Highly Compressed Aho-Corasick Automata For Efficient Intrusion Detection

Xinyan Zha & Sartaj Sahni
Computer and Information Science and Engineering
University of Florida
Gainesville, FL 32611
{xzha, sahni}@cise.ufl.edu

***Abstract*—We develop a method to compress the unoptimized Aho-Corasick automaton that is used widely in intrusion detection systems. Our method uses bitmaps with multiple levels of summaries as well as aggressive path compaction. By using multiple levels of summaries, we are able to determine a popcount with as few as 1 addition. On Snort string databases, our compressed automata take 24% to 31% less memory than taken by the compressed automata of Tuck et al. [23]. and the number of additions required to compute popcounts is reduced by about 90%.**

Keywords: Intrusion detection, Aho-Corasick trees, compression, efficient popcount computation, performance.

I. Introduction

Network intrusion detection systems (NIDS) examine network traffic (both in- and out-bound packets) looking for traffic patterns that indicate attempts to break into a target computer, port scans, denial of service attacks, and other malicious behavior.Bro [16], [6], [9], [19], [5] and Snort [17] are two of the more popular public-domain NIDSs. Both maintain a database of signatures (or rules) that include a string as a component. These intrusion detection systems examine the payload of each packet that is matched by a rule and reports all occurrences of the string associated with that rule. It is estimated that about 70% of the time it takes Snort, for example, to process packets is spent in its string matching code and this code accounts for about 80% of the instructions executed [2]. Consequently, much research has been done recently to improve the efficiency of string matching ([4], [11], [23], for example). The focus of this paper is to improve the storage and search cost of NIDS string matching using Aho-Corasick trees [1].

In Section II, we review related work. The Aho-Corasick automaton, which is central to our work, is described in Section III. The compression method of Tuck et al. [23] is described in Section IV. In Section V we propose three designs to compute popcounts efficiently in 256-bit bitmaps. These designs make it possible to use popcounts efficiently without any hardware support whatsoever! Our method to compress the Aho-Corasick automaton is described in Section VI and experimental results comparing our method with that of Tuck et al. [23] are presented in Section VII.

II. Related Work

Snort [17] and Bro [16], [6], [9], [19], [5] are two of the more popular public domain NIDSs. Both are software solutions to intrusion detection. The current implementation of Snort uses the optimized version of the Aho-Corasick automaton [1]. Snort also uses SFK search and the Wu-Manber [24] multi-string search algorithm. To reduce the memory requirement of the Aho-Corasick automaton, Tuck et al. [23] have proposed starting with the unoptimized Aho-Corasick automaton and using bitmaps and path compression. In the network algorithms area, bitmaps have been used also in the tree bitmap scheme [7] and in shape shifting and hybrid shape shifting tries [20], [12]; path compression has been used in several IP lookup structures including tree bitmap [7] and hybrid shape shifting tries [12]. With these compression methods, the memory required by the compressed unoptimized Aho-Corasick automaton becomes about 1/50 to 1/30 of that required by the optimized automaton and the Wu-Manber structure and is slightly less than that required by SFK search [23]. However, a search requires us to perform a large number of additions at each node and so requires hardware support for efficient implementation.

Hardware and hardware assisted solutions have been proposed [22], [8], [26], [25], [4], [1], [21], [11], [23], [14], [13].

III. The Aho-Corasick Automaton

The Aho-Corasick finite state automaton [1] for multi-string matching is widely used in IDSs. In the unoptimized version, which we use in this paper, there is a failure pointer for each state and each state has success pointers;each success pointer has a label, which is a character from the string alphabet, associated with it. Also, each state has a list of strings/rules (from the string database) that are matched when that state is reached by following a success pointer. This is the list of matched rules. The search starts with the automaton start state designated as the current state and the first character in the text string, S, that is being searched designated as the current character. At each step, a state transition is made by examining the current character of S. If the current state has a success pointer labeled by the current character, a transition to the state pointed at by this success pointer is made and the next character of S becomes the current character. When there is no corresponding success pointer, a transition to the state pointed at by the failure pointer is made and the current character is not changed. Whenever a state is reached by following a success pointer, the rules in the list of matched rules for the reached state are output along with the position in S of the current character. This output is sufficient to identify all occurrences, in S, of all database strings. Aho and Corasick [1] have shown that when their unoptimized automaton is used, the number of state transitions is $2n$, where n is the length of S.

IV. The Method of Tuck et al. [23] To Compress Non-Optimized Automaton

Assume that the alphabet size is 256 (e.g., ASCII characters). A natural way to store the Aho-Corasick automaton, for a given database D of strings, is to represent each state of the unoptimized automaton by a node that has 256 success pointers, a failure pointer, and a list of rules that are matched when this state is reached via a success pointer. Assuming that a pointer takes 4 bytes and the rule list is simply pointed at by the node, each state node is 1032 bytes. Using bitmap and path compression, we may use nodes whose size is 52 bytes [24].

V. Popcounts With Fewer Additions

A serious deficiency of the compression method of [23] is the need to perform up to 31 additions at each bitmap node. This seriously degrades worst-case performance and increases the clamor for hardware support for a popcount in network processors [23]. Since popcounts are used in a variety of network algorithms ([3], [7], [12], [20], for example) in addition to those for intrusion detection, we consider, in this section, the problem of determining the popcount independent of the application. This problem has been studied extensively by the algorithms community ([10], [15], [**?**], for example).

Motivated by the work of Munro [15], [**?**], we propose 3 designs for summaries for a 256-bit bitmap. The first two of these use 3 levels of summaries and the third uses 2 levels.

1) *Type I Summaries*
   - *Level 1 Summaries* For the level 1 summaries, the 256-bit bitmap is partitioned into 4 blocks of 64 bits each. $S1(i)$ is the number of 1s in blocks 0 through $i-1$, $1 \le i \le 3$.
   - *Level 2 Summaries* For each block j of 64 bits, we keep a collection of level 2 summaries. For this purpose, the 64-bit block is partitioned into 16 4-bit subblocks. $S2(j,i)$ is the number of 1s in subblocks 0 through $i-1$ of block j, $0 \le j \le 3$, $1 \le i \le 15$.
   - *Level 3 Summaries* Each 4-bit subblock is partitioned into 2 2-bit subsubblocks. $S3(j,i,1)$ is the number of 1s in subsubblock 0 of the ith 4-bit subblock of the jth 64-bit block, $0 \le j \le 3$, $0 \le i \le 15$.

   Figure 1 shows the setup for Type I summaries. When Type I summaries are used, the popcount for position q

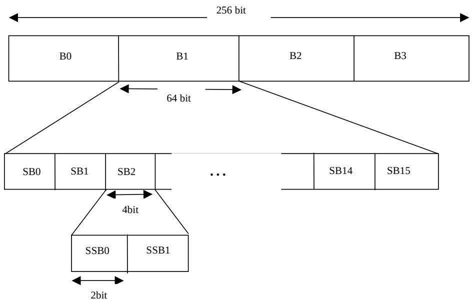

Fig. 1. Type I summaries

degree	number of nodes	percentage
0	1964	7.75
1	22453	88.6
2	591	2.33
3	149	0.58
4	43	0.17
5	35	0.14
6	14	0.055
7	23	0.090
8	14	0.055
9	8	0.031
10,11,12,13,14,15	6,3,4,5,3,2	< 0.03
17,18,21,51,78	1	< 0.03

Fig. 2. Distribution of states in a 3000 string Snort database

(i.e., the number of 1s preceding position q), $0 \le q < 256$, of the bitmap is obtained as follows:

   a) Position q is in subblock $sb = \lfloor (q \bmod 64)/4 \rfloor$ of block $b = \lfloor q/64 \rfloor$. The subsubblock ssb is 0 when $q \bmod 4 < 2$ and 1 otherwise.
   b) The popcount for position q is $S1(b) + S2(b,sb) + S3(b,sb,ssb) + bit(q-1)$, where $bit(q-1)$ is 0 if $q \bmod 2 = 0$ and is bit $q-1$ of the bitmap otherwise; $S1(0)$, $S2(b,0)$ and $S3(b,sb,0)$ are all 0. 1

   So, using Type I summaries, we can determine a popcount with at most 3 additions whereas using only 1 level of summaries as in [23], up to 31 additions are required. This reduction in the number of additions comes at the expense of memory. An $S1(*)$ value lies between 0 and 192 and so requires 8 bits; an $S2$ value requires 6 bits and an $S3$ value requires 2 bits. So, we need $8*3 = 24$ bits for the level-1 summaries, $6*15*4 = 360$ bits for the level-2 summaries, and $2*1*16*4 = 128$ bits for the level-3 summaries. Therefore, 512 bits (or 64 bytes) are needed for the summaries. In contrast, the summaries of the 1-level scheme of [23] require only 56 bits (or 7 bytes).

2) *Type II Summaries* These are exactly what is prescribed by Munro [15], [**?**]. $S1$ and $S2$ are as for Type I summaries. However, the $S3$ summaries are replaced by a summary table $T4(0:15, 0:3)$ such that $T4(i,j)$ is the number of 1s in positions 0 through $j-1$ of the binary representation of i. The popcount for position q

of a bitmap is $S1(b)$ + $S2(b, sb)$ + $T4(d, e)$, where d is the integer whose binary representation is the bits in subblock sb of block b of the bitmap and e is the position of q within this subblock; $S1$ and SB are for the current state/bitmap.

Since $T4(i, j) \leq 3$, we need 2 bits for each entry of $T4$ for a total of 128 bits for the entire table. Recognizing that rows $2j$ and $2j + 1$ are the same for every j, we may store only the even rows and reduce storage cost to 64 bits. A further reduction in storage cost for $T4$ is possible by noticing that all values in column 0 of this array are 0 and so we need not store this column explicitly. Actually, since only 1 copy of this table is needed, there seems to be little value (for our intrusion detection system application) to the suggested optimizations and we may store the entire table at a storage cost of 128 bits.

The memory required for the level 1 and 2 summaries is 24 + 360 = 384 bits (48 bytes), a reduction of 16 bytes compared to Type I summaries. When Type II summaries are used, a popcount is determined with 2 additions rather than 3 using Type I summaries and 31 using the 1-level summaries of [23].

3) *Type III Summaries* These are 2 level summaries and using these, the number of additions needed to compute a popcount is reduced to 1. Level-1 summaries are kept for the bitmap and a lookup table is used for the second level. For the level-1 summaries, we partition the bitmap into 16 blocks of 16 bits each. $S1(i)$ is the number of 1s in blocks 0 through $i - 1$, $1 \leq i \leq 15$. The lookup table $T16(i, j)$ gives the number of 1s in positions 0 through $j - 1$ of the binary representation of i, $0 \leq i < 65,536 = 2^{16}$, $0 \leq j < 16$. The popcount for position q of the bitmap is $S1(\lfloor q/16 \rfloor)$ + $T16(d, e)$, where d is the integer whose binary representation is the bits in block $\lfloor q/16 \rfloor$ of the bitmap and e is the position of q within this subblock; $S1$ and SB are for the current state/bitmap.

$8 * 15 = 120$ bits (or 15 bytes) of memory are required for the level-1 summaries of a bitmap compared to 7 bytes in [23]. The lookup table $T16$ requires $2^{16} * 16 * 4$ bits as each table entry lies between 0 and 15 and so requires 4 bits. The total memory for $T16$ is 512KB. For a table of this size, it is worth considering the optimizations mentioned earlier in connection with $T4$. Since rows $2j$ and $2j+1$ are the same for all j, we may reduce table size to 256KB by storing explicitly only the even rows of $T16$. Another 16KB may be saved by not storing column 0 explicitly. Yet another 16KB reduction is achieved by splitting the optimized table into 2. Now, column 0 of one of them is all 0 and is all 1 in the other. So, column 0 may be eliminated. We note that optimization below 256KB may not be of much value as the increased complexity of using the table will outweigh the small reduction is storage.

node type 3bits	firstchild type 3bits	L1(B0,..B2) 8bit*3=24bits	L2(SB0,..SB14) 6bit*4*15=360bits	L3(SSB0) 2bit*16*4*1=128bits

bitmap 256bits	failptr offset 8bits	failptr 32bits	ruleptr 32bits	firstchildptr 32bits

Fig. 3. Our bitmap node

VI. Our Method To Compress The Non-Optimized Aho-Corasick Automaton

A. *Classification of Automaton States*

The Snort database had 3,578 strings in April, 2006. Figure 2 profiles the states in the corresponding unoptimized Aho-Corasick automaton by degree (i.e., number of non-null success pointers in a state). This profile motivated us to classify the states into 3 categories–B (states whose degree is more than 8), L (states whose degree is between 2 and 8) and O (all other states). B states are those that will be represented using a bitmap, L states are low degree states, and O states are states whose degree is one or zero. In case the distribution of states in future string databases changes significantly, we can use a different classification of states.

Next, a finer (2 letter) state classification is done as below and in the stated order.

1) BB All B states are reclassified as BB states.
2) BL All L states that have a sibling BB state are reclassified as a BL states.
3) BO All O states that have a BB sibling are reclassified as BO states.
4) LL All remaining L states are reclassified as LL states.
5) LO All remaining O states that have an LL sibling are reclassified as LO states.
6) OO All remaining O states are reclassified as OO states.

B. *Node Types*

Our compressed representation uses three node types–bitmap, low degree, and path compressed. These are described below.

1) Bitmap: A bitmap node has a 256-bit bitmap together with summaries; any of the three summary types described in Section V may be used. We note that when Type II or Type III summaries are used, only one copy of the lookup table ($T4$ or $T16$) is needed for the entire automaton. All bitmap nodes may share this single copy of the lookup table. When Type II summaries are used, the 128 bits needed by the unoptimized $T4$ are insignificant compared to the storage required by the remainder of the automaton. For Type III summaries, however, using a 512KB unoptimized $T16$ is quite wasteful of memory and it is desirable to go down to at least the 256KB version.

When Type I summaries are used, each bitmap node (Figure 3) is 110 bytes (we need 57 extra bytes compared to the 52-byte nodes of [23] for the larger summaries and an additional extra byte because we use larger failure pointer offsets). When Type II summaries are used, each bitmap node is 94 bytes and the node size is 61 bytes when Type III summaries are used.

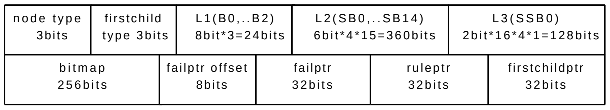

Fig. 4. Our path compressed node

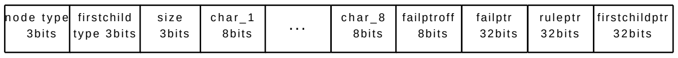

Fig. 5. Our low degree node

2) Low Degree Node: Low degree nodes are used for states that have between 2 and 8 success transitions. Figure 5 shows the format of such a node. In addition to fields for the node type, failure pointer, failure pointer offset, rule list pointer, and first child pointer, a low degree node has the fields $char1$, ..., $char8$ for the up to 8 characters for which the state has a non-null success transition and $size$, which gives us the number of these characters stored in the node. Since this number is between 2 and 8, 3 bits are sufficient for the size field. Although it is sufficient to allocate 22 bytes to a low degree node, we allocate 25 bytes as this allows us to pack a path compressed node with up to 2 characters (i.e., an $O2$ node as described later) into a low degree node.

3) Path Compressed Node: Unlike [23], we do not limit path compression to end-node sequences. Instead, we path compress any sequence of states whose degree is either 1 or 0. Further, we use variable-size path compressed nodes so that both short and long sequences may be compressed into a single node with no waste. In the path compression scheme of [23] an end-node sequence with 31 states will use 7 nodes and in one of these the capacity utilization is only 20% (only one of the available 5 slots is used). Additionally, the overhead of the type, next node, and size fields is incurred for each of the path compressed nodes. By using variable-size path compressed nodes, all the space in such a node is utilized and the node overhead is paid just once. In our implementation, we limit the capacity of a path compressed node to 256 states. This requires that the failure pointer offsets in all nodes be at least 8 bits. A path compressed node whose capacity is c, $c \leq 256$, has c character fields, c failure pointers, c failure pointer offsets, c rule list pointers, 1 type field, 1 size field, and 1 next node field (Figure 4). We refer to the path compressed node of Figure 4 as an O node. Five special types of O nodes–$O1$ through $O5$–also are used by us. An Ol node, $1 \leq l \leq 5$, is simply an O node whose capacity is exactly l characters. For these special O-node types, we may dispense with the capacity field as the capacity may be inferred from the node type.

The type fields (node type and first child type) are 3 bits. We use Type = 000 for a bitmap node, Type = 111 for a low degree node and Type = 110 for an O node. The remaining 5 values for Type are assigned to Ol nodes. Since the capacity of an O node must be at least 6, we actually store the node's true capacity minus 6 in its capacity field. As a result, an 8-bit capacity field suffices for capacities up to 261. However, since failure pointer offsets are 8 bits, using an O node with capacity between 257 and 261 isn't possible. So, the limit on O node capacity is 256. The total size of a path compressed node O is $10c + 6$ bytes, where c is the capacity of the O node. The size of an Ol node is $10l + 5$ as we do not need the capacity field in such a node.

C. Memory Accesses

The number of memory accesses needed to process a node depends on the memory bandwidth W, how the node's fields are mapped to memory, and whether or not we get a match at the node. We provide the access analysis primarily for the case $W = 32$ bits.

D. Bitmap Node With Type I Summaries, $W = 32$

We map our bitmap node into memory by packing the node type, first child type, failure pointer offset fields as well as 2 of the 3 L1 summaries into a 32-bit block; 2 bits of this block are unused. The remaining L1 summary ($S1(3)$) together with $S2(0, *)$ are placed into another 32-bit block. The remaining L2 summaries are packed into 32-bit blocks; 5 summaries per block; 2 bits per block are unused. The L3 summaries occupy 4 memory blocks; the bitmap takes 8 blocks; and each of the 3 pointers takes a block.

When a bitmap node is reached, the memory block with type fields is accessed to determine the node's actual type. The rule pointer is accessed so we can list all matching rules. A bitmap block is accessed to determine whether we have a match with the input string character. If the examined bit is 0, the failure pointer is accessed and we proceed to the node pointed by this pointer; the failure pointer offset, which was retrieved from memory when the block with type fields was accessed, is used to position us at the proper place in the node pointed at by the failure pointer in case this node is a path compressed node. So, the total number of memory accesses when we do not have a match is 4. When the examined bit of the bitmap is 1, we compute a popcount. This may require between 0 and 3 memory accesses (for example, 0 are needed when bit 0 of the bitmap is examined or when the only summary required is $S1(1)$ or $S1(2)$). Using the computed popcount, the first child pointer (another memory access) and the first child type (cannot be that of an O node), we move to the next node in our data structure. A total of 4 to 7 memory accesses are made.

E. Low Degree Node, $W = 32$

Next consider the case of a low degree node. We pack the type fields, size field, failure pointer offset field, and the char 1 field into a memory block; 7 bits are unused. The remaining 7 char fields are packed into 2 blocks leaving 8 bits unused. Each of the pointer fields occupies a memory block. When a low degree node is reached, we must access the memory block with type fields as well as the rule pointer. To determine whether we have a match at this node, we do an ordered sequential search of the up to 8 characters stored in the node. Let i denote the number of characters examined. For i = 1, no additional memory access is required, one additional access is required

	W=32		W=1024	
	match	mismatch	match	mismatch
$B(I)$	4 to 7	4	1	1
$B(II)$	4 to 6	4	1	1
$B(III)$	4 to 5	4	1	1
L	3 to 5	3 to 5	1	1
$O1$	3	3	1	1
$O2$	4	3 or 5	1	1
$O3$	6	3, 5, or 6	1	1
$O4$	7	3 or 5 to 7	1	1
$O5$	8	3, 5, 6, 8, or 9	1	1
O	3, $\lceil\frac{2+5i}{4}\rceil$+1	3, $\lceil\frac{2+5i}{4}\rceil$+2	1, $\lceil\frac{2+5i}{128}\rceil$+1	1, $\lceil\frac{2+5i}{128}\rceil$+1
TB([23])	4 to 5	4	1	1
TO([23])	$1+i$,6 or 8	3,$3+i$	1	1

Fig. 6. Memory accesses to process a node

Node Type	B	L	Ol	O	TB	TO
DataSet 1284	133	595	850	454	1057	2955
DataSet 2430	100	769	938	576	1527	3310

Fig. 7. Number of nodes of each type, Ol and O counts are for Type I summaries

when $2 \leq i \leq 5$, and 2 accesses are required when $6 \leq i \leq 8$. In case of no match we need to access also the failure pointer; the first child pointer is retrieved in case of a match. The total number of memory accesses to process a low degree node is 3 to 5 regardless of whether there is a match.

F. Summary

Using a similar analysis, we can derive the memory access counts for different values of the memory bandwidth W, other summary types, and other node types. Figure 6 gives the access counts for the different node and summary types for a few sample values of W. The rows labeled B (bitmap), L (low degree), Ol ($O1$ through $O5$), and O refer to node types for our structure while those labeled TB (bitmap) and TO (one degree) refer to node types in the structure of Tuck et al. [23].

G. Mapping States to Nodes

We map states to nodes as follows and in the stated order.

1) Category $BX, X \in \{B, L, O\}$, states are mapped to 1 bitmap node each; sibling states are mapped to nodes that are contiguous in memory. Note that in the case of BL and BO states, only a portion of a bitmap node is used.
2) Maximal sets of $LX, X \in \{L, O\}$, states that are siblings are packed into unused space in a bitmap node created in (1) using 25 bytes per LX state and the low degree structure of Figure 5. The packing of sibling LX nodes is done in non-increasing order of the number of siblings.
3) The remaining LX states are mapped into low degree nodes (LL states) or $O2$ nodes (LO states). LL states are mapped one state per low degree node. As before, when an LO state whose child is an OO state is mapped in this way, it is mapped together with its lone OO-state child into a single 25-byte $O2$ node. Sibling states are mapped to nodes that are contiguous in memory.
4) The chains of remaining OO states are handled in groups where a group is comprised of chains whose first nodes are siblings. In each group, we find the length, l, of the shortest chain. If $l > 5$, set $l = 5$. Each chain is mapped to an Ol node followed by an O node. The Ol nodes for the group are in contiguous memory. Note that an O node can only be the child of an Ol node or another O node.

Data set 2430				
Methods	[23]	Type I	Type II	Type III
Memory(bytes)	251524	177061	175511	172523
Normalized	1	0.70	0.70	0.69

Fig. 8. Memory requirement for data set 2430 (*Excludes memory for $T4$ and $T16$)

VII. Experimental Results

We benchmarked our compression method of Section VI against that proposed by Tuck et al. [23] using two data sets of strings extracted from Snort [18] rule sets. The first data set has 1284 strings and the second has 2430 strings. We name each data set by the number of strings in the data set.

A. Number of Nodes

Figure 7 gives the number of nodes of type I and Tuck et al. [23] in the compressed Aho-Corasick structure for each of our string sets. The maximum capacity of an allocated O node was 141 for data set 1284 and 256 for data set 2430.

B. Memory Requirement

Although the total number of nodes used by us is less than that used by Tuck et al. [23], our nodes are larger and so the potential remains that we actually use more memory than used by the structure of Tuck et al. [23]. Figure 8 gives the number of bytes of memory used by the structure of [23] as well as that used by our structure for each of the different summary types of Section V. The row labeled *Normalized* gives the memory required normalized by that required by the structure of Tuck et al. [23]. As can be seen, our structures take between 24% and 31% less memory than is required by the structure of [23]. With the 256KB required by $T16$ added in for Type III summaries, the Type III representation takes twice as much memory as does [23] for the 1284 data set and 75% more for the 2430 data set. As the size of the data set increases, we expect Type II summaries to be more competitive than [23] on total memory required.

C. Popcount

Figure 9 gives the total number of additions required to compute popcounts when using each of the data structures. For this experiment, we used 3 query strings obtained by concatenating a differing number of real emails that were classified as spam by our spam filter. The string lengths varied

Methods	[23]	Type I	Type II	Type III
strlen=1002832	11.54M	1.46M	1.33M	0.79M
strlen=2032131	34.97M	4.43M	4.02M	2.42M
strlen=3002665	69.54M	8.78M	7.96M	4.80M
Normalized	1	0.127	0.114	0.069

Fig. 9. Number of popcount additions, data set 2430

from 1MB to 3MB and we counted the number of additions needed to report all occurrences of all strings in the Snort data sets (1284 or 2430) in each of the query strings. The last row of each figure is the total number of adds for all 3 query strings normalized by the total for the structure of [23]. When Type III summaries are used, the number of popcount additions is only 7% that used by the structure of [23]. Type I and Type II summaries require about 13% and 12%, respectively, of the number of additions required by [23].

VIII. Conclusion

We have proposed the use of 2- and 3-level summaries for efficient popcount computation and have suggested ways to minimize the size of the lookup table associated with the popcount scheme of Munro [15]. Using the summaries proposed here, the number of additions required to compute popcount is between 7% and 13% of that required by the scheme of [23]. We also have proposed an aggressive compression scheme. When this scheme is used on our test sets, the memory required by the search structure is between 24% and 31% less than that required when the compression scheme of [23] is used.

References

[1] A. Aho and M. Corasick, Efficient string matching: An aid to bibliographic search, CACM, 18, 6, 1975, 333-340.

[2] S. Antonatos, K. Anagnostakis and E. Markatos, Generating realistic workloads for network intrusion detection systems, *ACM Workshop on Software and Performance*, 2004.

[3] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink, Small forwarding tables for fast routing lookups, *ACM SIGCOMM*, 1997, 3-14.

[4] S. Dharamapurikar and J. Lockwood, Fast and scalable pattern matching for content filtering, *ANCS*, 2005.

[5] H. Dreger, A. Feldmann, M. Mai, V. Paxson and R. Sommer, Dynamic application-layer protocol analysis for network intrusion detection, *USENIX Security Symposium*, 2006.

[6] H. Dreger, C. Kreibach, V. Paxson, and R. Sommer, Enhancing the accuracy of network-based intrusion detection with host-based context, *DIMVA*, 2005.

[7] W. Eatherton, G. Varghese, Z. Dittia, Tree bitmap: hardware/software IP lookups with incremental updates, *Computer Communication Review*, 34(2): 97-122, 2004.

[8] Y. Fang, R. Katz and T. Lakshman, Gigabit rate packet pattern-matching using TCAM, *ICNP*, 2004

[9] J. Gonzalez and V. Paxson, Enhancing network intrusion detection with integrated sampling and filtering, *RAID*, 2006.

[10] G. Jacobson, Succinct Static Data Structure, *Carnegie Mellon University Ph.D Thesis*, 1998.

[11] J. Lockwood, C. Neely, and C. Zuver, An extensible system-on-programmable-chip, content-aware Internet firewall.

[12] W. Lu and S. Sahni, Succinct representation of static packet classifiers, *IEEE Symposium on Computers and Communications*, 2007.

[13] J. Lunteran and A. Engbersen, Fast and scalable packet classification using, *IEEE JSAC*, 21, 4, 2003, 560-571.

[14] J. Lunteren, High-performance pattern-matching for intrusion detection, *INFOCOM*, 2006

[15] J. Munro, Tables, *Foundations of Software Technology and Theoretical Computer Science*, LNCS, 1180, 37–42, 1996.

[16] V. Paxson, Bro: A system for detecting network intruders in real-time, *Computer Networks*, 31, 1999, 2435–2463.

[17] Snort users manual 2.6.0, 2006.

[18] http://www.snort.org/dl.

[19] R. Sommer and V. Paxson, Exploiting independent state for network intrusion detection, *ACSAC*, 2005.

[20] H. Song, J. Turner, and J. Lockwood, Shape shifting tries for faster IP route lookup, *ICNP*, 2005.

[21] H. Song, et al. Snort offloader: A reconfigurable hardware NIDS filter, *FPL 2005*.

[22] H. Song and J. Lockwood, Efficient packet classification for network intrusion detection, *FPGA*, 2005.

[23] N. Tuck, T. Sherwood, B. Calder and G. Varghese, Deterministic memory-efficient string matching algorithms for intrusion detection, *INFOCOM*, 2004.

[24] S. Wu and U. Manber, Agrep–a fast algorithm for multi-pattern searching, Technical Report, Department of Computer Science, University of Arizona, 1994.

[25] M. Yazdani, W. Fraczak, F. Welfeld, and I. Lambadaris, Two level state machine architecture for content inspection engines, *INFOCOM 2006*.

[26] F. Yu and R. Katz, Efficient multi-match packet classification with TCAM.