hunt dataset.

References

1. P. Ammann, D. Wijesekera, and S. Kaushik. Scalable, graph-based network vulnerability analysis. In *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS'02)*, pages 217–224, 2002.
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 and A. Miege. Alert correlation in a cooperative intrusion detection framework. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy (S&P'02)*, pages 187–200, 2002.
4. F. Cuppens and 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)*, pages 197–216, 2001.
5. O. Dain and R.K. Cunningham. Building scenarios from a heterogeneous alert system. In *Proceedings of the 2001 IEEE Workshop on Information Assurance and Security*, 2001.
6. O. Dain and R.K. Cunningham. Fusing a heterogeneous alert stream into scenarios. In *Proceedings of the ACM Workshop on Data Mining for Security Applications*, pages 1–13, 2001.
7. 2000 darpa intrusion detection evaluation datasets. http://www.ll.mit.edu/IST/ideval/data/2000/2000_data_index.html, 2000.
8. H. Debar and A. Wespi. Aggregation and correlation of intrusion-detection alerts. In *Proceedings of the 3rd International Symposium on Recent Advances in Intrusion Detection (RAID'01)*, pages 85–103, 2001.
9. S.T. Eckmann, G. Vigna, and R.A. Kemmerer. STATL: An attack language for state-based intrusion detection. *Journal of Computer Security*, 10(1/2):71–104, 2002.
10. D. Farmer and E.H. Spafford. The COPS security checker system. In *USENIX Summer*, pages 165–170, 1990.
11. N. Habra, Charlier B.L., A. Mounji, and 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)*, pages 430–450, 2004.
12. IBM. IBM tivoli risk manager. Available at http://www.ibm.com/software/tivoli/products/risk-mgr/.
13. SRI International. Event monitoring enabling responses to anomalous live disturbances (EMERALD). Available at http:// www.sdl.sri.com/projects/emerald/.
14. S. Jajodia, S. Noel, and B. O'Berry. Topological analysis of network attack vulnerability. In V. Kumar, J. Srivastava, and A. Lazarevic, editors, *Managing Cyber Threats: Issues, Approaches and Challenges*. Kluwer Academic Publisher, 2003.
15. S. Jha, O. Sheyner, and J.M. Wing. Two formal analysis of attack graph. In *Proceedings of the 15th Computer Security Foundation Workshop (CSFW'02)*, 2002.
16. Klaus Julisch and Marc Dacier. Mining intrusion detection alarms for actionable knowledge. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 366–375, 2002.
17. W. Lee, J.B.D. Cabrera, A. Thomas, N. Balwalli, S. Saluja, and 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.
18. B. Morin, L. Mé, H. Debar, and 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)*, pages 115–137, 2002.
19. P. Ning, Y. Cui, and D.S. Reeves. Constructing attack scenarios through correlation of intrusion alerts. In *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS'02)*, pages 245–254, 2002.

20. P. Ning and D. Xu. Adapting query optimization techniques for efficient intrusion alert correlation. Technical report, NCSU, Department of Computer Science, 2002.
21. P. Ning and D. Xu. Learning attack strategies from intrusion alerts. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS'03)*, 2003.
22. P. Ning, D. Xu, C.G. Healey, and 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)*, pages 97–111, 2004.
23. S. Noel and S. Jajodia. Correlating intrusion events and building attack scenarios through attack graph distance. In *Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC'04)*, 2004.
24. S. Noel, S. Jajodia, B. O'Berry, and M. Jacobs. Efficient minimum-cost network hardening via exploit dependency grpahs. In *Proceedings of the 19th Annual Computer Security Applications Conference (ACSAC'03)*, 2003.
25. R. Ortalo, Y. Deswarte, and M. Kaaniche. Experimenting with quantitative evaluation tools for monitoring operational security. *IEEE Trans. Software Eng.*, 25(5):633–650, 1999.
26. OSSIM. Open source security information management. Available at http://www.ossim.net.
27. V. Paxson. Bro: A system for detecting network intruders in real-time. *Computer Networks*, 31(23-24):2435–2463, 12 1999.
28. X. Qin and W. Lee. Statistical causality analysis of INFOSEC alert data. In *Proceedings of the 6th International Symposium on Recent Advances in Intrusion Detection (RAID 2003)*, pages 591–627, 2003.
29. X. Qin and W. Lee. Discovering novel attack strategies from INFOSEC alerts. In *Proceedings of the 9th European Symposium on Research in Computer Security (ESORICS 2004)*, pages 439–456, 2004.
30. R. Ritchey and P. Ammann. Using model checking to analyze network vulnerabilities. In *Proceedings of the 2000 IEEE Symposium on Research on Security and Privacy (S&P'00)*, pages 156–165, 2000.
31. R. Ritchey, B. O'Berry, and S. Noel. Representing TCP/IP connectivity for topological analysis of network security. In *Proceedings of the 18th Annual Computer Security Applications Conference (ACSAC'02)*, page 25, 2002.
32. M. Roesch. Snort - lightweight intrusion detection for networks. In *Proceedings of the 1999 USENIX LISA Conference*, pages 229–238, 1999.
33. O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J.M. Wing. Automated generation and analysis of attack graphs. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy (S&P'02)*, pages 273–284, 2002.
34. S. Staniford, J.A. Hoagland, and J.M. McAlerney. Practical automated detection of stealthy portscans. *Journal of Computer Security*, 10(1/2):105–136, 2002.
35. S. Templeton and K. Levitt. A requires/provides model for computer attacks. In *Proceedings of the 2000 New Security Paradigms Workshop (NSPW'00)*, pages 31–38, 2000.
36. Treasure hunt datasets. http://www.cs.ucsb.edu/~vigna/treasurehunt/index.html, 2004.
37. A. Turner. Tcpreplay: Pcap editing and replay tools for *nix. Available at http://tcpreplay.sourceforge.net/.
38. A. Valdes and K. Skinner. Probabilistic alert correlation. In *Proceedings of the 4th International Symposium on Recent Advances in Intrusion Detection*, pages 54–68, 2001.
39. L. Wang, A. Liu, and S. Jajodia. Real-time analyses of intrusion alert streams. Technical report, Center for Secure Information Systems, George Mason University, 2005.
40. D. Zerkle and K. Levitt. Netkuang - a multi-host configuration vulnerability checker. In *Proceedings of the 6th USENIX Unix Security Symposium (USENIX'96)*, 1996.
41. Y. Zhai, P. Ning, P. Iyer, and D. Reeves. Reasoning about complementary intrusion evidence. In *Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC'04)*, pages 39–48, 2004.