MULAN: Multi-Level Adaptive Network Filter*

Shimrit Tzur-David, Danny Dolev, and Tal Anker

The Hebrew University, Jerusalem, Israel
shimritd, dolev, anker@cs.huji.ac.il

Abstract. A security engine should detect network traffic attacks at line-speed. When an attack is detected, a good security engine should screen away the offending packets and continue to forward all other traffic. Anomaly detection engines must protect the network from new and unknown threats before the vulnerability is discovered and an attack is launched. Thus, the engine should integrate intelligent "learning" capabilities. The principal way for achieving this goal is to model anticipated network traffic behavior, and to use this model for identifying anomalies.
The scope of this research focuses primarily on denial of service (DoS) attacks and distributed DoS (DDoS). Our goal is detection and prevention of attacks. The main challenges include minimizing the false-positive rate and the memory consumption. In this paper, we present the MULAN-filter. The MULAN (MUlti-Level Adaptive Network) filter is an accurate engine that uses multi-level adaptive structure for specifically detecting suspicious traffic using a relatively small memory size.

1 Introduction

A bandwidth attack is an attempt to disrupt an online service by flooding it with large volumes of bogus packets in order to overwhelm the servers. The aim is to consume network bandwidth in the targeted network to such an extent that it starts dropping packets. As the packets that get dropped include also legitimate traffic, the result is denial of service (DoS) to valid users.

Normally, a large number of machines is required to generate volume of traffic large enough for flooding a network. This is called a distributed denial of service (DDoS), as the coordinated attack is carried out by multiple machines. Furthermore, to diffuse the source of the attack, such machines are typically located in different networks, so that a single network address cannot be identified as the source of the attack and be blocked away.

Detection of such attacks is usually done by monitoring IP addresses, ports, TCP state information and other attributes to identify the anomalous network sessions. The weakness of directly applying such a methodology is the large volume of memory required for a successful monitoring. Protocols that accumulate state information that grows linearly with the number of flows are not scalable.

In designing a fully accurate and scalable engine, one need to address the following challenges.

* This is the authors copy of the paper that will appear in SecureComm 2009.

1. Prevention of Threats: The engine should prevent threats from entering the network. Threat prevention (and not just detection) adds difficulties to the engine, most of which stem from the need to work at line-speed. This potentially makes the engine a bottleneck – increasing latency and reducing throughput.
2. Accuracy: The engine must be accurate. Accuracy is measured by false-negative and false-positive rates. A false-negative occurs when the engine does not detect a threat and a false-positive when the engine drops normal traffic.
3. Modeling the anticipated traffic behavior: A typical engine uses thresholds to determine whether a packet/flow is part of an attack or not. These thresholds are a function of the anticipated traffic behavior, which should reflect, as best as possible, actual "clean" traffic. Creating such a profile requires a continuous tracking of network flows.
4. Scalability: One of the major problems in supplying an accurate engine is the memory explosion. There is a clear trade-off between accuracy and memory consumption. It is a challenge to design a scalable engine using a relatively small memory that does not compromise the engine accuracy.

This paper presents the MULAN-filter. The MULAN-filter detects and prevents DoS/DDoS attacks from entering the network. The MULAN-filter maintains a hierarchical data structure to measure traffic statistics. It uses a dynamic tree to maintain the information used in identifying offending traffic. Each level of the tree represents a different aggregation level. The main goal of the tree is to save statistics only for potentially threatening traffic. Leaf nodes are used to maintain the most detailed statistics. Each inner-node of the tree represents an aggregation of the statistics of all its descendants.

Periodically, the algorithm clusters the nodes at the first level of the tree, it identifies the clusters that might hold information of suspicious traffic, for each node in such clusters, the algorithm retrieves its children and apply the clustering algorithm on the node's children. The algorithm repeats this process until it gets to the lower level of the tree. This way, the algorithm identifies the specific traffic of the attack and thus, this traffic can be blocked.

The MULAN-filter removes from the tree nodes that are not being updated frequently. This way, it maintains detailed information for active incoming flows that may potentially become suspicious, without exhausting the memory of the device on which it is installed.

The MULAN-filter uses samples. At the end of each sample it analyzes the tree and identifies suspicious traffic. When the MULAN-filter identifies a suspicious path in the tree, it examines this path to determine whether or not the path represents an attack, this may take a few more samples. As a result, there might be very short attacks, that start and end within few samples that the MULAN-filter will not detect. In [1], the authors conclude that the bulk of the attacks last from three to twenty minutes. By determining the duration of a sample to few seconds, our MULAN-filter detect almost all such attacks.

The MULAN-filter was implemented in software and was demonstrated both on traces from the MIT DARPA project [2] and on 10 days of incoming traffic of the Computer Science school in our university. Our results show that the MULAN-filter works at wire speed with great accuracy. The MULAN-filter preferably be installed a on a router,

so the attacks are detected before they harm the network, but its design allows it to be installed anywhere.

2 Related Work

Detection of network anomalies is currently performed by monitoring IP addresses, ports, TCP state information and other attributes to identify network sessions, or by identifying TCP connections that differ from a profile trained on *attacks-free* traffic.

PHAD [3] is a packet header anomaly detector that models protocols rather than user behavior using a time-based model, which assumes that the network statistics can change rapidly, in a short period of time. According to PHAD, the probability, P, of an event occurring is inversely proportional to the length of time since it last occurred. $P(NovelEvent) = r/n$, where r is the number of observed values and n is the number of observations. PHAD assigns an anomaly score for novel values of $1/P(NovelEvent) = tn/r$, where t is the time since the last detected anomaly. PHAD detects $\sim 35\%$ of the attacks at a rate of ten false alarms per day after training on seven days on attack-free network traffic.

MULTOPS [4] is a denial of service bandwidth detection system. In this system, each network device maintains a data structure that monitors certain traffic characteristics. The data structure is a tree of nodes that contains packet rate statistics for subnet prefixes at different aggregation levels. The detection is performed by comparing the inbound and outbound packet rates. MULTOPS fails to detect attacks that deploy a large number of proportional flows to cripple the victim, thus, it will not detect many of the DDoS attacks.

ALPI [5] is a DDoS defense system for high speed networks. It uses a leaky-bucket scheme to calculate an attribute-value-variation score for analyzing the deviations of the values of the current traffic attributes. It applies the classical proportion integration scheme in control theory to determine the discarding threshold dynamically. ALPI does not provide attribute value analysis semantics; i.e., it does not take into consideration that some TCP control packets, like SYN or ACK, are being more disruptive.

Many DoS defense systems, like Brownlee et al. [6], instrument routers to add flow meters at either all, or at selected, input links. The main problem with the flow measurement approach is its lack of scalability. For example, in [6], if memory usage rises above a high-water mark they increase the granularity of flow collection and decrease the rate at which new flows are created. Updating per-packet counters in DRAM is impossible with today's line speed. Cisco NetFlow [7] solves this problem by sampling, which affects measurement accuracy. Estan and Varghese presented in [8] algorithms that use an amount of memory that is a constant factor larger than the number of large flows. For each packet arrival, a flow id lookup is generated. The main problem with this approach is in identifying large flows. The first solution they presented is to sample each packet with a certain probability, if there is no entry for the flow id, a new entry is created. From this point, each packet of that flow is sampled. The problem with that is its accuracy. The second solution uses hash stages that operate in parallel. When a packet arrives, a hash of its flow id is computed and the corresponding counter is updated. A large flow is a flow whose counter exceeds some threshold. Since the number

of counters is lower than the number of flows, packets of different flows can result in updating the same counter, yielding a wrong result. In order to reduce the false-positives, several hash tables are used in parallel.

Schuehler et al. present in [9] an FPGA implementation of a modular circuit design of a content processing system. The implementation contains a large per-flow state store that supports 8 million bidirectional TCP flows concurrently. The memory consumption grows linearly with the number of flows. The processing rate of the device is limited to 2.9 million 64-byte packets per second.

Another solution, presented in [10], uses aggregation to scalably detect attacks. Due to behavioral aliasing the solution doesn't produce good accuracy. Behavioral aliasing can cause false-positives when a set of well behaved connections aggregate, thus mimicking bad behavior. Aliasing can also result in false negatives when the aggregate behavior of several badly behaved connections mimics good behavior. Another drawback of this solution is its vulnerability against spoofing. The authors identify flows with a high imbalance between two types of control packets that are usually balanced. For example, the comparison of SYNs and FINs can be exploited by the attacker to send spurious FINs to confuse the detection mechanism.

3 DoS Attacks

Denial of service (DoS) attacks cause service disruptions when too many resources are consumed by the attack instead of serving legitimate users. A distributed denial of service (DDoS) attack launches a coordinated DoS attack toward the victim from geographically diverse Internet nodes. The attacking machines are usually compromised zombie machines controlled by remote masters. Typical attacked resources include link bandwidth, server memory and CPU time. DDoS attacks are more potent because of the aggregated effect of the traffic converging from many sources to a single one. With knowledge of the network topology the attackers may overwhelm specific links in the attacked network.

The best known TCP DoS attack is the SYN flooding [11]. Cisco Systems Inc. implemented a TCP Intercept feature on its routers [12]. The router acts as a transparent TCP proxy between the real server and the client. When a connection request is made from the client, the router completes the handshake for the server, and opens the real connection only after the handshake is completed. If the amount of half-open connections exceeds a threshold, the timeout period interval is lowered, thus dropping the half-open connections faster. The real servers are shielded while the routers aggressively handle the attacks. Another solution is SYN cookies [13], which eliminates the need to store information per half open connection. This solution requires design modification, in order to change the system responses.

Another known DoS attack is the SMURF [14]. SMURF uses spoofed broadcast ping messages to flood a target system. In such an attack, a perpetrator sends a large amount of ICMP echo (ping) traffic to IP broadcast addresses, with a spoofed source address of the intended victim. The hosts on that IP network take the ICMP echo request and reply with an echo reply, multiplying the traffic by the number of hosts responding.

An optional solution is to never reply to ICMP packets that are sent on a broadcast address [15].

Back [16] is an attack against the Apache web server in which an attacker submits requests with URL containing many front-slashes. Processing these requests slows down the server performance, until it is incapable of processing other requests. Sometimes this attack is not categorized as high rate DoS attacks, but we mention it since the MULAN-filter discovers it. In order to avoid detection, the attacker sends the front-slashes in separate HTTP packets, resulting in many ACK packets from the victim server to the attacker. An existing solution suggests counting the front-slashes in the URL. A request with 100 front-slashes in the URL would be highly irregular on most systems. This threshold could be varied to find the desired balance between detection rate and false alarm rate.

In all the above examples, the solutions presented are specific to the target attack and can be implemented just after the vulnerabilities are exploited. The MULAN-filter identifies new and unknown threats, including all the above attacks, before the vulnerability is discovered and the exploit is created and launched, as detailed later.

4 Notations and Definitions

- A *metric* is defined as the chosen rate at which the measurements by the algorithm are executed, for example, bit per second, packets per second etc.
- An L_n is the number of levels in the tree data structured used by the algorithm.
- *Sample value* is defined as the aggregated value that is collected from the captured packets in one sample interval.
- *Window interval* is defined as $m\times$ *sample interval*, where $m > 0$ and the *sample interval* is the length of each sampling.
- *Clustering Algorithm* is defined as the process of organizing *sample values* into groups whose members have "similar" values. A *cluster* is therefore a collection of *sample values* that are "similar" and are "dissimilar" to the *sample values* belonging to other clusters (as detailed later).
- *Cluster Info* is the number of samples in the cluster, the cluster mean and the cluster standard deviation, denoted by $C.Size$, $C.Mean$ and $C.Std$ respectively.
- *Anticipated Behavior Profile* (*ABP*) is a set of k Clusters Info, where k is the number of clusters.
- *Clusters Weighted Mean* ($WMean$) is the weighted mean of all clusters, alternatively, the mean of all samples.
- *Clusters Weighted Standard Deviation* ($WStd$) is the weighted standard deviation of all clusters, alternatively, the standard deviation of all samples.
- *High General Threshold* ($HGThreshold$) is $WMean + t_1 \times WStd$ and *Low General Threshold* ($LGThreshold$) is $WMean + t_2 \times WStd$, where $t_1 > t_2$.
- *Marked Cluster* is a cluster with mean greater than $LGThreshold$.

5 The MULAN-filter Design

The MULAN-filter uses inbound rate anomalies in order to detect malicious traffic. The statistics are maintained in a tree-shaped data-structure. Each level in the tree represents an aggregation level of the traffic. For instance, the highest level may describe inbound packets rate per-destination, the second level may represent per-protocol rate for a specific destination and the third level hold per-destination port rate for a specific destination and protocol. Each node maintains the aggregated statistics of all its descendants. A new node is created only for packets with a potentially suspicious parent node. This way, for example, there is a need to maintain a detailed statistics only for potentially suspicious destinations, protocols or ports. Another advantage of using the tree is the ability to find specific anomalies for specific nodes. For example, one rate can be considered normal for one destination, but is anomalous for the normal traffic of another destination.

The MULAN-filter can be used in two modes, training mode and verification mode. The output of the training mode is the *ABP* and the thresholds. For each cluster C in the *ABP*, if $C.Mean > LGThreshold$, the cluster is denoted as a *marked cluster*. This information is used to compare the online rates in the verification mode process.

In order to calculate this information, the anticipated traffic behavior profile must be measured. There are two ways to measure such a profile: Either training a profile on identification of attack-free traffic, or by trying to filter the traffic from prominent bursts, which might indicate attacks and then creating the profile from the filtered traffic.

5.1 Anticipated Traffic Behavior Profile

In order to create the *ABP*, it is better to use an attack-free traffic. Alternative solutions strongly assume attack-free traffic, an assumption that may be impractical for several reasons. First, unknown attacks may be included in that traffic, so the baseline traffic is not attack-free. Furthermore, traffic profiles vary from one device to another, and unique attack-free training profiles need to be obtained for each specific device on which the system is deployed. Moreover, traffic profiles on any given device may vary over time. Thus, a profile may be relevant only at a given time, and may change a few hours later.

We propose a methodology in which anomalies are first identified, and then *refined*. The cleaner the initial traffic the more precise the initial state is, but our methodology works well with non-clean traffic. To achieve both goals, the algorithm aggregates per-*sample interval* statistics, creating a *sample value* for each such interval. At the end of each *window interval*, the algorithm employs a clustering algorithm in order to obtain a set of clustered *sample values*. If there are one or more clusters with significantly high mean values (3 standard deviations from $WMean$), the algorithm discovers the samples that are key contributors to the resulting mean values. The algorithm refines those samples by setting their value to the cluster mean value and then recalculates the clusters' means values. The "refinement" rule states that lower levels always override refinement of higher levels. This means that if the algorithm detects a high burst at one of the destinations and then detects a high burst at a specific protocol for that destination, it refines the node value associated with the protocol, which also impacts the value

associated with the destination. The *refinement* process is performed at every *window interval* for maintaining a dynamic profile.

5.2 Data Structure

The MULAN-filter uses a tree-shaped data structure. The tree enables maintaining distinct statistics of *all* the relevant traffic. Traffic is considered relevant if there is a high probability that it contains an attack.

In our implementation example, there are three levels in the tree. The nodes at the first level hold statistics per-destination IP address, the nodes at the second level hold statistics per-protocol and the nodes at the third level hold statistics per-destination port (see Fig. 1). During the verification mode, when a *sample value* is calculated, the algorithm saves the aggregation for the first level. In our implementation, assume that the *sample value* is equal to SV and there are N_s packets that arrived during the *sample interval* with n different IP addresses. We define $Metric(Packet_j)$ to be the contribution of $Packet_j$ to SV, and SV_i to be the part of SV that is calculated from packets with IP_i in their header. Formally, $SV_i = \sum_{j,IP_i \in Packet_j} Metric(Packet_j)$, thus, $SV = \sum_i SV_i$, where $1 \leq j \leq N_s$ and $1 \leq i \leq n$. The tree structure is flexible to hold special levels for specific protocols, see Section 5.3.

In the verification mode, the tree is updated following two possible events. One is a completion of each *sample interval*. In this case, the algorithm compares SV to the clusters' means from the *ABP*. If the closest mean belongs to a *marked cluster*, a node for each IP_i is added to the first level in the tree. The second event at which the tree is updated may occur at packet's arrival. If the destination IP address in the packet header has a node in the first level, a node for the packet protocol is created at the second level, and so on. In any case, the metric's values along the path from the leaf to the root are updated. This way, each node in the tree holds the aggregated sum of the metric's values of its descendants.

A node that is not updated for long enough is removed from the tree. A node can not be removed unless it is a leaf, and it can become a leaf if all of its descendants have been removed. Thus, we focus only on nodes (or on traffic) that are suspected of comprising an attack; thus, saving on memory consumption.

5.3 Special Levels for Specific Protocols

Some protocols have special information in their header that can help the algorithm in blocking more specific data. Since our tree is very flexible in the data it can hold, we can add special levels for specific protocols. In our experiments we added a level for the TCP protocol that distinguishes between the TCP flags. This addition results in dropping only the SYN packets when the algorithm detects a SYN attack. The same can be done for the ICMP types in order to recognize the ECHO packets in the SMURF attack.

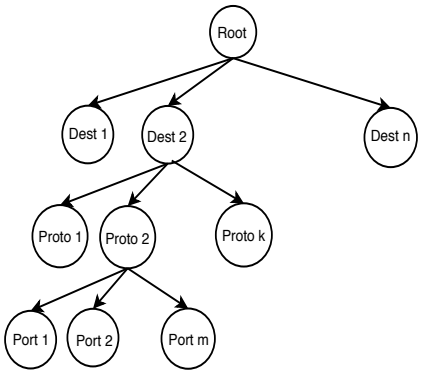

Fig. 1. The Tree

6 The Algorithm

Prior to implementing the algorithm, the following should be defined: depth of the tree, characteristics of the data aggregated at each level, sample interval, window interval, metrics to be measured, and $t1$ and $t2$ used for calculating $LGThreshold$ and $HGThreshold$.

The MULAN-filter has been implemented in software. The input to our engine is taken both from the MIT DARPA project [2] and from the Computer Science school in our university. The input from MIT contains two stretches of five-days traffic (fourth and fifth week) with documented attacks and the input from the university contains 10 days of incoming traffic, this containing both real and simulated attacks that we injected.

The algorithm operates in two modes, the training mode and the verification mode. The training mode is generated at the end of a *window interval*. The input for this mode is a set of N samples and the output is ABP with indication of the *marked clusters*.

6.1 Training Mode

In order to create the ABP, the algorithm generates the K-means [17] clustering algorithm every *window interval* to cluster the N *sample values* into k *clusters*. For each cluster, the algorithm holds its size, mean and standard deviation, after which the algorithm can calculate the weighted mean, $WMean$, and the weighted standard deviation, $WStd$, and determine the value of $LGThreshold$. Since the samples may contain attacks, $LGThreshold$ might be higher than it should. Therefore, for each cluster C, if $C.Mean > LGThreshold$, the algorithm retrieves C's *sample values*. In our implementation, each *sample value* in the cluster holds *metric values* of IP addresses that were seen in that sample interval. For each sample, the algorithm gets the aggregation per IP address and generates new set of samples. The algorithm then generates K-means again, where the input is the newly created set. Running K-means on this set produces a set of clusters, a cluster with a high mean value holds the destinations with the highest metric value. The algorithm continues recursively for each level of the tree.

At each iteration in the recursion, the algorithm detects the high bursts and refines the samples in the cluster to decrease the bursts influence on the thresholds, see Section 5.1. As mentioned, the "refinement" rule states that lower levels always override refinement of higher levels. This means that if the algorithm detects a high burst at one of the destinations and then a high burst at a specific protocol for that destination, it refines the node value associated with the protocol, impacting on the value associated with the destination. When the refinement process is completed, the refined samples are clustered again to produce the updated *ABP* information and the $LGThreshold$. A cluster C is indicated a *marked cluster* if $C.Mean > LGThreshold$.

There can be cases in which the bursts that the algorithm refines represent normal behavior. In such cases $LGThreshold$ may be lower than expected. Since the algorithm uses the training mode in order to decide whether to save information when running in verification mode, the only adverse impact of this refinement is in memory utilization, as more information is saved than actually needed. This is preferable to overlooking an attack because of a mistakenly calculated high threshold. The training mode algorithm is presented in Algorithm 1.

At each sample completion, the algorithm gets the *sample value* and finds the most suitable cluster from the *ABP*. In order for the profile to stay updated and for the clustering algorithm to be generated only at the end of each *window interval*, the cluster mean and standard deviation are updated by the new *sample value*.

6.2 Verification Mode

The verification mode starts after one iteration of the training mode (after one *window interval*). The verification mode is executed either on a packet arrival or following a sample completion.

To simplify the discussion, as a working example in this section we assume that the tree has three levels and the aggregation levels are IP address, protocol and port number in the first, second and third level, respectively.

On each packet arrival, the algorithm checks whether there is a node in the first level of the tree for the destination IP address of the packet. If there is no such node, nothing is done. If the packet has a representative node in the first level, the algorithm updates the node's metric value. From the second level down, if a child node exists for the packet, the algorithm updates the node's metric value, otherwise, it creates a new child.

At each sample completion, the algorithm gets the *sample value* and finds the most suitable cluster from the *ABP*. If the suitable cluster is a *marked cluster*, the algorithm adds nodes for that sample in the first level of the tree. In our example, the algorithm adds per-destination aggregated information from the sample to the tree. I.e. for each destination IP address that appears in the sample, if there is no child node for that IP address, the algorithm creates a child node with the aggregated metric value for that address (see Section 5.2).

The algorithm runs K-means on the nodes at the first level of the tree. Each sample value is per-destination aggregated information (SV_i with the notations from Section 5.2). As in the training mode, the clustering algorithm produces the set of *clusters info*, but in this case the algorithm calculates the threshold $HGThreshold$. If a cluster's

Algorithm 1 Training Mode Algorithm

```
1: packet ⇐ ReceivePacket();
2: UpdateMetricValue(sample, packet);
3: if End of Sample Interval then
4:     samples.AddSample(sample);
5: end if;
6: if End of Window Interval then
7:     UpdateTrainProfile(samples);
8: end if.
```

UpdateTrainProfile(samples)

```
1: clusters ⇐ KMeans(samples);
2: samples ⇐ Refine(clusters, samples);
3: ABP ⇐ BuildProfile(clusters);
4: LGThreshold ⇐ calcThreshold(ABP);
5: for all clusterInfo ∈ ABP do
6:     if clusterInfo.Mean > LGThreshold then
7:         setMarked(cluster);
8:     end if;
9: end for.
```

mean is above the $HGThreshold$, a deeper analysis is performed. For each sample in the cluster (or alternatively, for each node at the first level), the algorithm retrieves the node's children and generates K-means again. The algorithm continues recursively for each level in the tree. At each iteration, the algorithm also checks the *sample values* in the cluster nodes. If a *sample value* is greater than $HGThreshold$, it marks the node as suspicious.

The last step is to walk through the tree and to identify the attacks. The analysis is done in a DFS manner. A leaf that has not been updated long enough is removed from the tree. Each leaf that is suspected too long is added to the black list, thus preventing suspicious traffic until its rate is lowered to a normal rate. For each node on the black-list, if its high rate is caused as a results of only a few sources, the algorithm raises an alert but does not block the traffic; If there are many sources, the traffic that is represented by the specific path is blocked until the rate becomes normal. The verification mode algorithm is presented in Algorithm 2.

In addition of the above, to prevent attacks that do not use a single IP destination, like attacks that scan the network looking for a specific port on one of the IP addresses, the algorithm identifies sudden increase in the size of the tree. When such increase is detected, the algorithm builds a hash-table indexed by the source IP address. The value of each entry in the hash-table is the number of packets that were sent by the specific source. This way, the algorithm can detect the attacker and block its traffic (see Section 8). The algorithm maintains a constant number of entries and replaces entries with low values. The hash-table size is negligible and does not affect the memory consumption of the algorithm.

Since the algorithm detects anomalies at each level of the tree, it can easily recognize exceptions in the anomalies it generates. For example, if one IP address appears in many samples as an anomaly, the algorithm learns this IP address and its anticipated

Algorithm 2 Verification Mode Algorithm

```
packet ⇐ ReceivePacket();
UpdateMetricValue(sample, packet);
PlacePctInTree(packet, root, 0);
if End of Sample Interval then
    SetFirstLevel(sample, root);
    Verify(root.children);
    AnalyzeTree(root);
end if.
```

PlacePctInTree(packet, node, level)

```
if level == lastLevel then
    return;
end if;
if node.HasChild(packet) then
    child ⇐ node.GetChild(packet);
    child.AddToSampleValue(packet);
    PlacePctInTree(packet, child, ++level);
else
    if level > 0 then
        child ⇐ CreateNode(packet);
        node.AddChild(child);
    end if;
end if.
```

SetFirstLevel(sample, root)

```
cluster ⇐ GetClosestClusterFromABP(sample);
cluster.UpdateMeanAndStd(sample);
if MarkedCluster(cluster) then
    AddFirstLevelInfo(sample);
end if.
```

Verify(nodes)

```
clustersInfo ⇐ KMeans(nodes);
CalcThresholds(clustersInfo);
for all cluster ∈ clustersInfo do
    if cluster.Mean > LGThreshold then
        for all node ∈ cluster do
            if node.sampleValue > HGThreshold then
                MarkSuspect(node);
            end if;
            Verify(node.children);
        end for;
    end if;
end for.
```

AnalyzeTree(node)

```
for all child ∈ node.children do
    if child.NoChildren() then
        if child.UnSuspectTime > cleanDuration then
            RemoveFromTree(child);
        end if;
        if child.SuspectTime > suspectDuration then
            AddToBlackList(child);
        end if;
    else
        AnalyzeTree(child);
    end if;
end for.
```

rate and adds it to an *exceptions list*. From this moment on, the algorithm compares this IP address to a specific threshold.

6.3 The Algorithm Parameters

In our simulation, the algorithm builds three levels in the tree. The first level holds aggregated data for the destination IP addresses, the second level holds aggregated data for the protocol for a specific IP address, and the third level holds aggregated data for a destination port for specific IP and port. Since we look for DoS/DDoS attacks, these levels are sufficient to isolate the attack's traffic.

At the end of each window interval the algorithm updates the *ABP* and, since the network can be very dynamic, we chose the window interval to be five minutes. The bulk of DoS/DDoS attacks lasts from three to twenty minutes, we have therefore chosen the sample interval to be five seconds. This way the algorithm might miss few very short attacks. An alternative solution for short attacks is presented in Section 6.5. A node is considered as indicating an attack if it stays suspicious for *suspect duration*; In our implementation the *suspect duration* is 18 samples. A node is removed from the tree if it is not updated for *clean duration*; In our implementation the *clean duration* is 1 sample. DoS/DDoS attacks can be generated by many short packets, like in the SYN attack example, thus, a bit-per-second metric may miss those attacks. In our implementation we use a packet-per-second metric.

The last parameters to be determined are $t1$ and $t2$ that are used for calculating $LGThreshold$ and $HGThreshold$. These parameters are chosen using Chebyshev inequality. The Chebyshev inequality states that in any data sample or probability distribution, nearly all the values are close to the mean value, in particular, no more than $1/t^2$ of the values are more than t standard deviations away from the mean. Formally, if $\alpha = t\sigma$, the probability of an attribute length, can be calculated using the inequality:

$$p(|x-\mu| > \alpha) < \frac{\sigma^2}{\alpha^2}.$$

The Chebyshev bound is weak, meaning the bound is tolerant to deviations in the samples. This weakness is usually a drawback. In our case, since DoS/DDoS attacks are characterized by a very high rate, the engine has to detect just significant anomalies and this weakness of the Chebyshev boundary becomes an advantage. In our experiment we set $t1 = 1$ and $t2 = 5$.

Non-self-similar traffic may be found at the lower levels of the tree (per destination rate, per protocol rate etc.). Another problem at the lower levels is the reduced number of samples, complicating the ability to anticipate traffic behavior at these levels. In order to identify the anomalies at those levels, we introduce an alternative measurement model, see Section 6.4.

6.4 Modeling Non-Self-Similar Traffic

The MULAN-filter has to model anticipated traffic. There are two main challenges in modeling anticipated traffic: the complexity of network traffic, and its variance over time.

Bellovin [18] and Paxson [19] found that wide network traffic contains a wide range of anomalies and bizarre data that is not easily explained. Instead of specifying the extremely complex behavior of network traffic, they use a machine learning approach to model actual traffic behavior. Research by Adamic [20] and Huberman and Adamic [21] implies that this approach would fail since the list of observed values grows at a constant rate and is never completed, regardless of the length of the training period. However, Leland et al. [22] and Paxson & Floyd [23] show that this is not valid for many types of events, like measuring packets per second rate.

Non-self-similar traffic may be found at the lower levels of the tree (per destination rate, per protocol rate etc.). Another problem at the lower levels is the reduced number of samples, complicating the ability to anticipate traffic behavior at these levels. In order to identify the anomalies at those levels, an alternative measurement model should be introduced. Let N_c be the number of children of a node, and s be the sum of all sample values of the node children. If a "small" subset of N_c represents a "high percentage" of s, an anomaly is alerted. For example, consider a destination for which there are seven protocol nodes, of which six have sample values of approximately ten packets per second, and a seventh node has a sample value of 400 packets per second. This would result in a mean value of 65.7, with rather high standard deviation of 147.4. Using traditional models, it will be difficult to identify the seventh child as an anomaly. Using the proposed model, one child represents $\sim 87\%$ of all samples, so this node is identified as an anomaly.

6.5 Handling Short Attacks

MIT traces contain short DDoS attacks (some of them are 1 second long). An example from MIT traces is the SMURF attack. In the SMURF attack, the attacker sends ICMP 'echo request' packets to the broadcast address of many subnets with the source address spoofed to be that of the intended victim. Any machine that is listening on these subnets responds by sending ICMP 'echo reply' packets to the victim. Short attacks can exhaust a victim but usually cannot defeat it. Since our algorithm blocks the anomalies from entering the network, it declares an anomaly only after a node has being suspected for some time. By reducing the sample interval, our algorithm can easily detect the short attacks so an alert mechanism is added for them. As opposed to the common DoS or DDoS attacks, in order to exhaust a service, the rate of the short attacks must be significantly high so the anomaly will be much more conspicuous. Thus, in order to reduce the false-positives we use more stringent detection rules for the short attacks.

7 Optimal Implementation

The main bottleneck that might occur in our engine is the tree lookup, which is performed on arrival of each packet. Since the engine has to work at wire speed, software solutions might be unacceptable. We suggest an alternative implementation.

The optimal implementation is to use a TCAM (Ternary Content Addressable Memory) [24]. The TCAM is an advanced memory chip that can store three values for every

bit: zero, one and "don't care". The memory is content addressable; thus, the time required to find an item is considerably reduced. The RTCAM NIPS presented in [25] detects signatures-based attacks that were drawn from Snort [26]. In the RTCAM solution, the patterns are populated in the TCAM so the engine detects a pattern match in one TCAM lookup. We can similarly deploy the MULAN filter in the TCAM. A TCAM of size M can be configured to hold $\lfloor M/w \rfloor$ rows, where w is the TCAM width. Let $|L_i|$ be the length of the information at level i in the tree. Assuming that there are m levels, w is taken to be $\sum_i |L_i|$, where $1 \leq i \leq m$. In our example, the IP address at the first level contains 4 bytes (for IPv4). An additional byte is needed to code the protocol at the second level. For the port number at the third level we need another two bytes. Thus, in our example $w = 7$. Since the TCAM returns the first match, it is populated as follows: the first rows hold the paths for all the leaves in the tree. A row for a leaf at level i, where $i < L_n$ is appended with "don't care" signs. After the rows for the leaves, we add rows for the rest of the nodes, from the bottom of the tree up to the root. Each row for a non-leaf node at level l is appended with "don't care" signs for the information at each level $j \leq n$ such that $l < j$. The last row contains w "don't care" bytes, thus indicating that there is no path for the packet and providing the default match row.

Fig. 2 presents an example of the tree structure and the populated TCAM for that tree. As shown, each node (except the root) has a row in the TCAM. When a packet arrives, the algorithm extracts the relevant data, creates a TCAM key and looks for a TCAM match. Each row in the TCAM holds a counter and a pointer to the row of the parent node. When there is a TCAM match (except in the last row), the algorithm updates the counter at the matched row and follows the pointer to the parent node's row. The algorithm updates the counters for each row along the path from the node corresponding to the matched row to the row corresponding to the ancestor at the first level.

In our algorithm, there are only two places where the algorithm might add nodes to the tree, when nodes are set for the first level, and on packet arrival. In both cases, the algorithm adds leaves represented by the corresponding rows at the beginning of the TCAM. Similarly, when the algorithm"cleans" the tree, it removes leaves, again, handling the beginning of the TCAM. In order to easily update the TCAM while keeping the right order of the populated rows, the TCAM is divided into L parts, where L is the number of the levels in the tree.

The last obstacle our algorithm has to deal with is the TCAM updates. TCAM updates are done when adding nodes to the tree and when removing nodes from the tree. The TCAM can be updated either with a software engine or with a hardware engine. Using software engine is much simpler but is practical only when there is a low number of updates. Fig. 3 presents the average number of TCAM updates for each 100 packets of the incoming traffic of the Computer Science school. The figure clearly illustrates the creation of the tree. During the creation of the tree there are many insertions, thus the number of updates is relatively high.

Each value is an average of values of all the days of MIT traces. The total average update rate is ~ 1.5 updates for 100 packets, more than 99% of the values are below 50 updates, with a small number of scenarios when the engine has to make up to ~ 1700 TCAM updates. Today's enterprise network equipment supports hundreds of Giga bits

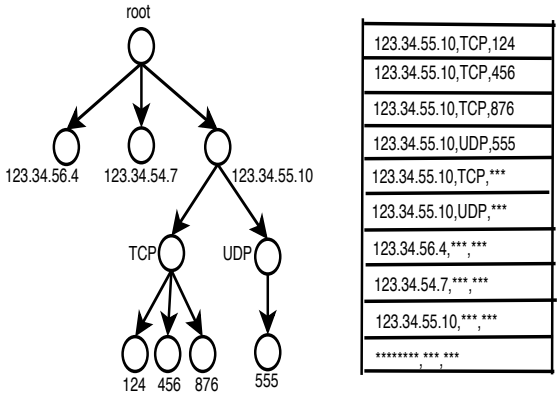

Fig. 2. TCAM Population

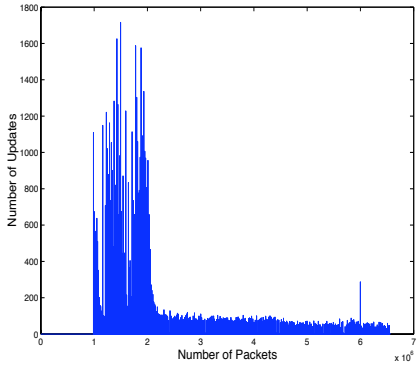

Fig. 3. TCAM Updates

per second of traffic and small and medium business devices handle $60-100$ Giga bits per second and above. One Giga interface supports 1.5 million packets per second, thus enterprise network devices need to deal with about 500 millions packets per second, and small and medium business need to deal with about 150 millions packets per second. A software engine will not be able to fulfil these requirements and thus is not acceptable. A hardware engine can achieve line speed rates. The available TCAM update speed with hardware engine is in the range of 3 to 4 nano seconds, which is $250,000,000$ to $330,000,000$ updates per second. In light of the rate of TCAM updates, it can be deduced that on average, one TCAM update is performed for every 67 packets. With a traffic rate of 500 million packets per second, the engine has to make $500M/67 \approx 7.5$ millions updates per second, which is significantly less than the available TCAM update rates limit. Even with 50 TCAM updates, the engine executes $500M/2 = 250$ millions updates per second which is still in range.

8 Experimental Results

The quality of performance of the algorithm is determined by two factors: scalability and accuracy. In order to analyze the performance of the algorithm, a simulation was implemented and tested with MIT DARPA traces and real traffic from our School of Computer Science.

8.1 Scalability

Demonstration of scalability of the algorithm requires analysis of the memory requirement at every stage of execution. We measured the number of nodes on each sample and we found an average size of the tree is 1028 nodes. This result is very encouraging since it is a very reasonable memory consumption.

Another major advantage of our algorithm is the fact that the increase in tree size is very moderate compared to the increase in the number of flows. This is clearly demonstrated in Fig. 4 (Note that the x axis is a logarithm scale). In general, for any number of flows the tree size is below 10000 nodes. There are few cases where the size of the tree exceeds 30000 nodes, these cases occur when the traffic contains attacks. An optimization to the algorithm, thats prevent such cases is presented in Section 8.3.

Memory consumption is one of the major difficulties when trying to extract per-flow information in a security device. The main problem with the flow measurement approach is its lack of scalability. Memory consumption of algorithms presented in previous works is directly influenced by the number of flows, and in many cases the algorithm performance is affected. Cisco NetFlow [7] solves this problem by sampling, which affects measurement accuracy. Another work [8] develops algorithms that use an amount of memory that is a constant factor larger than the number of the large flows. The main problem in this approach is how to identify large flows. Two possible solutions were presented, both of which lack accuracy. In [4], the authors try to aggregate data by IP prefixes. For more than 1024 IPs, the data structure size does not fit in cache, so that the algorithm rates drop proportionally to the total memory consumption. In our engine, memory consumption does not grow linearly with the number of flows and the algorithm accuracy is therefore not affected.

8.2 Accuracy

Accuracy is measured by the false-negative and the false-positive rates. False-negative is a case in which the system does not detect a threat and false-positive is the case in which the system drops normal traffic. This section presents the accuracy results both on MIT DARPA traces and on the real traffic from our School of Computer Science.

MIT DARPA Traces There are only two documented bandwidth attacks in the MIT DARPA traces, both are SYN-attacks. Our algorithm finds these attacks. In addition, our algorithm detects several other anomalous behaviors. The analysis indicates that in one of the days, there are many retransmissions packets and a large number of sequential TCP-keep-alive packets, which is consistent with anomalous behavior. Another example is the back attack targeted at the Apache web servers by submitting requests

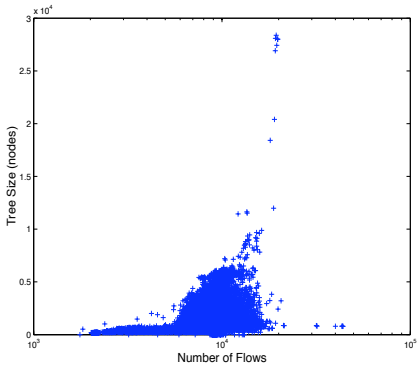

Fig. 4. Tree Size vs Number of Flows

with URL's containing many front-slashes. As the server tries to process these requests, it slows down and is unable to process other requests. In order to avoid detection, the attacker sends each front-slash in a different HTTP packet. The victim sends many TCP ACK packets back to the attacker. Since the engine compares traffic per destination (at the first level) it detects this traffic as anomaly. There is another case where our algorithm detects many TCP SYN, RST and FIN packets. In one of the SYN attacks, the source of the attack is an IP address within the network. As a result of the attack, the victim sends many TCP RST packets back to the attacker. Consequently, the engine detects two anomalies: the SYN packets to the victim and the RST packets to the attacker.

The School of Computer Science Traces We analyzed the traffic in two modes. In the first mode, we ran the algorithm on the original data and we looked for real attacks. In the second mode we randomly added attacks to random destinations and verified that the algorithm detects the injected attacks.

In the first mode we found some very interesting anomalous behaviors. In one alert, the algorithm detects inbound scan on TCP, port 1433. In this attack, the attacker scans the network, looking for a Microsoft SQL Server installations with weak password protection and, if successful, looks to steal or corrupt data or use some features with SQL Server to compromise the host system. Another alert indicates a single source that scans the network for a listening HTTP server (scanning many IP addresses on port 80). One more interesting alert indicates an inbound scan on TCP, port 139. Such inbound scans are typically systems that are trying to connect to file shares that might be available on the system and therefore should be blocked. While most of this traffic is the result of worms or viruses, which can use open file shares to propagate, they can be also the result of malicious users attempt to connect to the victim. Once connected, they can download, upload or even delete or edit files on the connected file share.

The algorithm detected 4 exceptional IP addresses, all of them servers in the network. The algorithm generated 87 alerts, almost all of them are IP addresses that communicate with an IP address from the *exceptions list*. Since the *exceptions list* is a safe list of servers, these alerts were omitted from the final results. We were left with 24 alerts. There can be cases where a single host downloads a heavy file or backup heavy

material etc. In such cases, there will be a high rate between a single host to a single destination. Our algorithm detects these channels as an attack. Since we don't want to prevent this legal traffic, our algorithm alerts these connections but it does not block the connection's traffic. Analysis of the results indicates that there are only 5 alerts containing more than one source. These 5 alerts are false positives and they were generated from the highest level in the tree, e.g. the alerts refer to IP addresses without indications for a specific protocol or port. In case of an attack on a specific service, the tree detects the attack also in lower levels, thus, an attack on this level may imply only some kind of network scan, i.e. a port scan. When an attacker tries to scan the network, the size of the tree significantly increases. Thus, by combining both anomalies, high rate on the highest level and the size of the tree, we can eliminate these false positives.

In the second mode we randomly injected DoS/DDoS attacks of different kinds, our algorithm found all of them. The injected attacks included the following attacks: ICMP flood, where a host is bombarded by many ICMP echo requests in order to consume its resources by the need to reply. Syn Attack, where random Syn packets are sent to the attacked host with intent to fill the number of open connections it can hold and therefore leave no free resources for new valid connections. DNS flood, roughly similar to ICMP flood, only more efficient against DNS servers as usually these requests require more time spent on the server side. Smurf, where the attacker spoofs many echo requests coming from the attacked host, and consequently the host is swamped by echo replies.

To reinforce our results, we compare the MULAN filter against LAD [27]. LAD is a triggered, multi-stage infrastructure for the detection of large-scale network attacks. In the first stage, LAD detects volume anomalies using SNMP data feeds. These anomalies are then used to trigger flow collectors and then, on the second stage, LAD performs analysis of the flow records by applying a clustering algorithm that discovers heavy-hitters along IP prefixes. By applying this multi-stage approach, LAD targets the scalability goal. Since SNMP data has coarse granularity, the first stage of LAD produces false-negatives. The collection of flow records on the second phase requires a buffer to hold the data and adds bandwidth overhead to the network, thus LAD uses a relatively high threshold that leads to the generation of false-negatives. One major difference between the MULAN filter and LAD is that LAD only supplies detection of attacks, which a network operator needs to process. This eases the implementation by two aspects; first, the attacks are not detected online and the second is the tolerance to false positives. The MULAN filter prevents the attacks with a negligible rate of false positives.

8.3 Controlling the Tree Size

In order to control the size of the tree in a way that it does not explode as it may do during scanning attacks, we added the following rule: When the algorithm detects an attack on any of the nodes in the tree, it stops adding children to that node until the node's rate falls below the threshold. As mentioned in Section 8.1, the reason the tree had $\sim 15,000$ nodes on that day is that two IP addresses received TCP traffic for many different ports. For each unique port, the algorithm created a node in the tree. With the above rule, when the anomalies are detected, the algorithm does not add more nodes for new ports, although it does update the counter at the parent node (in this example, the node that represents TCP). The algorithm resume adding children when the counter

at the parent is reduced and the parent is no longer categorized as an anomaly. The tree size results after applying this optimization is presented in Table 1.

Day (W.D)	*Packets Number*	*Average (Nodes)*	*Maximum (Nodes)*
4.1	1,320,049	11.3	108
4.2	1,251,319	9.3	101
4.3	1,258,076	10.2	84
4.4	1,580,440	11.2	121
4.5	1,261,848	10.4	116
5.1	1,320,049	10.8	108
5.2	2,502,808	10.9	134
5.3	1,329,336	10.5	90
5.4	2,259,146	19.7	1,035
5.5	2,602,565	11.5	104

Table 1. Tree Size Results after Optimization.

9 Discussion and Future Work

The engine presented in this paper detects DoS/DDoS attacks. We fully simulated and tested it with MIT DARPA traces and on real and recent traffic. There are two major advantages of our algorithm. One is the ability to save detailed information of the attacks while using a limited amount of memory. The second advantage is the fact that our engine finds all the attacks we expect it to find with a negligible number of false-positives. These two advantages were achieved by the use of a hierarchical data structure.

A future work can identify a way to generalize this algorithm so it can detect other types of attacks. One can create a state machine for each protocol, and identify patterns that repeat in the different state machines. Thus, the nodes in the tree will hold the state machine operations and suspicious behavior will be an anomaly from these operations.

Another algorithm could be developed for finding anomalies in different parts of a packet or a flow. For example, a normal pattern can be the number of HTTP headers, in which case, HTTP request with many headers (Apache2 attack) would be reported as an anomaly. Another example is addressing a Mailbomb attack in which the attacker sends many messages to a server, overflowing that server's mail queue and causing system crash. Each site has a different threshold of e-mail messages that can be sent by (or to) one user before the messages are considered a Mailbomb. Thus, a high rate detection engine might not discover this kind of attack. If the nodes in the tree will contain per protocol information, the algorithm will detect the unexpected number of emails.

References

1. Moore, D., Voelker, G.M., Savage, S.: Inferring internet denial-of-service activity. In: 10th Usenix Security Symposium, pp. 9–22. (2001)

2. Mit darpa project data set, http://www.ll.mit.edu/IST/ideval/index.html
3. Mahoney, M., Chan, P.: Phad: Packet header anomaly detection for identifying hostile network traffic. Technical report, Florida Tech., CS-2001-4 (2001)
4. Gil, T.M., Poletto, M.: MULTOPS: A Data-Structure for bandwidth attack detection. In: Proceedings of USENIX Security Symposium, pp. 23–38. (2001)
5. Ayres, P.E., Sun, H., Chao, H.J., Lau, W.C.: Alpi: A ddos defense system for high-speed networks. IEEE JOURNAL ON SELECTED AREAS IN COMMUNICATIONS. vol. 24, no. 10, pp. 1864–1876. (2006)
6. Brownlee, N., Mills, C., Ruth, G.: Traffic flow measurement: Architecture, http://www.ietf.org/rfc/rfc2063.txt
7. Cisco netflow, http://www.cisco.com/en/US/products/sw/netmgtsw/ps1964/index.html
8. Estan, C., Varghese, G.: New directions in traffic measurement and accounting. In: Proceedings of the 2001 ACM SIGCOMM Internet Measurement Workshop, pp. 75–80. (2002)
9. Schuehler, D.V., Lockwood, J.W.: A modular system for fpga-based tcp flow processing in high-speed networks. In: 14th International Conference on Field Programmable Logic and Applications (FPL), pp. 301–310. (2004)
10. Kompella, R.R., Singh, S., Varghese, G.: On scalable attack detection in the network. IEEE/ACM Trans. Netw. vol. 15, no. 1, 14–25 (2007)
11. Cert coordination center: tcp syn flooding and ip spoofing attacks, http://www.cert.org/advisories/CA-1996-21.html
12. Eddy, W.M.: Cisco: Defenses against tcp syn flooding attacks, http://www.cisco.com/web/about/ac123/ac147/archived_issues/ipj_9-4/syn_flooding_attacks.html
13. Bernstein, D.J.: Syn cookies, http://cr.yp.to/syncookies.html
14. Cert coordination center: smurf ip denial-of-service attacks, http://www.cert.org/advisories/CA-1998-01.html
15. Ferguson, P., Senie, D.: Rfc 2827. network ingress filtering: Defeating denial of service attacks which employ ip source address spoofing, http://www.faqs.org/rfcs/rfc2827.html
16. Kendall, K.: A database of computer attacks for the evaluation of intrusion detection systems. Master Thesis, MIT Department of Electrical Engineering and Computer Science (1999)
17. MacQueen, J.B.: Some methods for classification and analysis of multivariate observations. Proc. of the fifth 5th Berkeley Symposium on Mathematical Statistics and Probability. L. M. L. Cam and J. Neyman, Eds., University of California. vol. 1, pp. 281–297 (1967)
18. Bellovin, S.M.: Packets found on an Internet. Technical report, Computer Communications Review (1993)
19. Paxson, V.: Bro: a system for detecting network intruders in real-time. Computer Networks (Amsterdam, Netherlands: 1999). vol. 31, no. 23–24, pp. 2435–2463 (1999)
20. Adamic, L.A.: Zipf, power-laws, and pareto - a ranking tutorial, http://www.hpl.hp.com/research/idl/papers/ ranking/ranking.html
21. Adamic, L.A., Huberman, B.A.: The nature of markets in the world wide web, http://www.hpl.hp.com/research/idl/papers/webmarkets/webmarkets.pdf
22. Leland, W.E., Taqq, M.S., Willinger, W., Wilson, D.V.: On the self-similar nature of Ethernet traffic. In ACM SIGCOMM, pp. 183–193 Deepinder P. Sidhu, San Francisco, California (1993)
23. Paxson, V., Floyd, S.: Wide area traffic: the failure of Poisson modeling. IEEE ACM Transactions on Networking. vol. 3, no. 3, pp. 226–244 (1995)
24. Arsovski, I., Chandler, T., Sheikholeslami, A.: A ternary content-addressable memory (tcam) based on 4t static storage and including a current-race sensing scheme. IEEE Journal of Solid-State Circuits. vol. 38, no. 1 (2003)

25. Weinsberg, Y., Tzur-David, S., Anker, T., Dolev, D.: High performance string matching algorithm for a network intrusion prevention system (nips). In: High Performance Switching and Routing (HPSR06) (2006)
26. Snort, http://www.snort.org/
27. Sekar, V., Duffield, N., Spatscheck, O., Merwe, J.V.D., Zhang, H.: Lads: Large-scale automated ddos detection system. In: USENIX ATC, pp. 171–184 (2006)