Adapting Query Optimization Techniques for Efficient Intrusion Alert Correlation[*]

Peng Ning and Dingbang Xu
Department of Computer Science, North Carolina State University
Raleigh, NC 29695-7534

Abstract

Intrusion alert correlation is the process to identify high-level attack scenarios by reasoning about low-level alerts raised by intrusion detection systems (IDS). The efficiency of intrusion alert correlation is critical in enabling interactive analysis of intrusion alerts as well as prompt responses to attacks. This paper presents an experimental study aimed at adapting main memory index structures (e.g., T Trees, Linear Hashing) and database query optimization techniques (e.g., nested loop join, sort join) for efficient correlation of intensive alerts. By taking advantage of the characteristics of the alert correlation process, this paper presents three techniques named *hyper-alert container, two-level index,* and *sort correlation*. This paper then reports a series of experiments designed to evaluate the effectiveness of these techniques. These experiments demonstrate that (1) hyper-alert containers improve the efficiency of order-preserving index structures (e.g., T Trees), with which an insertion operation involves search, (2) two-level index improves the efficiency of all index structures, (3) a two-level index structure combining Chained Bucket Hashing and Linear Hashing is the most efficient for streamed alerts with and without memory constraint, and (4) sort correlation with heap sort algorithm is the most efficient for alert correlation in batch.

Keywords: Intrusion detection, intrusion alert correlation, query optimization

1 Introduction

Traditional intrusion detection systems (IDS) focus on low-level attacks or anomalies, and raise alerts independently, though there may be logical connections between them. In situations where there are intensive intrusions, not only will actual alerts be mixed with false alerts, but the amount of alerts will also become unmanageable. As a result, it is difficult for human users or intrusion response systems to understand the alerts and take appropriate actions.

To assist the analysis of intrusion alerts, several alert correlation methods (e.g., [8,9,21]) have been proposed recently to process the alerts reported by IDS. (Please see Section 5 for details.) As one of

[*]This work is partially supported by the National Science Foundation under grants CCR-0207297 and ITR-0219315, by the U.S. Army Research Office under grant DAAD 19-02-1-0219. The authors would like to thank the anonymous reviewers for their valuable suggestions.

these methods, we have been developing intrusion alert correlation and analysis techniques based on prerequisites and consequences of attacks [15, 16]. Intuitively, the prerequisite of an intrusion is the necessary condition for the intrusion to be successful, while the consequence of an intrusion is the possible outcome of the intrusion. Based on the prerequisites and consequences of different types of attacks, our method correlates alerts by (partially) matching the consequence of some previous alerts and the prerequisite of some later ones.

We have implemented an offline intrusion alert correlator using our approach, and our initial experiments with 2000 DARPA intrusion detection scenario specific datasets[1] indicate that our approach is promising in constructing attack scenarios and differentiating true and false alerts [16]. On the basis of our intrusion alert correlator, we also developed three utilities named *adjustable graph reduction, focused analysis*, and *graph decomposition*, to facilitate the interactive analysis of intensive intrusion alerts. Our study with the network traffic collected at the DEF CON 8 Capture The Flag (CTF) event[2] demonstrated that these three utilities can simplify the analysis of intensive alerts and help identify the attack strategies behind them [15].

Although we have demonstrated the effectiveness of our alert correlation techniques, our solution still faces some challenges. In particular, we implemented the previous intrusion alert correlator as a DBMS-based application [16]. Involving a DBMS in the alert correlation process provided enormous convenience and support in our initial implementation; however, relying entirely on the DBMS also introduced performance penalty. For example, to correlate about 65,000 alerts generated from the DEFCON 8 CTF dataset, it took the DBMS-based alert correlator around 45 minutes with the JDBC-ODBC driver included in Java 2 SDK, Standard Edition, and more than 4 minutes with the Microsoft SQL Server 2000 Driver for JDBC. Such performance is clearly not sufficient to make alert correlation practical, especially for interactive analysis of intensive alerts. Our timing analysis indicates that the performance bottleneck lies in the interaction between the intrusion alert correlator and the DBMS. Since this implementation completely relies on the DBMS, processing of each single alert entails interaction with the DBMS, which introduces significant performance penalty.

In this paper, we address this problem by performing alert correlation entirely in main memory, while only using the DBMS as the storage of intrusion alerts. We study several main memory index structures, including Array Binary Search [2], AVL Trees [1], B Trees [4], Chained Bucket Hashing [11], Linear Hashing [13], and T Trees [12], as well as some database query optimization techniques such as nested loop join and sort join [10] to facilitate timely correlation of intrusion alerts. By taking advantage of the characteristics of the alert correlation process, we develop three techniques named *hyper-alert container, two-level index*, and *sort correlation*, which further reduce the execution time required by alert correlation.

We performed a series of experiments to evaluate these techniques with the DEF CON 8 CTF data set. The experimental results demonstrate that (1) hyper-alert containers improve the efficiency of index structures with which an insertion operation involves search (e.g., B Trees, T Trees), (2) two-level index improves the efficiency of all index structures, (3) a two-level index structure combining Chained Bucket Hashing and Linear Hashing is most efficient for correlating streamed alerts with and without memory constraint, and (4) sort correlation with heap sort algorithm is most efficient for alert correlation in batch. With the most efficient method, the execution time for correlating the alerts generated from the DEF CON 8 CTF data set is reduced from over four minutes (when the

[1]`http://www.ll.mit.edu/IST/ideval/data/2000/2000_data_index.html.`

[2]`http://www.defcon.org/html/defcon-8-post.html.`

Microsoft SQL Server 2000 driver for JDBC is used) to less than one second.

The remainder of this paper is organized as follows. To be self contained, Section 2 briefly describes our alert correlation method and some previous results. Section 3 presents our adaptations of the main memory index structures and some join methods. Section 4 reports our implementation and experimental results. Section 5 discusses the related work, and Section 6 concludes this paper and points out some future research directions.

2 An Overview of Alert Correlation

In this section, we briefly describe our model for correlating alerts using prerequisites and consequences of intrusions. Further details can be found in [16].

The alert correlation model is based on the observation that in series of attacks, the component attacks are usually not isolated, but related as different stages of the attacks, with the early ones preparing for the later ones. For example, an attacker has to install Distributed Denial of Service (DDOS) daemon programs before he can launch a DDOS attack. To take advantage of this observation, we correlate alerts using prerequisites and consequences of the corresponding attacks. Intuitively, the *prerequisite* of an attack is the necessary condition for the attack to be successful. For example, the existence of a vulnerable service is the prerequisite of a remote buffer overflow attack against the service. Moreover, an attacker may make progress (e.g., install a Trojan horse program) as a result of an attack. Informally, we call the possible outcome of an attack the *consequence* of the attack. In a series of attacks where attackers launch earlier ones to prepare for later ones, there are usually strong connections between the consequences of the earlier attacks and the prerequisites of the later ones. Accordingly, we identify the prerequisites (e.g., existence of vulnerable services) and the consequences (e.g., discovery of vulnerable services) of each type of attacks and correlate detected attacks (i.e., alerts) by matching the consequences of previous alerts and the prerequisites of later ones.

We use predicates as basic constructs to represent prerequisites and consequences of attacks. For example, a scanning attack may discover UDP services vulnerable to certain buffer overflow attacks. We can use the predicate *UDPVulnerableToBOF* (*VictimIP, VictimPort*) to represent this discovery. In general, we use a logical formula, *i.e.*, logical combination of predicates, to represent the prerequisite of an attack. Thus, we may have a prerequisite of the form *UDPVulnerableToBOF* (*VictimIP, VictimPort*) $\wedge$ *UDPAccessibleViaFirewall* (*VictimIP, VictimPort*). Similarly, we use a *set* of logical formulas to represent the consequence of an attack.

With predicates as basic constructs, we use a *hyper-alert type* to encode our knowledge about each type of attacks. A *hyper-alert type T* is a triple (*fact, prerequisite, consequence*) where (1) *fact* is a set of attribute names, each with an associated domain of values, (2) *prerequisite* is a logical formula whose free variables are all in *fact*, and (3) *consequence* is a set of logical formulas such that all the free variables in *consequence* are in *fact*. Intuitively, the *fact* component of a hyper-alert type gives the information associated with the alert, *prerequisite* specifies what must be true for the attack to be successful, and *consequence* describes what could be true if the attack indeed happens. For brevity, we omit the domains associated with attribute names when they are clear from context.

Given a hyper-alert type *T* = (*fact, prerequisite, consequence*), a *hyper-alert (instance) h of type T* is a finite set of tuples on *fact*, where each tuple is associated with an interval-based timestamp [*begin_time, end_time*]. The hyper-alert h implies that *prerequisite* must evaluate to True and all the logical formulas in *consequence* might evaluate to True for each of the tuples. The *fact* component

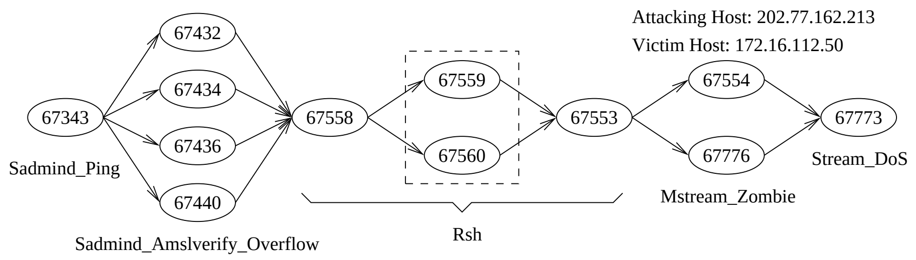


Figure 1: A hyper-alert correlation graph

of a hyper-alert type is essentially a relation schema (as in relational databases), and a hyper-alert is a relation instance of this schema. A hyper-alert *instantiates* its *prerequisite* and *consequence* by replacing the free variables in *prerequisite* and *consequence* with its specific values. Note that *prerequisite* and *consequence* can be instantiated multiple times if *fact* consists of multiple tuples. For example, if an IPSweep attack involves several IP addresses, the *prerequisite* and *consequence* of the corresponding hyper-alert type will be instantiated for each of these addresses.

To correlate hyper-alerts, we check if an earlier hyper-alert *contributes* to the prerequisite of a later one. Specifically, we decompose the prerequisite of a hyper-alert into parts of predicates and test whether the consequence of an earlier hyper-alert makes some parts of the prerequisite True (*i.e.,* makes the prerequisite easier to satisfy). If the result is positive, then we correlate the hyper-alerts. In our formal model, given an instance h of the hyper-alert type *T* = (*fact, prerequisite, consequence*), the *prerequisite set (or consequence set, resp.) of* h, denoted $P(h)$ (or $C(h)$, resp.), is the set of all such predicates that appear in *prerequisite* (or *consequence*, resp.) whose arguments are replaced with the corresponding attribute values of each tuple in h. Each element in $P(h)$ (or $C(h)$, resp.) is associated with the timestamp of the corresponding tuple in h. We say that hyper-alert h_1 *prepares for* hyper-alert h_2 if there exist $p \in P(h_2)$ and $C \subseteq C(h_1)$ such that for all $c \in C$, $c.end\_time < p.begin\_time$ and the conjunction of all the logical formulas in C implies p.

Given a sequence S of hyper-alerts, a hyper-alert h in S is a *correlated hyper-alert* if there exists another hyper-alert h' such that either h prepares for h' or h' prepares for h. We use a *hyper-alert correlation graph* to represent a set of correlated hyper-alerts. Specifically, a *hyper-alert correlation graph* CG = (N, E) is a connected graph, where N is a set of hyper-alerts and for each pair $n_1, n_2 \in N$, there is a directed edge from n_1 to n_2 in E if and only if n_1 prepares for n_2. Figure 1 shows one of the hyper-alert correlation graphs discovered in our experiments with the 2000 DARPA intrusion detection evaluation datasets [16]. Each node in Figure 1 represents a hyper-alert. The numbers inside the nodes are the alert Id's generated by the IDS.

Previous Implementation and Evaluation. We have implemented an intrusion alert correlator using our method [16], which is a Java application that interacts with the DBMS via JDBC. In this implementation, we expand the consequence set of each hyper-alert by including all the predicates implied by the consequence set. We call the result the *expanded consequence set* of the hyper-alert. The predicates in both prerequisite and expanded consequence sets of the hyper-alerts are then encoded into strings called *Encoded Predicate* and stored in two tables, *PrereqSet* and *Expanded-ConseqSet*, along with the corresponding hyper-alert ID and timestamp. Both tables have attributes *HyperAlertID, EncodedPredicate*, *begin_time*, and *end_time*, with meanings as indicated by their names. As a result, alert correlation can be performed using the following SQL statement.

```
SELECT DISTINCT c.HyperAlertID, p.HyperAlertID
```

```
FROM PrereqSet p, ExpandedConseqSet c
WHERE p.EncodedPredicate = c.EncodedPredicate AND c.end_time < p.begin_time
```

To evaluate the effectiveness of our method, we performed a series of experiments using the 2000 DARPA intrusion detection scenario specific datasets (LLDOS 1.0 and LLDOS 2.0.2). Our experiments indicated that our approach can effectively construct attack scenarios from low-level alerts [16]. Moreover, we also developed three utilities called *adjustable graph reduction, focused analysis,* and *graph decomposition* to facilitate the interactive analysis of large sets of intrusion alerts [15]. We did a case study with the network traffic captured at the DEF CON 8 CTF event; the results showed that these utilities effectively simplified the analysis of large amounts of alerts, and revealed several attack strategies used in the DEF CON 8 CTF event [15].

As discussed earlier, one problem of the intrusion alert correlator is its efficiency because of its dependence on, and intensive interaction with the DBMS. In this paper, we address this problem by performing alert correlation entirely in main memory, while only using the DBMS as the storage of intrusion alerts. In the following, we study how to improve performance of alert correlation by adapting database query optimization techniques, including various main memory index structures.

3 Adapting Query Optimization Techniques

The essential problem in this work is how to perform the SQL query in Section 2 efficiently. One option is to use database query optimization techniques, which have been studied extensively for both disk based and main memory based databases. However, alert correlation has a different access pattern than typical database applications, which may lead to different performance than traditional database applications. In addition, the unique characteristics in alert correlation give us an opportunity for further improvement. Thus, in this section, we seek ways to improve alert correlation by adapting existing query optimization techniques.

3.1 Main Memory Index Structures

Main memory index structures have been studied extensively in the context of search algorithms and main memory databases. Many different kinds of index structures have been proposed in the literature. In our study, we focus on the following ones: Array Binary Search [2], AVL Trees [1], B Trees [4], Chained Bucket Hashing [11], Linear Hashing [13], and T Trees [12]. In the following, we briefly describe these index structures. Detailed information can be found in the corresponding references. For comparison purpose, we also implement a naive, sequential scan method, which simply scans in an (unordered) array for the desired data item.

Array Binary Search [2, 11] stores sorted data items in an array and locates the desired item via binary search. Array Binary Search is pretty efficient when searching in a static array. However, it has certain drawbacks in a dynamic environment. First, the array has to have enough space to accommodate new data items; otherwise, memory reallocation and copy of the entire array will have to be performed. In addition, insertion or deletion involves $O(N)$ data movements.

AVL Trees [1] are balanced binary search trees. Each node in an AVL Tree contains a data item, control information, a left pointer which points to the subtree that contains the smaller data items (than the current data item), and a right pointer which points to the subtree that contains the bigger items (than the current data item). Search in an AVL Tree is very fast, since the binary search is intrinsic to the tree structure [12]. Insertion into an AVL Tree always involves a leaf node, and both insertion and deletion may lead to rotation operations if it results in an unbalanced tree.

B Trees [4] are also balanced search trees. Unlike an AVL Tree, a node in a B Tree may have multiple data items and pointers. Data items in a B Tree node are ordered, and each pointer points to a subtree that consists of the data items that fall into the range identified by the adjacent data items. B trees are shallower than AVL Trees, and thus involve less node accesses for a search operation. Insertion and deletion in a B Tree is fast, which usually involves only one node.

T Trees [12] are binary trees with many elements in a node, which evolved from AVL Trees and B Trees. T Trees retain the intrinsic binary search nature of AVL Trees, but it also has the good update and storage characteristics of B Trees, since a T Tree node contains many elements. Search in a T Tree consists of a search in the binary tree followed by a search within a node. Insertion or deletion in a T Tree involves data movements within a single node, and possible rotations to rebalance the tree structure.

Chained Bucket Hashing [11] uses a static hash table and a chain of buckets for each hash entry. It is efficient in a static environment where the number of data items can be predetermined. However, in a dynamic environment in which the number of data items is not known, Chained Bucket Hashing may have poor performance. If the hash table is too small, too many buckets may be chained for each hash entry; if the hash table is too large, space may be wasted due to empty entries.

Linear Hashing [13] uses a dynamic hash table, which splits hash buckets in predefined linear oder. Each time when the candidate bucket (i.e., the next bucket to split according to the linear order) overflows, Linear Hashing splits the candidate bucket into two, and the size of the hash table grows by one. The overflowed data items in the non-candidate buckets are placed in the overflow buckets for the same hash entries. The buckets are ordered sequentially, allowing the bucket address to be computed from a base address.

3.2 Correlating Streamed Intrusion Alerts

We first study alert correlation methods that deal with intrusion alert streams continuously generated by IDS. With such methods, an alert correlation system can be pipelined with IDS and produce correlation result in a timely manner.

Figure 2 presents a nested loop method that can accommodate streamed alerts. (As the name suggests, nested loop correlation is adapted from nested loop join [10].) It assumes that the input hyper-alerts are ordered ascendingly in terms of their beginning time. The nested loop method takes advantage of main memory index structures such as Linear Hashing. While processing the hyper-alerts, it maintains an index structure $\mathcal{I}$ for the instantiated predicates in the expanded consequence sets along with the corresponding hyper-alerts. Each time when a hyper-alert h is processed, the algorithm searches in $\mathcal{I}$ for each instantiated predicate p that appears in h's prerequisite set. A match of a hyper-alert h' implies that h' has the same instantiated predicate p in its expanded consequent set. If h'.EndTime is before h.BeginTime, then h' prepares for h. If the method processes all the hyper-alerts in the ascending order of their beginning time, it is easy to see that the nested loop method can find all and only the prepare-for relations between the input hyper-alerts.

The nested loop correlation method has different performance if different index structures are used. Thus, one of our tasks is to identify the index structure most suitable for this method. In addition, we further develop two adaptations to improve the performance of these index structures. Our first adaptation is based on the following observation.

Observation 1 *Multiple hyper-alerts may share the same instantiated predicate in their expanded consequence sets. Almost all of them prepare for a later hyper-alert that has the same instantiated predicate in its prerequisite set.*

```
Outline of Nested Loop Correlation
Input: A list $H$ of hyper-alerts ordered ascendingly in their beginning times.
Output: All pairs of ($h'$, $h$) such that both $h$ and $h'$ are in $H$ and $h'$ prepares for $h$.
Method:
  Maintain an index structure $\mathcal{I}$ for instantiated predicates in the expanded consequence sets
  of hyper-alerts. Each instantiated predicate is associated with the corresponding hyper-alert.
  Initially, $\mathcal{I}$ is empty.
  1. for each hyper-alert $h$ in $H$ (accessed in the given order)
  2.     for each instantiated predicate $p$ in the prerequisite set of $h$
  3.         Search the set of hyper-alerts with index key $p$ in $\mathcal{I}$. Let $H'$ be the result.
  4.         for each $h'$ in $H'$
  5.             if ($h'$.EndTime$<$ $h$.BeginTime) then output ($h'$, $h$).
  6.     for each $p$ in the expanded consequence set of $h$
  7.         Insert $p$ along with $h$ into $\mathcal{I}$.
  end
```

Figure 2: Outline of the nested loop alert correlation methods

Observation 1 implies that we can associate hyper-alerts with an instantiated predicate p if p appears in the expanded consequence sets of all these hyper-alerts. As a result, locating an instantiated predicate directly leads to the locations of all the hyper-alerts that share the instantiated predicate in their expanded consequence sets. We call the set of hyper-alerts associated with an instantiated predicate a *hyper-alert container*.

Using hyper-alert containers does not always result in better performance. There are two types of accesses to the index structure in the nested loop correlation method: insertion and search. For the index structures that preserve the order of data items, insertion implies search, since each time when an element is inserted into the index structure, it has to be placed in the "right" place. Using hyper-alert container does not increase the insertion cost significantly, while at the same time reduces the search cost. However, for the non-order preserving index structures such as Linear Hashing, insertion does not involve search. Using hyper-alert containers would force to perform a search, since the hyper-alerts have to be put into the right container. In this case, hyper-alert container decreases the search cost but increases the insertion cost, and it is not straightforward to determine whether the overall cost is decreased or not. We study this through experiments in Section 4.

Observation 2 *There is a small, static, and finite set of predicates. Two instantiated predicates are the same only if they are instantiated from the same predicate.*

Observation 2 leads to a *two-level index structure*. Each instantiated predicate can be split into two parts, the predicate name and the arguments. The top-level index is built on the predicate names. Since we usually have a static and small set of predicate names, we use Chained Bucket Hashing for this purpose. Each element in the top-level index further points to a second-level index structure. The second-level index is built on the arguments of the instantiated predicates. When an instantiated predicate is inserted into a two-level index structure, we first locate the right hash bucket based on the predicate name, then locate the second-level index structure within the hash bucket (by scanning the bucket elements), and finally insert it into the second-level index structure using the arguments.

We expect the two-level index structure to improve the performance due to the following reasons. First, since the number of predicates is small and static, using Chained Bucket Hashing on predicate names is very efficient. In our experiments, the size of the hash table is set to the number of

```
Outline of Sort Correlation
Input: A set $H$ of hyper-alerts.
Output: All pairs of ($h'$, $h$) such that both $h$ and $h'$ are in $H$ and $h'$ prepares for $h$.
Method:
  Prepare two arrays $A_{pre}$ and $A_{con}$, each entry of which is a hyper-alert associated with a $key$
  field. Each array is initialized with a reasonable size, and reallocated with doubled sizes if
  out of space. Existing content is copied to the new buffer if reallocation happens.
  1. for each $h$ in $H$
  2.     for each $p$ in the prerequisite set of $h$
  3.         Append $h$ to $A_{pre}$ with $key = p$.
  4.     for each $p$ in the expanded consequence set of $h$
  5.         Append $h$ to $A_{con}$ with $key = p$.
  6. Sort $A_{pre}$ and $A_{con}$ ascendingly in terms of the $key$ field (with, e.g., heap sort).
  7. Partition the entries in $A_{pre}$ and $A_{con}$ into maximal blocks that share the same instantiated
     predicate. Assume $A_{pre}$ and $A_{con}$ have $B_{pre}$ and $B_{con}$ blocks, respectively.
  8. $i = 0, j = 0$.
  9. while ($i < B_{pre}$ and $j < B_{con}$) do
  10.    if ($A_{pre}.Block_i.InstantiatedPredicate < A_{con}.Block_j.InstantiatedPredicate$) then
  11.      $i = i + 1$.
  12.    else if ($A_{pre}.Block_i.InstantiatedPredicate > A_{con}.Block_j.InstantiatedPredicate$) then
  13.      $j = j + 1$.
  14.    else for each $h$ in $A_{pre}.Block_i$ and each $h'$ in $A_{con}.Block_j$
  15.            if $h'.EndTime < h.BeginTime$ then output ($h'$, $h$).
  16.        $i = i + 1, j = j + 1$.
  end
```

Figure 3: The sort correlation method

predicates, and it usually takes one or two accesses to locate the second-level index structure for a given predicate name. Second, the two-level index structure decomposes the entire index structure into smaller ones, and thus reduces the search time in the second-level index. We verify our analysis through extensive experiments in Section 4.

3.3 Correlating Intrusion Alerts in Batch

Some applications allow alerts to be processed in batch (e.g., forensic analysis with an alert database). Though the nested loop method discussed earlier is still applicable, there are more efficient ways for alert correlation in batch.

Figure 3 presents a sort correlation method, which is adapted from sort join [10]. The sort correlation method achieves good performance by taking advantage of efficient main memory sorting algorithms. Specifically, it uses two arrays, A_{pre} and A_{con}. A_{pre} stores the instantiated predicates in the prerequisite sets of the hyper-alerts (along with the corresponding hyper-alerts), and A_{con} stores the instantiated predicates in the expanded consequence sets (along with the corresponding hyper-alerts). This method then sorts both arrays in terms of the instantiated predicate with an efficient sorting algorithm (e.g., heap sort).

Assume both arrays are sorted ascendingly in terms of instantiated predicate. The sort correlation method partitions both arrays into blocks that share the same instantiated predicate, and scans both arrays simultaneously. It maintains two indices, i and j, that references to the current blocks in A_{pre}

and A_{con}, respectively. The method compares the instantiated predicates in the two current blocks. If the instantiated predicate in the current block of A_{pre} is smaller, it advances the index i; if the instantiated predicate in the current block A_{con} is smaller, it advances the index j; otherwise, the current blocks of A_{pre} and A_{con} share the same instantiated predicate. The method then examines each pair of hyper-alerts h' and h, where h' and h are in the current block of A_{con} and A_{pre}, respectively. If the end time of h' is before the beginning time of h, then h' prepares for h.

It is easy to see that the sort correlation method can find all pairs of hyper-alerts such that the first prepares for the second. Consider two hyper-alerts h and h' where h' prepares for h. There must exist an instantiated predicate p in both the expanded consequence set of h' and the prerequisite set of h. Thus, p along with h' must be placed in the array A_{con}, and p along with h must be placed in the array A_{pre}. The scanning method (lines 9–16) will eventually point i to p's block in A_{pre} and j to p's block in A_{con} at the same time, and output h' prepares for h. Therefore, the sort correlation can discover all and only pairs of hyper-alerts such that the first prepares for the second.

We also study the possibility of adapting two-index join and hash join methods [10] to improve the performance of batch alert correlation. However, our analysis indicates they cannot outperform nested loop correlation due to the fact that alert correlation is performed entirely in main memory.

A naive adaptation of two-index join leads to the following correlation method: Build two index structures for the instantiated predicates in the prerequisite sets and the expanded consequence sets, respectively. For each instantiated predicate p, locate the hyper-alerts related to p in both index structures, and compare the corresponding timestamps. However, this method cannot perform better than the nested loop method. The nested loop method only involves insertion of instantiated predicates in the expanded consequence sets and search of those in the prerequisite sets. In contrast, the above adaptation requires insertion of instantiated predicates in both prerequisite and expanded consequence sets, and search of instantiated predicates in at least one of the index structures.

A possible improvement over the naive adaptation is to merge the two index structures. We can associate two sets of hyper-alerts with each instantiated predicate p, denoted $H_{pre}(p)$ and $H_{con}(p)$, and build one index structure for the instantiated predicates. $H_{pre}(p)$ and $H_{con}(p)$ consist of the hyper-alerts that have p in their prerequisite sets and expanded consequence sets, respectively. After all the instantiated predicates in the prerequisite or the consequence set of the hyper-alerts are inserted into the index structure, we can simply scan all the instantiated predicates, and compare the corresponding timestamps of the hyper-alerts in $H_{pre}(p)$ and $H_{con}(p)$ for each instantiated predicate p. However, each insertion of an instantiated predicate entails a search operation, since the corresponding hyper-alert has to be inserted into either $H_{pre}(p)$ or $H_{con}(p)$. Thus, this method cannot outperform the nested loop method, which involves one insertion for each instantiated predicate in the expanded consequence sets, and one search for each instantiated predicate in the prerequisite sets. A similar conclusion can be drawn for hash join.

Another possibility to have a faster batch correlation is to use Chained Bucket Hashing. Since the number of alerts is known beforehand, we may be able to decide a relatively accurate hash table size, and thus have a better performance than its counter part for streamed alerts. We study this through experiments in Section 4.

3.4 Correlating Intrusion Alerts with Limited Memory

The previous approaches to in-memory alert correlation have assumed that all index structures fit in memory during the alert correlation process. This may be true for analyzing intrusion alerts collected during several days or weeks; however, in typical operational scenarios, the IDS produce intrusion alerts continuously and the memory of the alert correlation system will eventually be exhausted. A typical solution is to use a “sliding window” to focus on alerts that are close to each other; at any given point in time, only alerts after a previous time point are considered for correlation.

We adopt a sliding window which can accommodate up to t intrusion alerts. The parameter t is determined by the amount of memory available to the intrusion alert correlation system. Since our goal is to optimize the intrusion alert correlation process, we do not discuss how to choose the appropriate value of t in this paper. Each time when a new intrusion alert is coming, we check if inserting this new alert will result in more than t alerts in the index structure. If yes, we remove the oldest alert from the index structure. In either case, we will perform the same correlation process as in Section 3.2. It is also possible to add multiple intrusion alerts in batch. In this case, multiple old alerts may be removed from the index structure. Note that though choosing a sliding *time* window is another option, it doesn’t reflect the memory constraint we have to face in this application.

Using a sliding window in our application essentially implies deleting old intrusion alerts when there are more than t alerts in the memory. This problem appeared to be trivial at the first glance, since all the data structures have known deletion algorithms. However, we soon realized that we had to go through a little trouble to make the deletion efficient. The challenge is that the index structures we build in all the previous approaches are in terms of instantiated predicates to facilitate correlation. However, to remove the oldest intrusion alerts, we need to locate and remove alerts in terms of their timestamps. Thus, the previous index structures cannot be used to perform the deletion operation efficiently. Indeed, each deletion implies a scan of all the alerts in the index structures.

To address this problem, we add a *secondary data structure* to facilitate locating the oldest intrusion alerts. Since the intrusion alerts are inserted as well as removed in terms of their time order, we use a queue (simulated with a circular buffer) for this purpose. Each newly inserted intrusion alert also has an entry added into this queue, which points to its location in the *primary index structure* in terms of the instantiated predicates. Thus, when we need to remove the oldest intrusion alert, we can simply dequeue an alert, find its location in the primary index structure, and delete it directly. Indeed, this is more efficient than the generic deletion method of the order preserving index structures (e.g., AVL Trees), since deletion usually implies search in those index structures.

4 Experimental Results

We have implemented all the techniques discussed in Section 3. All the programs are written in Java, with JDBC to connect to the DBMS. We performed a series of experiments to compare these techniques. All the experiments were run on a DELL Precision Workstation with 1.8GHz Pentium 4 CPU and 512M bytes memory. The alerts used in our experiments were generated by a RealSecure Network Sensor 6.0[3], which monitors an isolated network in which we replayed the network traffic collected at the DEF CON 8 CTF event. The Network Sensor was configured to use the *Maximum_Coverage* policy with a slight change, which forced the Network Sensor to save all the reported alerts. (Further details on implementation and experiments can be found in [17].)

In these experiments, we mapped each alert type reported by the RealSecure Network Sensor to a hyper-alert type, and generated one hyper-alert from each alert. The prerequisite and consequence of each hyper-alert type were specified according to the descriptions of the attack signatures provided by RealSecure. There are totally 65,058 hyper-alerts generated by the RealSecure Network Sensor,

[3]`http://www.iss.net.`

among which 52,318 hyper-alerts have prerequisite or consequence. The remaining hyper-alerts are mainly *Windows_Access_Error* and *IPDuplicate*. These hyper-alerts cannot be correlated with any other ones due to the overly general semantics, and do not contribute to the time required by alert correlation. To precisely evaluate the relationship between the execution time and the number of hyper-alerts, we did not include them in our experiments.

It is desirable to evaluate the quality of alert correlation (i.e., completeness and soundness) as we have done in [16]. However, there is no information about the actual attacks and their relationships in the DEF CON 8 dataset, while the DARPA datasets are too small to evaluate the new techniques. Further considering that the techniques proposed in this paper do not affect the quality of alert correlation, in the following, we only evaluate the efficiency of these techniques.

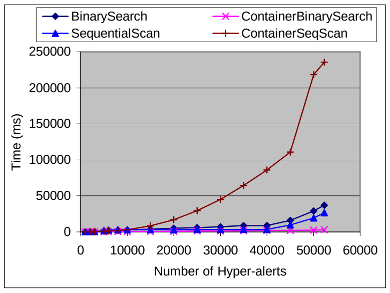
Nested-Loop Correlation without Memory Constraint. Our first set of experiments was intended to evaluate the effectiveness of hyper-alert container in nested loop correlation. According to our analysis, hyper-alert container may reduce the execution time if we use the order-preserving index structures. We compared the execution time for Sequential Scan, Array Binary Search, and Linear Hashing, with or without hyper-alert container. We did not perform a similar comparison for the tree index structures (i.e., T Tree, B Tree, and AVL Tree), since not having hyper-alert container not only increases both insertion and search cost, but also the complexity of the programs. As shown in Figures 4(a) and 4(b), hyper-alert container reduces the execution time for Array Binary Search, but increases the execution time for Sequential Scan significantly, and Linear Hashing slightly.

Our second set of experiments was intended to evaluate the effectiveness of two-level index structure in the nested loop correlation method. According to our analysis and the earlier experimental results, we used hyper-alert container in Array Binary Search and tree index structures, but not in Sequential Scan and Linear Hashing. As indicated by Figures 4(c) to 4(e), two-level index reduces execution time for all index structures.

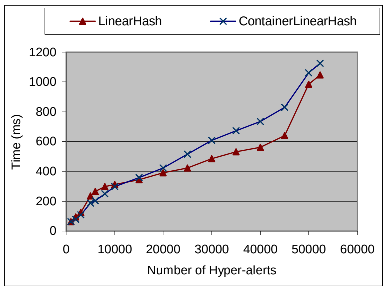
In Figure 4(c), the lines for Sequential Scan and two-level Sequential Scan have an interesting flat area when the number of input hyper-alerts is between 8,000 and 40,000. Our investigation revealed that the majority of hyper-alerts in this range do not have any prerequisite. Thus, processing of these hyper-alerts does not involve search (i.e., sequential scan) in a large array, and there is no big increase in execution time. In the other index structures, there is no significant difference between insertion and search costs. Thus, there is no dramatic change in execution time for the hyper-alerts between 8,000 and 40,000, though we can observe the slow down in the increase of execution time.

Our next goal is to find out which index structure (with or without the two adaptations) has the best performance for nested loop correlation. We take the fastest methods from Figure 4(c), 4(d), and 4(e), which are two-level Array Binary Search with hyper-alert container, two-level AVL Tree, and two-level Linear Hashing, and put them in Figure 4(f). The resulting figure shows both two-level AVL Tree and two-level Linear Hashing are significantly faster than two-level Array Binary Search with hyper-alert container, and two-level Linear Hashing outperforms two-level AVL Tree by up to 20%. Thus, nested loop correlation achieves the best performance with two-level Linear Hashing.

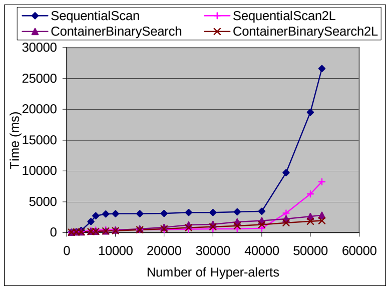
Batch Correlation without Memory Constraint. Our next set of experiments is focused on methods for correlating alerts in batch. Certainly, all the previously evaluated methods can be used for batch processing of intrusion alerts. Our evaluation here is to determine whether any method can achieve better performance than nested loop correlation with two-level Linear Hashing, the best method for correlating streamed alerts. Besides two-level Linear Hashing, we tested Chained Bucket Hashing and the sort correlation methods, because they have the potential to perform better than two-level Linear Hashing. To further examine the impact of the time order of input hyper-



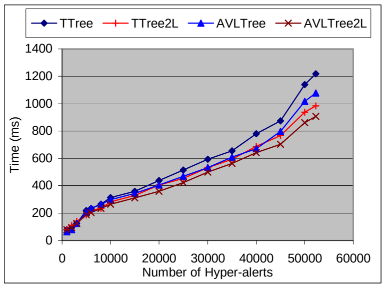
(a) Hyper-alert containers (1)



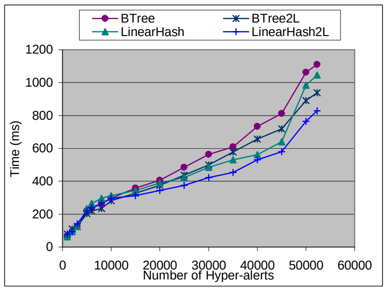
(b) Hyper-alert containers (2)



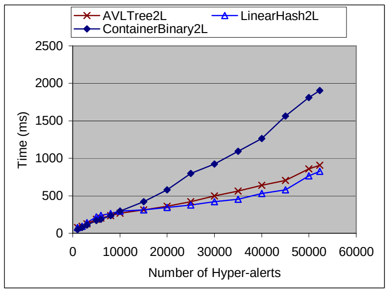
(c) Two-level indices (1)



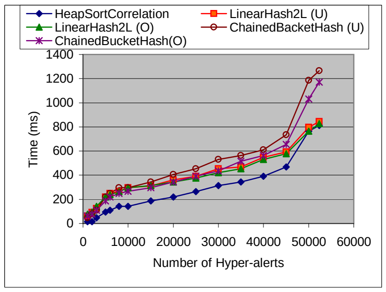
(d) Two-level indices (2)



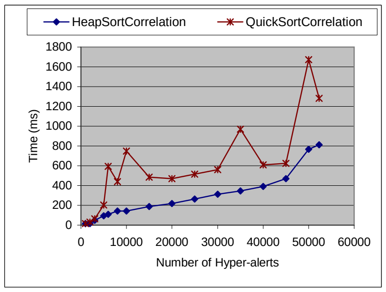
(e) Two-level indices (3)



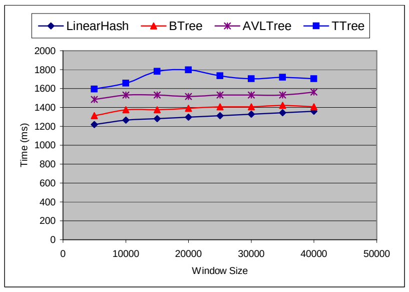
(f) Efficient two-level indices



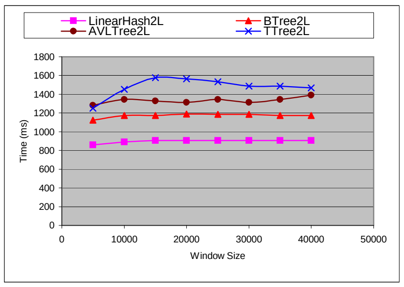
(g) Efficient batch correlation



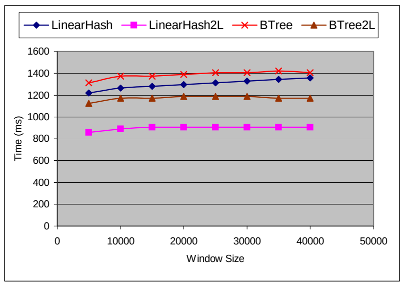
(h) Impact of sorting algorithms



(i) With memory constraint (1)



(j) With memory constraint (2)



(k) With memory constraint (3)

Figure 4: Experimental results

alerts, we examined the timing results with ordered and unordered input. With input hyper-alerts not ordered in their beginning time, the algorithm must insert all of the instantiated predicates in the expanded consequence sets before it processes any instantiated predicate in the prerequisite sets.

Figure 4(g) shows the timing results of these methods. Surprisingly, Chained Bucket Hashing has the worst performance. Our further investigation explains this result: The average number of data items per hash entry is between 1.0 and 1.52; however, the maximum number of data items per hash entry is between 162 and 518. That is, the distribution of the instantiated predicates resulted in uneven distribution of hyper-alerts in the buckets. Having input hyper-alerts ordered by beginning time only reduced the execution time slightly differences for nested loop correlation with both two-level Linear Hashing and Chained Bucket Hashing. Finally, sort correlation with heap sort achieves the best performance among these methods.

We also studied the impact of different sorting algorithms on the execution time of sort correlation. We compared two sorting algorithms, heap sort and quick sort. Heap sort has the least complexity in the worst case scenarios, while quick sort is considered the best practical choice among all the sorting algorithms [5]. Figure 4(h) shows the timing results of both algorithms: Sort correlation with quick sort performs significantly worse than the heap sort case. In addition, the execution time is not very stable in terms of the number of input hyper-alerts. This is because quick sort is sensitive to the input. In contrast, heap sort has stably increasing execution time as the number of hyper-alerts increases. Thus, we believe heap sort is a good choice for sort correlation.

Nested-Loop Correlation with Memory Constraint. Our last set of experiments is focused on evaluating the efficiency of different indexing structures when there is memory constraint. Based on our prior experimental results, we only compare the execution time of AVL Tree, T Tree, B Tree, and Linear Hashing. We do not consider Sequential Scan and Array Binary Search because of their poor performance (in insertion and search). It's quite clear that their performance will not be comparable with the other methods.

We performed a set of experiments with varying sliding window sizes, using all of the hyper-alerts as input. Figures 4(i), 4(j), and 4(k) show the results. These results indicate that two-level Linear Hashing is the most efficient and the two level index structure improves the performance for all four methods. An interesting observation is that there is a bump in the line for T Tree in both Figure 4(i) and Figure 4(j) when the window size is between 15,000 and 25,000. Our investigation reveals that the numbers of node balancing operations for these window sizes are more than the other window sizes. (There are 16,611, 16,684, and 16,232 node balancing operations for the window sizes 15,000, 20,000, and 25,000, respectively.)

5 Related Work

Our alert correlation method is intended to process alerts generated by IDS. Although we used RealSecure Network Sensor in our experiments, our method can correlate alerts from any other IDS (e.g., Snort [18]), as long as we can specify the prerequisite and consequence for the alerts.

The result in this paper is a continuance of our previous work [15, 16], which has been described in Section 2. Our method was initially developed to address the limitations of JIGSAW [20]. Our method has several features beyond JIGSAW. First, our method allows partial satisfaction of prerequisites, recognizing the possibility of undetected and unknown attacks, while JIGSAW cannot deal with missing detections. Second, our method allows aggregation of alerts, and thus can reduce the complexity involved in alert analysis, while JIGSAW does not have any similar mechanism.

The work closest to ours is the alert correlation method by Cuppens and Miege in the context of MIRADOR project [6], which has been done independently and in parallel to our previous work. The MIRADOR approach also correlates alerts using partial match of prerequisites (pre-conditions) and consequences (post-conditions) of attacks, which are derived from attack databases described in LAMBDA [7]. However, our method allows alert aggregation during and after correlation, while the MIRADOR approach treats alert aggregation as an individual stage before alert correlation. This difference has led to three utilities for interactive alert analysis [15].

Several other alert correlation methods have been proposed. Spice [19] and the probabilistic alert correlation method [21] correlate alerts based on the similarities between alert attributes. Though they are effective in correlating some alerts (e.g., alerts with the same source and destination IP addresses), they cannot fully discover the causal relationships between alerts. Another type of alert correlation methods (e.g., the data mining approach [8]) bases alert correlation on attack scenarios specified by human users or learned through training datasets. These methods are restricted to *known* attack scenarios. A variation in this class uses a consequence mechanism to specify what types of attacks may follow a given attack, partially addressing this problem [9].

Our work is also related to query optimization for streaming data (e.g., [3] and [14]). These techniques are usually intended to provide DBMS support for generic streaming applications. In contrast, the techniques developed in this paper are specifically aimed at alert correlation, which can be considered a special form of queries.

6 Conclusions and Future Work

In this paper, we studied main memory index structures and database query optimization techniques to facilitate timely correlation of intensive alerts. We developed three techniques named *hyper-alert container, two-level index,* and *sort correlation* by taking advantage of the characteristics of the alert correlation process. The experimental study demonstrated that (1) hyper-alert containers improve the efficiency of order-preserving index structures, with which an insertion operation involves search, (2) two-level index improves the efficiency of all index structures, (3) a two-level index structure combining Chained Bucket Hashing and Linear Hashing is most efficient for streamed alerts with and without memory constraint, and (4) sort correlation with heap sort algorithm is the most efficient for alert correlation in batch. Our future work includes incorporating the efficient methods in this paper into the intrusion alert correlation toolkit and developing more techniques to facilitate timely interactive analysis of intrusion alerts.

References

[1] A. Aho, J. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[2] A. Ammann, M. Hanrahan, and R. Krishnamurthy. Design of a memory resident DBMS. In *Proc. of IEEE COMPCON*, February 1985.

[3] S. Chandrasekaran, M.J. Franklin. Streaming queries over streaming data. In *Proc. of VLDB*, August 2002.

[4] D. Comer. The ubiquitous B-Tree. *ACM Computeing Surveys*, 11(2):121–137, 1979.

[5] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1989.

[6] F. Cuppens and A. Miege. Alert correlation in a cooperative intrusion detection framework. In *Proc. of the 2002 IEEE Symposium on Security and Privacy*, May 2002.

[7] F. Cuppens and R. Ortalo. LAMBDA: A language to model a database for detection of attacks. In *Proc. of Recent Advances in Intrusion Detection*, pages 197–216, September 2000.

[8] O. Dain and R.K. Cunningham. Fusing a heterogeneous alert stream into scenarios. In *Proc. of the ACM Workshop on Data Mining for Security Applications*, pages 1–13, November 2001.

[9] H. Debar and A. Wespi. Aggregation and correlation of intrusion-detection alerts. In *Recent Advances in Intrusion Detection*, LNCS 2212, pages 85–103, 2001.

[10] H. Garcia-Molina, J. Widom, and J. D. Ullman. *Database System Implementation*. Prentice Hall, 2000.

[11] D. Knuth. *The Art of Computer Programming*. Addison-Wesley, 1973.

[12] T. J. Lehman and M. J. Carey. A study of index structure for main memory database management systems. In *Proc. of the 12th Int'l Conf. on Very Large Databases*, pages 294–303, August 1986.

[13] W. Litwin. Linear hashing: A new tool for file and table addressing. In *Proc. of the 6th Conf. on Very Large Data Bases*, pages 212–223, October 1980.

[14] S. Madden, M.A. Shah, J.M. Hellerstein, V. Raman. Continuously adaptive continuous queries over streams. In *Proc. of the ACM SIGMOD Conference*, June 2002.

[15] P. Ning, Y. Cui, and D. S Reeves. Analyzing intensive intrusion alerts via correlation. In *Proc. of the 5th Int'l Symposium on Recent Advances in Intrusion Detection*, October 2002.

[16] P. Ning, Y. Cui, and D. S Reeves. Constructing attack scenarios through correlation of intrusion alerts. In *Proc. of the 9th ACM Conf. on Computer and Communications Security*, 2002.

[17] P. Ning and D. Xu. Adapting query optimization techniques for efficient intrusion alert correlation. Technical Report TR-2002-14, NCSU, Dept. of Computer Science, 2002.

[18] M. Roesch. Snort - lightweight intrusion detection for networks. In *Proc. of the 1999 USENIX LISA conference*, 1999.

[19] 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.

[20] S. Templeton and K. Levit. A requires/provides model for computer attacks. In *Proc. of New Security Paradigms Workshop*, pages 31–38. September 2000.

[21] A. Valdes and K. Skinner. Probabilistic alert correlation. In *Proc. of the 4th Int'l Symposium on Recent Advances in Intrusion Detection*, pages 54–68, 2001.