Asura: A huge PCAP file analyzer for anomaly packets detection using massive multithreading

Ruo Ando
National Institute of Informatics
DEF CON 26

***Abstract*—Recently, the inspection of huge traffic log is imposing a great burden on security analysts. Unfortunately, there have been few research efforts focusing on scalability in analyzing very large PCAP file with reasonable computing resources. Asura is a portable and scalable PCAP file analyzer for detecting anomaly packets using massive multithreading. Asura's parallel packet dump inspection is based on task-based decomposition and therefore can handle massive threads for large PCAP file without considering tidy parameter selection in adopting data decomposition. Asura is designed to scale out in processing large PCAP file by taking as many threads as possible.**

Asura takes two steps. First, Asura extracts feature vector represented by associative containers of $< sourceIP, destIP >$ pair. By doing this, the feature vector can be drastically small compared with the size of original PCAP files. In other words, Asura can reduce packet dump data into the size of unique ¡sourceIP, destIP¿ pairs (for example, in experiment, Asura's output which is reduced in first step is about 2a parallel clustering algorithm is applied for the feature vector which is represented as $< sourceIP, destIP >, V[i]$ where $V[i]$ is aggregated flow vector. In second step, Asura adopts an enhanced Kmeans algorithm. Concretely, two functions of Kmeans which are (1)calculating distance and (2)relabeling points are improved for parallel processing.

In experiment, in processing public PCAP datasets, Asura can detect 750 packets which are labeled as malicious from among 70 million (about 18GB) normal packets. In a nutshell, Asura successfully found 750 malicious packets in about 18GB packet dump. For Asura to inspect 70 million packets, it took reasonable computing time of around 350-450 minutes with 1000-5000 multithreading by running commodity workstation. Asura will be released under MIT license and available at author's GitHub site on the first day of DEF CON 26.

I. INTRODUCTION

The scale of network traffic are growing year by year. Besides, as the emergence of sophisticated cyber attack such as APT (Advanced Persistent Threat), we are forced to cope with the detailed packet inspection as well as checking other traffic logs. Currently, the inspection of huge traffic log is imposing a great burden on security analysts.

Traffic irregularities or traffic anomalies is usually described as the result of one or more occurrences which changes the normal flow of data. Anomaly detection is the process of finding patterns in the audit network traffic log which do not conform to legitimate or normal behavior. The design of an anomaly detection method relies on a baseline model which represents the normal behavior of network traffic. The model is generally trained using audit data extracted from traffic for which all items are labeled in advance as either normal or anomalous. However, labelling traffic data is often expensive and time-consuming partly because it requires human experts.

II. OVERVIEW

A. Task based decomposition

If we want to transform code into a concurrent version, there are two ways. First one is data decomposition, in which the program cope with a large collection of data and can compute every chunk of the data independently. Second one is is task decomposition, in which the process is partitioned into a set of independent tasks that threads can execute in any order. Data decomposition has some drawbacks. For example, the size of PCAP file varies according to the situation in which the file is dumped. Besides each divided process is NOT homogeneous. Asura adopts task-based parallel processing which is shown in Figure1. Asura's parallel packet dump inspection is based on task-based decomposition and therefore can handle massive threads for large PCAP file without considering tidy parameter selection in adopting data decomposition. Asura is designed to scale out in processing large PCAP file by taking as many threads as possible. Assume that we have n threads and each thread and each thread is associated with one PCAP file. By doing this, Asura can take advantages of dynamic scheduling for coping with a variety of kinds and size of PCAP file. Asura consists of two thread sets: a master thread which enqueues the name of PCAP file and a worker thread which processes the each PCAP file. The master thread enqueues the list of PCAP file, and passes it to the worker thread. More specifically, the master thread traverses PCAP file directory and enqueue the file name. When the queue is full, the master thread waits until the worker thread processing packets consumes a file name and removes it from the queue. The worker (packet processing thread) dequeues a file name and parses the PCAP file. The task of the worker thread is flow vector aggregation using associative container discussed later in III-A.

B. Feature selection and structures

In general, most system of the identification of anomalies on traffic volume is based on features. The features could be many: source and destination addresses, port numbers, static signatures, statistics signatures and so on. The difficulty of the anomaly traffic identification is to handle the huge traffic volume. The challenging part here is to select which header item are worth and suitable to be learned. For simplicity, we pick up the five items as follows:

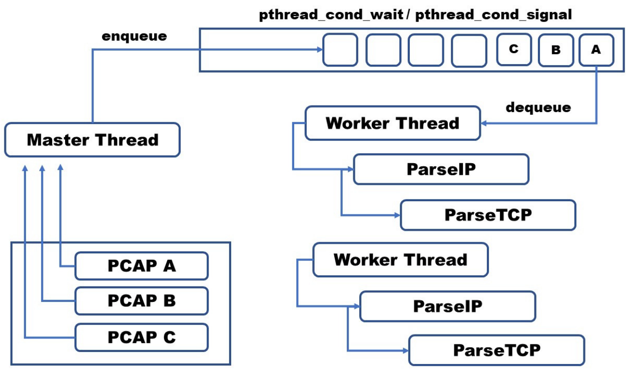

Fig. 1. Task based decomposition

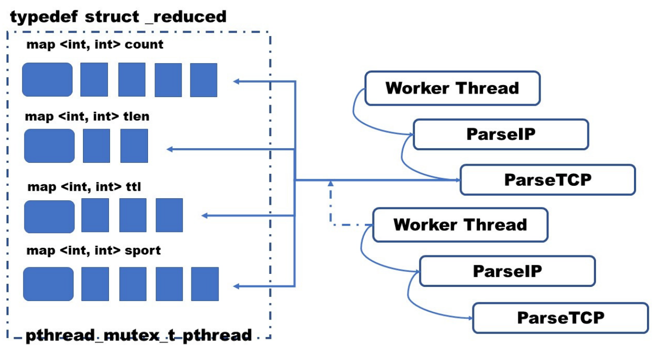

Fig. 2. Flow vector computation by multithreading

1) Source and destination IP addresses: unique identiers of sender and receiver.
2) The number of captured packets: the fields is reflected by the traffic volume.
3) TLEN (total length of packets): the 16-bit field defines the entire packet volume in bytes, including both header and data.
4) TTL (The time-to-live): the time-to-live value can be referred as an upper bound on the time which an IP datagram can be processed.
5) source and destination port: port number is checked for the type of application.

Structure is shown as follows. For each fields, the numerical fields of packet header are reduced using associative container of $map < key, value >$. The key stores number of packets. In other words,

```
/* STRUCTURE II: reduced */
typedef struct _reduced {
map<int, int> count;
map<int, int> tlen;
map<int, int> ttl;
map<int, int> sport;
map<int, int> dport;
pthread_mutex_t mutex;
} reduced_t;
reduced_t reduced;
```

Another challenging part is to transform code into a concurrent version with massive multithreading. For reduction, we use the structure as follows.

```
/* STRUCTURE I: srcIP, destIP */
typedef struct _addrpair {
map<string, string> m;
pthread_mutex_t mutex;
} addrpair_t;
addrpair_t addrpair;
```

Here pthread_mutex_t is the notation for mutual execution. Mutual exclusion locks (mutexes) prevent multiple threads from simultaneously executing critical sections of code which access shared data (that is, mutexes are used to serialize the execution of threads). All mutexes must be global. Threads within the same process or within other processes can share mutexes. Mutexes can synchronize threads within the same process or in other processes. Mutexes can be used to synchronize threads between processes if the mutexes are allocated in writable memory and shared among the cooperating processes ,and have been initialized for this task.

Algorithm 1 Flow vector aggregation 1

```
Input: $file\{< S, D >, V\}$
Output: Unique pair of $< s[i], d[i] >, v[i]$
while for i in each thread do
    $lock(map < S, D >)$
    $local\_map < S, D >= map < S, D >$
    $unlock(map < S, D >)$
    while $!end\_of\_file[i]$ do
        $< s, d ><= readline(FILE)$
        $i = 0$
        while $local\_map < S, D >! = EMPTY$ do
            if $< s, d >! = local\_map < S[i], D[i] >$ then
                $local\_map.insert(< s, d >)$
                $lock(map < S, D >)$
                $map < S, D >= local_map < S, D >$
                $unlock(map < S, D >)$
                $i++$
            end if
        end while
    end while
end while
```

III. Proposal method

A. Extracting feature vector using associative container

Associative containers are a generic group of class templates in the standard library of the C++ programming language which cope with ordered associative arrays. The containers are implemented in the current revision of the C++ standard: set, map, multiset, multimap.

1) Key uniqueness: in map and set each key must be unique.

Algorithm 2 Flow vector aggregation 2

```
Input: $R1\{< src, dest >, V\}, R2\{counter, sum\}$
Output: $< src, dest >, V$
 1: while  for i in each thread  do
 2:   while $!end\_of\_file$ do
 3:     $\{< s, d >, v\} <= readline(FILE)$
 4:     $i = 0$
 5:     for $itr = R1\_local.begin; itr! = R1\_local.end$ do
 6:       if $< s, d >! = R1\_local < S[i], D[i] >$ then
 7:         $R1\_local. < Vi > + = v$
 8:         $lock(R1\{< S, D >, V\})$
 9:         $R1\{< S, D >, V\} = R1\_local$
10:         $unlock(R1\{< src, dest >, V\})$
11:       end if
12:       $i + +$
13:     end for
14:   end while
15: end while
```

2) Element composition: in map and multimap each element should be composed from a key and a mapped value.
3) Element ordering: elements must follow a strict weak ordering

Proposal method is divided into two steps. Two pseudocodes are shown in the Algorithm 1 and 2. The core idea of our algorithm is to divide data into chunks which includes the pair $< src, dest >$. The implementation is based on associative container which stores the unique pair of $< src, dest >$. This procedure is shown in the pseudocode of Algorithm 1. The second nested loop retrieves the pair $< s, d >$ from file stream at line 6. In the third nested loop, the program traverse local_map ¡S, D¿ for looking V[i] corresponding to $< s, d >$ The final output of Algorithm 1 is the unique pair of $< s[i], d[j] >$

The purpose of Algorithm 2 is the contraction. The most important line is 7. At line 7, edges between src and dest are contracted by adding V[i]. More precisely, the edge $< src[i], dest[i] >$ and $< src[j], dest[j] >$ are reduced by adding V[i] to V. After figuring out $< src, dest >, V$, contracted edges V are stored in global variable with the index of $< src, dest >$. The second nested loop beginning at line 2 retrieves $\{< s, d >, v\}$ where s, d is single point with intermediate points s. The third nested loop starting at line 5 finds the corresponding $\{< s[j], d[j] >, v[j]\}$in $< s[i], d[i] >, v[i]$ at line 6. Similar to previous steps, $\{< S, D >, V\}$ is locked. The final output is $< src, dest >, V$.

B. Parallel Kmeans

The general design of Kmeans is that it iteratively partitions a given dataset into k clusters. At first, it choices k data point as the initial centroids. It could be the first data points or a set of randomly selected k data points in the data set. We are given an integer k and a set of n data points $x \in R^d$. We which to choose k centers C so as to minimize the potential function,

$$\phi = \sum_{x \in D}^{n} min_{c \in C} \parallel x - c \parallel^2$$

A clustering each centroid implicitly requires choosing these centroids. Here we set one cluster to be the set of data points which are closer to the center than to any therefore. It is known that finding exact solution to the Kmeans problem is NP-hard. For implementing our method, Kmeans are divided into four steps: (1) initialization, (2) distance calculation, (3) relabeling points, (4) centroid recalculation and (5)coverage condition. Here the step of (3) relabeling points is explicitly appended. The core idea of our algorithm is based data decomposition which divide the dataset into j chunks. $R = 1, ...j$. We can parallelize steps of (1) - (4). The pseudocode our our algorithm is shown in Algorithm 3. Line 3 - 11 represents the step of distance calculation of $\{< R, S >, D\}$ where R is data points, S is cluster set, and D is the distance between R and S. The table of distance is manipulated on shared memory, therefore line 6-8 lock/unlock $\{< R, S >, D\}$ defined as global variable.

The second step of relabeling clusters which finds the minimum distance between R and S is depicted in line 14-26. Precisely, this step aims to discover the unique pair of $< r, s >$ with minimum distance d. Aster calculating the minimum distance, this pair $< r, s >$ is stored in $\{< D, S >, R\}$.

The third parallelized step of (4) is represented in line 27-31. In this step, we have already relabeled the points R to cluster S. Hence line 29 can be parallelized.

IV. Experimental result

In experiment, in processing public PCAP datasets, Asura can identified 750 packets which are labeled as malicious from among 70 million (about 18GB) normal packets. In a nutshell, Asura successfully found 750 malicious packets in about 18GB packet dump. For Asura to inspect 70 million packets, it took reasonable computing time of around 350-450 minutes with 1000-5000 multithreading by running commodity workstation.

V. Conclusion

Recently, the inspection of huge traffic log is imposing a great burden on security analysts. Unfortunately, there have been few research efforts focusing on scalability in analyzing very large PCAP file with reasonable computing resources. Asura is full-scratch (and painful) implementation with POSIX Pthreads and C++ STL. PCAP files are NOT organized in a regular pattern or the parsing of PCAP files is different or unpredictable for each element in the stream. Leveraging Pthreads which represents assembly language in parallelism provides maximum flexibility. Asura adopts task-based implementation for processing huge, heterogeneous and unpredictable PCAP file stream.

Currently, Asura has thousands lines of code. can process about 75,000,000 - 80,000,000 packets in about 250-500 minutes. Drawbacks of adopting Pthreads are caused by lock

Algorithm 3 Parallel Kmeans

```
Input: $< R, S >$
Output: $\{< D, S >, R\}$, $\{< R, S >, D\}$
 1: $i = 0$
 2: for each threads j do
 3:     for each r in R[j] do
 4:         for each s in S do
 5:             $d[i] = distance(r, s)$
 6:             $lock(\{< R[j], S >, D\})$
 7:             $store < R[j], S >, d[i]$
 8:             $unlock(\{< R[j], S >, D\})$
 9:             $i++$
10:         end for
11:     end for
12: end for
13: $i = 0$
14: for each threads j do
15:     for each r in R[j] do
16:         for each s in S do
17:             if $d[i](r, s) < min\_D$ then
18:                 $min\_D = d[i]$
19:             end if
20:             $i++$
21:         end for
22:         $lock(\{< D, S >, R\})$
23:         $store < min\_D, s >, r$
24:         $unlock(\{< D, S >, R\})$
25:     end for
26: end for
27: for each threads j do
28:     for each s in S do
29:         recalculate new centroid with $\{< D, S >, R[j]\}$
30:     end for
31: end for
```

connection and context switch. For future work, other parallel processing libraries will be promising technology for the speedup of current Asura.

References

[1] Linux Programmer's Manual PTHREADS(7), http://man7.org/linux/man-pages/man7/pthreads.7.html

[2] Threading Building Blocks, https://www.threadingbuildingblocks.org/

[3] Thrust, https://developer.nvidia.com/thrust

[4] Asura, https://github.com/RuoAndo/Asura