One Algorithm to Match Them All: On a Generic NIPS Pattern Matching Algorithm

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
The Hebrew University Of Jerusalem
Email: tala@marvell.com

***Abstract*— Today's Network Intrusion Prevention Systems (NIPS) provide an important defense mechanism against security threats. The detection of network attacks utilizes a high-speed pattern matching algorithm that can be implemented in either hardware or software. Adapting a software-based pattern matching algorithm to hardware-based device is a complicated task. This paper presents a cost effective multi-pattern matching algorithm based on Field Programmable Gate Arrays (FPGAs) and standard RAM. The algorithm achieves line-rate speed, which is several orders of magnitude faster than the current state of the art, while attaining similar accuracy of detection. The algorithm can be easily adapted to operate in hardware-based NIPS and attain even higher speed by utilizing a TCAM memory.**

I. Introduction

Network Intrusion Prevention Systems (NIPS) detect malicious packets by constantly monitoring the network traffic. Once such a packet is detected, a NIPS device usually triggers an alert and blocks the offending packet. 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 signatures.

Snort [1] is an open source NIPS that is commonly used in the 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 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. The Snort rule syntax is the de-facto industrial standard. 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.

The current trend for integrating security with network switches and routers implies that the NIPS device must meet stringent network performance and reliability requirements. As network traffic speeds increase, PC-based solutions cannot continue to process all traffic in real time. For example, Snort's string matching algorithm by itself consumes more than 80% of the CPU time [2]. The need for a hardware-based pattern matching algorithm is apparent.

This work is part of a research project aimed at designing and implementing a hardware based NIPS device [3]. This paper presents a novel pattern matching algorithm using FPGA, which enables the NIPS appliance to operate at an aggregate rate of several gigabits per second.

II. Notations and Definitions

The following section provides the necessary notations and definitions which are used in the proposed pattern matching algorithm (Section VI).

Definition 1 *A pattern P is defined as a string of characters from an alphabet Σ that need to be identified within the input text. A sub-pattern, P_s is defined as a sub-string of pattern P.*

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

Definition 3 *A Shift is defined as the number of bytes the algorithm can safely skip without losing an occurrence of any pattern in the text. Formally, a pattern P of length m and a text T of length n has a shift value s, iff $\forall s' < s$, $T_{[s'+1..s'+m]} \neq P_{[1..m]}$, s.t. $0 \leq s \leq n-m$. A shift value of $n-m+1$ indicates that the text does not contain the pattern.*

A *string-matching algorithm* is defined as the problem of finding the first appearance of a pattern in the text. One can look at the the string-matching algorithm as finding the maximal shift value s s.t. $s \leq n-m$.

The extended problem of finding multiple patterns $P = \{P_1, \cdots, P_\ell\}$ in a given text T is called "multiple pattern matching" and its goal is to identify the minimal position at which a *pattern* $P_j \in P$ occurs.

III. Related Work

The string matching algorithm is an essential building block for NIPS. Most algorithms either provide poor performance or are too complex to be implemented in hardware.[1] This section presents the state-of-the-art *hardware* based algorithms.

[1]Several common software based algorithms can be found in [4], [5], [6], [7], [8], [9], [10] and [11].

1-4244-1206-4/07/$25.00 ©2007 IEEE

A. Parallel Bloom Filters

The Parallel Bloom Filters [12], [13] algorithm uses a bloom filter for each possible pattern length. Briefly, a bloom filter uses several hash methods that reduce the potential pattern 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. [14] 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 average length of signatures used today is 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. FPGA Solutions

In [15], a hardwired design is presented that provides good area efficiency and good time performance by using replicated hardwired 32-bit comparators in a pipeline structure. The matching technique proposed there uses four 32-bit hardwired comparators, each with the same pattern offset successively by 8 bits, allowing the running time to be reduced by 4x for an equivalent increase in hardware. Furthermore, they use about 100 rules ("the most common attacks") and have implemented only these patterns in the FPGA. The main weakness is the p^2 increase in hardware for a p increase in throughput.

Similarly, Other FPGA solutions (like [16]) usually take one of the available software algorithms and deploy it in the parallel environment facilitated by the FPGA.

IV. RTCAM: A TCAM Based Algorithm

A Ternary Content Addressable Memory (TCAM) [17] 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 relatively to regular memory lookups.

We have previously presented a TCAM-based algorithm called RTCAM [18]. In the RTCAM algorithm, a TCAM of size M is configured to hold $\lceil M/w \rceil$ rows, where w is the TCAM width. The rule's signatures (patterns) were split to fit in the chosen w. Patterns longer than w occupy more than one row. Each TCAM row has a corresponding shift value that states the number of bytes the algorithm can safely shift in the packet when a match occurs. In addition, 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. The last row corresponds to the maximum shift value and contains w *don't care* bytes, thus providing the default match row.

The algorithm constructs a "key" of size w bytes based on the packet and produces a TCAM lookup. The corresponding shift value is retrieved. A shift value greater than zero indicates the algorithm can construct a new key at the shifted position. A zero value implies that a prefix pattern has matched the key, the algorithm queries its internal data structures in order to locate the potential attack patterns.

The RTCAM algorithm considers the TCAM as a big hash table, thus, it can be easily replaced by simple FPGA logic supporting a standard SRAM. In this paper we present a FPGA version of our algorithm. The RTCAM algorithm was ported to simple SRAM with minimal performance penalty (See Section VIII for comparison). Figure 1 presents both architectures, the RTCAM and the FPGA-based one.

The packet is first received at the static flow classifier component which only extracts information from the header, a process that yields a flow descriptor. The packet is then transferred to the stateful inspection engine that uses the pattern-matching algorithms, which in turn looks for attacks. The pattern matching algorithm can be either the RTCAM or the FPGA-based algorithm.

In either case, the input to the algorithm is the packet payload and the flow signature (protocol, source and destination IPs and source and destination ports) and its output is the matched patterns. Both algorithms contain the list of the sub-patterns (TCAM or hash table) with pointers to the corresponding pattern nodes. In case of a sub-pattern match, these nodes are examined to verify the sub-pattern full context (see Section VI). If the full pattern is matched, the pattern matching algorithm adds a record in the matched pattern list.

V. Motivation

A NIPS device is required to deploy a multi-pattern matching method that can meet the line-speed of packet transfer. The multi-pattern matching method should efficiently handle a large number of patterns with a wide range of pattern lengths. Due to the high price of TCAM memory and the increasing number of signatures, a TCAM oriented solution may not be acceptable for industrial purposes. For example, one can easily calculate the memory requirements of the RTCAM solution 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 reduces the amount of false positives and increases the average shift value. Thus, the algorithm has an inherent tradeoff between cost and performance. Figure 2 shows the TCAM size required in order to accommodate all the patterns in Snort and in ClamAV signature databases.[2]

[2]ClamAV is a free open source antivirus software, See [19].

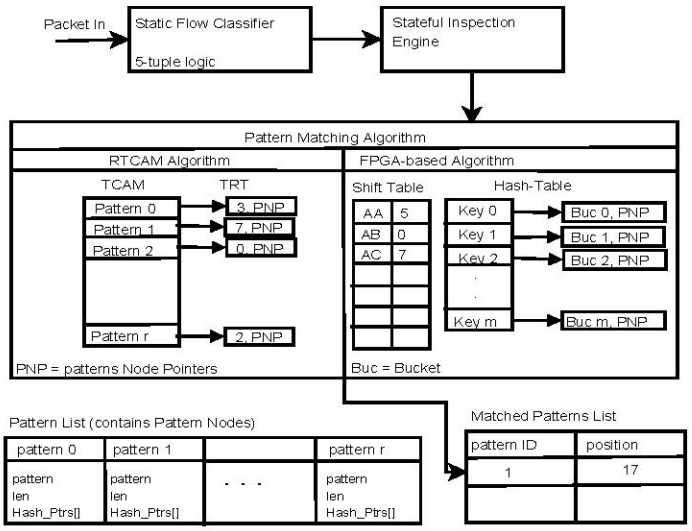

Fig. 1. RTCAM vs FPGA Architecture

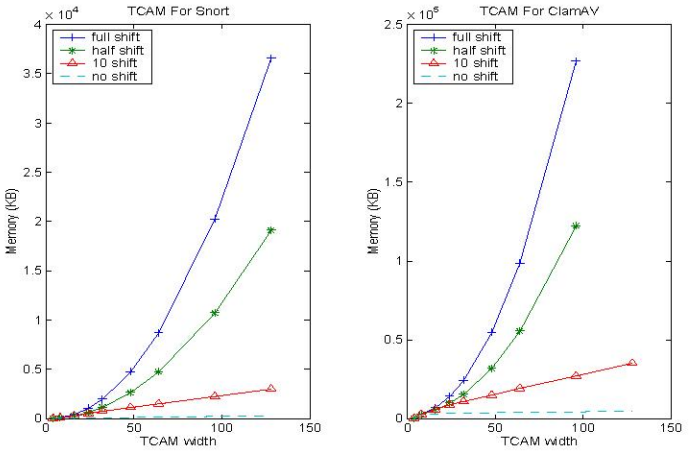

Fig. 2. TCAM Size Requirement

Several techniques can be used in order to decrease the amount of required TCAM (like shifting part of the pattern length), however the total cost of a TCAM-based solution is usually much higher than software or FPGA ones. Therefore, it is desirable to be able to construct an algorithm that can balance between its use of TCAM versus regular, and much cheaper, SRAM. The uniqueness of our prior algorithm is that it can be easily adapted to use regular SRAM. In this paper we will present how it is done and will provide a detailed evaluation of a fully implemented system.

We will start by giving a high-level description of the algorithm and describe the data structures needed for the algorithm. We will conclude with a simple example.

VI. FPGA Algorithm

A. High-Level Runtime Operation

Algorithm 1 presents the FPGA pattern matching algorithm in pseudo-code. The algorithm operates in several steps:

(i) A "text-key" of size w bytes is constructed from the packet at $position = pos$ (pos is initially 0).

(ii) A block of size B bytes is constructed from the *end* of the text-key. The block's corresponding entry in the shift table is obtained.

(iii) The corresponding shift value is retrieved. A shift value greater than zero indicates that none of the patterns end with the given block and thus none of the patterns match the given text-key; we can therefore repeat step (i) with $position = position + shift$. A zero value implies that the block is a suffix of one of the patterns and step (iv) is invoked, to check if there is a full pattern prefix match.

(iv) The algorithm calculates a hash-key for the text-key and looks for an entry in the hash table. If there is no such entry, none of the patterns match the text-key and the algorithm repeats step (i) with $position = position + 1$. If an entry with the same hash-key exists in the hash table, the pattern's bucket is located and step (v) is invoked.

If the pattern's bucket contains the text-key, step (v) queries the internal data structures (discussed in VI-B) in order to locate the potential attack patterns. The easy case occurs if the matched pattern's length is exactly that of the key size w (perhaps with the padding discussed later). 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. Step (v) repeatedly uses the shift and hash tables to match the rest of the pattern by taking the next w bytes from the text and applying steps (ii) through (iv) at the text-keys. The pattern can be marked as fully-matched, only if all of its sub-patterns are matched. Otherwise, step (i) is invoked again with the position increased by one.

B. Data Structures

For brevity we assume that all patterns have length of m which is greater or equal to the sliding window of w bytes. Later in this paper, we will show how the algorithm is used for matching patterns of smaller lengths.

Hash Table – This table resides in SRAM and contains the r patterns divided by w, the search window's width. Entries with less than w bytes are prepended (padding at the prefix) with the suffix of the previous part. A key is calculated on the sub-pattern and the sub-pattern is placed in the hash table "bucket" according to the sub-pattern key. Each value in a bucket contains the sub-pattern's text and a list of associated patterns that have this sub-pattern as their prefix. Sub-patterns with the same hash-key are linked to a "bucket" under the same hash-key. Patterns longer than w occupy more than one value in the hash table. The first subpattern of each pattern is marked as the "pattern's prefix".

Shift Table – The shift table enables us to skip the text by more than one character at each iteration. We create shift values for each possible block of B characters where $B \leq w$ (as in the algorithm presented by Manber [20], [21]). By using blocks (and not a single character), we reduce the amount of

Algorithm 1 Hash Table Pattern Matching

```
 1: T(Packet) = {T_i, 1 ≤ i ≤ n}
 2: pos ⇐ 1; shift ⇐ 0
 3: while pos ≤ n − width do
 4:    Step (i)
 5:    key ⇐ T_[pos,..,pos+width−1]
 6:    Step (ii)
 7:    block ⇐ Key_[width−B+1,..,width]
 8:    Step (iii)
 9:    if shift_table(block) ≠ 0 then
10:       pos ⇐ pos + shift
11:       continue
12:    end if
13:    Step (iv)
14:    if hash_table.containsKey(key) then
15:       Step (v)
16:       hash_val ⇐ hash.GetVal(key)
17:       if (current = hash_val.bucket.GetNode(text − key)) != null
          then
18:          if current.len ≤ width or
             checkSubPatterns(current.len, pos, current.Hash_Ptrs)
             then
19:             MatchedList.add(current)
20:          end if
21:       end if
22:    end if
23:    pos ⇐ pos + 1
24: end while
 1: checkSubPatterns(len, pos, Hash_Ptrs)
 2: while pos ≤ len − width do
 3:    text − key ⇐ T_[pos,..,pos+width−1]
 4:    block ⇐ text − key_[width−B+1,..,width]
 5:    if shift_table(block) > 0 then
 6:       return false
 7:    end if
 8:    hash_val ⇐ hash.GetVal(text − key)
 9:    if (current = hash_val.bucket.GetNode(text − key)) == null
       then
10:       return false
11:    end if
12:    if current.patternId ∉ Hash_Ptrs then
13:       return false
14:    end if
15: end while
16: return true
```

false matches. For simplicity[3] assume that the table size is Σ^B, where Σ is our alphabet. Each entry corresponds to a distinct substring of length B. This is a B-dimensional table, thus, when we have a block of B bytes, we can obtain its corresponding entry in the table in one memory hit.

Let $X = \{x_1, x_2, .., x_B\}$ be a string corresponding to the i'th entry of the shift table. There are two cases: either X appears somewhere in one of the patterns or it does not. If X does not appear in any of the patterns (and thus none of the patterns *end* with X), we store $w - B + 1$ in the corresponding shift entry. Since we observe the rightmost B bytes of the search window, we can shift the text $w - B + 1$ bytes (note that part of the block might be a prefix of one of the patterns). The second case occurs when the block appears in one of the patterns. In this case, we find the rightmost occurrence of X in any of these patterns: suppose it is in P_j and that X ends at position q of P_j, then we store $w - q - 1$ in the table. In this way we align the text to a position where the block X in the texts matches the same block X in P_j.

[3]The table can be easily compacted by hashing the B characters and setting the shift value to the minimum of the values corresponding to the same bucket.

Patterns List – The patterns list data structure is accessed when the algorithm finds a match of a pattern prefix (step (v)). A patterns list entry contains several fields which hold the information needed to implement the various Snort keywords: *len* - is the pattern's length; *offset* - indicates the place in the packet from which 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 pattern 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; *Hash_Ptrs* - an array of the hashtable references that are used in step (v) of the algorithm whenever the pattern's length is greater than w. Each cell in the Hash_Ptrs array is a pair (i, j) where i is the key index and j is the index of the pattern within the pointed bucket.

It is important to note that the hash table, shift table and the patterns list are populated once at initialization time.

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, the algorithm checks if the pattern's position is compliant with the pattern position constraints within the rule.

C. A Packet Flow within the NIPS

In this section we will walk through the algorithm, using the example shown in Figure 3. Assume that the sliding window width is 4, the block size is 2, the input packet is "WWABCDEFTXYZA" and we search for the patterns: "ABCDEF", "XYZW", "ABCDARP".

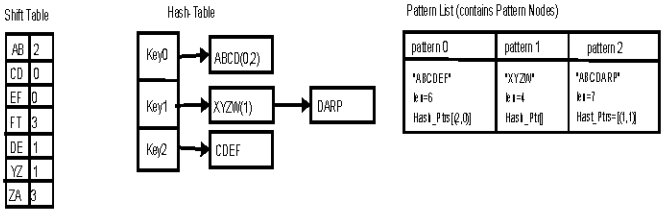

Fig. 3. A Packet Flow Walk-through

Initially, we search the packet at position 0. The first text key is *WWAB* and the block is *AB*, the shift table retrieves the *AB* entry and the shift value is *2*. The sliding window position within the packet is then increased by *2*. The next text key is *ABCD*, the block is *CD* so the shift value is *0*.

Now, step (iv) of the algorithm is invoked and a hash key is calculated for the sliding window value. A search of the hash table leads to the bucket that contains the prefix *ABCD*.

At step (v), the algorithm follows the association pointers. In our example there are two patterns that *ABCD* is their

prefix, *ABCDEF* and *ABCDARP*. The algorithm queries the pattern nodes in order to compare the full patterns with the packet's content. The first association pointer points to a node that contains the pattern *ABCDEF*. The pattern's length is *6*, so the algorithm takes the next *2* bytes and the text key is *CDEF* (prepending the 2 bytes with the last 2 bytes from the previous text key). The block is *EF* which also yields *0* as a shift value. Lookup in the hashtable returns key 2 and the pattern is the first in the bucket. Since the pair $(2,0)$ appears in the *Hash_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.

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 text key is *DEFT* and the shift value for the block *FT* is 3). The algorithm increases the search position within the packet by one, and constructs the search key *BCDE* and the block *DE*. The shift value for this block is 1, thus the next text key is *CDEF*, and the shift value for the block *EF* is 0. Step (iv) is then invoked but since *CDEF* is not a prefix, the algorithm returns and the position within the packet increases by one. The next text key is *DEFT*, the block is *FT* and the shift value is 3.

The next key is *TXYZ*, the block is *YZ* and the shift value is 1. The position increases by one, the next key is *XYZA*, the block is *ZA* and the shift value is 3. The entire packet has now been analyzed and the algorithm stops.

Note that the design of the algorithm 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 have been omitted. The interested reader is encouraged to read [18], [3].

VII. Dealing with Short Patterns

There is a major difficulty in designing a unified algorithm that deals both with long patterns and with short ones, since performance is typically influenced by the overhead caused by the short patterns. It is important to note that ClamAV signatures are quite long (an average of 124 bytes), where Snort's signatures are shorter (average is only 12 bytes).

In order to deal with short patterns while using a large w, we need to pad the short patterns to equal the width on which the hash is applied. Since there is no way to pad short patterns with *don't care* signs as we did in the RTCAM solution, a pad must be constructed out of actual characters. The pad can be usually extracted from the flow context.

The pad is constructed twice: once, when creating the hash function (the flow signature is extracted from the rule) and a second time, when constructing the search key (the flow signature is constructed from the received packet). Most of the short patterns reside in rules that specifically state their flow signature. A padding technique must be used in order to create a key for locating short patterns. A pseudo random pad is the most trivial solution. This method is suitable for rules with almost no flow signature. Using the flow signature to create a pad is a better practice since it reduces the amount of false positives while comparing the hash value with the text-keys. For example, if we are currently parsing an FTP packet, the hash function will use the protocol (sub-protocol) as pattern padding, thus reducing the amount of false matches.

Note that when we construct the shift table and we look for the rightmost appearance of a block in one of the patterns, we need to look at each pattern from its end to its beginning; i.e. if pattern p contains some block b that ends at position q, the de-facto rightmost end position will be $q+(w-m)$ where m is the sub-pattern length ($m \leq w$). For example, if $w=7$, $B=3$, we are looking for the block xyz and the only pattern that contains it is $axyzf$, the block end position within the pattern is $3+(7-5)=5$.

In order to locate short patterns we must calculate w text-keys for each sliding window, where w is the sliding window length. Each text-key is constructed by shifting the pattern to the right (losing the right most byte) and adding a padding byte from the left. For example, let's look at the case where $w=4$, the text-key is $xyzw$ and the packet's flow signature identifiers are P_1, P_2 and P_3. The text-keys are $xyzw$, P_1,x,y,z, P_2,P_1,x,y and P_3,P_2,P_1,x.

Calculating the hash value should be done quickly, and therefore it is preferable to do it incrementally. For example, one can create a full pad using a hash function and only use a simple XOR operation on the text. In order to recalculate the next hash one can XOR the text again, increase the text window width and XOR again with the pad.

VIII. Experimental Results

We have fully implemented the FPGA-based NIPS device written in VHDL using Altera Stratix GX EP1SGX25F development board. We have tested our device with a set of intrusion detection signatures taken from the Snort 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 [22].

We tested our NIPS device using several sliding window widths and a fixed block size $B=2$. Table I presents the sliding window width, the average shift value and the bitrate for each width.

Figure 4 shows the average shift value and the achieved search rate for different sliding window widths. In the TCAM-based algorithm, in order to increase the shift value one had to increase the sliding window width. The immediate implication is a significant increase in TCAM size and cost. The hash based algorithm uses an SRAM instead of TCAM, thus the cost is no longer a part of the "equation", i.e., the sliding window size can be increased for better performance. The figure clearly shows a linear correlation between the sliding window width and the gained average shift value, which leads to an increased bitrate. The maximal achieved bitrate using the FPGA solution is 10.3 Gbps for a sliding window of $16B$. Under the same environment settings, the achieved RTCAM bitrate was 19.8 Gbps.

Sliding Window Width Impact on Shift Average and Rate		
width (Bytes)	*Total shift average (Bytes)*	*Bit rate (Gbps)*
4	2.14	2.80
5	2.73	3.54
6	2.83	4.17
7	3.72	4.86
8	4.23	5.46
12	6.14	7.64
16	8.87	10.31

TABLE I

WINDOW'S WIDTH IMPACT ON SHIFT AVERAGE AND RATE

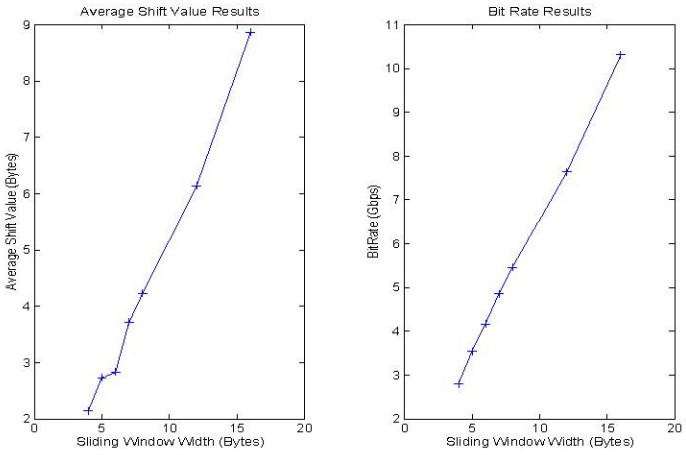

Fig. 4. Sliding Window Size Effect

Our FPGA implementation does not take advantage of the inherent parallelism capability of the FPGA. Thus, we compared our results against a single engine version of the other FPGA solutions (obviously we could run our FPGA algorithm in parallel on several chunks of the packets concurrently, in the same way other solutions did). For instance, in [15] the throughput is ~ 2 Gbps for a single engine implementation. Also, the work in [16] achieved over 10Gbps performance with four pipelined engines, thus we can assume worse performance when a single engine is used (probably about a fourth of their published result).

IX. DISCUSSION AND FUTURE WORK

We have designed and implemented a hardware based NIPS with a novel pattern matching algorithm at its core. We have shown that our previous TCAM based algorithm can be successfully ported to a hash-based algorithm using FPGA and still maintain line-speed rates. Our system is fully compatible with Snort's rules syntax, which is an important advantage over related work as Snort is becoming the de-facto standard. The proposed FPGA algorithm can be easily incorporated into a TCAM based device, thus reducing its total cost of ownership. In a future work, we plan to further improve the performance of our device by reducing the number of false positives identifications. This can be done by applying several parallel hash functions on a single text key and only proceed if all the hash functions indicate a match. In addition, we can significantly improve the algorithm performance by running it in parallel on several chunks of the packets concurrently.

ACKNOWLEDGMENTS

We would like to thank Tomer Gershoni and Yaniv Cohen for fully implementing the system and for providing valuable comments. We would also like to thank Holon institute of technology (HIT) for providing the hardware infrastructure for this research project. We would also like to thank Ariel Daliot for proofreading the paper.

REFERENCES

[1] "Snort," http://www.snort.org/.
[2] E. Markatos, S. Antonatos, M. Polychronakis, and K. Anagnostakis, "Exclusion-based signature matching for intrusion detection," CCN, 2002.
[3] "Hardware NIPS Project Home Page," Homepage at http://www.cs.huji.ac.il/labs/danss/nips.
[4] C. E. L. Thomas H. Cormen and R. L. Rivest, Introduction to Algorithms. The MIT Press, 1990.
[5] D. E. Knuth, J. Morris, and V. R. Pratt, "Fast pattern matching in strings," SIAM Journal of Computing, vol. 6, no. 2, pp. 323–350, 1977.
[6] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," Communications of the ACM, vol. 20, no. 10, pp. 62–72, 1977.
[7] D. R. Musser and G. V. Nishanov, "A fast generic sequence matching algorithm," May 13 2004. [Online]. Available: http://citeseer.ist.psu.edu/676947.html;http://www.cs.rpi.edu/research/ps/97-11.ps
[8] A. Czumaj, L. Gasieniec, S. Jarominek, T. Lecroq, W. Plandowski, and W. Rytter, "Fast practical multi-pattern matching," Oct. 11 1993. [Online]. Available: http://citeseer.ist.psu.edu/621721.html;http://www-igm.univ-mlv.fr/~lecroq/articles/igm9303.ps.gz
[9] A. Czumaj, L. Gasieniec, M. Crochemore, S. Jarominek, T. Lecroq, and W. Plandowski, "Fast practical multi-pattern matching," Sept. 02 1999. [Online]. Available: http://citeseer.ist.psu.edu/336719.html;http://www-igm.univ-mlv.fr/~lecroq/ipl3.ps.gz
[10] A. V. Aho and M. J. Corasick, "Efficient string matching: an aid to bibliographic search," Communications of the ACM, vol. 18, no. 6, pp. 333–340, 1975.
[11] Y. Fang, R. H. Katz, and T. V. Lakshman, "Gigabit rate packet pattern-matching using tcam." in ICNP, 2004, pp. 174–183.
[12] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood, "Deep packet inspection using parallel bloom filters," 2003. [Online]. Available: citeseer.csail.mit.edu/dharmapurikar03deep.html
[13] D. E. Taylor, P. Krishnamurthy, and S. Dharmapurikar, "Longest prefix matching using bloom filters," Sept. 03 2000. [Online]. Available: http://citeseer.ist.psu.edu/641375.html;http://www.arl.wustl.edu/Publications/2000-04/sigcomm03sd.pdf
[14] R.-T. Liu, N.-F. Huang, C.-H. Chen, and C.-N. Kao, "A fast string-matching algorithm for network processor-based intrusion detection system," Trans. on Embedded Computing Sys., vol. 3, no. 3, pp. 614–633, 2004.
[15] C. R. Clark and D. E. Schimmel, "Scalable pattern matching for high speed networks," FCCM, 2004.
[16] I. Sourdis and D. Pnevmatikatos, "Fast, large-scale string match for a 10gbps fpga-based network intrusion." [Online]. Available: citeseer.ist.psu.edu/sourdis03fast.html
[17] T. C. Igor Arsovski 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, January.
[18] Y. Weinsberg, S. Tzur-David, T. Anker and D. Dolev, "High performance string matching algorithm for a network intrusion prevention system (nips)," High Performance Switching and Routing (HPSR06), 2006.
[19] "Clamav," http://www.clamav.net/.
[20] S. Wu and U. Manber, "A fast algorithm for multi-pattern searching, Tech. Rep. TR-94-17, 1994. [Online]. Available: citeseer.ist.psu.edu/wu94fast.html
[21] S. Wu and U. Manber, "A grep – a fast approximate pattern-matching tool," in Proceedings USENIX Winter 1992 Technical Conference, San Francisco, CA, 1992, pp. 153–162. [Online]. Available: citeseer.ist.psu.edu/wu92agrep.html
[22] "Mit darpa project data set," http://www.ll.mit.edu/IST/ideval/index.html.