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
Email: ning@csc.ncsu.edu, dxu@unity.ncsu.edu

Abstract

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. Several complementary alert correlation methods have been proposed to address this problem. As one of these methods, we have developed a Database Management System (DBMS) based toolkit to correlate intrusion alerts, which have been shown to be effective through our previous studies. However, our experience also shows relying entirely on DBMS introduces unacceptable performance penalty, especially for interactive analysis of intensive alerts.

This paper adapts main memory index structures (e.g., B Trees, T Trees, Linear Hashing) and database query optimization techniques (e.g., nested loop join, sort join) to facilitate timely 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*. The performance of these techniques is studied through a series of experiments. The experimental results demonstrate that (1) hyper-alert containers improve the efficiency of order-preserving index structures, with which an insertion operation involves search (e.g., Array Binary Search, 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 the most efficient for streamed alerts, (4) sort correlation with heap sort algorithm is the most efficient for alert correlation in batch, (5) two-level Linear Hashing is the most efficient for alert correlation when sliding window is used to cope with memory constraint

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., [9, 10, 28]) have been proposed recently to process the alerts reported by IDS. (Please see Section 5 for details.) As one of these methods, we have been developing intrusion alert correlation and analysis techniques based on prerequisites and consequences of attacks [20, 21, 22, 24]. 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 [19] indicate that our approach is promising in constructing attack scenarios and differentiating true and false alerts [22]. 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 [11] demonstrated that these three utilities can simplify the analysis of intensive alerts and help identify the attack strategies behind them [21].

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 [22]. 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 DEF CON 8 CTF dataset, it took the DBMS-based intrusion alert correlator around 45 minutes with the JDBC-ODBC driver included in Java 2 SDK, Standard Edition[1], and more than 4 minutes with the Microsoft SQL Server 2000 Driver for JDBC[2]. Such performance is clearly not sufficient to make alert correlation a practical tool, especially for interactive analysis of intensive alerts. Our timing analysis indicates that the performance bottle neck lies in the interaction between the intrusion alert correlator and the DBMS. Since our current intrusion alert correlator completely relies on the DBMS, processing of each single alert entails interaction with the DBMS, which introduces significant performance overhead.

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 [5], Chained Bucket Hashing [15], Linear Hashing [18], and T Trees [16], as well as some database query optimization techniques such as nested loop join and sort join [27] 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 [11]. 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, 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 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 Alert Correlation Based on Prerequisites and Consequences of Intrusions

2.1 The Model

In this section, we briefly describe our model for correlating alerts using prerequisites of intrusions. Please read [22] for further details.

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., discover a vulnerable service, 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

[1] `http://java.sun.com/j2se/.`
[2] `http://www.microsoft.com/sql/.`

of attacks and correlate detected attacks (i.e., alerts) by matching the consequences of some previous alerts and the prerequisites of some later ones.

Note that an attacker does not have to perform early attacks to prepare for later ones. For example, an attacker may launch an individual buffer overflow attack against the service blindly. In this case, we cannot, and should not correlate it with others. However, if the attacker does launch attacks with earlier ones preparing for later ones, our method can correlate them, provided the attacks are detected by IDS.

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. To simplify the discussion, we restrict the logical operators to $\wedge$ (conjunction) and $\vee$ (disjunction).

With predicates as basic constructs, we use a *hyper-alert type* to encode our knowledge about each type of attacks.

Definition 1 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 succeeds. For brevity, we omit the domains associated with attribute names when they are clear from context.

Example 1 Consider the buffer overflow attack against the *sadmind* remote administration tool. We may have the following hyper-alert type for such attacks: *SadmindBufferOverflow* = ({*VictimIP, VictimPort*}, *ExistHost* (*VictimIP*) $\wedge$ *VulnerableSadmind* (*VictimIP*), {*GainRootAccess*(*VictimIP*)}). Intuitively, this hyper-alert type says that such an attack is against the host running at IP address *VictimIP*. (We expect the actual values of *VictimIP* are reported by an IDS.) As the prerequisite of a successful attack, there must exist a host at the IP address *VictimIP* and the corresponding *sadmind* service should be vulnerable to buffer overflow attacks. The attacker may gain root privilege as a result of the attack. □

Given a hyper-alert type, a *hyper-alert instance* can be generated if the corresponding attack is reported by IDS.

Definition 2 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 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.

In our model, we treat timestamps implicitly and omit them if they are not necessary for our discussion.

Example 2 Consider the hyper-alert type *SadmindBufferOverflow* defined in example 1. We may have a hyper-alert $h_{SadmindBOF}$ that includes the following tuples: {(*VictimIP* = 152.141.129.5, *VictimPort* = 1235), (*VictimIP* = 152.141.129.37, *VictimPort* = 1235)}. This implies that if the attack is successful, the following two logical formulas must be True as the prerequisites of the attack: *ExistHost* (152.141.129.5) $\wedge$ *VulnerableSadmind* (152.141.129.5), *ExistHost* (152.141.129.37) $\wedge$ *VulnerableSadmind* (152.141.129.37), and the following two predicates might be True as consequences of the attack: *GainRootAccess* (152.141.129.5), *GainRootAccess* (152.141.129.37). This hyper-alert says that there are buffer overflow attacks against *sadmind* at IP addresses 152.141.129.5 and 152.141.129.37, and the attacker may gain root access as a result of the attacks. □

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.

Definition 3 Consider a hyper-alert type *T* = (*fact*, *prerequisite*, *consequence*). The *prerequisite set (or consequence set, resp.) of* T, denoted $P(T)$ (or $C(T)$, resp.), is the set of all such predicates that appear in *prerequisite* (or *consequence*, resp.). Given a hyper-alert instance h of type T, the *prerequisite set (or consequence set, resp.) of* h, denoted $P(h)$ (or $C(h)$, resp.), is the set of predicates in $P(T)$ (or $C(T)$, 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.

Definition 4 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. Otherwise, h is called an *isolated hyper-alert*.

Let us further explain the alert correlation method with the following example.

Example 3 Consider the *Sadmind Ping* attack with which an attacker discovers possibly vulnerable *sadmind* services. The corresponding hyper-alert type can be represented by *SadmindPing* = ({*VictimIP*, *VictimPort*}, *ExistHost* (*VictimIP*), {*VulnerableSadmind* (*VictimIP*)}). It is easy to see that $P(SadmindPing) = \{ExistHost(VictimIP)\}$, and $C(SadmindPing) = \{VulnerableSadmind(VictimIP)\}$.

Suppose a hyper-alert $h_{SadmindPing}$ of type *SadmindPing* has the following tuples: {(*VictimIP* = 152.141.129.5, *VictimPort* = 1235)}. Then the prerequisite set of $h_{SadmindPing}$ is $P(h_{SadmindPing})$ = {*ExistHost* (152.141.129.5)}, and the consequence set is $C(h_{SadmindPing})$ = {*VulnerableSadmind* (152.141.129.5)}.

Now consider the hyper-alert $h_{SadmindBOF}$ discussed in Example 2. Similar to $h_{SadmindPing}$, we can easily get $P(h_{SadmindBOF})$ = {*ExistHost* (152.141.129.5), *ExistHost* (152.141.129.37), *VulnerableSadmind* (152.141.129.5), *VulnerableSadmind* (152.141.129.37)}, and $C(h_{SadmindBOF})$ = {*GainRootAccess* (152.141.129.5), *GainRootAccess* (152.141.129.37)}.

Assume that all tuples in $h_{SadmindPing}$ have timestamps earlier than every tuple in $h_{SadmindBOF}$. By comparing the contents of $C(h_{SadmindPing})$ and $P(h_{SadmindBOF})$, it is clear that the element *VulnerableSadmind* (152.141.129.5) in $P(h_{SadmindBOF})$ (among others) is also in $C(h_{SadmindPing})$. Thus, $h_{SadmindPing}$ prepares for, and should be correlated with $h_{SadmindBOF}$. □

The prepare-for relation between hyper-alerts provides a natural way to represent the causal relationship between correlated hyper-alerts. We also introduce the notion of a *hyper-alert correlation graph* to represent a set of correlated hyper-alerts.

Definition 5 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.

A hyper-alert correlation graph is an intuitive representation of correlated alerts. It can potentially reveal intrusion strategies behind a series of attacks, and thus lead to better understanding of the attacker's intention. We have performed a series of experiments with the 2000 DARPA intrusion detection evaluation datasets [22]. Figure 1 shows one of the hyper-alert correlation graphs discovered from these datasets. Each node in Figure 1 represents a hyper-alert. The numbers inside the nodes are the alert Id's generated by the IDS. This hyper-alert correlation graph clearly shows the strategy behind the sequence of attacks. (For details please refer to [22].)

2.2 Previous Implementation and Evaluation

We have implemented an intrusion alert correlator using our method [22], which is a Java application that interacts with the DBMS via JDBC[3]. 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 *ExpandedConseqSet*, along with the corresponding

[3]JDBC is an API that allows to access data sources from the Java programming language. Please visit `http://java.sun.com/products/jdbc/` for details.

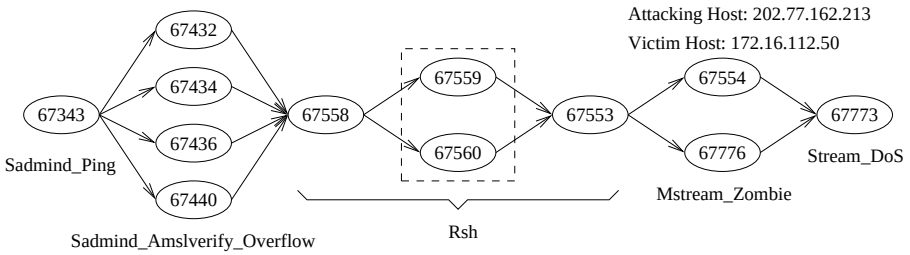


Figure 1: A hyper-alert correlation graph discovered in the 2000 DARPA intrusion detection evaluation datasets

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) [19]. We introduced two measures, completeness (R_c) and soundness (R_s), to measure the effectiveness of alert correlation: $R_c = \frac{\#correctly\ correlated\ alerts}{\#related\ alerts}$, $R_s = \frac{\#correctly\ correlated\ alerts}{\#correlated\ alerts}$. Intuitively, the completeness of alert correlation assesses how well we can correlate related alerts together, while the soundness evaluates how correctly the alerts are correlated. Our experiments showed that our method achieved $R_c = R_s = 93.18\%$ for the inside traffic of LLDOS 1.0, $R_c = R_s = 94.74\%$ for the DMZ traffic of LLDOS 1.0, $R_c = 66.7\%$ and $R_s = 92.3\%$ for the inside traffic of LLDOS 2.0.2, and $R_2 = 62.5\%$ and $R_s = 100\%$ for the DMZ traffic of LLDOS 2.0.2. Our further analysis revealed that all the hyper-alerts missed in LLDOS 2.0.2 were triggered by the same *telnet* that the attacker used to access a victim host, which were false alerts since these attacks never happened. Thus, our method is demonstrated experimentally to be effective to construct attack scenarios from intrusion alerts.

On the basis of the DBMS-based intrusion alert correlator, 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 [21]. These utilities are intended for human users to analyze and understand the correlated alerts as well as the strategies behind them. We have studied the effectiveness of these utilities through a case study with the network traffic captured at the DEF CON 8 CTF event. Our results showed that the three utilities effectively simplified the analysis of large amounts of alerts, and revealed several attack strategies used in the DEF CON 8 CTF event.

One challenge we face in the DBMS-based intrusion alert correlator is its efficiency. Although using DBMS to perform the alert correlation has provided enormous convenience and support in our first implementation, it also introduces performance penalty. Our experiments with the DEF CON 8 CTF data set showed that it took more than four minutes for the DBMS-based intrusion alert correlator to correlate about 65,000 alerts on a DELL Precision Workstation with a 1.8GHz Pentium 4 CPU. Such performance is clearly not sufficient to make alert correlation a practical tool, especially for interactive analysis of intensive alerts. Our timing analysis indicates that the performance bottle neck lies in the frequent interactions between the intrusion alert correlator and the DBMS. Due to the dependence on the DBMS, processing of each single alert entails interaction with the DBMS, which together introduces significant performance overhead.

In this work, 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; this may lead to different performance than traditional database applications. In addition, the unique characteristics in alert correlation may give us the opportunity for further improvement. Thus, in this and the next sections, we seek the possibilities to improve alert correlation by adapting existing query optimization techniques, evaluate various techniques and their adaptations, and identify the most suitable for intrusion alert correlation.

In the following, we first go over some main memory index structures, and then present our adaptations for correlating streamed as well as batch alerts. In Section 4, we report our experimental results.

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 [5], Chained Bucket Hashing [15], Linear Hashing [18], and T Trees [16].

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. We only care about insertion and search operations due to the need for alert correlation.

Sequential Scan is only implemented for reference purposes. In our study, Sequential Scan stores data items in an array. Search is performed by sequentially scanning the data items in the array, and insertion is simply to append to the end of the array.

Array Binary Search [15, 2] 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, even if there is enough space, insertion into the array 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 [16]. Insertion into an AVL Tree always involves a leaf node, and may lead to a rotation operation if it results in an unbalanced tree.

B Trees [5] 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 into a B Tree is fast, which usually involves only one node.

T Trees [16] are binary trees with many elements in a node, which evolved from AVL Trees and B Trees. The T Tree retains the intrinsic binary search nature of the AVL Tree, but it also has the good update and storage characteristics of the B Tree, 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 into a T Tree involves data movements within a single node, and possible rotations to rebalance the tree structure.

Chained Bucket Hashing [15] 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 (e.g., alert correlation), Chained Bucket Hashing may have poor performance. If the size of the hash table is too small, too many buckets may be chained for each hash entry; if the size of the hash table is too large, space may be wasted due to the empty entries.

Linear Hashing [18] 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

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

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 [12].) 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 and T Trees. While processing the hyper-alerts, the nested loop method 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 according to Definition 4. If the method processes all the hyper-alerts in the ascending order of their beginning time, it is not difficult 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.*

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*.

However, 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 (Figure 2: insertion and search. For the index structures that preserve the order of data items in them, insertion implies search, since each time when an element is inserted into the index structure, we have to place it in the "right" place. Using hyper-alert container does not increase the insertion

cost significantly in this case, 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.

Our second adaptation is based on the following observation.

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 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 [27]. 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. The sort correlation method 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 according to Definition 4.

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 in Figure 3 (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 thus 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 [27] 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.

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

A naive adaptation of two-index join leads to the following two-index 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 approachs 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 IDSs 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. Such a method has been adopted by many IDSs such as ADAM [4].

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 Implementation and Experiments

We have implemented all the techniques discussed in Section 3. All the programs are written in Java, with JDBC to connect to the DBMS. However, unlike our previous prototype system, the current implementation only uses the DBMS as the storage of hyper-alert types and hyper-alerts. All the processing of alerts is handled in main memory by the program. To make the execution time comparable, we reuse the code as much as possible, and make sure we use the most efficient way in coding.

Some index structures need array to store the data, which may need memory reallocation in dynamic environments. We implemented a simple memory reallocation strategy to handle all the array reallocation. Each array is initialized with a certain size. When the array is not enough, the program reallocates another array with a doubled size and copy over all the data items in the previous array.

Several index structures require some other parameters. For B Trees, we need to specify node size (i.e., how many data items to store in one B Tree node); for T Trees, we need minimum and maximum node sizes; for Chained Bucket Hashing and Linear Hashing, we need the bucket size (i.e., how many elements in each bucket). Different parameters may result in different performance. A common feature of these parameters is that both too large and too small values will result in poor performance. We found the experimentally optimal values for these parameters in the corresponding references, performed a series of experiments to compare the execution time, and picked the best values. As a result, the node size of B Trees is 7, the minimum and the maximum node sizes of a T Tree node are 8 and 10, respectively, the bucket size of Linear Hashing is 20, and the bucket size of Chained Bucket Hashing is 5.

4.1 Experimental Results

We performed a series of experiments to compare the techniques discussed in Section 3. All the experiments were run on a DELL Precision Workstation with 1.8GHz Pentium 4 CPU and 512M memory. The alerts used in our experiments were generated by a RealSecure Network Sensor 6.0 [14], which monitors an isolated network in which we replayed the network traffic collected at the DEF CON 8 CTF event [11]. 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.

In these experiments, we mapped each alert type reported by the RealSecure Network Sensor to a hyper-alert type (with the same name), 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, among which 52,318 hyper-alerts have prerequisite or consequence. The remaining hyper-alerts are mainly *Windows_Access_Error* and *IPDuplicate*, which we decided to ignore due to their overly general semantics. These hyper-alerts cannot be correlated with any other ones, and do not contribute to the time required by alert correlation. In order to precisely evaluate the relationship between the execution time and the number of hyper-alerts, we did not include them in our experiments.

4.1.1 Nested-Loop Correlation without Memory Constraint

Our first set of experiments was intended to evaluate the effectiveness of hyper-alert container in the nested loop correlation method. 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.

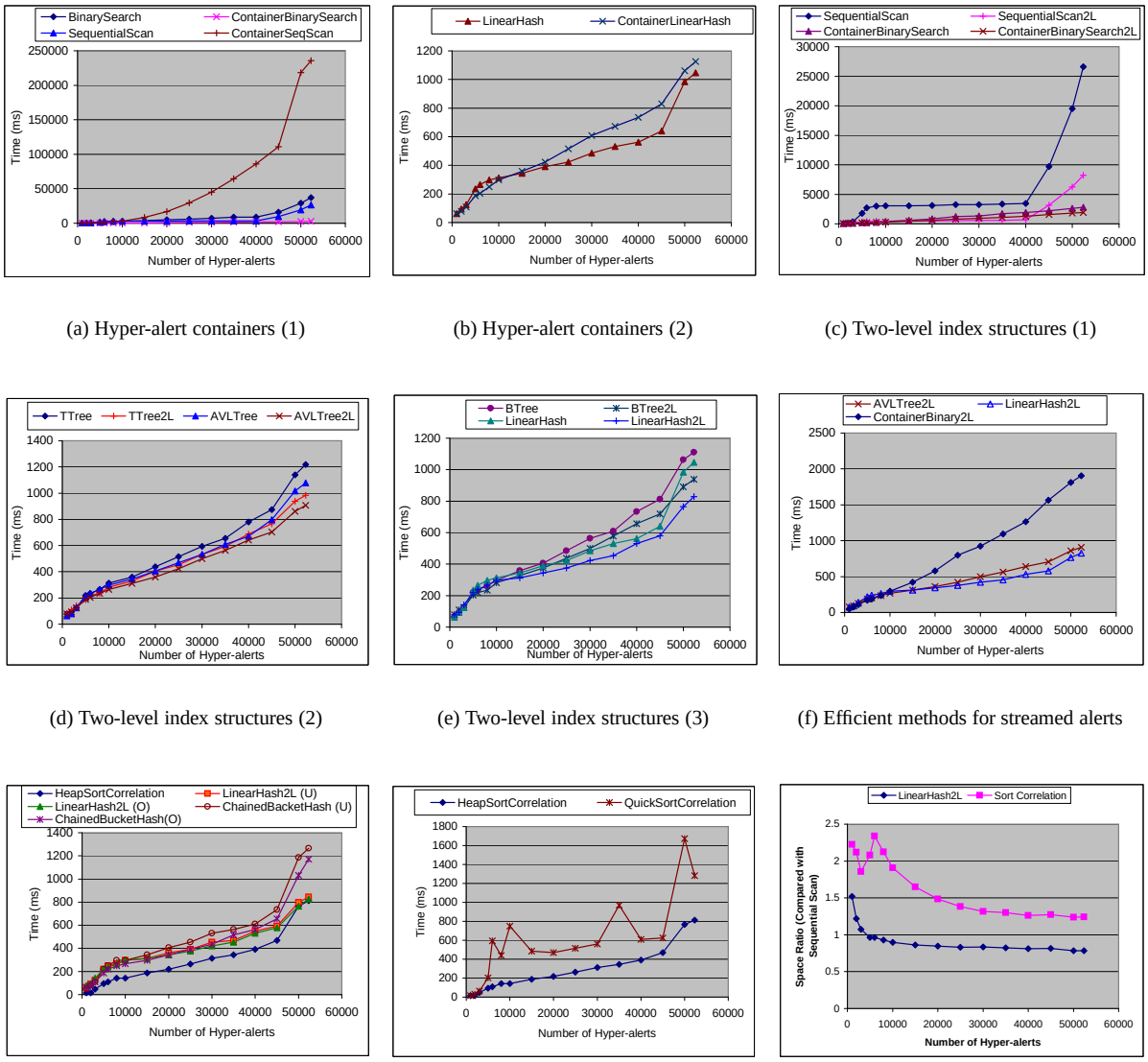
Our second set of experiments was intended to evaluate the effectiveness of two-level index structure in the nested loop correlation method. According to our analysis and the earlier experimental results, we used hyper-alert container in Array Binary Search and tree index structures, but not in Sequential 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 other words, the difference between insertion and search cost and the fact that there is not many searches for the hyper-alerts between 8,000 and 40,000 resulted in the flat area in Figure 4(c). 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.

4.1.2 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. For the index structures other than Chained Bucket Hashing, knowing the hyper-alerts before alert correlation will not change anything in the index structures. Thus, we believe their relative performance



(a) Hyper-alert containers (1)

(b) Hyper-alert containers (2)

(c) Two-level index structures (1)

(d) Two-level index structures (2)

(e) Two-level index structures (3)

(f) Efficient methods for streamed alerts

(g) Efficient methods for batch alerts

(h) Impact of sorting algorithms

(i) Space overhead

Figure 4: Experimental results

will not change for batch alert correlation. However, knowing how many hyper-alerts gives more information for Chained Bucket Hashing, since we can estimate the number of elements to be inserted into the hash table and thus have a good guess about the desired size of the hash table. In our experiments, we chose to set the hash table size the same as the number of input hyper-alerts. Moreover, the sort correlation method can potentially outperform nested loop correlation with two-level Linear Hashing, since it adopts a different way to correlate the hyper-alerts. Thus, we decided to compare the execution time of nested loop correlation with two-level Linear Hashing, nested loop with Chained Bucket Hashing, and sort correlation. To further examine the impact of the time order of input hyper-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. The time order of input does not have any impact on sort correlation.

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 four 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 [6]. 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.

4.1.3 Space Utilization

We examined the space overhead of these methods by comparing their space requirements with the sequential scan method. We use a quantitative measure *space ratio* = $\frac{\#bytes\ to\ use\ the\ method}{\#bytes\ to\ use\ sequential\ scan}$ for this purpose. As shown in Figure 4(i), the highest space ratios of sort correlation and nested loop correlation with two-level Linear Hashing are 2.34 and 1.52, respectively. Sort correlation requires about twice space as nested loop with two-level Linear Hashing, since it has to store instantiated predicates in both prerequisite and expanded consequence sets. In addition, nested loop correlation with two-level Linear Hashing requires more space than sequential scan when the input size is small, but less space when the input size is large. This is because two-level Linear Hashing usually has wasted cells in the hash table when the input size is small. When the input size is large, not only the hash table is better utilized, but storing predicate names in the top level index can also reduce the storage requirement. The spike in the line of sort correlation is due to the irregular distribution of instantiated predicates in the prerequisite sets, which are only saved in sort correlation, but not in the nested loop correlation method. (There is a sudden increase of hyper-alerts that only have prerequisites between 3,000 and 6,000 input hyper-alerts.)

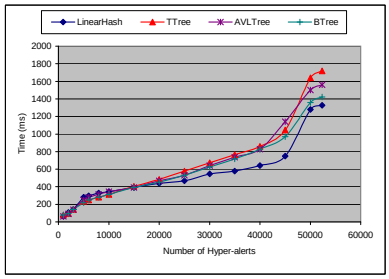
4.1.4 Nested-Loop Correlation with Memory Constraint

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.

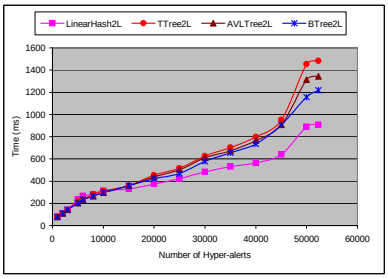
In this set of experiments, we first use a sliding window of size 30,000 to compare the execution time for different number of input hyper-alerts. As shown in Figure 5(a), when the two-level index structure is not used, Linear Hashing has the best performance compared with the three tree based indexing structures. Figure 5(b) shows a similar performance order, when the two level index structure is used. We also notice that B Trees perform the best among the tree based index structures, whereas AVL Tree is the best when there is no memory constraint. Our further investigation indicates that the deletion algorithm of AVL Tree is not only more complex than that of B Tree, but also more complex than the insertion algorithm of AVL Tree. In an AVL Tree, one deletion may trigger several subtree rotations. As a result, more operations are need to rebalance the tree. Figure 5(c) further shows the comparison of Linear Hashing and B Tree with and without the two-level index structure. The result shows that the two-level index does improve the efficiency of the index structures, and two-level Linear Hashing is the most efficient one among all the index structures.

To reconfirm the performance results, we perform another set of experiments with varying sliding window sizes, using all of the hyper-alerts as input. Figures 5(d), 5(e), and 5(f) 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.

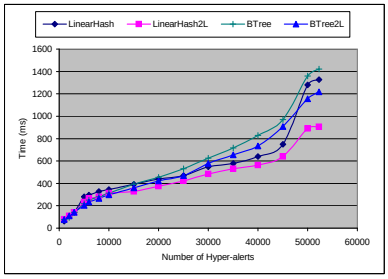
An interesting observation is that there is a bump in the line for T Tree in both Figure 5(d) and Figure 5(e) when the window size is between 15,000 and 25,000. Our investigation reveals that the numbers of node balancing operations



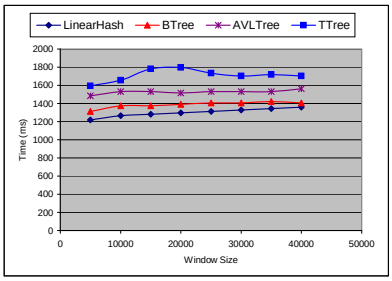
(a) Single-level index structures with fixed window size



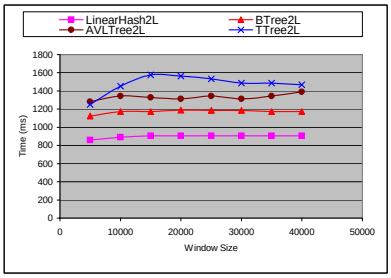
(b) Two-level index structures with fixed window size



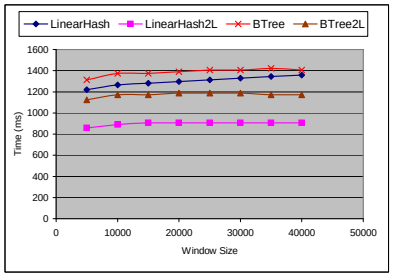
(c) Selected efficient index structures with fixed window size



(d) Single-level index structures with varying window size



(e) Two-level index structures with varying window size



(f) Selected efficient index structures with varying window size

Figure 5: Experimental results of correlations with memory constraint

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

Intrusion detection has been studied for about twenty years. An excellent overview of intrusion detection techniques and related issues can be found in a recent book [3].

The result reported in this paper is a further step of our work in [22] and [21], which has been described in Section 2. Our method was initially developed to address the limitation of JIGSAW [26]. Our method has several features beyond JIGSAW. First, our method allows partial satisfaction of prerequisites (i.e., required capabilities in JIGSAW [26]), recognizing the possibility of undetected attacks and that of attackers gaining information through non-intrusive ways (e.g., talking to a friend working in the victim organization), while JIGSAW requires all required capabilities be satisfied. Second, our method allows aggregation of alerts, and thus can reduce the complexity involved in alert analysis, while JIGSAW currently does not have any similar mechanisms. Third, we have implemented our method and developed a set of utilities for interactive analysis of correlated alerts, which are not available in JIGSAW.

The work closest to ours is the alert correlation method by Cuppens and Miege in the context of MIRADOR project [7], 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 database described in LAMBDA [8]. 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 the three utilities for interactive alert analysis [21]. In addition, we have built a toolkit based on our model, and studied ways to improve the efficiency of this toolkit in this paper.

Several other alert correlation methods have been proposed. Spice [25] and the probabilistic alert correlation method [28] correlates alerts based on the similarities between alert attributes. Though they are effective for correlating some alerts (e.g., alerts with the same source and destination IP addresses), they cannot fully discover the causal relationships between related alerts. Another type of alert correlation methods (e.g., the data mining approach [9]) 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 [10].

Several languages have been proposed to represent attacks, including STAT [29, 13], Colored-Petri Automata (CPA), and MuSig [17] and its successor [23]. However, all these languages specify entire attack scenarios, which are limited to known scenarios. In contrast, our method (as well as JIGSAW and the MIRADOR approach) describes prerequisites and consequences of individual attacks, and correlates detected attacks based on the relationship between the prerequisites and consequences. Thus, our method can potentially correlate alerts from unknown attack scenarios.

6 Conclusion and Future Work

This paper studied main memory index structures and database query optimization techniques to facilitate timely correlation of intensive alerts. In addition to experimental study of the performance of various main memory index structures, this paper presented 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, (4) sort correlation with heap sort algorithm is the most efficient for alert correlation in batch, and (5) two-level Linear Hashing is the most efficient for alert correlation when sliding window is used to cope with memory constraint. 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 *Proceedings of IEEE COMPCON*, San Francisco, February 1985.

[3] R.G. Bace. *Intrusion Detection*. Macmillan Technology Publishing, 2000.

[4] D. Barbará, N. Wu, and S. Jajodia. Detecting novel network intrusion using bayes estimators. In *Proceedings of the First SIAM Conference on Data Mining*, April 2001.

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

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

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

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

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

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

[11] DEFCON. Def con capture the flag (CTF) contest. http://www.defcon.org/html/defcon-8-post.html, July 2000. Archive accessible at http://wi2600.org/mediawhore/mirrors/shmoo/.

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

[13] K. Ilgun, R. A. Kemmerer, and P. A. Porras. State transition analysis: A rule-based intrusion detection approach. *IEEE Transaction on Software Engineering*, 21(3):181–199, 1995.

[14] ISS, Inc. RealSecure intrusion detection system. http://www.iss.net.

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

[16] T. J. Lehman and M. J. Carey. A study of index structure for main memory database management systems. In *Proceedings of the Twelfth International Conference on Very Large Databases*, pages 294–303, Kyoto, Japan, August 1986.

[17] J. Lin, X. S. Wang, and S. Jajodia. Abstraction-based misuse detection: High-level specifications and adaptable strategies. In *Proceedings of the 11th Computer Security Foundations Workshop*, pages 190–201, Rockport, MA, June 1998.

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

[19] MIT Lincoln Lab. 2000 DARPA intrusion detection scenario specific datasets. `http://www.ll.mit.edu/IST/ideval/data/2000/2000_data_index.html`, 2000.

[20] P. Ning and Y. Cui. An intrusion alert correlator based on prerequisites of intrusions. Technical Report TR-2002-01, North Carolina State University, Department of Computer Science, January 2002.

[21] P. Ning, Y. Cui, and D. S Reeves. Analyzing intensive intrusion alerts via correlation. In *Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection (RAID 2002)*, Zurich, Switzerland, October 2002.

[22] 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*, Washington, D.C., November 2002.

[23] P. Ning, S. Jajodia, and X. S. Wang. Abstraction-based intrusion detection in distributed environments. *ACM Transactions on Information and System Security*, 4(4):407–452, November 2001.

[24] P. Ning, D. S. Reeves, and Y. Cui. Correlating alerts using prerequisites of intrusions. Technical Report TR-2001-13, North Carolina State University, Department of Computer Science, December 2001.

[25] S. Staniford, J.A. Hoagland, and J.M. McAlerney. Practical automated detection of stealthy portscans. To appear in Journal of Computer Security, 2002.

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

[27] J. D. Ullman. *Principles of database and knowledge-base systems*, volume 2. Computer Science Press, 1989.

[28] A. Valdes and K. Skinner. Probabilistic alert correlation. In *Proceedings of the 4th International Symposium on Recent Advances in Intrusion Detection (RAID 2001)*, pages 54–68, 2001.

[29] G. Vigna and R. A. Kemmerer. NetSTAT: A network-based intrusion detection system. *Journal of Computer Security*, 7(1):37–71, 1999.