Real-time Alert Correlation Using Stream Data Mining Techniques

Reza Sadoddin and **Ali A. Ghorbani**
Information Security Centre of Excellence
Faculty of Computer Science, University of New Brunswick
Fredericton, NB, E3B 5A3, Canada
Email : {reza.sadoddin, ghorbani}@unb.ca

Abstract

With the large volume of alerts produced by low-level detectors, management of intrusion alerts is becoming more challenging. Alert Correlation addresses this issue by providing a condensed, yet more useful view of the network from the intrusion standpoint. In this paper, we propose a new framework for real-time alert correlation that incorporates novel techniques for aggregating alerts into structured patterns and incremental mining of frequent structured patterns. In the proposed framework, time-sensitive statistical relationships between alerts are maintained in an efficient data structure and are updated incrementally to reflect the latest trends of patterns. The results of experiments with synthetic and real-world datasets demonstrate the efficiency of the proposed techniques. Our Frequent Structure Mining algorithm scales linearly with the size of the dataset and the proposed framework can cope with the throughput of a large-scale network. The ability to answer time-sensitive queries about patterns is another advantage of this work compared to other methods.

Introduction

Intrusion detection is one of the major techniques for protecting information systems. Intrusion detection systems raise alerts when the network traffic either matches with a known attack scenario (Signature-based detection) or deviates from the previously learned normal behavior of the network (Anomaly-based detection). A signature-based technique has a lower false positive rate but it is intended for detecting known attacks. An anomaly-based technique, on the other hand, has the potential to detect novel attacks, but it suffers from a high false positive rate.

In order to compensate for limitations of individual intrusion detection systems, sensors with various detection mechanisms and other security tools (*firewalls, antivirus utilities*) are deployed in different locations of the protected network. Higher level management is required for analyzing low-level alerts produced by these devices for several reasons. First, the low-level sensors consider raw alerts in isolation and raise alarms for each of them, without considering logical connections between alerts. Secondly, in a typical environment there are a lot of false positive alerts reported by traditional IDSs, which are mixed with true positive alerts.

Alert Correlation provides the system with automatic analysis of alerts. An alert correlation module not only saves a lot of time on the administrative side, but also helps to deal with potential threats against the network more efficiently. Previous works in this area have addressed different aspects of alert correlation including filtering false positives, aggregating similar alerts into clusters and correlating alerts based on their relationship in an attack scenario.

Two main approaches have been used in the literature for correlating alerts in attack scenarios. In the first category of works, the relationships between alerts in the well-known attack scenarios are hard-coded in the system. These methods are limited to the knowledgebase of the system and cannot correlate alerts issued due to unseen attacks. To overcome this problem, machine learning and data mining techniques have been used to extract relationships between alerts automatically. In this category of works, temporal co-occurrence of alerts is used as an important feature for the statistical analysis of alerts. This involves pair-wise comparison between alerts since every two alerts are candidates to be correlated in a newly emerging pattern. Pair-wise comparison between alerts poses serious challenges on learning-based correlation techniques in large-scale networks, in which a throughput of 5,000 alerts per minute is expectable.

In this paper, we address the problem of real-time correlation of alerts in a learning-based approach. A new framework is proposed for real-time correlation of alerts based on their frequency of co-occurrences. The proposed framework provides a means for processing a stream of alerts continuously, maintaining their correlation significance with respect to the latest changes, and answering interesting time-sensitive queries about patterns. Our approach to dynamic creation and maintenance of patterns results in an abstract view of the most significant patterns along with their structures that contribute to the continuous stream of alerts.

Related Work

Different aspects of alert correlation have been addressed in the literature, however the most related ones to our work are aggregating alerts into clusters and extracting causality relationships between alerts. In aggregation, alerts are put into a group based on the similarity of their corresponding features. The most common attributes of alerts are *Source IP*, *Destination IP*, *Source Port*, *Destination Port*, *Attack Class* and *Timestamp*.

Valdes *et al.* propose grouping alerts with each other based on their overall similarity (Valdes & Skinner 2001). The overall similarity of two alerts is defined based on their similarities on the corresponding features. The proposed technique, even though it provides a basic probabilistic model for measuring similarities between alerts, has the drawback of relying on experts for specifying the similarity degree between attack classes manually.

A clustering technique is proposed by Julisch for grouping similar alerts (Julisch 2003). Central to this technique are the hierarchy structures (*generalization hierarchies* as they are called), which decompose the attributes of the alerts from the most general values to the most specific ones. The idea of generalization hierarchies seems promising for aggregation, but the proposed method has the drawback of assuming a predetermined size for all the clusters.

Finding causal relationships between alerts is another aspect of correlation. Previous works on causality analysis can be divided into two categories:

- **Knowledge-based correlation**: In this category of works, causality relationships between alerts are either specified by an attack language such as STATL (Eckmann, Vigna, & Kemmerer 2002) or LAMBDA (Cuppens & Ortalo 2000), encoded in terms of correlation rules (Templeton & Levitt 2000; Cuppens & Miege 2002), or extracted automatically using machine learning techniques (Dain & Cunningham 2001).

- **Statistical correlation**: Methods in this group correlate two alerts if they are statistically related to each other based on their temporal co-occurrence. Qin *et al.* model the correlation between alerts by Bayesian Networks and propose a technique for learning the structure of the network as well as the amount of correlation between alerts (Qin 2005). In another reported work, *Granger Causality Test* is used for statistical analysis of alerts (Qin & Lee 2003). Each type of alert is modeled by a time series variable representing the alert rate within equal time slots. The Granger Causality Test is used as an statistical test to see whether an alert of type *X* provides significant information about an alert of type *Y*. Zhu *et al.* propose two machine learning techniques (Multilayer Perceptron and Support Vector Machine) for estimating *correlation probability* between alerts (Zhu & Ghorbani 2006). The proposed features by the authors are similar to features proposed by Valdes *et al.* (Valdes & Skinner 2001), but the correlation degree between attack classes are learned incrementally (as apposed to having the experts specify it).

Generally, the knowledge-based methods are limited to the available expert knowledge that should be hard-coded in the system, whereas the statistical techniques are capable of correlating alerts that may contribute to unknown attacks. On the other hand, in practice these techniques seem to be less accurate and more time consuming compared to knowledge-based methods.

Proposed Framework for Incremental Frequent Structure Mining

Figure 1 shows our proposed framework for incremental alert correlation that is inspired by three major requirements. First, the framework should cope with the high rate of alerts. Secondly, incremental development of patterns is necessary as not all the alerts that contribute to a pattern are received simultaneously. Moreover, a statistical approach to alert correlation should deal with "*concept drift*".

Raw alerts are received continuously at the *Aggregation* component. This component correlates alerts into graph structures based on their connectivity information with respect to origin and target of alerts in the network. Each of these *structured patterns* might represent attack strategies launched step by step by an attacker or normal patterns generated due to false positive alerts. Constructed patterns are allowed to change dynamically until they become stable. Signatures of *stable* structured patterns are passed to the *Frequent Structure Mining* component as a new batch of transactions for further processing. This component takes care of extracting frequent patterns among the recent transactions and updating a running model with respect to recently seen transactions of alerts. The framework provides the system administrator with two interfaces. He/she might see all the significant patterns by browsing the tree or can query the running model for a specific set of patterns.

Aggregation

Attackers may launch their attacks in different compositions based on the resources available to them and their final goals. For instance, in a distributed denial of service (*DDoS*) attack, attackers take advantage of some intermediate victim hosts (such as *Zombies*) in order to flood their final victim (see Figure 2). The number of involved intermediate hosts depends on the number of required machines for saturating the final host and the number of hosts the attacker is successful in compromising. However, all of these attacks render a similar pattern: A set of alerts from one source to several destinations labeled as *XXX_Exploit* (the exploit for vulnerable service *XXX*) followed by a set of alerts from several sources to a single destination labeled with *DoS_Stream*.

In order to capture the common characteristics between different compositions of frequent patterns, three types of generalization rules are considered:

- *Many-to-one* generalization is the abstraction of a set of alerts of the same type, the same destination IP, and multiple source IPs. The generalized set of alerts is referred to as Many-to-one Hyper Alert and is represented by $A^{*>}$.
- *One-to-many* generalization, represented by $A^{*<}$, is defined similarly for a set of alerts of the same type, the same source IP and multiple destination IPs.
- *Many-to-Many* generalization, represented by $A^{*=}$, is the abstraction of a set of alerts of the same type, the same source IP, and the same destination IP.

The resulting graph structure after application of possible generalization rules is referred to as *Virtual Pattern*. Even

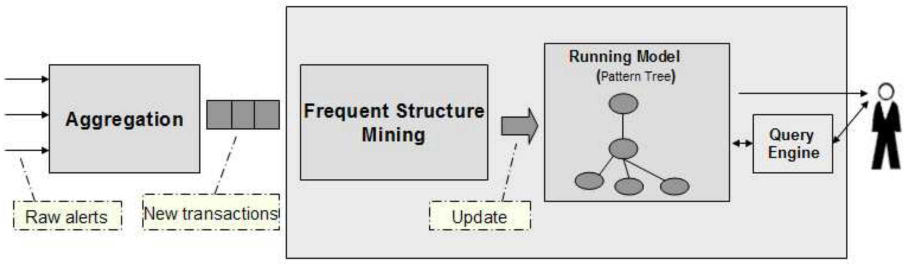


Figure 1: The Incremental Framework for Alert Correlation.

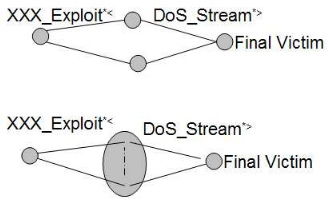


Figure 2: Variants of attack patterns (DDoS)

though we model patterns by a graph data structure, a one-to-one mapping between hosts and individual alerts of the network to nodes and edges of the graph is not possible due to the possibility of generalization. A real-time correlation system should be able to construct the frequent patterns incrementally. A hash table is used for adding newly received alerts to a potential pattern that has been created already. The *pattern hash table* provides the mapping between the IP addresses of real hosts and their corresponding virtual nodes in virtual patterns. Figure 3 represents how a pattern in a real-world network is mapped to a virtual pattern through a hash table. In this figure, three actual alerts of type A are aggregated into hyper alert $A^{*>}$ while their corresponding source hosts are mapped to the virtual node v_1. When an alert is received by the aggregator, it is hashed using its source and destination IP addresses as separate keys to the hash table. If the corresponding cells for both source and destination IPs in the hash table are empty, it means that none of the source and destination hosts are currently registered with a pattern in the system. In this case, a new pattern is created and is registered in the hash table with its source IP and destination IP as the keys. If a pattern is already registered in the hash table with corresponding cells to either source IP or destination IP, the alert is absorbed by the registered pattern. Absorbtion involves considering the new pattern for possible generalization rules and updating the structures of the pattern and hash table accordingly.

A pattern is assumed to be *stable* if it is not changed for a period longer than a user-specified time threshold referred to as *Keep_Active*. Stable patterns are appended to the list of stable transactions and delivered to the next component of the framework for further processing after being unregistered from the hash table.

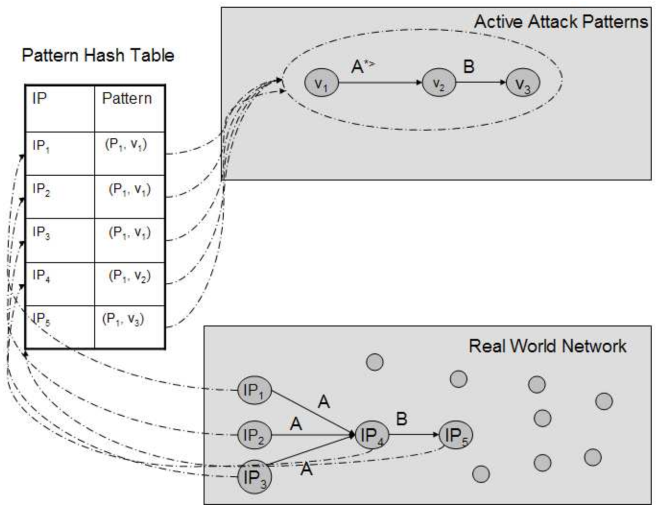


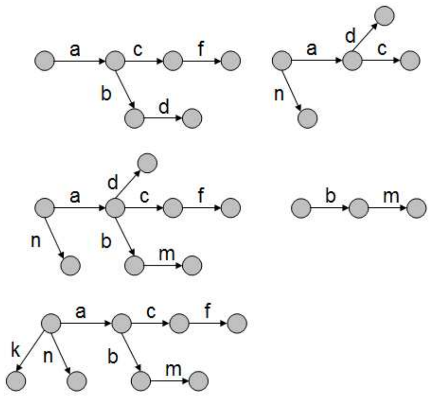
Figure 3: Pattern Hash Table.

Mining Frequent Graph Structures

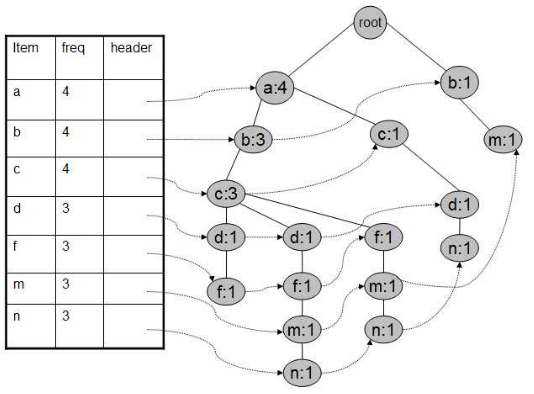
In our proposed approach, the transactions (structured patterns) are created based on the connectivity information of the corresponding alerts. In this way, the constructed transactions might include some additional steps that are not actually parts of the pattern. This might happen as a result of accidental co-occurrence of alerts within a time window. Statistical analysis of the candidate frequent patterns mitigates the mentioned problem by selecting the most significant patterns.

Our approach to mining frequent structures is based on the *FP_Growth* algorithm (Han, Pei, & Yin 2000). The FP_Growth algorithm makes use of *FP_Tree*, which provides a compact data structure for storing candidate frequent patterns. Once the initial FP_Tree is constructed from the dataset, the FP_Growth algorithm mines the frequent patterns recursively by adding one item to it at a time.

The *FP_Tree* data structure and *FP_Growth* algorithm were designed for mining simple transactions (as opposed to structured patterns). Two main issues should be addressed in using the *FP_Tree* data structure in mining frequent structured patterns. First, the structure of the graphs should be encoded in the inserted transactions in the tree. Secondly, frequent patterns in each recursion level of the min-



(a) Initial structured patterns.



(b) Initial FSP_Tree.

Figure 4: Sample FSP_Tree constructed from structured patterns.

ing algorithm should be expanded by taking into account the connectivity information of a candidate item to the current structured pattern. To address these issues, we propose the *FSP_Tree* (*Frequent Structured Pattern Tree*) data structure, which is extended from the *FP_Tree* structure, along with the *FSP_Growth* algorithm for mining frequent structured patterns from it.

Similar to *FP_Tree*, a transaction is inserted in the *FSP_Tree* based on the frequency order of its items, *i.e* the items with higher frequency are inserted first. The main difference between constructing the *FSP_Tree* and *FP_Tree* is that transactions are inserted in the former tree considering their structures. To insert a new transaction, an existing path of the tree is used as long as the structure of the pattern matches with the structure encoded along the path and the frequencies of the nodes along the matching path are incremented accordingly. Otherwise, the insertion is continued by creating a new branch in the tree. Figure 4(b) shows the initial *FSP_Tree* constructed from the structured patterns in Figure 4(a). Similar to *FP_Growth*, the *FSP_Growth* algorithm starts mining frequent patterns from the bottom of the tree and recursively expands the patterns taking into account their structures. Further details of the *FSP_Growth* algorithm are available in (Sadoddin & Ghorbani 2008).

Maintaining Alert Streams at Multiple Time Granularities

The proposed structured pattern mining algorithm (*FSP_Tree*) extracts the frequent patterns from the recent batch of transactions. Dealing with a stream of alerts involves maintaining a running model of the statistical relationships between the alerts. The running model should not only summarize the significant patterns emerged so far, but also be updated incrementally to reflect the latest trends of the alerts.

To achieve these goals, we adopt the technique proposed by Giannella *et al.* (Giannella *et al.* 2002). Using this technique, all the frequent structured patterns can be represented in a compact tree data structure. The *pattern tree* encodes the most significant patterns along with their time-sensitive information. The time-sensitive information of patterns are maintained by the *tilted-time window* model. Each tilted-time window basically contains time frames of multiple granularities. An example of a tilted-time window is shown in Figure 5. In this model, the frequencies of recent patterns, which occurred in the past quarter of an hour, are maintained at fine granularity while older patterns are kept at coarser granularities (*i.e.* hour and day).

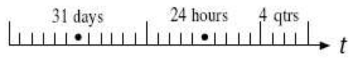


Figure 5: Sample tilted-time window (Giannella *et al.* 2002).

Each node in the pattern tree represents a pattern/sub_pattern (from root to this node). As shown in Figure 6, a tilted-time window table is associated with each node of the pattern tree that contains its time-sensitive information. The top rows of the table contain the fine-grained information while the bottom rows encode information at coarser granularities. In the current design, the time-sensitive information includes the frequency and *IP ranking table* of a pattern. However, other information of interest can be included with minor changes. The IP ranking table maintains the most frequent pairs of (Source IP, Destination IP) that have contributed to creation of the corresponding pattern.

Figure 7 illustrates the incremental process of mining the alert stream. Stable transactions (candidate frequent patterns) are periodically passed to the Frequent Structure Mining component. This period mainly depends on the smallest windows in which frequent patterns should be reported. Every batch of recent transactions is processed by the Frequent Structure Mining component and the running model (*pattern tree*) is updated simultaneously. An update procedure is proposed by Giannella *et al.* that incorporates a pruning mech-

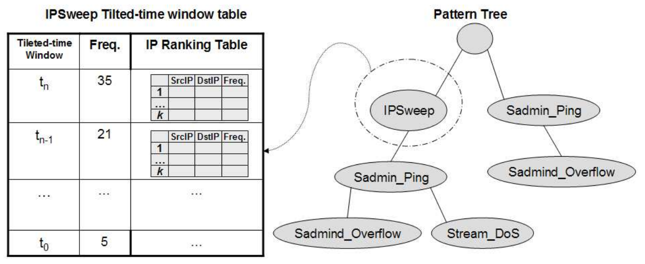


Figure 6: Pattern tree structure.

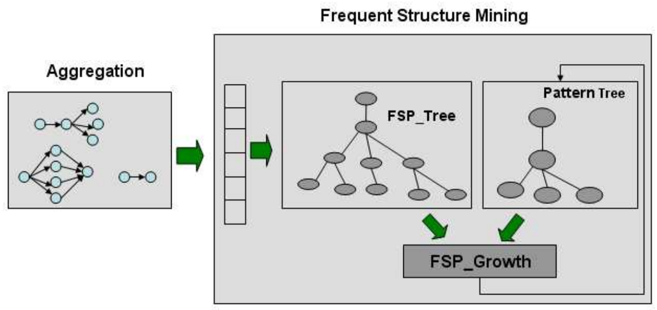


Figure 7: Incremental mining architecture.

anism for dropping the tilted-time windows (and even tree nodes) corresponding to *infrequent* patterns (i.e. the patterns that were frequent in the past, but are not frequent anymore considering the latest transactions). The pruning results in losing accuracy in frequency. However, the frequency error (ϵ) of the patterns in each time window is guaranteed to be less than a user-specified parameter (Giannella *et al.* 2002).

Test and Evaluation

The proposed framework was evaluated in two steps. The performance of the FSP_Growth algorithm was evaluated using an artificial random graph data generator. In addition, we replayed the alerts collected by Snort in Fred-eZone (the WiFi network provided by the City of Fredericton, NB, Canada) into our system in order to evaluate the proposed framework against alert logs of a real world network.

Experiments with Synthetic Datasets

The purpose of this experiment is to evaluate the performance of the FSP_Growth algorithm in terms of its time complexity with respect to the size of the dataset. A sample model for generating synthetic graph data was proposed by Kuramochi *et al.* with the following parameters (Kuramochi & Karypis 2001):

1. $|D|$: The total number of graphs.
2. $|T|$: The average size of graphs (in terms of the number of edges).
3. $|L|$: The number of potentially frequent subgraphs.
4. $|I|$: The average size of potentially frequent subgraphs (in terms of the number of edges).
5. $|N|$: The number of labels (assigned to edges).

The synthetic graph generator first creates a pool of $|L|$ potentially frequent subgraphs with the average size of $|I|$, which is called *seed pool*. Following that, a total number of $|D|$ graphs with the average size of $|T|$ are constructed by selecting potentially frequent subgraphs from the seed pool.

A similar model is adopted for generating input graphs for our experiments. Table 1 shows different values of the parameters used in the experiments. The labels are selected from a label pool of size 100. This parameter corresponds to the different attack classes used in the simulation. For each parameter setting, 10 different random datasets are created and the performance of the algorithm is reported based on the average of 10 trials. Since the performance of the algorithm depends on the number of reported frequent structures, we exclude the trials that returned significantly more frequent patterns compared to the average number of frequent patterns returned by other trials. The reason for excluding these trials is to make a fair comparison between the average running times with increasing size of the dataset. The minimum support value was set to 0.5%. The experiments were performed on a 1.86GHz Intel Xeon machine with 2GB main memory running the Ubuntu operating system.

Parameter	Value
$\lvert D\rvert$	10000, 20000, 30000, 40000, 50000, 60000
$\lvert T\rvert$	5, 10, 15
$\lvert I\rvert$	3, 5
$\lvert L\rvert$	200

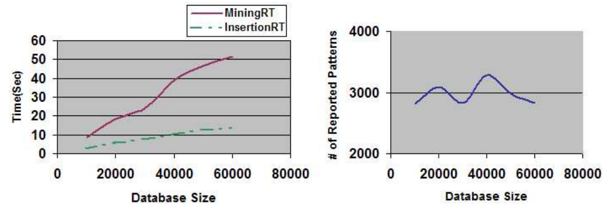
Table 1: Synthetic data parameter settings.

Figure 8(a) shows the results of experiments for the parameter setting ($|L|$,$|T|$, $|I|$) = $(200, 10, 3)$ (referred to as EX_200_10_3). In this figure, axes *X* and *Y* represent the size of the dataset and the running time in seconds, respectively, and solid and dash-dot-dot lines represent the mining time and the insertions time, respectively (insertion time represents the time required for creating the initial FSP_Tree). The average number of reported patterns is shown in the right curve of Figure 8(a). It is seen from the figures that both the mining time and the insertion time *roughly* scale linearly with the size of the dataset. The behavior of the algorithm should be analyzed by taking the average number of reported patterns into account. This effect can be verified in Figure 8(a). The running times of the algorithm are slightly less than their expected values at points 30,000 and 60,000, which correspond to the dropping points in the number of reported patterns in the right curve of Figure 8(a).

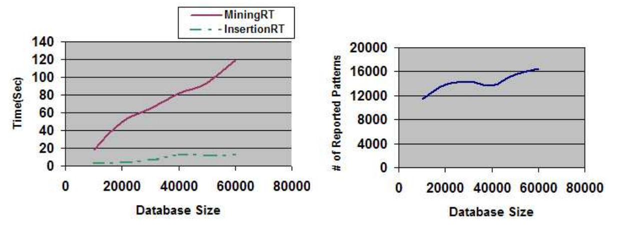
Experiments with a Real World Dataset

For this experiment, we used the alert logs collected by a Snort box in the Fred-eZone. The logs include 3,208,788 raw alerts within a period of one month.

Experiment Setup We replayed the alert logs into our system using a simple Log Scanner. The alerts are received by the framework one by one and the current time of the system is changed accordingly. Moreover, alert signatures that



(a) EX_200_10_3



(b) EX_200_10_5

Figure 8: Runtime and Number of Reported Patterns vs. Dataset Size.

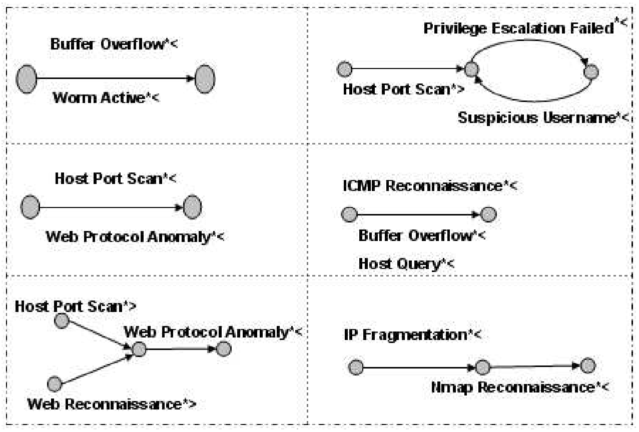


Figure 9: Some of the extracted patterns in the end of the month ($length(pattern) \geq 2$).

are reported by Snort are mapped to high-level categories of events using QRadar© (a Security Information Management product provided by Q1Labs Inc.[1]). The following parameters are used for processing the alert stream:

- *Keep_Active* is set to 60 seconds, which basically means that patterns are assumed to be stable if they are not changed for a period longer than 60 seconds.
- The value of Mining Interval is set to 15 minutes (*i.e.* stable patterns are mined every 15 minutes and the pattern tree structure is updated accordingly).
- The values of σ (minimum support) and ϵ (maximum support error) are set to 0.15 and 0.05, respectively.

Experimental Results Figure 9 shows some of the patterns included in the pattern tree in the end of the month. The patterns of length one that were not correlated with other alerts are excluded in this list. The nodes in each pattern correspond to a host (or a set of hosts in case of generalization) and edges correspond to a single alert or hyper alerts issued between hosts. The text labels belong to edges and represent the high-level attack classes. Moreover, notations $*>$, $*<$ and $*=$ stand for three kinds of generalized hyper alerts introduced earlier. Among the extracted patterns, some well-known attack patterns such as

1. $Buffer\ Overflow^{*<} \rightarrow Worm\ Active^{*<}$
2. $Host\ Query^{*<} \rightarrow ICMP\ Reconnaissance^{*<} \rightarrow Buffer\ Overflow^{*<}$

are mixed with other unknown patterns such as

$$\begin{aligned} IP\ Fragmentation^{*<} &\rightarrow \\ Privilege\ Escalation\ Failed^{*<} &\rightarrow \\ Suspicious\ Username^{*<} \end{aligned}$$

whose descriptions require further investigation.

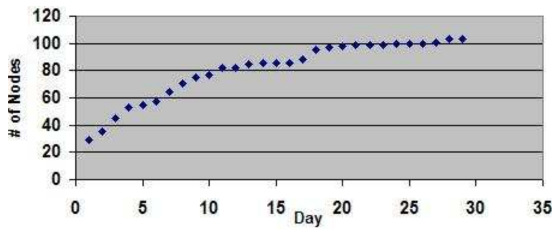
Figures 10(a) and 10(b) represent the size of pattern tree in terms of the number of nodes and number of tilted-time windows in different days, respectively. As it is seen in the figures, the size of tree increases within the first two weeks of the month and becomes more stable after that. These results are expectable as the number of *emerging* patterns is more than *disappearing* patterns at first, but they will compensate for each other after a while. The only exception is a sharp decrease in the number of tilted-time windows in day 24. Further investigation of alert logs showed us that no alerts were received for a period of around one day beginning from the end of the day 23.

Running on the same machine as that of the previous experiments, it takes our system around 8 minutes to process a stream of alerts for one month, which contains 3,208,788 raw alerts. This will amount to a throughput of 401,098 number of alerts in one minute, which is much more than the expected alert throughput in large scale networks (Having discussed it with network security experts, we believe that the alert throughput in large networks is less than 10000 per minute). Considering the fact that the Frequent Structure Mining component is the bottleneck of the framework and given the linearly scalable running time of the FSP_Growth algorithm with the size of the dataset, the proposed framework can cope with the alert throughput of large scale networks as well.

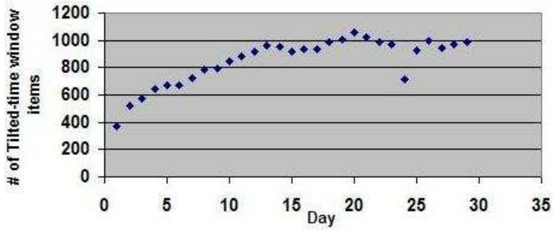
Note that each node in the *pattern tree* encodes the connectivity information of that node to its previous nodes in the path from the root (in addition to the tilted-time window table). We use this information to reconstruct the patterns stored in the tree. The size of the pattern tree is controlled through a pruning mechanism mentioned earlier. The pruning facility along with automatic summarization of tilted-time windows of old transactions makes the *pattern tree* an efficient data structure for maintaining frequent patterns.

Maintaining the alert stream in the *pattern tree* not only reflects the latest trends of patterns, but also provides a means for answering some time-sensitive interesting queries. For instance, the following queries can be answered efficiently using the pattern tree:

[1] *http://www.q1labs.com/*



(a) Number of nodes of the pattern tree.



(b) Number of tilted-time windows of pattern tree.

Figure 10: Size of the pattern tree in different days.

- Which alerts co-occurred with pattern $\underline{p}$ in hour $\underline{h}$, where $\underline{p}$ is an arbitrary pattern (*e.g.* $\underline{\text{Network Sweep}^{*<} \rightarrow \text{Web Exploit}^{*<}}$).
- Return the tilted-time windows in which pattern $\underline{p}$ occurred with support $\underline{\sigma_1}$, where $\sigma_1 >= \sigma$ (σ_1 is the specified support in the query and σ is the support of the incremental mining framework).

Conclusion and Future Work

A new framework for real-time analysis of intrusion alerts was proposed. The Aggregation component provides an abstract view of the candidate patterns whose significance is analyzed by a frequent structure mining technique periodically. The pattern tree contains the most significant patterns seen in the network along with other supplementary information. Maintaining the frequent patterns of alerts in the pattern tree not only reflects the latest trends of patterns, but also provides a means for answering interesting time-sensitive queries. The results of experiments with synthetic and real-world alert data show that the proposed framework can cope with the throughput of a large-scale network in terms of processing speed and memory management.

Even though the proposed framework can extract novel patterns of alerts, new attack strategies are mixed with false positive patterns. Developing a component for filtering false positive patterns would be of great benefit to the security administrator. Moreover, discovering slowly developing patterns (i.e. the patterns with large delays between their consecutive steps) is a difficult task for correlation systems since almost all of them incorporate the time difference between alerts in correlating them. Developing techniques for correlating alerts with adaptive time windows (which works well for different types of patterns) will improve the completeness of the mining process.

Acknowledgments

This work was funded by the Atlantic Canada Opportunity Agency (ACOA) through the Atlantic Innovation Fund (AIF). The authors are thankful to the City of Fredericton and Q1Labs Inc. for providing real world alert logs.

References

Cuppens, F., and Miege, A. 2002. Alert correlation in a cooperative intrusion detection framework. In *IEEE Symposium on Security and Privacy*, 202–215.

Cuppens, F., and Ortalo, R. 2000. Lambda: A language to model a database for detection of attacks. In *RAID*, volume 1907, 197–216. Toulouse, France: LNCS, Springer-Verlag Heidelberg.

Dain, O., and Cunningham, R. K. 2001. Fusing a heterogeneous alert stream into scenarios. In *ACM Workshop on Data Mining for Security Applications*, 1–13.

Eckmann, S. T.; Vigna, G.; and Kemmerer, R. A. 2002. Statl: an attack language for state-based intrusion detection. *J. Comput. Secur.* 10(1-2):71–103.

Giannella, C.; Han, J.; Pei, J.; Yan, X.; and Yu, P. 2002. Mining frequent patterns in data streams at multiple time granularities. In *Proceedings of the NSF Workshop on Next Generation Data Mining*.

Han, J.; Pei, J.; and Yin, Y. 2000. Mining frequent patterns without candidate generation. In *SIGMOD*, 1–12.

Julisch, K. 2003. Clustering intrusion detection alarms to support root cause analysis. *ACM Trans. Inf. Syst. Secur.* 6(4):443–471.

Kuramochi, M., and Karypis, G. 2001. Frequent subgraph discovery. In *International Conference on Data Mining*, 313–320. Washington, DC, USA: IEEE Computer Society.

Qin, X., and Lee, W. 2003. Statistical causality analysis of infosec alert data. In *RAID*, volume 2820, 73–93. Pittsburgh, USA: LNCS, Springer-Verlag Heidelberg.

Qin, X. 2005. *A Probabilistic-Based Framework for INFOSEC Alert Correlation*. Ph.D. Dissertation, Georgia Institute of Technology.

Sadoddin, R., and Ghorbani, A. A. 2008. FSPGrowth : A new algorithm for mining frequent graph structures. Technical Report TR08-191, Faculty of Computer Science, University of New Brunswick.

Templeton, S. J., and Levitt, K. 2000. A requires/provides model for computer attacks. In *Proceedings of the 2000 workshop on New security paradigms*, 31–38. New York, USA: ACM Press.

Valdes, A., and Skinner, K. 2001. Probabilistic alert correlation. In *RAID*, volume 3089, 54–68. Davis, CA, USA: LNCS, Springer-Verlag Heidelberg.

Zhu, B., and Ghorbani, A. A. 2006. Alert correlation for extracting attack strategies. *International Journal of Network Security* 3(2):244–258.