High Performance String Matching Algorithm for a Network Intrusion Prevention System (NIPS)

Yaron Weinsberg Shimrit Tzur-David Danny Dolev
The Hebrew University Of Jerusalem
Email: {wyaron,shimritd,dolev}@cs.huji.ac.il

Tal Anker
Radlan - a Marvell Company
Email: tala@marvell.com

Abstract— **Intrusion Detection systems (IDS) were developed to identify and report attacks in the late 1990s, as hacker attacks and network worms began to affect the internet. Traditional IDS technologies detect hostile traffic and send alerts but do nothing to stop the attacks. Network Intrusion Prevention Systems (NIPS) are deployed in-line with the network segment being protected. As the traffic passes through the NIPS, it is inspected for the presence of an attack. Like viruses, most intruder activities have some sort of signatures. Therefore, a pattern-matching algorithm resides at the heart of the NIPS. When an attack is identified, the NIPS blocks the offending data. There is an alleged trade-off between the accuracy of detection and algorithmic efficiency. Both are paramount in ensuring that legitimate traffic is not delayed or disrupted as it flows through the device. For this reason, the pattern-matching algorithm must be able to operate at wire speed, while simultaneously detecting the main bulk of intrusions. With networking speeds doubling every year, it is becoming increasingly difficult for software based solutions to keep up with the line rates. This paper presents a novel pattern-matching algorithm. The algorithm uses a Ternary Content Addressable Memory (TCAM) and is capable of matching multiple patterns in a single operation. The algorithm achieves line-rate speed of several orders of magnitude faster than current works, while attaining similar accuracy of detection. Furthermore, our system is fully compatible with Snort's rules syntax, which is the de facto standard for intrusion prevention systems.**

I. Introduction

VanDyke Software [1] has just announced the results of a security-related survey. Although viruses were the most significant threats faced by the respondents, 66% of the companies chose system penetration as the largest threat to their enterprises. The survey also revealed that firewalls are not always effective against penetrations as the average firewall is designed to deny clearly suspicious traffic; for example, an attempt to telnet to a device when corporate security policy completely forbids telnet access.

The inadequacies inherent in current network defense mechanisms have motivated the development of a new breed of security products, called *Network Intrusion Prevention Systems (NIPS)*. These systems deploy proactive defense mechanisms designed to detect malicious packets within normal network traffic. Once identified, the malicious traffic is usually blocked.

Most NIPS products are basically *Intrusion Detection Systems (IDS)* that operate **in-line** and are thus dependent on pattern-matching to recognize malicious content within individual packets (or across groups of packets). NIPS systems are usually comprised of two major components: a pattern-matching engine and a complementary packet classification engine. The pattern matching engine's input is a packet and its output is a set of matched patterns belonging to the set of well known attack's signatures.

There are a number of challenges in implementing a NIPS device; These all stem from the fact that a NIPS device is designed to work in-line, thus presenting both a bottleneck and a single point of failure. If the NIPS device fails, it can seriously impact the availability of the entire network. If a NIPS device struggles to keep up with traffic speeds, it becomes a bottleneck, thus increasing latency and reducing throughput.

The current trend for integrating security with network switches and routers both at the network edge and at the enterprise gateway, implies that the NIPS device must meet stringent network performance and reliability requirements.

This work is a part of a research project aimed at designing and implementing a hardware based NIPS device [2]. A core component of any NIPS appliance is its pattern matching component. In this paper we present a novel pattern matching algorithm, called *RTCAM* (Rotating TCAM), which suggests the usage of an off-the-shelf TCAM and some additional logic that can be implemented in HW. The RTCAM algorithm enables the NIPS appliance to operate at an aggregate rate of several gigabit per second.

A. Snort's Database

Snort [3] is an open source NIPS that is commonly used in industry. Snort contains a database of rules with several thousands of attack signatures. Each of Snort's rules contains a header and several content fields. The header part consists of a packet identifier (protocol, source/destination IPs and ports), while the content part contains one or more patterns that may have some correlation between them. A rule is matched only if all of its patterns are matched with the expected correlation among them. The Snort rule syntax is the de facto industrial standard. NIPS devices which are compliant with this standard have a great advantage - the same database can be transparently imported from one engine to another. As opposed to several hardware based NIPS devices, our solution is fully Snort compatible.

Internally, Snort uses a software based pattern matching algorithm, a variant of the Boyer-Moore algorithm, which is applied to a set of keywords held in an Aho-Corassick like

keyword tree. Current Snort performance on high-end PCs is 400 Mbps for large packet sizes.

II. Notations and Definitions

DEFINITION 1 *A pattern P is define as a string of characters from an alphabet Σ which need to be identified within the input text. A sub-pattern P_s is define as a sub-string of a pattern P.*

DEFINITION 2 *A search window is define as a part of the input text within which a sub-pattern is searched.*

DEFINITION 3 *A shift is define as the number of bytes the algorithm can safely skip without losing any of the patterns match. Formally, a pattern P of length m occurs with shift s in text T of length n if $0 \leq s \leq n-m$ and $T_{[s+1..s+m]} = P_{[1..m]}$.*

DEFINITION 4 *A string-matching algorithm is define as follows: Define the text as an array $T_{[1..n]}$ and the pattern as an array $P_{[1..m]}$, and assume that the elements of P and T are characters drawn from a finite alphabet Σ. The string-matching problem is the problem of finding the shifts. The extended problem of finding multiple patterns in a given text is called "multiple pattern matching" and its goal is to output the positions of all occurrences of the patterns in the text. We formalize the multiple pattern string-matching problem by: $T = t_{[1..n]}$ and a set of r patterns P, such that $P_j \in P$, where $1 \leq j \leq r$ (the patterns may have different lengths).*

III. Related Work

The string matching algorithm is an essential building block for NIPS. Most algorithms are either too complex to be implemented in hardware, or provide poor performance (See [4], [5], [6], [7], [8], [9], [10], [11], [12] for some known algorithms). This section presents state-of-the-art *hardware* based algorithms.

A. Parallel Bloom Filters

The Parallel Bloom Filters [13], [14] algorithm uses a bloom filter for each possible pattern length. Briefly, a bloom filter uses several hash methods that reduce the potential patterns space that may match the search window. The paper gives a reasonable cost-space tradeoff by using four parallel engines. The algorithm can push four bytes in a single clock cycle, with a throughput of approximately 2.46Gbps. The fact that each different pattern length requires a separate bloom filter is a limiting factor, especially when dealing with very long virus definitions that can be thousands of bytes long.

B. Network Processor Pattern Matching

The work of Liu et al. [15] describes a shift based algorithm that uses a network processor with a memory based hashing engine. It uses a prefix sliding window of length w, which shifts from the leftmost byte to the rightmost byte of the text. Their algorithm only supports simple patterns, with no identification of correlation of patterns. The algorithm uses a shift table (of size $(2^8)^w$) that includes all possible w bytes combinations. At the time of the introduction of this algorithm, setting w to 4 was sufficient, since most Snort signatures were that long. However, the length of signatures used today is an average 12 bytes. Maintaining a table for $w = 12$ is impractical. A major limitation in using a small window size is the large number of false positive matches. For the proposed $w = 4$, the algorithm obtains an average shift of ~ 2.

C. TCAM Pattern Matching

A TCAM [16] 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 state-of-the-art in pattern matching algorithms that uses TCAM has been presented by Lakshman et al. [12]. The proposed algorithm places the set of the attacks' signatures in the TCAM and deploys a simple "brute-force" pattern-matching algorithm. A key of w bytes is consecutively constructed from the packet (by shifting the text one byte at a time), and the TCAM looks for a match. Assuming that the packet length is n, the algorithm requires n TCAM lookups. If a single TCAM lookup takes 4 ns, this brute force algorithm yields a matching speed of $8 \times n[Bytes]/4n[ns] = 2$ Gbps (denoted as *Naive_Scan_Rate*).

IV. The Rotating TCAM (RTCAM) Algorithm

This section presents the RTCAM pattern matching algorithm. We begin with a high level description of the RTCAM algorithm. We will then introduce its internal data structures and conclude with an example.

A. Populating The TCAM

A TCAM of size M, can be configured to hold $\lceil M/w \rceil$ rows, where w is the TCAM width. The TCAM is populated in an *offline* process with two phases. In the first phase, we split the rule's signatures (patterns) to fit in the chosen w. Patterns longer than w occupy more than one row. The pattern in the first row is marked as the 'pattern's prefix". Short patterns, marked as prefixes, are padded to the size of w by using the TCAM "don't care" bit. Each TCAM row has a corresponding shift value that states the number of bytes we can safely shift in the packet when a match occurs. In this phase all TCAM rows have a shift value of 0. In the next phase, a set of shifted sub-patterns is created for each pattern prefix, by shifting the prefix to the right, losing the rightmost character and adding *don't care* at the left. Such a rotation increases the shift value by one[1]. We repeat this process until all of the pattern's bytes are don't care (Note that similar sub-patterns are collided together). Since a TCAM lookup returns the first matched row, the TCAM rows are ordered according to their shift values in an ascending order. The last row corresponds to the maximum shift value and contains w *don't care* bytes, thus providing the default match row.

Note that a TCAM manufacturer can easily implement the rotating effect by only storing the prefix pattern and utilizing internal clock cycles with a slightly modified TCAM logic. This will further reduce the amount of required TCAM space.

[1]Thus the name Rotating TCAM

B. High-Level Runtime Operation

This section provides a high-level description of the RTCAM algorithm. The algorithm operates in 3 or 4 steps: (i) A "key" of size w bytes is constructed from the packet at $position = pos$ (initially $pos = 0$). (ii) A TCAM lookup is invoked and the matched row is obtained. (iii) The corresponding row shift value is retrieved. A shift value greater than zero indicates that none of the patterns matched the given key and we can repeat step (i) with $position = position + shift$. A zero value implies that a prefix pattern has matched the key and step (iv) is invoked. This step queries the internal data structures (discussed in IV-C) in order to locate the potential attack patterns. The simple case occurs if the matched pattern is smaller than the key size (w). In this case, the relevant pattern is added to a dedicated "Matched Patterns List" and step (i) is invoked with $position = position + 1$. If the pattern is a partial match (i.e., it matched the prefix of a longer pattern), the rest of the pattern should be matched as well. One way to do this is to compare the rest of the packet and pattern using memory instructions. A faster way is to use the TCAM again to reduce the memory access latency. Step (iv) repeatedly uses the TCAM to match the rest of the pattern by splitting it into several w sub-patterns and iteratively looking it up in the TCAM. This operation is performed in the context of a specific rule (the rule to which the pattern belongs). The pattern can be marked as fully-matched, only if all of the TCAM matched rows correspond to a shift value of zero. Otherwise, step (i) is invoked again with the position increased by one. Pseudo code is presented in algorithm 1.

Algorithm 1 TCAM Pattern Matching

```
1: T(Packet) = {T_i, 1 ≤ i ≤ n}
2: pos ⇐ 1
3: shift ⇐ 0
4: while pos ≤ n − width do
5:    Step (i)
6:    key ⇐ T_[pos,..,pos+width−1]
7:    Step (ii)
8:    entry = TCAM.Lookup(key)
9:    Step (iii)
10:   shift ⇐ entry.shift
11:   if shift ≠ 0 then
12:      pos ⇐ pos + shift
13:      continue
14:   end if
15:   Step (iv)
16:   for all current = entry.PatternNode.next ≠ null do
17:      if current.len ≤ width OR
         checkSubPatterns(current.len, pos, current.TCAM_Ptrs) ==
         True then
18:         MatchedList.add(current)
19:      end if
20:   end for
21: end while
1: checkSubPatterns(len, pos, TCAM_Ptrs)
2: while pos ≤ len − width do
3:    key ⇐ T_[pos,..,pos+width−1]
4:    entry = TCAM.lookup(key)
5:    if entry.shift ≠ 0 or entry.id ∉ TCAM_Ptrs then
6:       return false
7:    end if
8:    return true
9: end while
```

C. Data Structures

We will now present the data structures that are used by the RTCAM algorithm.

1) Patterns List: The patterns list data structure is accessed in step (iv) of the algorithm (e.g. for comparing the suffix of a pattern with the packet content). A patterns list entry contains several fields which hold the information needed to implement the various Snort keywords: *len* - is the pattern's length; *root* - is a boolean that indicates whether this pattern is the first pattern of a rule; *offset* - indicates from where in the packet the pattern should be searched; *distance* - the minimum number of bytes allowed between two successive matches (i.e. the previous pattern match and the current match); *within* - the maximum number of bytes allowed between two successive pattern matches; *depth* - how far into the packet the algorithm should search for the specified pattern; *TCAM_Ptrs* - an array of TCAM references that are used in step (iv) of the algorithm whenever the pattern's length is greater than w.

2) TCAM Rules Table (TRT): This table correlates between a TCAM row and the patterns list. Each table entry contains the shift value, an inclusion patterns list and a list of associated patterns. When a TCAM match occurs, we associate the matched TCAM row content with the set of patterns containing the matched row as their prefix. The inclusion list is used to identify the patterns shorter than w that are prefixes of the matched pattern (TCAM row).

3) Matched Patterns List: This list holds the matched patterns for the current processed packet. Each entry contains the matched patterns and their corresponding end position in the packet. In case of a match, if the pattern is correlated, the algorithm checks if the previous pattern is already in the list. Additionally, the algorithm checks if the pattern's position is compliant with the pattern position constrains within the rule.

4) Rules List: The rules list maps between a single rule and its corresponding patterns. Each entry contains the number of patterns in the rule, and a bitmap with a bit for each pattern. This data structure is needed to support Snort's negation property as well as rules which contain uncorrelated patterns (more information is available in [2]).

D. A Packet Flow within the NIPS

In this section we will walk through the RTCAM algorithm, using the example shown in figure 1. Assume that the TCAM width is 4, the input packet is "WWABCDEFTXYZABC-DARP" and we will search for the rules appearing in table I. At stage 0, the Matched Patterns list is empty.

Initially, we search the packet at position 0. The first key is *wwab* (first 4 bytes of the packet), the TCAM lookup retrieves the *??ab* entry and the shift value is *2*. The key position within the packet is then increased by *2*. The next key is *abcd*, the lookup with this key matches the first TCAM entry and the corresponding shift value is *0*. Now step (iv) of the algorithm

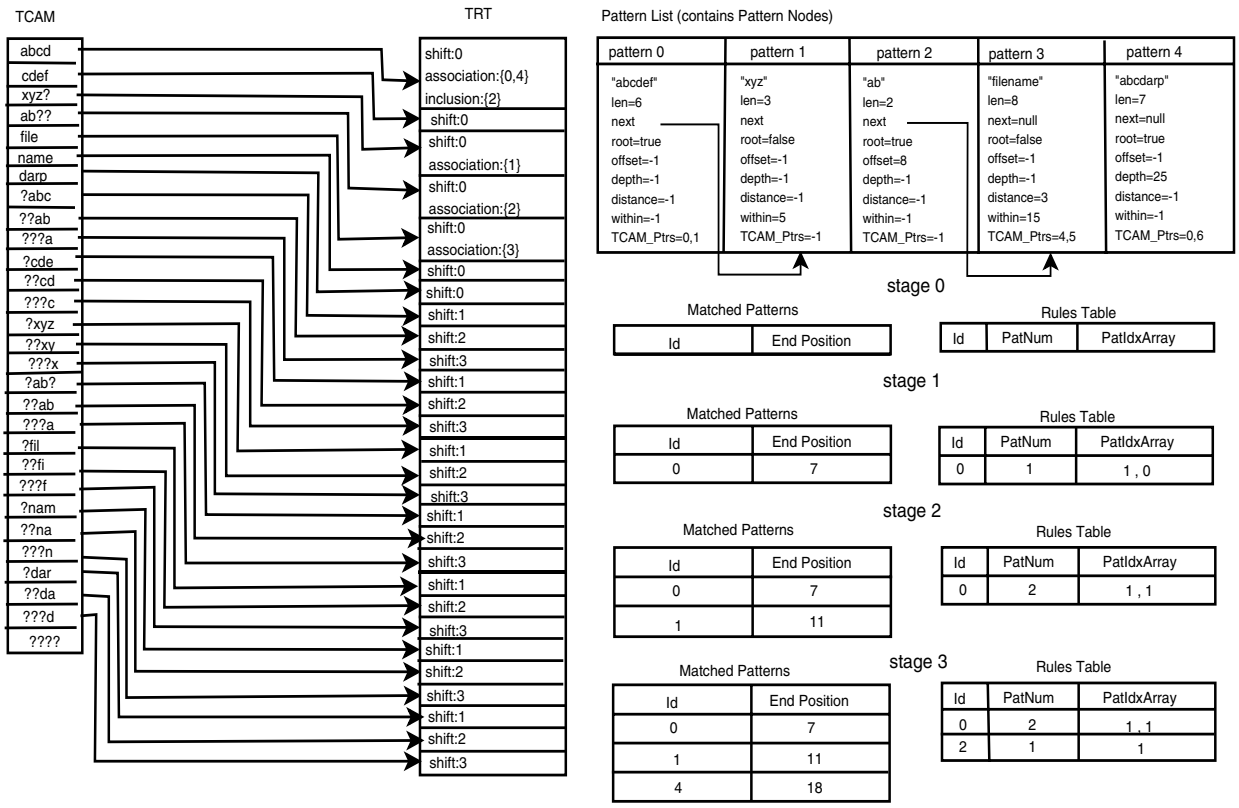


Fig. 1. An Example

is invoked and the association pointers are used in order to compare the full pattern with the packet's content. The first association pointer points to a node that contains the pattern *abcdef*. The pattern has no *offset* or *depth* so there are no position constraints. The pattern's length is *6*, so the algorithm takes the next *2* bytes and performs a lookup on the TCAM with the key *cdef* (prepending the 2 bytes with the last 2 bytes from the previous lookup). This key also yields *0* as a shift value and the index of the TCAM entry appears in the *TCAM_Ptrs* array. The algorithm finds a match, and adds the pattern to the *matched patterns list*. The algorithm also turns on the pattern's bit in the rule entry bitmap, see stage 1. The next pointer in the association list for the key *abcd* points to a node containing the *abcdarp* pattern which does not match the packet content (the lookup on the key *darp* fails).

Now, the algorithm follows the inclusion list and checks pattern number *2*. Since the constraint on the offset value is *8* and the current position is *2*, the algorithm does not match this pattern. The algorithm increases the search position within the packet by one, and constructs the search key *bcde*. This key matches the last TCAM entry so the shift value is *4* thus the next key is *ftxy*. Within two more lookup operations, a key of *xyza* yields a shift value of *0*.

Once again the algorithm follows the association pointer which in this case contains the pattern *xyz*. Since this is not a root pattern, the algorithm checks the *matched patterns list* for the previous pattern (by id). The previous pattern appears in the list and its end position is *7*. The *xyz* pattern has a *within* value of *5*, its length is *3* and the current position is *9*. The algorithm confirms that $9+3 \leq 7+5+1$; thus, the matched pattern is inserted to the *matched patterns list*. The algorithm also sets the pattern's bit in the rule entry bitmap (see stage 2). Since all the bits in the bitmap are set, the algorithm can announce an attack. Finally, the search position within the packet is shifted by one.

Two more lookups result in a match on the key *abcd* with a shift value of 0. Yet again, the algorithm follows the association pointers. The first node contains the pattern *abcdef*. Since the lookup for the key *cdar* yields a shift value greater than 0, we conclude that the pattern did not match and continue with the next association pointer (*abcdarp*). The depth value is *25*, the current position is *11* and the pattern's length is *7*. The algorithm checks that $11+7 \leq 25$. Since the pattern length is *7* (greater than *w*), the algorithm takes the next byte and hits the TCAM with the key *darp*. This key yields 0 as the shift value and the index of the TCAM entry is in the TCAM_Ptrs array, so the algorithm adds the pattern to the matched list and sets the bit corresponding to this pattern, in the rule entry (see stage 3). Since this is the only pattern in the rule, the

The contents part of Snort's rules
content:"abcdef"; content:"xyz"; within:5;
content:"ab"; offset:8; content:"filename"; distance:3; within:15;
content:"abcdarp"; depth:25;

TABLE I
SNORT'S RULES EXAMPLE

algorithm triggers an attack alert.

The algorithm can continue to scan the packet for further attacks until the entire packet is analyzed.

V. Snort Compatibility

The design of RTCAM and its associated data structures is highly influenced by the requirement to be compatible with Snort. We have successfully imported Snort's database directly to our simulated NIPS and were able to deal with Snort's keywords. Due to space limitations, some of the details concerning Snort compatibility were omitted. The interested reader is encouraged to read [2].

VI. Worst Case Performance Considerations

The motivation behind using the TCAM in step (iv) of the algorithm is to benefit from the fast TCAM access speed for matching long patterns. However, since pattern lengths are not bounded, there can be no guarantee on the worst case performance. In order to provide a worst case boundary, we introduced a simple pre-processing phase to the algorithm. In this phase, we take every long pattern and convert it to a set of short patterns (each of length equal or less than w). We then correlate these sets of short patterns using the *distance* and *within* keywords.

For example, we can split the pattern: *abcdefgh* into the two following patterns: *abcd* (all the keywords of the original pattern remain in the first sub-pattern), and *efgh* with $distance = 0$ and $within = 4$.

The result of this optimization is that "checkSubPatterns" of step (iv) will never be invoked. The theoretical worst case scenario is a TCAM hit for each byte in the packet with a corresponding shift value of either 0 or 1 (i.e. search position increase of 1), thus yielding a 2 Gbps scan rate.

VII. Window Size Considerations

TCAM memory is currently one of the more expensive components in a NIPS device. Reducing the amount of required TCAM space is a major concern. The TCAM width is configurable and is the greatest contributor to space consumption. The RTCAM memory requirement is calculated as follows: for a TCAM width of w and k patterns, each of length m_i, the number of TCAM rows is: $r = \sum\lceil m_i/w \rceil$. So the total TCAM memory required is $w^2 * r$ bytes.

Increasing the TCAM width significantly reduces the amount of false positives and increases the average shift value (discussed below). Thus, choosing w is a tradeoff between cost and performances.

A. Effects on the Shift Value

This section analyzes the theoretical expected shift value, assuming that the packet bytes are evenly distributed. In order to calculate the shift average, we need to calculate the probability of matching each of the TCAM rows. Section VIII provides the shift value measured from a real packet trace.

If the number of the *don't care* signs in the pattern P_i is K_i, there are $(2^8)^{K_i}$ strings that match P_i. If the shift value of P_i is S_i, P_i contributes $(2^8)^{K_i} * S_i$ to the total shift sum. From the sum $(2^8)^{K_i}$, we subtract $(2^8)^{K_j}$ for each pattern P_j that we count more than once and whose shift value is lower than S_i (for example we subtract the sum of the pattern *??cd* from the sum of the pattern *?bcd*). Let $Desc(P_i)$ be a set containing all such P_j's for a pattern P_i.

Finally, if S_i is the shift value and $S_i^n = (2^8)^{K_i} - \sum_{P_j \in Desc(P_i)} (2^8)^{K_j}$, the total average shift value is $\frac{\sum_{S_i \in 0..w} S_i^n * S_i}{(2^8)^w}$.

We calculated the shift average this way and we got that the average shift value is $w/2$. If TCAM lookup time is 4ns, the matching speed is $\frac{8*n}{4(n/(w/2))} = w$ Gbps. This result will encourage us to increase the TCAM width. Note, however, that there is a clear tradeoff between window size, required memory, speed and cost.

TCAM width Impact on Shift Average with ClamAv Patterns Set		
TCAM width	*TCAM memory size (KB)*	*Shift average value*
8	2797	7.53
16	7049	15.73
32	24261	31.75

TABLE II
TCAM width Impact on Shift Average

VIII. Experimental Results

We have fully implemented a software version of an RTCAM-NIPS device written in Java and have tested our simulation with two complex pattern sets. The first of this sets is a virus signature set from ClamAV [17], which contains only simple patterns (relatively long). The second set is comprised of intrusion detection signatures taken from the Snort [3] tool. Most of these signatures are comprised of correlated patterns. The input for our NIPS was comprised of a real packet trace from the MIT DARPA project [18].

We simulated our NIPS device using several TCAM widths. Choosing the right width is a very crucial decision in introducing this solution to the industry, since there is a clear tradeoff between TCAM cost and overall system performance. We compared our results with the ones presented by Lakshman et al., since these are currently the only two algorithms that use a TCAM.

A. Results on ClamAV Pattern Set

ClamAV version 0.82 has 26987 simple patterns and the average pattern length is 124.1. The performance results of our NIPS, when operating on the ClamAv database, appear in table II.

Let d denote the ratio of speeds of TCAM vs SRAM access. Our NIPS performance, for a system with 4ns TCAM access time, is about $\frac{Naive\_Scan\_Rate \cdot Shift\_Avg}{1+d}$ Gbps (Note that the 1 in the denominator denotes the TCAM access time and d is the SRAM access time). d is influenced by S_n, which is the number of SRAM accesses for each TCAM access and by the speed of each SRAM access.

We define a *memory ratio*, M_r, to be the ratio between the TCAM access speed and SRAM access speed. For example, a memory ratio of 0.2 means that memory access speed is 5 times faster than the TCAM access speed. Let S_n be the

TCAM width Impact on Shift Average with SNORT Pattern Set		
TCAM width	*TCAM memory size (KB)*	*Total shift average*
4	26	2.62
8	99	4.42
16	443	6.06
24	912	7.41
32	1990	7.67
48	4760	8.31
64	8735	9.11
96	20273	9.17
128	36591	9.57

TABLE III
TCAM WIDTH IMPACT ON SHIFT AVERAGE

number of SRAM accesses for each TCAM access and let d be $S_n \times M_r$.

When there are no attacks, $S_n = 1$, resulting in $d = M_r$. Since the values of the memory ratio are in the range of 0.2 to 1, the throughput average, when there are no attacks, is in the range of $\frac{63.5}{1+0.2} = 52.9$ Gbps to $\frac{63.5}{1+1} = 31.75$ Gbps.

Since ClamAV signatures are relatively long, most of the time the algorithm yields 0 as a shift value when a real attack occurs. Thus, at ClamAV, $S_n = 1$ also for the average case, so this range is true for this case as well.

B. Results on SNORT Pattern Set

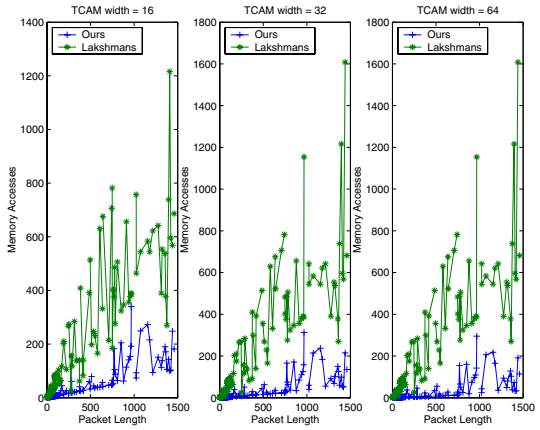
Snort's case is more complicated, since Snort patterns include many short patterns (of size ≤ 4). In most of the cases, these patterns are part of correlating rules. Our algorithm matches these patterns, resulting in a very low average shift (around 2, independent of the TCAM width). Thus, we eliminate the TCAM matches by adding more information to each TCAM row besides the pattern itself. We created a hash function h whose input is the additional data D and whose output is a key k, $h(D) = k$. We added k to each entry in the TCAM. The assumption of this solution is that most of the short patterns are related to specific flows and do not have to be checked for any received packet.

We added a bitmap of one byte for the main protocols and another value that represents the range of the lowest port and the highest port associated with this pattern. For example, if a pattern appears with the ports: 2048, 3400 and 7777, the port key will be: 11???0??0?00?.

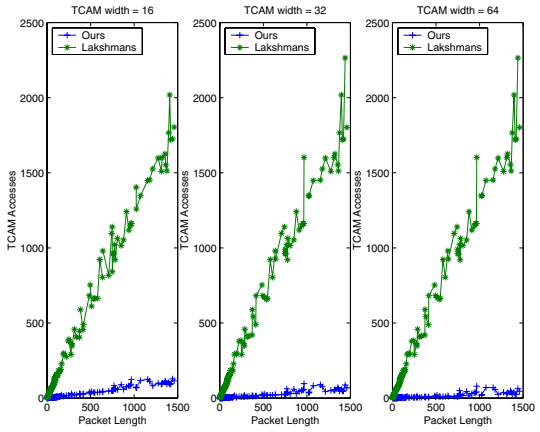
With this extension we saw significantly better results. Table III presents the TCAM memory size requirement and the shift average value for each TCAM width. Figure 2 shows the tradeoff between TCAM memory and the average shift value. We can see that even with the additions of the flow data to the TCAM key, the shift average grows slowly when the TCAM width is greater than 24. After this value, the tradeoff is not significant. In contrast to the moderated growth of the shift average, the TCAM memory size grows exponentially.

1) Scanning Time Results: The algorithm presented in Lakshman et al. presents a scan rate of 2 Gbps in the best case. The algorithm achieves this rate when the scan ratio is 1, meaning that there are no memory accesses at all.

The most significant factor affecting the scanning time is the TCAM lookups, as well as the memory accesses. As opposed to the algorithm presented by Lakshman et al. where a TCAM hit does not necessarily require a memory hit, our algorithm hits the memory at each TCAM hit in order to procure the shift value. Our main advantage over the Lakshman et al. solution is that our algorithm hits the TCAM significantly fewer times. Even though our algorithm hits the memory for each TCAM hit, the total number of memory hits is also significantly less than those of Lakshman et al. Figure 3(a) shows our improvement in the number of memory hits. Figure 3(b) presents the number of TCAM lookups our algorithm required for different packets length. In terms of this factor, our algorithm is significantly better than Lakshman et al. algorithm.



(a) Memory Accesses



(b) TCAM Accesses

Fig. 3. Memory Accesses

Running the simulation using a MIT DARPA trace showed that for a TCAM width of 24, 60% of the TCAM hits result in a shift value greater than 0. Since the RTCAM algorithm accesses the SRAM every TCAM lookup, the scan ratio (as defined in [12]) is 2. The simulation results for this TCAM width showed that the average shift value is 7.4. Since SRAM memory access speed is usually faster than the TCAM speed, we define a *memory ratio* to be the relation between these speeds. For example, a memory ratio of 1 means that SRAM access speed is equal to TCAM access speed. In current memory technologies, the ratio is 0.2, i.e. memory access speed is 5 times faster than TCAM access speed. Taking this figure, we can achieve a throughput of $\frac{2 \times 7.4}{1+0.2} = 12.35$ Gbps.

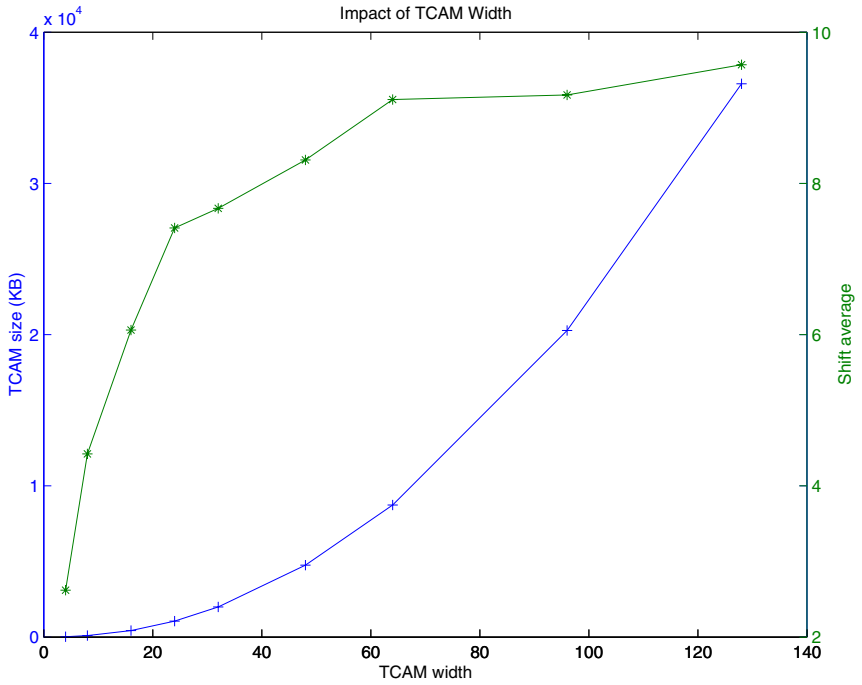


Fig. 2. TCAM vs. Shift Average Tradeoff

IX. Discussion and Future Work

We have designed and implemented a full NIPS system that is based on a novel pattern matching algorithm, called RTCAM. We have shown that our solution is adequate for NIPS device developers, as it achieves line-speed rates. Specifically, for about 60% of real network traffic, an average line-speed of 12.35 Gbps can be achieved. This paper presents several major advantages over existing NIPS devices. First, the achieved line-rate speed is several orders of magnitude faster than related works. Second, as opposed to other solutions, our system is fully compatible with Snort's rules syntax. This is an important advantage as Snort is becoming the de facto standard for intrusion detection and prevention systems. We have created a simple tool that is capable of importing Snort's database at the click of a mouse.

We have already initiated a lab project for prototyping a hash-based algorithm using FPGA. We would like to be able to compare the performance of the FPGA and the RTCAM solutions. Another important research area is cross packets inspection. Intuitively, it seems that the level of support for this problem is proportional to the amount of memory available on the intrusion detection device. Still, we would like to explore the various possibilities for dealing with this problem and to provide some experimental results. Finally, we plan to design an integrated RTCAM circuit that will automatically compare the provided key with the rotations of each pattern in the TCAM (using dedicated circuitry). This will significantly reduce the amount of TCAM memory needed by the algorithm.

References

[1] VanDyke Software. Homepage at http://www.vandyke.com/, keywords = vandyke, source = http://www.vandyke.com/.

[2] Hardware NIPS Project Home Page. Homepage at http://www.cs.huji.ac.il/labs/danss/nips.

[3] Snort. http://www.snort.org/.

[4] Charles E. Leiserson Thomas H. Cormen and Ronald L. Rivest. Introduction to Algorithms. The MIT Press, 1990.

[5] B. W. Watson, G. Zwaan, and Mrs F. Van Neerven. A taxonomy of keyword pattern matching algorithms. Technical report, January 28 1992.

[6] Donald E. Knuth, J.H. Morris, and Vaughan R. Pratt. Fast pattern matching in strings. SIAM Journal of Computing, 6(2):323–350, 1977.

[7] David R. Musser and Gor V. Nishanov. A fast generic sequence matching algorithm, May 13 2004.

[8] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. Communications of the ACM, 20(10):62–72, 1977.

[9] Artur Czumaj, Leszek Gasieniec, Stefan Jarominek, Thierry Lecroq, Wojciech Plandowski, and Wojciech Rytter. Fast practical multi-pattern matching, October 11 1993.

[10] A. Czumaj, L. Gasieniec, M. Crochemore, S. Jarominek, T. Lecroq, and W. Plandowski. Fast practical multi-pattern matching, September 02 1999.

[11] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: an aid to bibliographic search. Communications of the ACM, 18(6):333–340, 1975.

[12] Yu Fang, Randy H. Katz, and T. V. Lakshman. Gigabit rate packet pattern-matching using tcam. In ICNP, pages 174–183, 2004.

[13] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood. Deep packet inspection using parallel bloom filters, 2003.

[14] David E. Taylor, Praveen Krishnamurthy, and Sarang Dharmapurikar. Longest prefix matching using bloom filters, September 03 2000.

[15] Rong-Tai Liu, Nen-Fu Huang, Chih-Hao Chen, and Chia-Nan Kao. A fast string-matching algorithm for network processor-based intrusion detection system. Trans. on Embedded Computing Sys., 3(3):614–633, 2004.

[16] Trevis Chandler Igor Arsovski and Ali 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, 38(1), January.

[17] Clamav. http://www.clamav.net/.

[18] Mit darpa project data set. http://www.ll.mit.edu/IST/ideval/index.html.