A Cache Oblivious based GA Solution for Clustering Problem in IDS

Vignesh, R
School of Computing,
SASTRA UNIVERSITY,
Thirumalaisamudram,
Thanjavur
Tamil Nadu, India.

Ganesh, B
School of Computing,
SASTRA UNIVERSITY,
Thirumalaisamudram,
Thanjavur
Tamil Nadu, India.

Aarthi, G
School of Computing,
SASTRA UNIVERSITY,
Thirumalaisamudram,
Thanjavur
Tamil Nadu, India.

Iyswarya, N
School of Computing,
SASTRA UNIVERSITY,
Thirumalaisamudram,
Thanjavur
Tamil Nadu, India.

ABSTRACT
In this we present an efficient solution for eliminating false positives in intrusion detection systems using a parallelized version of Genetic Algorithm. Genetic algorithm uses selection, mutation and crossover operations eliminating most of the false positives in a reasonable time. Almost all existing versions are sequential without exploiting the capabilities of newer multiprocessors or distributed systems. By parallelizing genetic operations in the context of intrusion detection systems we reduce the total complexities. This parallelized approach gives better solution than sequential one by taking advantage of the parallel architecture. We propose the use of cache oblivious technique in our algorithm to provide efficient memory transfers. The complexity of this algorithm is $O((N/B) \log_{M/B} N^{1/3}/3 + N^{1/3})$ which is very much lesser when compared to other sorting algorithms.

Categories and Subject Descriptors
C.1.4 **[Processor Architectures]:** Parallel Architectures-*Distributed Architectures.*
D.1.3 **[Programming Techniques]:** Concurrent Programming-*Parallel programming.*
G.1.0 **[Numerical Analysis]:** General-*Parallel algorithms*.

General Terms
Algorithms, Security, Theory

Keywords
Cache Oblivious, Clustering, Genetic algorithm, False Positive, Funnel Sort.

1. INTRODUCTION
Intrusion detection system is a hardware or software, monitoring the anomalous events that can be a potential threat to computer systems. It may be implemented in firewalls. There are two main types of IDS being used today, Network Based and Host Based. An intrusion detection system raises an alarm when an anomalous behavior is detected. These alarms are presented to a human operator who evaluates them and initiates an adequate response. Examples of possible responses include law suits, firewall reconfigurations and fixing of discovered vulnerabilities [8]. Practitioners [5][10] as well as researchers [1][4][6][7] have observed that IDS can easily trigger thousands of alarms per day up to 99% of which are false positives. This flood of mostly false positives makes it very difficult to identify the hidden true positives.

Genetic algorithms can be used to evolve simple rules for network traffic [12]. Network events are assessed with these rules giving an indication of whether the particular event is an intrusion or not. The final goal of applying GA is to generate rules that match only the anomalous connections. These rules are tested on historical connections and are used to filter new connections to find suspicious network traffic [9]. GA operations selection, mutation and crossover influence in efficient elimination of false positives with a noticeable time complexity.

Cache oblivious approach exploits the CPU cache without having the size of cache as an explicit parameter. It is designed to perform well without modifications on multiple machines with different cache sizes. Cache oblivious sorting is a parallel sorting algorithm requiring at least $N^{1/3}$ processors where N is the number of elements. To perform cache oblivious we require a memory size of M which is at least B^2 where B is the size of a single block in the cache. To implement cache oblivious sorting we use a K-funnel merger. A K-funnel merger consists of K sorted list each of size greater than or equal to K^3.

2. IDS ALARMS
Every alarm event that happens in the network is a vital clue to understand its true operational status. By maintaining a history of alarm events, one can track trends and locate problem areas in the network. This information can help to revise maintenance schedules, determine equipment replacement plans, and anticipate and prevent future problems.

A high-quality alarm management system can record each alarm event in a history log. History logs can include alarms, control operations, alarm acknowledgements, internal alarms, power failures and user activity.

Several metrics are used to evaluate and compare the performance of IDSs. The most basic metrics are the detection and false alarm rates. The detection rate is equal to the number

of intrusions detected divided by the total number of intrusions in a data set, while the false alarm rate is equal to the number of normal instances detected as intrusions divided by the number of normal instances in a data set. These false alarms are also referred to as false positives.

2.1. Alarm Clustering

The alarm clustering problem deals with clustering alarms based on the root cause. An exact solution for the clustering problem will eliminate the redundancy of finding the source every time as they are grouped according to the cause for the problem. But unfortunately, there is no exact solution. This is because; the computer programs are not aware of the root causes and therefore do not enforce the requirement that all the alarms of the alarm cluster must share the same root cause.

The alarm clustering problem can also be defined as the one, where large alarm clusters that are adequately modeled by generalized alarms when the alarm log is given.

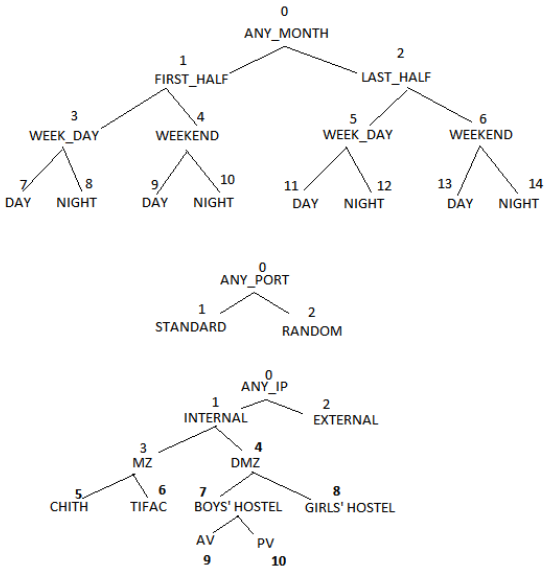

Figure 1: Taxonomy of clustering

3. CACHE OBLIVIOUS METHOD

An algorithm is cache oblivious if no program variables dependent on hardware configuration parameters, such as cache size and cache-line length, need to be tuned to minimize the number of cache misses. The cache-oblivious sorting algorithm presented here is a version of funnel sort, which is similar to merge sort. In cache-oblivious data structures, the quotient M/B must be at least 1 so that useful amount of data can be stored in the cache. This assumption is rather weak and in funnel sort it is replaced with a stronger assumption, called the tall-cache assumption. This generally states that the cache is taller than it is wide. A funnel merges several sorted lists into one sorted list in an output buffer.

3.1. Cache Oblivious Sorting

Cache oblivious sorting is a parallel sorting algorithm requiring at least $N^{1/3}$ processors where N is the number of elements. The base case occurs if $N < O(B^2)$, where, by the tall-cache assumption, we can move the entire list into the cache and sort in O (B) time.

To implement cache oblivious sorting we use a K-funnel merger. A K-funnel merger consists of K sorted list each of size greater than or equal to K^3. Clearly,

i. Conceptually split the elements into $N^{1/3}$ segments of length $N^{2/3}$ each.
ii. Call Funnel Sort recursively on each segment.
iii. Merge the sorted segments into the output stream using an $N^{1/3}$ -funnel.

The K-funnel sorts these K lists using $O((N/B)\log_{(M/B)}(N/B))$ memory transfers.

4. GENETIC-ALGORITHM BASED SOLUTION

The problem is converted to GA domain by encoding the alarm into chromosomes. The chromosome is made of n pieces, one from each tuple, where the length of the piece varies from tree to tree. This chromosome can be decoded back to an alarm. Crossover and mutation operations are performed on this chromosome. This produces a new generation of alarm. This helps in clustering related events which can be identified as a false positive. The algorithm first selects the individual chromosomes and creates a new generation of alarms and selects best X alarms. Then local optimization will be performed on each of the alarms by considering its best nearest neighbor. Nearest neighbor is one in which an element in the tuple is replaced with either its parent or its offspring. This neighbor is considered from the taxonomy given [Figure 1]. This prevents the premature result.

4.1. Modified Cache Oblivious Method

This algorithm is a modified version of GA algorithm presented by[14]. Here we select N individuals and perform crossover and mutation. They calculate the fitness of each individual in the new generation, which is based on the occurrence of that alarm. This global optimization may settle down on a local minimum, so we perform Local optimization

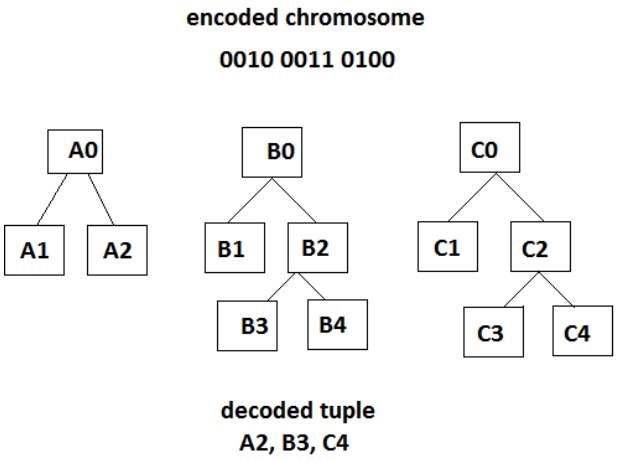

Figure 2: Encoding and Decoding

Here (A2, B3, C4) is taken as binary of (2,3,4)

.

Algorithm:

Input: Security events, Threshold value

Output: A solution (rule)

Global Variables:

N	Number of individuals
K	$N^{1/3}$
i	Processor index
X	No of individuals for local optimizations.
Best	Solution (rule)
Y	Any input alarm

```
Begin
    Create N random individuals
    Spawn (Processors P0 to Pk) where k= N^(1/3).
    For all Pi where 0≤i<k
    do
      While (Error is large)
      do
            Crossover () // with probability 100%
            Mutation () //with probability 5%
            Fitness ()
            CacheObliviousSort ()
            Retain the N best individuals
      EndWhile
    EndFor
    # Local optimization.
    Choose X best individuals
    For each Yi in X
    do
      Best[i] = Local_Optimize(Yi)
    EndFor
    CacheObliviousSort (Best)
    return Best
End
```

This algorithm takes a finite number of events, say N, as input from the log. A threshold value is also considered for evaluation purpose in the fitness function.

A number of processes, say $K=N^{1/3}$, are spawned to execute in parallel where each processor performs crossover and mutation on individual chromosome. Once the operations are performed, the fitness of the chromosome is evaluated and sorted based on the result of the fitness function using funnel sort (cache oblivious). After sorting, N best individuals are retained and the procedure continues till the error reduces to acceptable value. When the procedure ends, local optimization is carried out on these individuals to obtain the best event. Local optimization is performed by identifying a neighbour which is better than the current alarm, selected by the fitness function. These can be done in parallel provided we span the necessary number of processes. Here cache oblivious sorting is used to identify the best neighbour.

Algorithm Local_Optimize(Y)

Input: An Alarm Y, Threshold T, Set of trees

Output: An Alarm

Global Variables:

Y	An input Alarm
NO_OF_CHILDREN	m in an m-ary tree
K	Processor Index
Tuple	An Alarm
|Tuple|	Cardinality of a tuple

Local Variable:

Best_neigh	Best Neighbor of a given alarm

```
Begin
    Spawn (Processors P0 to Pk)
        where k = (NO_OF_CHILDREN+1)*|Y|
    While(Y can be optimized further)
    do
      For all Pi where 0≤i<k
      do
            Assign_neighbours(Y)
       Fitness ()
      End For
      Best_neigh = CacheObliviousSort ()
      If (Best_neigh is better than Y)
      then
            Y← Best_neigh
      Else
            return Y
      EndIf
    EndWhile
End
```

This algorithm uses adjacency lists to represent the neighbors of a particular node in a forest. The size of the adjacency list depends on the tree adapted. For instance, a node in an m-ary tree has at the maximum m+1 node in its adjacency list.

The procedure **Assign_neighbours(Y)** assigns a processor i to generate new chromosomes (neighbor alarms) by substituting the neighbors of element (i/m)+1 in the tuple. For instance,

processors 0, 1 and 2 change the first element of Y in case of a binary tree.

K processors, where k= (m+1)*cardinality of tuple, uses these adjacency lists to create new chromosomes, in O (1) asymptotic time, which are applied to fitness function in parallel. Based on the fitness value the best chromosome is chosen using cache oblivious sort.

4.2. Complexity Analysis

In the worst case the recursive substitution procedure will be bound by the height of the tree. Since we have n trees, where n is the cardinality of the tuple, the local optimization procedure will also be bound by O (H*n), where H is the height of the largest tree.

Time complexity of the cache oblivious sort is $O((K/B)(\log_{M/B}K^{1/3}/3))$.where M is the Memory size and B is Block size. Hence the total complexity of Local_Optimize () is $O(n*K/B)H(\log_{M/B}K^{1/3}/3)$.

Table 1: Complexity Measures

Function	Time Complexity
Spawn	O(log N)/3
CrossOver	$O(N^{2/3})$
Mutation	$O(N^{2/3})$
Fitness	$O(N^{2/3})$
CacheObliviousSort	$O((N/B)\log_{M/B}N^{1/3}/3 + N^{1/3})$
Locally_Optimize	$(\|Y\|*K/B)H(\log_{M/B}K^{1/3}/3)$

Where,
N is the Number of elements
H is the Height of the m-ary tree
M is the Memory size
B is the Cache Block size
|Y| is the Cardinality of alarm Y.
K is equal to (NO_OF_CHILDREN+1)*|Y|

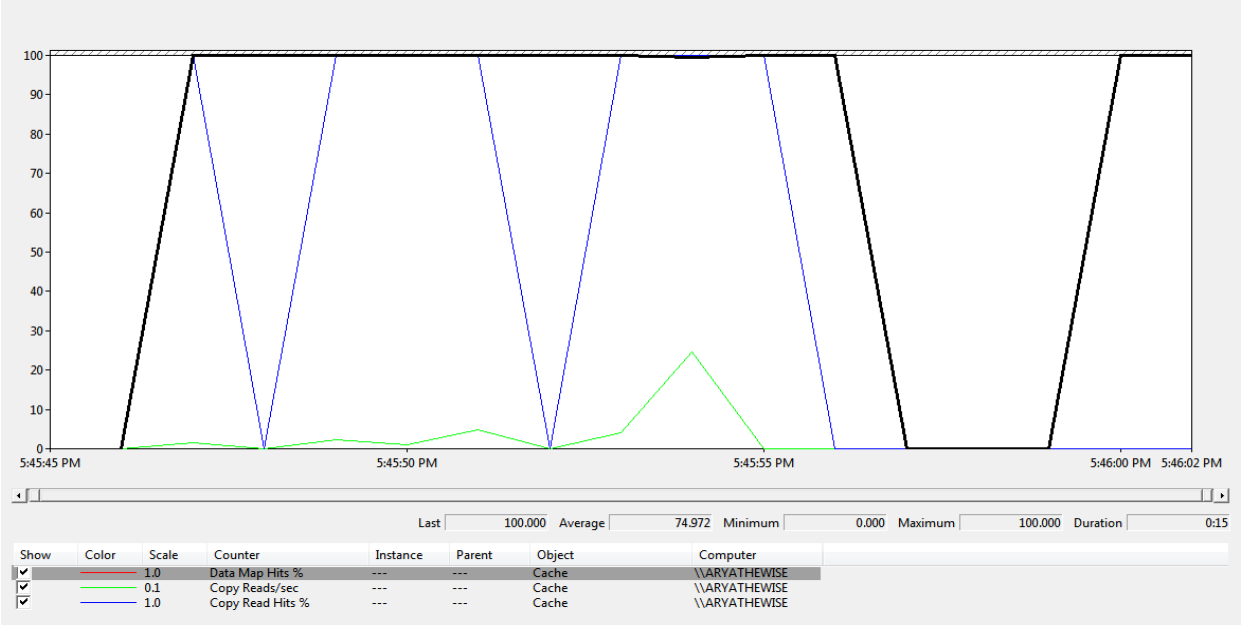

Figure 3: Data Map Hit percentage using cache oblivious sort, where x axis is time and y axis is % Data Map Hit

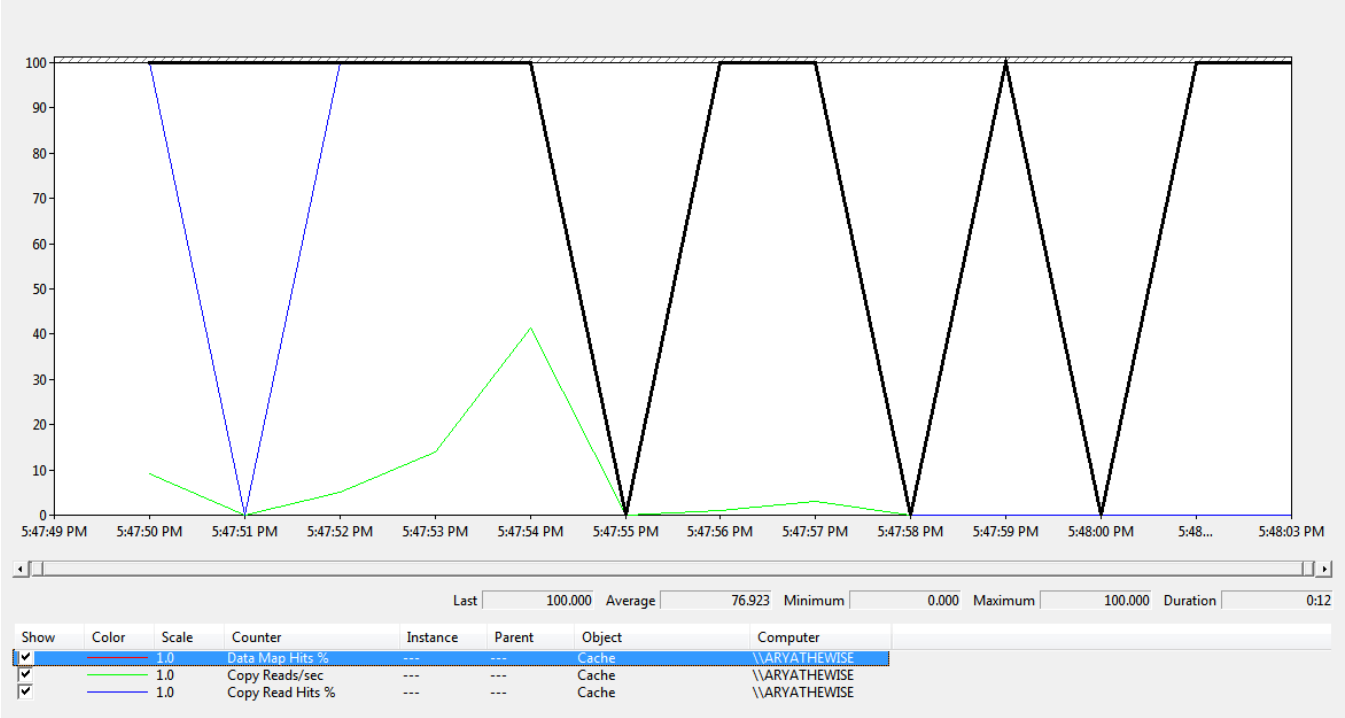

Figure 4: Data Map Hit percentage using inbuilt sort, where x axis is time and y axis is % Data Map Hit

No. of chromosomes	1 processor (Execution time) s	2 processor(Executio time) s
50	0.094089001773682	0.087680475066008 5
100	0.198661883980094	0.109687482887239
150	0.345966098844656	0.145876838971162
200	0.538873995952599	0.235071212373441

Table 2: 1 vs 2 processor comparison

5. RESULTS

In cache oblivious sort, the data map hit percentage is high and thus, a high consistency of cache hits is maintained. Whereas in Array.Sort(), the cache hits are not consistent. Thus, Cache Oblivious sort can be preferred for better cache exploitation.

In the tree apriori approach, the usage of hierarchy in classification of tuples is not possible. So, dissimilarity based clustering cannot be formed. Thus, it is only a count based clustering approach. But in GA approach hierarchical decomposition is also possible. Thus, GA is a better approach than tree apriori in analysis of network data and also for better performance and results.

6. LIMITATIONS

This cache oblivious method requires a large data set, requiring large CPU work for any page fault. The alarm tuple should neither be very wide nor be very narrow, so its appropriate selection is a key factor in determining performance. And the solution given by GA is an approximate solution, so future work can be done on tree based apriori, which gives most appropriate result.

7. CONCLUSION

From the above analysis, we conclude that many of the algorithms are aimed at utilizing the improvements in CPU processing, not the memory though. Cache oblivious algorithm tries to fill the gap, which is also portable among various architectures as they are oblivious towards cache parameters.

8. REFERENCES

[1] Axelsson, S. 2000. The Base-Rate Fallacy and the Difficulty of Intrusion Detection. ACM Transactions on Information and System Security (TISSEC) 3(3), 186-205.

[2] Bankovic, Z., Moya, José M., Araujo, A., Bojanic, S., and Nieto-Taladriz, O. September, 2007. Improving Network Security Using Genetic Algorithm Approach. Computers & Electrical Engineering, Vol.33, Issue 5-6. 438-451.

[3] Bankovic, Z., Moya, José M., Araujo, A., Bojanic, S., and Nieto-Taladriz, O. 2009. A Genetic Algorithm-based Solution for Intrusion Detection, Journal of Information Assurance and Security 4. 192-199.

[4] Bloedorn, E., Hill, B., Christiansen, A., Skorupka, C., Talbot, L., and Tivel, J. 2000. Data mining for improving intrusion detection Technical report, MITRE Corporation.

[5] Broderick, J. (ed.). 1998. IBM outsourced solution. http://www.infoworld.com/cgi-bin/displayTC.pl?/980504sb3-ibm.htm.

[6] Clifton, C., Gengo, G. 2000. Developing custom intrusion detection filters using data mining. In 2000 Military Communications International Symposium. USA. 22-25.

[7] Julisch, K. 2001. Mining Alarm Clusters to Improve Alarm Handling Efficiency. In 17th Annual Computer Security Applications Conference (ACSAC). 12-21.

[8] Julisch, K. 2003. Clustering Intrusion Detection Alarms to Support Root Cause Analysis. 8-16.

[9] Li, W. 2004. Using Genetic Algorithm for Network Intrusion Detection.

[10] Manganaris, S., Christensen, M., Zerkle, D., and Hermiz, K. 2000. A Data Mining Analysis of RTID Alarms. Computer Networks 34(4), 571-577.

[11] Olsen, Jesper H., Skov, S. December, 2002. Cache-Oblivious Algorithms in Parctice, Master's Thesis. University of Copenhagen

[12] Perdisci, R., Giacinto, G., Roli, F. Alarm clustering for intrusion detection systems in computer networks. 2006. Engineering Applications of Artificial Intelligence, Science Direct.429–438.

[13] Sinclair, C., Lyn P., and Matzner, S. 1999. "An Application of Machine Learning to Network Intrusion Detection." In Proceedings of 1999 Annual Computer Security Applications Conf. (ACSAC). 371-377. Phoenix, Arizona. URL: http://www.acsac.org/1999/papers/fri-b-1030-sinclair.pdf (30 Oct. 2003).

[14] Wang, J., Wang, H., Zhao, G. 2006. A GA-based Solution to an NP-hard Problem of Clustering Security Events. IEEE 2093- 2097.