SPADE: Statistical Packet Acceptance Defense Engine

Shimrit Tzur-David Harel Avissar Danny Dolev Tal Anker
The Hebrew University Of Jerusalem
Email: {shimritd,harela01,dolev,anker}@cs.huji.ac.il

[1]

***Abstract*— A security engine should detect network traffic attacks at line-speed. "Learning" capabilities can help detecting new and unknown threats even before a vulnerability is exploited. The principal way for achieving this goal is to model anticipated network traffic behavior, and to use this model for identifying anomalies.**

This paper focuses on denial of service (DoS) attacks and distributed DoS (DDoS). Our goal is detecting and preventing of attacks. The main challenges include minimizing the false-positive rate and the memory consumption.

SPADE: a Statistical Packet Acceptance Defense Engine is presented. SPADE is an accurate engine that uses an hierarchical adaptive structure to detect suspicious traffic using a relatively small memory footprint, therefore can be easily applied on hardware. SPADE is based on the assumption that during DoS/DDoS attacks, a significant portion of the traffic that is seen belongs to the attack, therefore, SPADE applies a statistical mechanism to primarily filter the attack's traffic.

I. 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 resources at the targeted network to such an extent that it starts dropping packets. As the dropped packets may also include legitimate traffic, the result is denial of service (DoS) to valid users.

Normally, a large number of machines is required to generate a volume of traffic large enough to flood a network. This is referred to as 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 it is impossible to identify a single network address as the source of the attack and block it. Most access links on the internet are limited in their capacity, therefore, a successful DDoS attack may only involve a single attacker that generates flows from multiple sources to bring down the system. Moreover, the size of some reported attack networks [1] suggests that one determined attacker might be capable of overloading even the largest access link.

Currently, detection of such attacks is done by monitoring IP addresses, ports, TCP state information and other attributes to identify anomalous network sessions. The weakness of simply applying such a methodology is that accumulated state information grows linearly with the number of flows; thus lacking scalability.

In designing a fully accurate and scalable engine, one needs to address the following challenges:

1) Prevention of Threats: The engine should be able to prevent threats from entering the network. Threat prevention (and not just detection) makes the engine more complicated, mostly because of the need to work at line-speed. This makes the engine potentially a bottleneck - thus 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 conversely, when the engine drops normal traffic.
3) 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.

This paper presents SPADE: Statistical Packet Acceptance Defense Engine. SPADE detects and prevents DoS/DDoS attacks from entering the network. SPADE maintains multiple dynamic hierarchical data structures to measure traffic statistics. These dynamic structures maintain the information used in identifying offending traffic. Each hierarchical structure is represented as a tree. Each level of the tree represents a different aggregation level. The main goal of the tree is reflecting the network behavior for efficient identification of an attack. 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.

SPADE starts dropping packets only when the device is overloaded and might crash. I.e. the maximum service

[1]This is the authors copy of the paper that will appear in HPSR 2010.

rate is about to be reached. At such a point the algorithm enters a defense mode to focus on traffic load reduction.

Following each packet's arrival the algorithm updates the relative tree nodes (or adds new nodes). Every *N* packets. the engine examines the rate of each node; if it is a leaf node and its rate is above a threshold, a spread-flag is marked and a descending nodes are added. Once the rate drops, the node's descendants are removed and the spread-flag is turned off. This enables holding detailed information of active incoming flows that may potentially become suspicious, with a reasonable memory size. In addition, the algorithm predicts the rate of the next *N* packets; if the predicted rate is above a threshold, the algorithm analyzes the tree to identify which packets to remove, relative to their contribution to the excess load.

Many DoS/DDoS detection systems use time intervals. At the end of each interval they try identifying suspicious traffic. As a result, short attacks that start and end within one interval might be missed. For example, the system presented in [2] tracks changes in traffic, if a change occurs at the same time-scale of the measurement interval, the system may miss the attack. In SPADE an interval spans over *N* packets, therefore when the load increases the intervals become shorter and very short attacks can be detected.

Another major advantage of SPADE over present solutions is that it does not require any learning period of "attack free" traffic in order to operate correctly. SPADE can gain form such traffic, but in essence it is "Plug and Play", as it can start protecting against an ongoing attack from the first round onwards and will stabilize to discard only attack packets over time. Moreover, SPADE has a mechanism to detect and prevent long and moderate attacks. Such attacks usually are too "slow" for current engines to detect, and thus avoid detection.

In our previous work [3] MULAN-filter is presented. Both systems, MULAN and SPADE, protect the network against DoS and DDoS attacks using a tree data-structure to obtain a scalable solution. The two systems differ by their methods. MULAN learns the network and when it detects an anomaly in the traffic rate, it identifies the attack and filters the offending traffic. SPADE, on the other hand, filers packets only when the network is under a risk of crashing. This difference has a major impact on performance. MULAN is exposed to false-positives, i.e. if the rate of the network grows sharply by normal traffic, MULAN identifies this as an anomaly and it raises an alert. This happens regardless of the load at the target. SPADE is much more resistant, and thus even if there are sharp changes in the rate, SPADE does not interfere as long as the total traffic can be handled. Moreover, since SPADE filters packets only when there is an actual risk of crashing, packets should be dropped one way or another.

Another difference is the use of the tree. MULAN clusters the nodes at each level in the tree by one of the chosen attributes, e.g. IP addresses. This clustering decelerates the process of identification and filtering of attack's packets. SPADE uses the tree to reflect the incoming traffic, and its consolidated nodes do not affect the algorithm effectiveness, as we detail below.

MULAN uses time intervals to examine the tree and decide whether to spread a cluster or not. This cannot provide protection against short attacks not only for the reason mentioned above, but also because of the aggregation method. When MULAN identifies high rate at a cluster, it splits the cluster and examines its specific samples to find the attacked IP. From the time the high rate of the cluster is identified until the specific target is located, the attack may be over. SPADE maps each IP address seen by the engine to a node in the first level of the tree. To target the scalability problem, SPADE consolidates IP addresses with low rate into a range by holding a node for the prefix. A prefix splits again when the rate of the prefix node grows. The use of prefixes was previously presented in [4] and [5]. However, the approaches and purposes are different. The authors of [4] provide valuable information for the network administrator by identifying the dominant traffic type, and the goal of [5] is packet classification; our goal is to automatically defend the network.

The last difference is the algorithm complexity. SPADE uses statistical methods to identify attacks and as a result it is much more efficient than MULAN, as detailed later. SPADE was implemented in software and was demonstrated on 10 days of incoming traffic at our School of Computer Science. The results show that with the use of TCAM SPADE can work at high wire speed with great accuracy.

II. 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 *attack-free* traffic.

MULTOPS [6] is a denial of service bandwidth detection system. In that system, each network device maintains a data structure that monitors certain traffic characteristics. The data structure is a tree of nodes containing packet rate statistics for subnet prefixes at different aggregation levels. The detection is performed by comparing the inbound and outbound packet rates. As MULTOPS fails to detect attacks that deploy a large

number of proportional flows to cripple the victim, it will not detect many of the DDoS attacks.

PacketScore [2] is a statistics-based system against DDoS attacks. It is based on packet-scoring approach. Arriving packets are given scores based on their TCP/IP attribute values, as compared to nominal traffic profile. In addition to the difficulty in detecting short attacks, PacketScore does not handle properly mixed attacks and for this reason generates a high false-positive rate. PacketScore assigns score to each attribute based on currently measured histogram and the nominal profile. The implementation complexity arises from calculating these two histograms for each packet attribute.

ALPI [7] is an extension of PacketScore. It uses a leaky-bucket scheme to calculate an attribute-value-variation score for analyzing deviations of the current traffic attributes. The authors proposed two additional scoring methods, one simpler but less accurate and the complexity of the second is between the other two. All the three schemes need to build histograms in order to generate nominal profiles and they all generate relatively high false-negative rate.

Many DoS defense systems, like [8], instrument routers to add flow meters at either all or at selected input links. Flow measurement approach does not scale. Updating per-packet counters in DRAM is impossible given today's line speed. Cisco NetFlow [9] attempts to solve this problem by sampling, which affects measurement accuracy. Some of the systems, like [10] and [11], use Cisco NetFlow as their input. Estan and Varghese presented in [12] algorithms that use an amount of memory that is a constant factor larger than the number of large flows. The solutions presented lack accuracy.

Schuehler et al., [13], use an FPGA implementation of a modular circuit design of a content processing system. It requires a large per-flow state, supporting 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.

Other solutions, [14], [15], use aggregation to scalably detect attacks. Aggregation affects accuracy; for example, due to behavioral aliasing, the solution presented in [14] doesn't produce good accuracy. Behavioral aliasing may cause false-positives and false-negatives results. Another drawback of this solution is its vulnerability against spoofing.

LAD [16] 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 in the second stage 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.

III. Notations and Definitions

- L_n - the number of levels in the tree.
- N - the size of the sample interval.
- *History Tree* - the regular traffic distribution tree.
- *Attack Tree* - the attack traffic distribution tree.
- *N-Tree* - the N packets traffic distribution tree.
- *Abandon Tree* is a tree representing the odds that packets have to be discarded in the next round.
- *Network Capacity* is a predefined rate that the target network can handle without a risk to crash.
- *Attack Threshold* is a predefined fraction of the *Network Capacity* that once crossed will make SPADE switch to behaving like under attack.

IV. SPADE Design

A. Data Structures

All tree structures aggregate packets' statistics using predefined parameters.

The *N-Tree* is a tree containing distribution of the current N packets - where each branch holds a counter of the number of packets with the branch's IP destination, protocol and specific data. When a packet is processed, it is first added to its branch in the tree (if the branch does not exist, it is created) and counters along the whole branch are updated accordingly. This tree profiles the current traffic and allows us to find anomalies by comparing it against the other trees.

The *History Tree* is a weighted average over the *N-Trees* with a long tail to ensure stability. The long tail is used to ensure that long moderate attacks cannot become part of the normal history - however the length of the tail is a parameter left to the administrator to determine according to the threat level it anticipates and the response time it expects. In our numeric tests at each round the *History Tree* is updated as follows:

$$\textit{History Tree} = \textit{History Tree} \times 0.95 + \textit{N-Tree} \times 0.05.$$

The *Attack Tree* is a weighted average over the *N-Trees* during an attack, with short tail as attacks are not necessarily stable. This tree is created when communications load crosses the predefined *Attack Threshold*, and while load remains over the threshold. It is updated instead of the *History Tree*. This helps in preventing long and moderate attacks to be inserted into the history. In our

numeric tests the *Attack Tree* is updated as follows:

Attack Tree = *Attack Tree* $\times$ 0.5 + *N-Tree* $\times$ 0.5.

The *Abandon Tree* is constructed differently. The nodes in the *Abandon Tree* do not count packets, but rather the odds of packets to be discarded in the next round. The *Abandon Tree* is constructed only when the load exceeds the *Attack Threshold*. Packets are dropped only when the prediction is that at the next sample the load might exceed the *Network Capacity*. The *Abandon Tree* is constructed by subtracting the normal traffic profile represented by the *History Tree* from our current traffic profile represented by the *Attack Tree*. This construction identifies the branches responsible for the increase in traffic and these branches are assigned odds accordingly. Experimentally, the *Abandon Tree* size quickly stabilizes to hold only branches of real attack, leading to lower false-positives.

B. Predicting the Rate of Next Sample

Rate prediction is calculated at the end of each sample. This prediction determines whether the engine should start constructing the *Attack Tree* or to continue adding nodes to the *History Tree*. The prediction rate is calculated as follows:

AvgRate = CurrentRoundRate $\times$ 0.5 + AvgRate $\times$ 0.5
AvgDelta = (CurrentRoundRate-PreviousRoundRate) $\times$ 0.5 + AvgDelta $\times$ 0.5.

AvgRate recurrence relation is responsible for predicting the current typical rate with bias towards current data, while AvgDelta is responsible for predicting the change based on recent data. This simple mechanism is powerful over traffic data, since traffic usually has a large variance. The above approach may not be accurate at each round, but the average stabilizes after few rounds. The predicted rate is (1+*Margin*)$\times$(AvgRate+AvgDelta). The *Margin* is a small percentile (1%-5%) that allows control over how much we regularly overshoot beyond the *Network Capacity*. As our engine is probabilistic, there is a certain chance that, for few seconds, the rate will exceed the assigned *Network Capacity*, and this can be controlled using a larger *Margin*.

V. The Algorithm

1) The Algorithm Parameters: The simulations constructed three levels' trees:

Level 1 - the destination host IP, or network IP.

Level 2 - the upper layer protocol encapsulated within the IP packet. This level is less susceptible to overflow by malicious attacker trying to cause overflow in the tree data structures. Specific protocols are manually configurable with basic defaults, and therefore the maximum number of nodes in this level is known.

Level 3 - the protocol specific information, which is, for example, port number for TCP and UDP or operation code in ICMP. This level can be configured for specific and non-standard protocols or, in case no protocol specific information is available, a wildcard can be used.

An additional parameter to determine is *N*. The value of *N* depends on the network's typical rate.

In our simulation, we run our engine on a small network with a *Network Capacity* of 10000 packets per second so we set *N* to be 1000. The rate of large networks can reach 1000000 packets per second, and an appropriate value of *N* should be around 100000.

When the network rate exceeds the *Attack Threshold*, the engine enters into a defensive mode. In the SPADE implementation we choose this threshold to be 80% of the *Network Capacity*.

Other parameters that can be tuned are the *Margin* percentile and thresholds that control the size of the trees. We use a *Margin* of 2% which, in turn, makes sure over 99% of the samples have legal rates; in the single samples when *Network Capacity* is breached, it is by less than 400 packets - well within the capabilities of a standard buffer.

The last parameter is a threshold that helps the engine control the size of the *History Tree* and the *Attack Tree*. In both trees, if a node at the first level gets less than 1% of the traffic that the tree represents, the branch rooted by this node is pruned and the node is consolidated with other nodes with the same prefix into one node that represents the network address. During each sample, if a consolidated node gets traffic beyond this threshold, the branches are reconstructed.

2) Non-Attack Mode: As long as rates do not go over our *Attack Threshold*, the operation of the algorithm is very simple - on each packet's arrival the engine updates the *N-Tree*; and at the end of a sample it updates the *History Tree*, and predicts the rate for the next sample.

3) Attack Mode: The attack mode starts when the predicted rate goes over the *Attack Threshold*. In the attack mode, the operation of the algorithm is changed - updating the *History Tree* is stopped to avoid further "poisoning" of the history and the *Attack Tree* starts getting updated. The *N-Tree* and the *Attack Tree* are updated as in the non-attack mode. In addition, at the end of each sample, the engine creates the *Abandon Tree* by subtracting the *Attack Tree* from the *History Tree*. As long as the predicted traffic rate is between the *Attack Threshold* and the *Network Capacity*, no packet will be discarded. However once capacity is breached, the engine uses the *Abandon Tree* to discard packets biased towards attack packets. The discarding odds of each node in the *Abandon Tree* are calculated using each branch's

relative weight in the tree. Let's assume, for example, that the discarding percentage should be $T\%$, and the tree holds two branches, one accounts $A\%$ of total traffic in the *Abandon Tree* and the other $B\%$ of it. These branches correspond to two branches of the *N-Tree* with weight $a\%$ and $b\%$ of the entire round traffic. The first branch packets will be discarded with odds $(T*(A/100))/a$ and the second with odds $(T*(B/100))/b$. This ensures that overall we have a discarding percentage of $T\%$ with bias according to the tree's entries. From now on, on each packet's arrival, the algorithm draws a random number in the range $0-1$ and discards the packet if the drawn number is less than the discarding odds. When a packet is discarded it is also subtracted from the *History Tree* to ensure that, over time, our history will become clean of attacks from previous rounds.

VI. Optimal Implementation

The main bottleneck that might occur in our engine are the *N-tree* lookup, which is performed on arrival of each packet, and the updates of the *History Tree* and the *Attack Tree* at the end of each sample. 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) [17]. 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 [18] detects signatures-based attacks that were drawn from Snort [19]. 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 SPADE's trees 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, w is taken to be $\sum_i |L_i|$. 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, $l < j \leq n$. The last row contains w "don't care" bytes, thus indicating that there is no path for the packet and

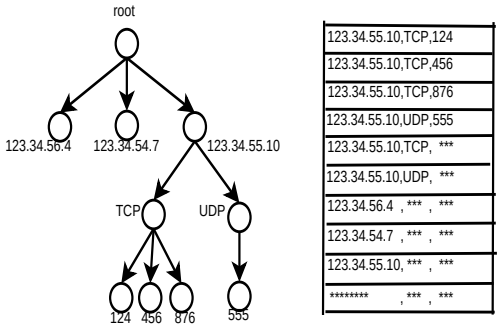

Fig. 1. TCAM Population

providing the default match row. This population is done for each one of the above trees, the *N-tree*, the *History Tree* and the *Attack Tree*. Figure 1 presents an example of a tree structure and the populated TCAM for it. When a packet arrives, the algorithm extracts the relevant data, creates a TCAM key and looks for a TCAM match. For each row in the TCAM we associate a counter and a pointer to the TCAM row of the parent node. When there is a TCAM match (except in the last row), the algorithm updates the counter of the matched row and follows the pointer to the parent node's row. The algorithm adds and removes only leaves. Since leaves can appear at any level in the tree, and there is no importance to the order of the rows in the same level, the TCAM is divided into L_n parts. This way the TCAM can be easily updated while keeping the right order of the populated rows.

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 small number of updates. Figure 2 presents the average number of TCAM updates for each 1000 packets of the incoming traffic of the School of Computer Science. The figure illustrates the creation of the tree. During the creation of the tree there are many insertions, thus the number of updates is relatively high.

The total average update rate is 593.82 updates for 1000 packets, more than 99.9995% of the values are below 1750, with a small number of scenarios when the engine has to deal with up to 5000 updates. A typical rate of large networks can be million packets per second. A software engine will not be able to perform the updates in time, though 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. Our findings show that the required performance is well within the range of the available TCAM update rates.

VII. Experimental Results

The quality of the algorithm is determined by two factors: scalability and accuracy. In order to analyze

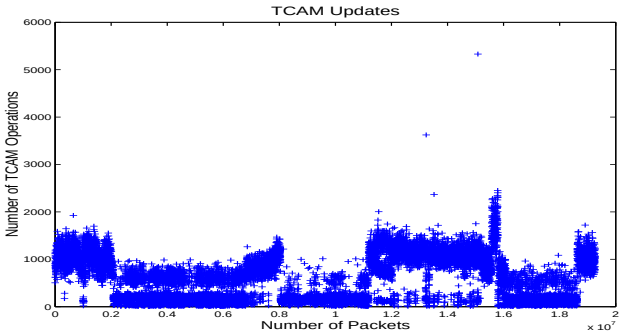

Fig. 2. TCAM Updates

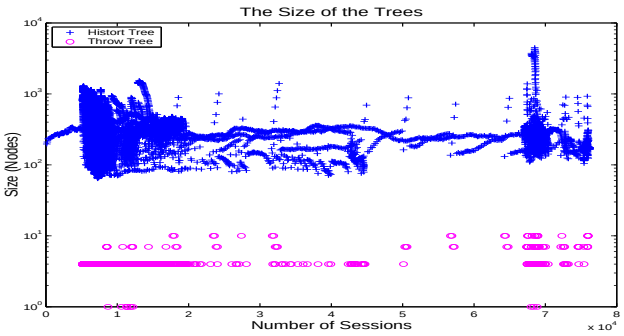

Fig. 3. Tree Size vs. Number of Flows

the performance of the algorithm, a simulation was implemented and tested on real traffic from our School of Computer Science.

A. Scalability

Demonstration of scalability requires analyzing the memory requirement at every stage of execution. We found that the average size of the *History Tree* is 290.96 nodes and the average size of the *Abandon Tree* is 1.52. The *Attack Tree* size is similar to the size of the *History Tree*. The size of the *N-Tree* depends on N. In the worse case scenario, if for each packet the algorithm adds a complete branch in the tree, i.e. L_n nodes, the size of the tree will be $N \times L_n$. In our implementation, where $L_n = 3$, even for large networks, where N is around 10^5, the maximum number of nodes is 300000. Since each node only holds a counter (2 bytes) and a pointer (4 bytes), the required memory for all trees is below 2 MB, that can be saved in a TCAM with a reasonable cost.

Another major advantage of the algorithm is that the tree size increase very slowly with respect to the the number of flows. This is clearly demonstrated in Figure 3 (Note that the y axis is a logarithm scale). In general, for any number of flows the size of the *History Tree* is below 1100 nodes. There are few cases where the size of the tree exceeds 1100 nodes. These cases occur when the traffic contains attacks as can be seen by the corresponding *Abandon Tree* size.

Memory consumption is one of the major limitations when trying to extract per-flow information. The main problem with flow measurement approach is lack of scalability. Memory consumption of previously proposed algorithms is directly influenced by the number of flows. In our engine the memory consumption does not grow linearly with the number of flows, thus the accuracy is not affected.

B. 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 when it drops normal traffic. This section describes the simulated attacks and presents the accuracy results on the real traffic from our School of Computer Science.

1) Attacks Generation: We randomly added six types of bandwidth attacks, attacking random or predefined hosts in the school network. Normal packet is different from an attack packet only by a tag that was not used by the algorithms. The tags were used for the analysis.

1) ICMP flood. An attack where a host is bombarded by many ICMP echo requests in order to consume its resources by the need to reply.
2) ICMP reflection. An attack where the attacker spoofs many echo requests coming from the attacked host, and thus the host is swamped by echo replies.
3) Smurf. An ICMP reflection attack amplified using network configuration.
4) DNS Flood. Resembles ICMP flood, but more efficient against DNS servers, as such requests require more time spent on the server side. This attack can also be amplified by registering the attacked machine as DNS for many bogus addresses and using real world DNS servers as Smurf amplifiers, both attacks appear identical at the attacked side.
5) Syn Storm. An attack where Syn packets are sent to a specific port on the attacked host with intent to overflow the resources of the server side application on the attacked port. This type of attack is most efficient when a FW/IPS blocks some ports as it targets the same port as the real communication.
6) Rand Syn. An attack where random Syn packets are sent to ports on the attacked host with intent to fill the number of open connections it can hold and leave no free resources for new valid connections.

The tests were executed on 3 different configurations:

Configuration A - one host is attacked with increasing intensity, trying to mask away the behavior by letting the

algorithm "learn" the flow to make it difficult to detect. This configuration specifically tests our long tail history and the *Attack Threshold*.

Configuration B - several hosts (some random) are attacked with a combination of attacks, including simultaneous attacks on the same host.

Configuration C - many random hosts are attacked by short and very intense bursts. This configuration specifically tests how contained are the false-positives that stem from the time it takes SPADE to stabilize.

2) Results: Figure 4 presents the original traffic rates compared to the rates of the traffic after inserting attacks and rates after cleaning. The x axis is the number of samples (each of size $N = 1000$), and the y axis is the average rate over this sample calculated by dividing N packets by sample time. The graphs are not of the same length as the original traffic had 1000000 packets (1000 samples), after adding attacks, there are more packets and therefore more samples. We see that the clean traffic does not resemble the original. This is expected since our goal is not to eliminate all attacks' traffic but merely to limit it to the *Network Capacity*. This ensures that when a communications spike is over our assigned capacity, we will make optimal use of the capacity to allow as much traffic as possible. We note that in the original traffic itself there are already abnormal points of overflow of the capacity due to normal packet rate variability.

Table I shows results of 7 tests undergone with each configuration file, listing the number of normal traffic packets discarded, number of attack packets discarded and FP%. The optimal number was calculated by summing up the overflowing packets at each sample based on the known capacity to illustrate that SPADE discards approximately the optimal number. This is important when our defenses are triggered by natural communication spikes rather than deliberate attack. The worst results are for configuration *B*, which is to be expected as it is the most challenging and include many simultaneous attacks, with average FP% of 1.77. Configuration A has average FP% of 1.49, since one large attack is quickly characterized and FP% becomes negligible, especially taking into effect real traffic spikes like noted earlier. The best results with FP% of 1.04 are for configuration *C*, where the large number of attacks implies many instances of learning new attack characteristics, discarding some normal traffic in the process. The large intensity of each attack makes sure this effect is minimal.

In Figure 5 we can see the fraction of normal packets discarded (normal discarded/total discarded) plotted as y over the samples range (note that y axis is in units of 0.001). The configuration is *A*, where the host is moderately but increasingly attacked from sample 2000 onwards, with a large spike in attack intensity at sample 4300. It is clearly visible that, as the attack continues, the FP% decreases, and while the spike causes some false positives, due to our *Margin* overshoot, we can see that a lengthy attack causes a decrement in FP%. Since this attack is moderately ceasing, we see that, at the end of the attack, the ratio becomes almost constant; if however the attack was to continue with the same intensity, this relation would have gone down towards 0. [20] concludes that the bulk of the attacks last from 3 to 20 minutes, meaning about $1800-12000$ samples, giving SPADE enough time to characterize and effectively block the attack for a significant part of its duration.

Figure 6 shows, in configuration *B*, the advantages of SPADE over a solution that discards packets without bias towards attack packets. The y axis is the total number of packets discarded, plotted against the number of samples. The "steps"-like behavior of the graphs is caused by the different attacks and variations in attack intensities. Major attacks were injected at samples 2000, 4600, 8000 and 16000, each corresponding to a new step in the attack graphs. The sections between the steps are not entirely flat as they are not entirely clean of attacks. An ongoing background attack with low intensity, like the one injected at sample 4000, causes a slow but steady rise. Near the end, where the traffic is completely free of attacks, we notice a totaly flat line. Attacks with constant intensity (8000 or 16000) cause a linear like behavior of discarded attack packets. After a preliminary learning stage the normal traffic stops being affected, as clearly illustrated by the long flat sections in SPADE normal graph. Notice that in the naive normal graph there is a steady increase. In summary, we can clearly see that SPADE is a significant improvement over the naive solution due to better attack characterization. The conclusion is that better characterization of attacks and of future traffic can even improve the results of SPADE.

VIII. Discussion and Future Work

The engine presented is used to detect DoS/DDoS attacks. We fully simulated our algorithm and tested it on real and recent traffic. There are two major advantages of our algorithm versus previous works. One is the ability to save detailed information of the attacks whilst using a limited amount of memory. The second advantage is the fact that our engine finds all the bandwidth attacks with a negligible number of false-positives. These two advantages were achieved by the use of a multitude of hierarchical data structures and statistical tools.

A future work may target attacks that slow down the server without significantly affect the utilization of the

Configuration A				Configuration B				Configuration C			
Normal	*Attack*	*Optimal*	*FP%*	*Normal*	*Attack*	*Optimal*	*FP%*	*Normal*	*Attack*	*Optimal*	*FP%*
41	103208	97923	0.039	2065	83996	83361	2.399	807	525145	499691	0.153
724	76131	74358	0.942	417	52577	49661	0.786	2697	373002	376724	0.717
283	90600	91074	0.311	379	73401	67643	0.513	5346	391110	384590	1.348
11718	319077	322240	3.542	85538	2016913	2056761	4.068	24516	1540151	1544661	1.566
45	122591	116255	0.036	326	61765	57467	0.525	678	516434	486710	0.131
43141	829376	866682	4.944	55834	2427916	2448462	2.247	51348	1889554	1947882	2.645
3112	501007	481588	0.617	20198	1046486	924609	1.893	19691	2581519	2576794	0.756

TABLE I

FALSE POSITIVES RATE

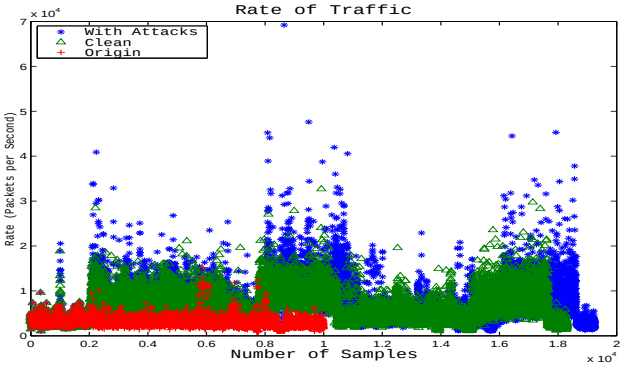

Fig. 4. Packet Rate

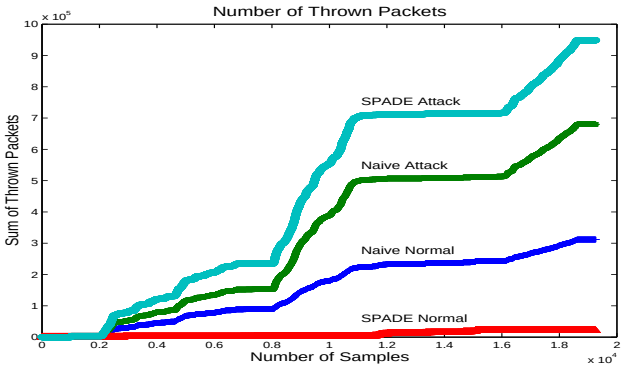

Fig. 6. Overall Discarding of Packets - Conf. B

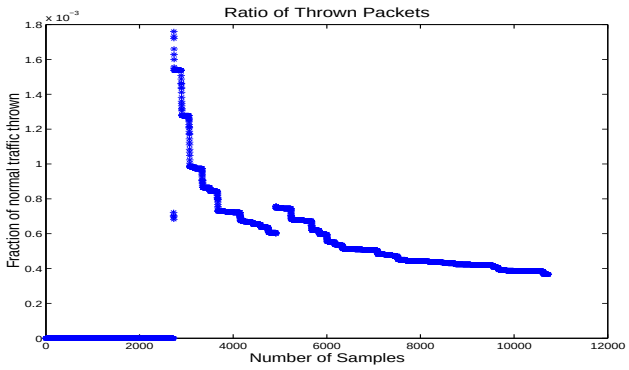

Fig. 5. FP Fraction - Configuration A

incoming link, i.e. requests for long-duration tasks at the server.

REFERENCES

[1] "Cops smash 100,000 node botnet, 2005," http://www.cert.org/advisories/CA-1996-21.html.

[2] Y. Kim, W. Cheong, L. Mooi, C. Chuah, and H. J. Chao, "Packetscore: Statistics-based overload control against distributed denial-of-service attacks," in IEEE Infocom, 2004, pp. 2594–2604.

[3] Shimrit Tzur-David and Danny Dolev and Tal Anker, "Mulan: Multi-level adaptive network filter," in SecureComm: 5th International ICST Conference on Security and Privacy in Communication Networks, 2009, pp. 71–90.

[4] C. Estan, S. Savage, and G. Varghese, "Automatically inferring patterns of resource consumption in network traffic," in SIGCOMM, 2003, pp. 137–148.

[5] Y. Zhang, S. Singh, S. Sen, N. G. Duffield, and C. Lund, "Online identification of hierarchical heavy hitters: algorithms, evaluation, and applications," in Internet Measurement Conference, 2004, pp. 101–114.

[6] T. M. Gil and M. Poletto, "Multops: a data-structure for bandwidth attack detection," in Proceedings of 10th Usenix Security Symposium, 2001, pp. 23–38.

[7] P. E. Ayres, H. Sun, H. J. Chao, and W. C. Lau, "Alpi: A ddos defense system for high-speed networks," IEEE JOURNAL ON SELECTED AREAS IN COMMUNICATIONS, vol. 24, no. 10, pp. 1864–1876, 2006.

[8] N. Brownlee, C. Mills, and G. Ruth, "Traffic flow measurement: Architecture," http://www.ietf.org/rfc/rfc2063.txt.

[9] "Cisco netflow," www.cisco.com/en/US/products/ps6601/products_ios_protocol_group_home.html.

[10] H. Choi, H. Lee, and H. Kim, "Fast detection and visualization of network attacks on parallel coordinates," Computers & Security, 2008.

[11] A. Lakhina, M. Crovella, and C. Diot, "Diagnosing network-wide traffic anomalies," in ACM SIGCOMM, 2004, pp. 219–230.

[12] C. Estan and G. Varghese, "New directions in traffic measurement and accounting," in Proceedings of the 2001 ACM SIGCOMM Internet Measurement Workshop, 2002, pp. 75–80.

[13] D. V. Schuehler and J. W. Lockwood, "A modular system for fpga-based tcp flow processing in high-speed networks," in 14th International Conference on Field Programmable Logic and Applications (FPL, 2004, pp. 301–310.

[14] R. R. Kompella, S. Singh, and G. Varghese, "On scalable attack detection in the network," in Internet Measurement Conference, 2004, pp. 187–200.

[15] S. S. Kim and A. N. Reddy, "A study of analyzing network traffic as images in real-time," in IEEE Infocom, 2005, pp. 2056–2067.

[16] V. Sekar, N. Duffield, O. Spatscheck, J. V. D. Merwe, and H. Zhang, "Lads: Large-scale automated ddos detection system," in Proceedings of USENIX ATC, 2006, pp. 171–184.

[17] I. Arsovski, T. Ch, and A. Sheikholeslami, "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, pp. 155–158, January 2003.

[18] Y. Weinsberg, S. Tzur-david, T. Anker, and D. Dolev, "High performance string matching algorithm for a network intrusion prevention system (nips)," in HPSR, 2006, p. 7.

[19] "Snort," http://www.snort.org/.

[20] D. Moore, G. M. Voelker, and S. Savage, "Inferring internet denial-of-service activity," in Proceedings of the 10th Usenix Security Symposium, 2001, pp. 9–22.