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. [32]. 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.

1 Introduction

Intrusion detection systems (IDS) monitor events within a network or computer system with the objective of detecting "attempts to compromise the confidentiality, integrity, availability, or to bypass the security mechanisms of a computer or network" [4]. The intrusion detected by an IDS may manifest itself as a denial of service, unauthorized login, a user performing tasks that he/she is not authorized to do (e.g., access secure files, create new accounts, etc), execution of malware such as viruses and worms, and so on. An IDS accomplishes its objective by analyzing data gathered from the network, host computer, or application

*This research was supported, in part, by the National Science Foundation under grant ITR-0326155

that is being monitored. The analysis usually takes one of two forms–misuse (or signature) detection and anomaly detection. In misuse detection, the IDS maintains a database of *signatures* (patterns of events) that correspond to known attacks and searches the gathered data for these signatures. In anomaly detection the IDS maintains statistics that describe normal usage and checks for deviations from these statistics in the monitored data. While misuse detection usually has a low rate of false positives, it is able to detect only known attacks. Anomaly detection usually has a higher rate of false positives (because users keep changing their usage pattern thereby invalidating the stored statistics) but is able to detect new attacks never seen before.

Several types–network, host, application, protocol and hybrid–of IDSs are available commercially. 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. Host intrusion detection systems (HIDS) monitor the activity within a computing system looking for activity that violates the computing systems internal security policy (e.g., a program attempting to access an unauthorized resource). Application intrusion detection systems (AIDS) monitor the activity of a specific application while protocol intrusion detection systems (PIDS) ensure that specific protocols such as HTTP behave as they should. Each type of IDS has its capabilities and limitations and attempts have been made to put together hybrid IDSs that combine the capabilities of the described base IDSs.

Bro [22, 8, 11, 27, 7] and Snort [25] 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 ([6, 13, 32], 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 2, we review related work. The Aho-Corasick automaton, which is central to our work, is described in Section 3. The compression method of Tuck et al. [32] is described in Section 4. In Section 5 we describe the method of Munro [20, 21] to compute popcounts with 2 additions and 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 6 and experimental results comparing our method with that of Tuck et al. [32] are presented in Section 7.

2 Related Work

The development of high-speed intrusion detection systems and components has been the focus of significant recent research. Although there are many components in a NIDS that need to be optimized to achieve line-rate processing, we focus our discussion here to the string matching component, which is the most time consuming and which has been the focus of much of the prior work on NIDS optimization. String matching requires the examination of the network traffic to determine all matches with the strings in the string database. Although through pre-filtering [22, 29] we can reduce the effective workload on the NIDS considerably, there remains a need for powerful and compact data structures for string matching.

Snort [25] and Bro [22, 8, 11, 27, 7] 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 [34] multi-string search algorithm. The memory required to store the optimized Aho-Corasick and Wu-Manber data structures is excessive [32]. To reduce the memory requirement of the Aho-Corasick automaton, Tuck et al. [32] have proposed starting with the unoptimized Aho-Corasick automaton and using bitmaps and path compression. We note that the use of bitmaps to obtain compact representations was proposed first by Jacobson [21]. In the network algorithms area, bitmaps have been used also in the tree bitmap scheme [9] and in shape shifting and hybrid shape shifting tries [28, 17]; path compression has been used in several IP lookup structures including tree bitmap [9] and hybrid shape

shifting tries [17]. 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 [32]. However, a search requires us to perform a large number of additions at each node and so requires hardware support for efficient implementation. String matching using a purely software implementation of the bitmap and path-compressed Aho-Corasick automaton takes about 10% to 20% more time, on average, than when an optimized Aho-Corasick automaton is used.

Hardware and hardware assisted solutions also have been proposed. Song and Lockwood [30], Fang et al. [10], and Yu and Katz [36], for example, propose the use of TCAMs (in the case of [30], the TCAM is supplemented with bit-vector hardware) for NIDS applications. Yazadani et al. [35] propose a two-level state machine architecture that employs a TCAM for packet content examination. Dharmapurikar and Lockwood [6] have proposed a hardware implementation of the Aho-Corasick [1] string matching algorithm for NIDS applications. They assert that their hardware design is more scalable than FPGA and TCAM based designs because of its reliance on "embedded on-chip memory blocks in VLSI hardware." Song et al. [29] propose the use of an FPGA pre-filter to reduce the network traffic actually examined by a NIDS and Lockwood et al. [13] propose an extensible system-on-programmable-chip design for content-aware filtering. Their design employs TCAMs and FPGAs. Tuck et al. [32] propose a way to represent unoptimized Aho-Corasick automata in a compact format. They predict a processing rate of about 8Gbps for an ASIC design. Lunteran [19] has proposed a B-FSM (Bart Finite State Machine) for NIDS applications. The proposed B-FSM employs a finite state machine similar to that used in the Aho-Corasick string matching algorithm and the packet classification scheme Bart developed earlier by Lunteran [18]. It is estimated that an FPGA version of the B-FSM will process at 10Gbps and an ASIC version at 20Gbps.

3 The Aho-Corasick Automaton

The Aho-Corasick automaton [1] for multi-string matching is widely used in IDSs. There are two versions of this automaton–unoptimized and optimized. While both versions are finite state machines, the unoptimized version has a failure pointer for each state while in the optimized version, no state has a failure pointer. In both versions, each state has success pointers and 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. In the unoptimized version, 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.

In the optimized version, each state has a success pointer for every character in the alphabet and so, there is no failure pointer. Aho and Corasick [1] show how to compute the success pointer for pairs of states and characters for which there is no success pointer in the unoptimized automaton thereby transforming a unoptimized automaton into an optimized one. The number of state transitions made by an optimized automaton when searching for matches in a string of length n is n.

Figure 1 shows an example string set drawn from the 3-letter alphabet {a,b,c}. Figures 2 and 3, respectively, show its unoptimized and optimized Aho-Corasick automata. For this

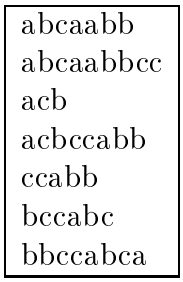

Figure 1: An example string set

example, we assume that the string alphabet is {A, B, C}.

It is important to note that when we remove the failure pointers from an uncompressed Aho-Corasick automaton, the resulting structure is a trie [23] rooted at the automaton start node. However, an optimized automaton has the structure of a graph that may not be a trie. This difference in the structure defined by the success pointers has a profound impact on our ability to compress unoptimized automata versus optimized automata.

4 The Method of Tuck et al. [32] To Compress Non-Optimized Automaton

Assume that the alphabet size is 256 (e.g., ASCII characters). Although the development is generalized readily to any alphabet size, it is more convenient to do the development using a fixed and realistic alphabet size. A natural way to store the Aho-Corasick automaton, for a given database D of strings, in a computer is to represent each state of the unoptimized automaton by a node that has the following fields:

1. $Success[0:255]$, where $Success[i]$ gives the state to transition to when the ASCII code for the current character is i ($Success[i]$ is $null$ in case there is no success pointer for the current state when the current character is i).

2. $RuleList$ ... a list of rules that are matched when this state is reached via a success pointer.

3. $Failure$ ... the transition to make when there is no success transition, for the current character, from the current state.

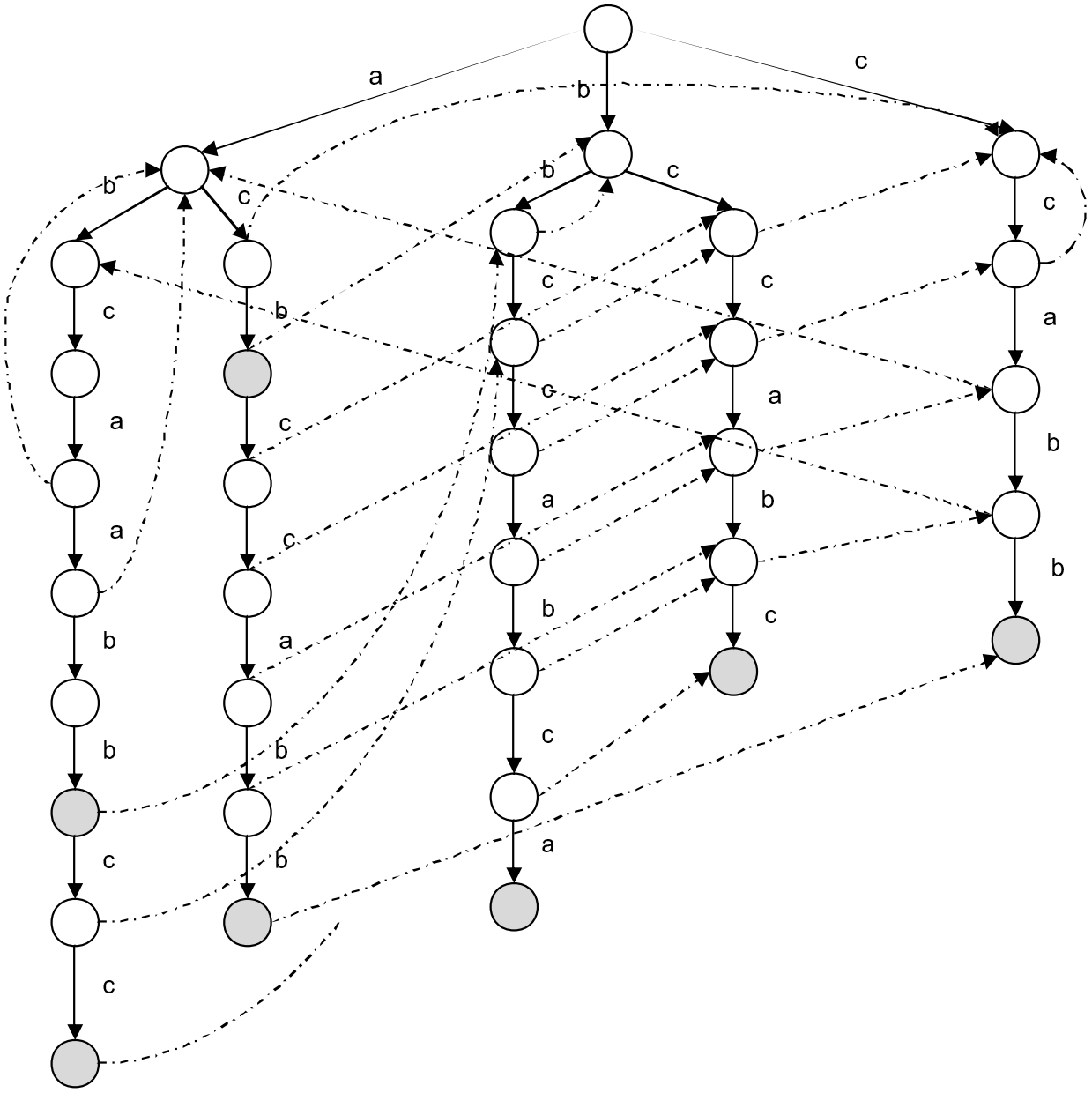

Figure 2: Unoptimized Aho-Corasick automata for strings of Figure 1

Assume that each pointer requires 4 bytes. So, each node requires 1024 bytes for the *Success* array and 4 bytes for the failure pointer. In keeping with Tuck et al. [32], when accounting for the memory required for *RuleList*, we shall assume that only a 4-byte pointer to this list is stored in the node and ignore the memory required by the list itself. Hence, the size of a state node for an unoptimized automaton is 1032 bytes. In the optimized version, the *Failure* field is omitted and the memory required by a node is 1028 bytes. While each

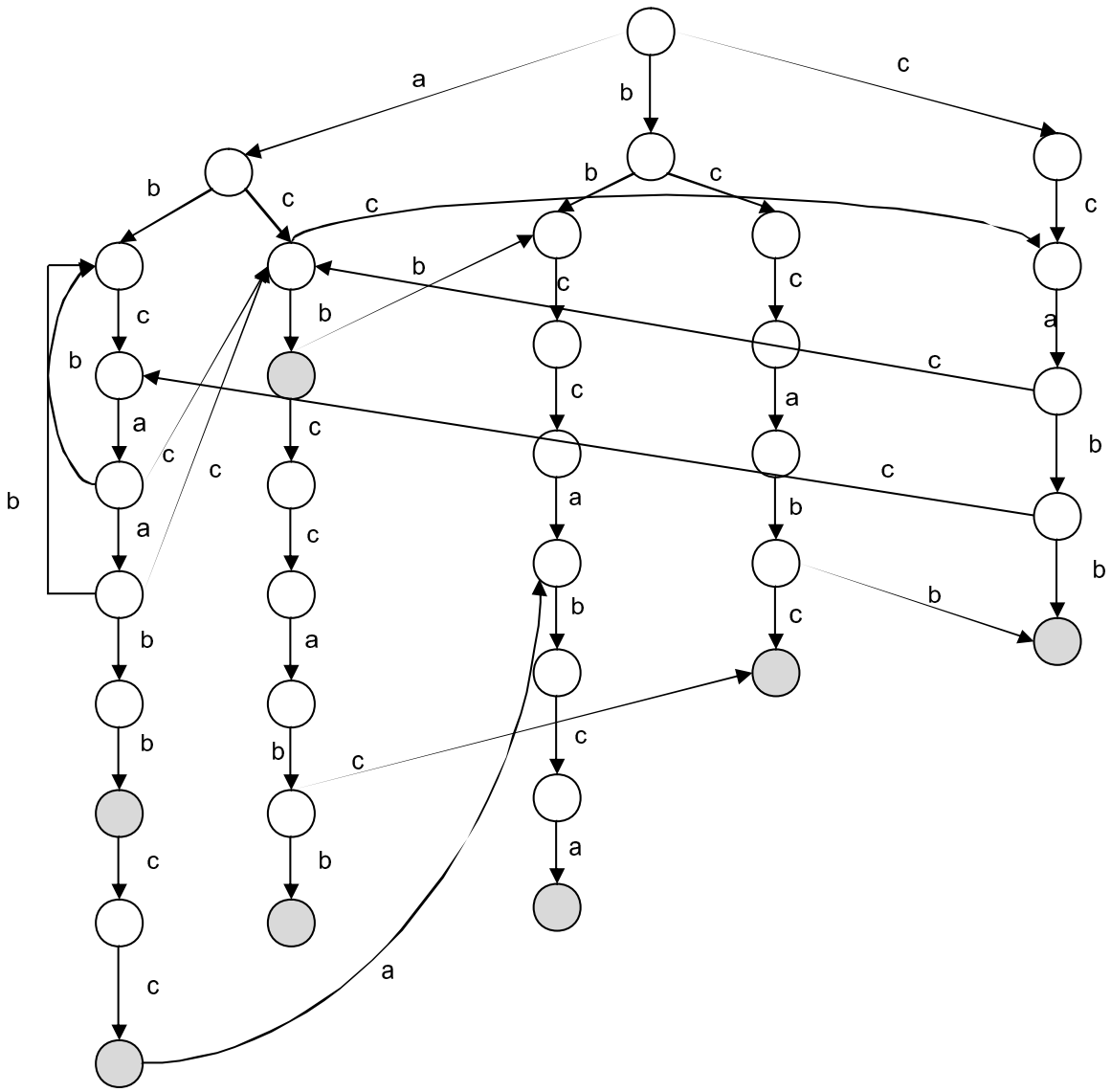

Figure 3: Optimized Aho-Corasick automata for strings of Figure 1

node of the optimized automaton requires 4 bytes less than required by each node of the unoptimized automaton, there is little opportunity to compress an optimized node as each of its 256 success pointers is non-null and the automaton does not have a tree structure. However, many of the success pointers in the nodes of a unoptimized automaton are null and the structure defined by the success pointers is a trie. Therefore, there is significant opportunity to compress these nodes. Following up on this observation, Tuck et al. [32] propose two transformations to compress the nodes in an unoptimized automaton:

1. *Bitmap Compression.* In its simplest form, bitmap compression replaces each 1032-byte node of an unoptimized automaton with a 44-byte node. Of these 44 bytes, 8 are used for the failure and rule list pointers. Another 32 bytes are used to maintain a 256-bit bitmap with the property that bit i of this map is 1 iff $Success[i] \neq null$. The nodes corresponding to the non-null success pointers are stored in contiguous memory and a pointer ($firstChild$) to the first of these stored in the 44-byte node. To make a state transition when the ASCII code for the current character is i, we first determine whether $Success[i]$ is $null$ by examining bit i of the map. In case this bit is $null$, the failure pointer is used. When this bit is not $null$, we determine the number of bits (popcount or rank) in bitmap positions less than i that are 1 and using this count, the size of a node (44-bytes), and the value of the first child pointer, determine the location of the node to transition to. Since, determining the popcount involves examining up to 255 bits, this operation is quite expensive (at least in software). To reduce the cost of determining the popcount, Tuck et al. [32] propose the use of summaries that give the popcount for the first $32 * j$, $1 \leq j < 8$ bits of the bitmap. Using these summaries the popcount for any i may be determined by adding together a summary popcount and up to 31 bit values. Each summary needs to be 8 bits long (the maximum value is 255) and 7 summaries are needed. The size of a bit compressed node with summaries is, therefore, 51 bytes. We note that the notion of using bitmaps and summaries for the compact representation of data structures (in particular, trees) was first advanced by Jacobson [12, 21] and has been used frequently in the context of data structures for network applications (see [5, 9, 17, 28], for example). While Jacobson [12, 21] suggests using several levels of summaries, [5, 32] use a single level. Also, Munro [21] has proposed a scheme that uses 3 levels of summaries, requires $O(m)$ space, where m is the size of the bitmap, and enables the computation of the popcount by adding three summaries, one from each level. The size of a bitmap node becomes 52 bytes when we add in the node type and failure pointer offset fields that are needed to support path compression (Figure 4).

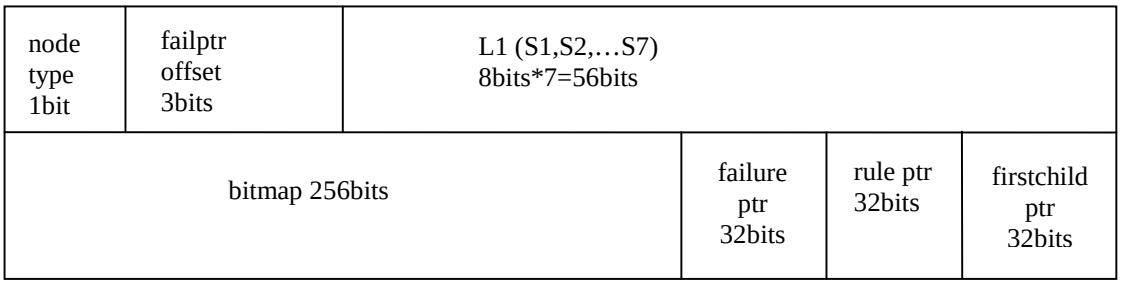

Figure 4: A bitmap node of [32]

2. *Path Compression.* Path compression is similar to end-node optimization [9, 17]. An *end-node sequence* is a sequence of states at the bottom of the automaton (the start state is at the top of the automaton) that are comprised of states that have a single non-null success transition (except the last state in the sequence, which has no non-null success transition). States in the same end-node sequence are packed together into one or more *path compressed* nodes. The number of these states that may be packed into a compressed node is limited by the capacity of a path compressed node. So, for example, if there is an end-node sequence $s_1, s_2, ..., s_6$ and if the capacity of a path compressed node is 4 states, then $s_1, ...s_4$ are packed into one node (say A) and s_5 and s_6 into another (say B). For each s_i packed into a path compressed node in this way, we need to store the 1-byte character for the transition plus the failure and rule list pointers for s_i. Since several automaton states are packed into a single compressed node, a 4-byte failure pointer that points to a compressed node isn't sufficient. In addition, we need an offset value that tells us which state within the compressed node we need to transition to. Using 3 bits for the offset, we can handle nodes with capacity $c \leq 8$. Note that now, $\lceil 3c/8 \rceil$ bytes are needed for the offsets. Hence, a path compressed node whose capacity is $c \leq 8$ needs $9c + \lceil 3c/8 \rceil$ bytes for the state information. Another 4 bytes are needed for a pointer to the next node (if any) in the sequence of path compressed nodes (i.e., a pointer from A to B). An additional byte is required to identify the node type (bitmap and compressed) and the size (number of states packed into this compressed node). So, the size of a compressed node is $9c + \lceil 3c/8 \rceil + 5$ bytes. The node type bit is

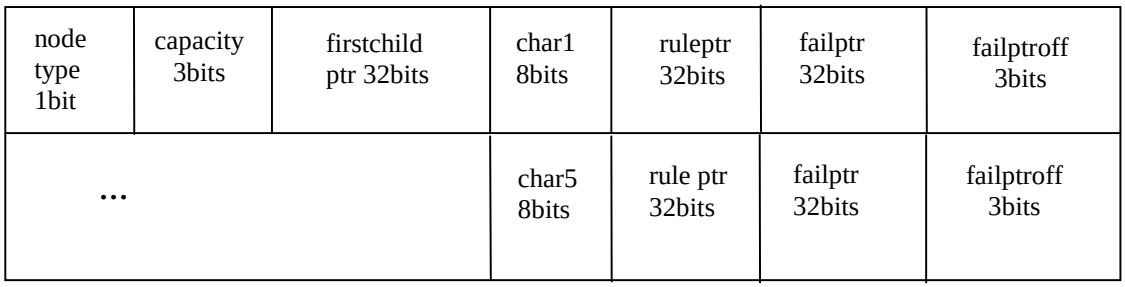

Figure 5: A path compressed node of [32]

required now in bitmap nodes as well as is an offset for the failure pointer. Accounting for these fields, the size of a bitmap node becomes 52 bytes. Since a compressed node may be a sibling (states/nodes reachable by following a single success pointer from any given state/node are siblings) of a bitmap node, we need to keep the size of both bitmap and path compressed nodes the same so that we can access easily the jth child of a bitmap node by performing arithmetic on the first child pointer. This requirement limits us to $c = 5$ and a path compressed node size that is 52 bytes. Figure 5 shows a path compressed node.

On the 1533-string Snort database of 2003, the memory required by the bitmapped-path compressed automaton using 1 level of summaries is about 1/50 that required by the optimized automaton, about 1/27 that required by the Wu-Manber data structure, and about 10% less than that required by the SFK search data structure [32]. However, the average search time, using a software implementation, is increased by between 10% and 20% relative to that for the optimized automaton, by between 30% and 100% relative to the Wu-Manber algorithm, and is about the same as for SFK search. The real payoff from the Aho-Corasick automaton comes with respect to worst-case search time. The worst-case search time using the Aho-Corasick automaton is between 1/4 and 1/3 that when the Wu-Manber or SFK search algorithms are used. The worst-case search time for the bitmapped-path compressed unoptimized automaton is between 50% and 100% more than for the optimized automaton [32].

5 Popcounts With Fewer Additions

A serious deficiency of the compression method of [32] 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 [32]. Since popcounts are used in a variety of network algorithms ([5, 9, 17, 28], 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 ([12, 20, 21], for example). In the algorithms community, the popcount problem is referred to as the bit-vector-rank problem, where the terms bitmap and bit vector are synonyms and popcount and rank are synonyms. We recast the best result for the bit-vector-rank problem using the bitmap-popcount terminology.

Munro [20, 21] has proposed a method to determine the popcount for m-bit bitmap using 3 levels of summaries that together take $o(m)$ bits of space. The popcount is determined by adding together 3 $O(\log m)$-bit numbers, one from each of the 3 levels of summaries. Munro's method is described below:

- *Level 1 Summaries* Partition the bitmap into blocks of $s1 = \lceil \log_2^2 m \rceil$ bits. The number of such blocks is $n1 = \lceil m/s1 \rceil$. Compute the level 1 summaries $S1(1 : n1)$, where $S1(i)$ is the number of 1s in blocks 0 through $i-1$, $1 \leq i \leq n1$.

- *Level 2 Summaries* Each level 1 block j is partitioned into subblocks of $s2 = \lceil \frac{1}{2} \log_2 m \rceil$ bits. The number of such subblocks is $n2 = \lceil s1/s2 \rceil$. $S2(j, i)$ is the number of 1s in subblocks 0 through $i-1$ of block j, $0 \leq j < n1$, $1 \leq i < n2$.

- *Level 3 Summaries* For the level 3 summaries, a lookup table $Ts2$ that gives the popcount for every possible position in every possible subblock is computed. The number of possible subblocks is $2^{s2} = O(\sqrt{m})$ and there are $s2$ possible positions in a subblock. Also, each entry of the table has $O(\log s2) = O(\log \log m)$ bits. So, the size of the lookup table is $O(\sqrt{m} \log m \log \log m)$ bits. Figure 6 gives the lookup table $T4$, which is for the case $s2 = 4$. $T4(i, j)$ is the number of 1s in positions 0 through $j-1$ in the

i	in binary	$T4(i,0)$	$T4(i,1)$	$T4(i,2)$	$T4(i,3)$
0	0000	0	0	0	0
1	0001	0	0	0	0
2	0010	0	0	0	1
3	0011	0	0	0	1
4	0100	0	0	1	1
5	0101	0	0	1	1
6	0110	0	0	1	2
7	0111	0	0	1	2
8	1000	0	1	1	1
9	1001	0	1	1	1
10	1010	0	1	1	2
11	1011	0	1	1	2
12	1100	0	1	2	2
13	1101	0	1	2	2
14	1110	0	1	2	3
15	1110	0	1	2	3

Figure 6: Lookup table for 4-bit blocks

binary representation of i; positions are numbered left to right beginning with 0 and a 4-bit representation of i is used.

One may verify that the total space required by the summaries is $o(m)$ bits and that a popcount may be determined by adding one summary from each of the three levels. For a 256-bit bitmap, using Munro's method [20, 21], the level-1 blocks are $s1 = 64$ bits long and there are $n1 = 4$ of these; each level-1 block is partitioned into $n2 = 16$ subblocks of size $s2 = 4$; and the lookup table $Ts2$ is $T4$.

Motivated by the work of Munro [20, 21], 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 $i-1$, $1 \leq i \leq 3$.

   - *Level 2 Summaries* For each block j of 64 bits, we keep a collection of level 2 sum-

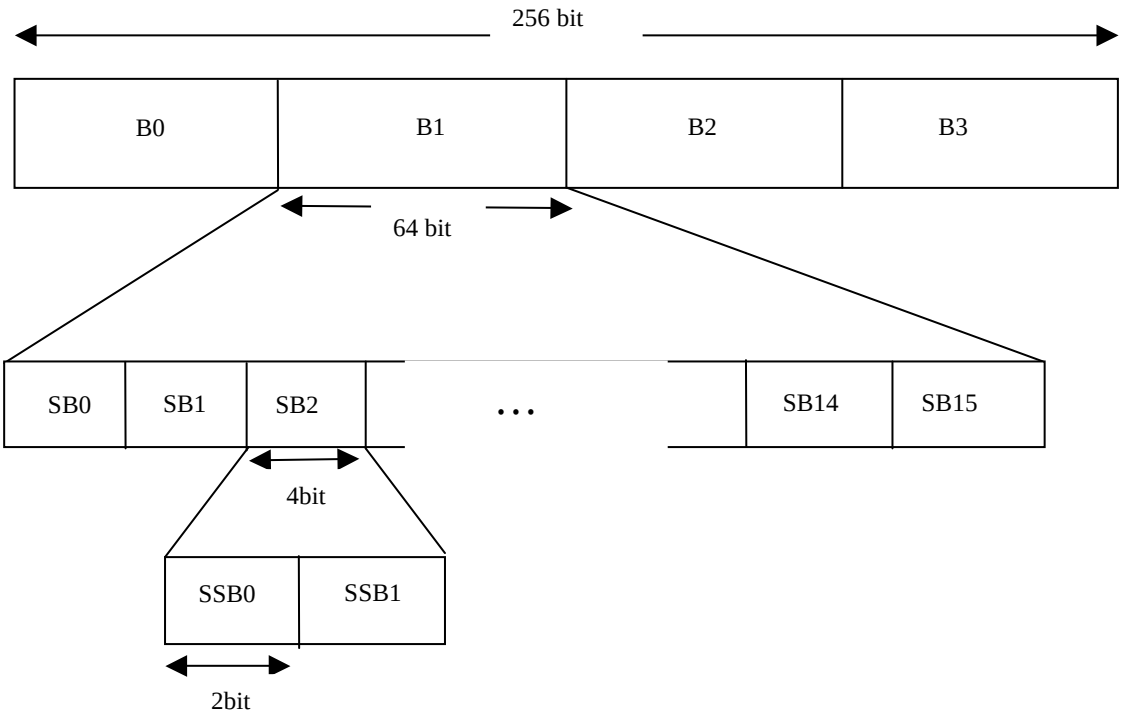

Figure 7: Type I summaries

maries. 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, $0 \leq j \leq 3$, $1 \leq i \leq 15$.

- *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, $0 \leq j \leq 3$, $0 \leq i \leq 15$.

Figure 7 shows the setup for Type I summaries. When Type I summaries are used, the popcount for position q (i.e., the number of 1s preceding position q), $0 \leq q < 256$, of the bitmap is obtained as follows:

Step 1: 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.

Step 2: 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.

As an example, consider the case $q = 203$. This bit is in subblock $sb = \lfloor(203 \bmod 64)/4\rfloor = \lfloor 11/4\rfloor = 2$ of block $b = \lfloor 203/64\rfloor = 3$. Since $203 \bmod 4 = 3$, the subsubblock ssb is 1. The popcount for bit 203 is the number of 1s in positions 0 through 191 + the number in positions 192 through 199 + those in positions 200 through 201 + the number in position 202 = $S1(3) + S2(3,2) + S3(3,2,1) + bit(202)$.

Since we do not store summaries for b, sb, and ssb equal to zero, the code to compute the popcount takes the form

```
if (b) popcount = S1(b)
else popcount = 0;
if (sb) popcount += S2(b,sb);
if (ssb) popcount += S3(b,sb,ssb);
if (q) popcount += bit(q-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 [32], 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 [32] require only 56 bits (or 7 bytes).

2. *Type II Summaries* These are exactly what is prescribed by Munro [20, 21]. $S1$ and $S2$ are as for Type I summaries. However, the $S3$ summaries are replaced by a summary table (Figure 6) $T4(0:15, 0:3)$ 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 [32].

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 [32]. 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.

6 Our Method To Compress The Non-Optimized Aho-Corasick Automaton

6.1 Classification of Automaton States

The Snort database had 3,578 strings in April, 2006. Figure 8 profiles the states in the corresponding unoptimized Aho-Corasick automaton by degree (i.e., number of non-null success pointers in a state). As can be seen, there are only 36 states whose degree is more than 8 and the number of states whose degree is between 2 and 8 is 869. An overwhelming number of states (24,417) have a degree that is less than 2. However, 1639 of these 24,417 states are not in end-node sequences. These observations 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.

BB All B states are reclassified as BB states.

BL All L states that have a sibling BB state are reclassified as a BL states.

BO All O states that have a BB sibling are reclassified as BO states.

LL All remaining L states are reclassified as LL states.

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	6	< 0.03
11	3	< 0.03
12	4	< 0.03
13	5	< 0.03
14	3	< 0.03
15	2	< 0.03
17,18,21,51,78	1	< 0.03

Figure 8: Distribution of states in a 3000 string Snort database

LO All remaining *O* states that have an *LL* sibling are reclassified as *LO* states.

OO All remaining *O* states are reclassified as *OO* states.

6.2 Node Types

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

Bitmap

A bitmap node has a 256-bit bitmap together with summaries; any of the three summary types described in Section 5 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,

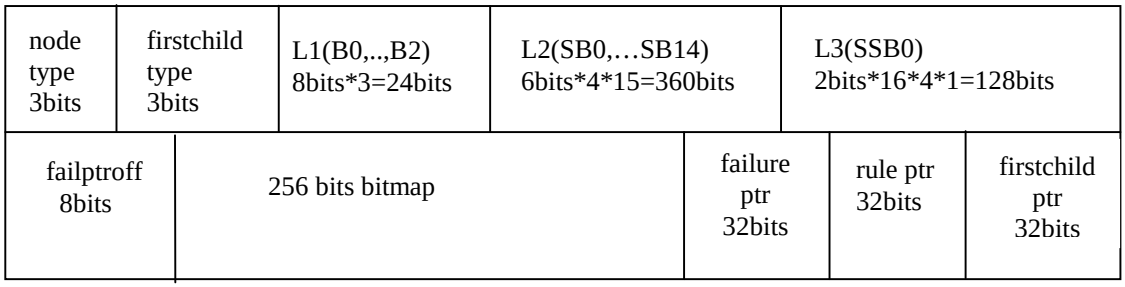

Figure 9: Our bitmap node

using a 512KB unoptimized $T16$ is quite wasteful of memory and it is desirable to go down to at least the 256KB version.

The memory required for a bitmap node depends on the summary type that is used. When Type I summaries are used, each bitmap node (Figure 9) is 110 bytes (we need 57 extra bytes compared to the 52-byte nodes of [32] 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.

Low Degree Node

Low degree nodes are used for states that have between 2 and 8 success transitions. Figure 10 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.

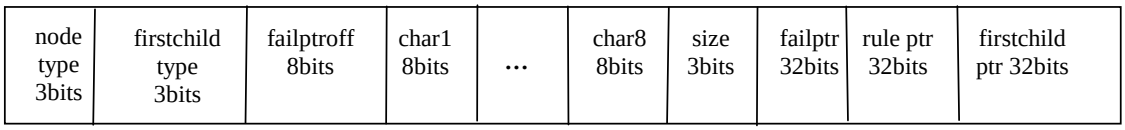

Figure 10: Our low degree node

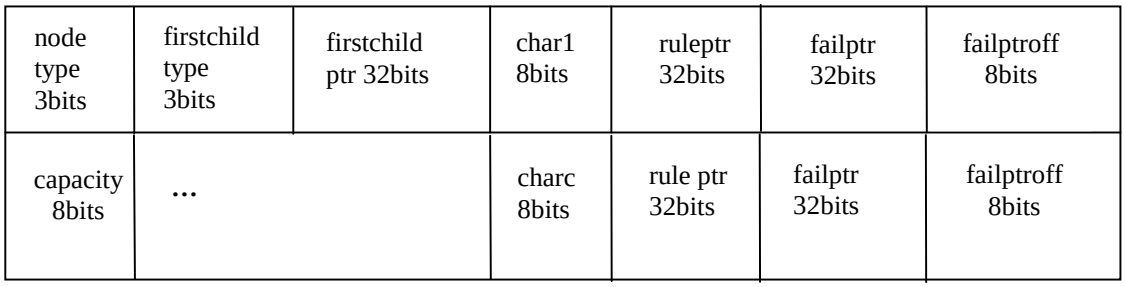

Figure 11: Our path compressed node

Path Compressed Node

Unlike [32], 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 [32] 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 11).

We refer to the path compressed node of Figure 11 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.

6.3 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.

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.

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 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.

$Ol, 1 \leq l \leq 5$, Nodes, $W = 32$

For an $O1$ node, we place the type, failure pointer offset, and char 1 fields into a memory block; the rule, failure and first child pointers are placed into individual memory block. To process an $O1$ node, we first retrieve the type block and then the rule pointer. The rule pointer is used to list the matching rules. Then, we compare with char 1 that is the retrieved type block. If there is a match, we retrieve the first child pointer and proceed to the node pointed at. In case of no match, we retrieve the failure pointer, which together with the

offset in the type block leads us to the next node. So, 3 accesses are needed when an $O1$ node is reached.

The mapping for an $O2$ is similar to that used for an $O1$ node. This time, the type block contains char 1 and char 2, the additional rule pointer and failure offset pointers are placed in separate blocks. The number of memory accesses needed to process such a node is 3 when only char 1 is examined (this happens when there is a mismatch at char 1). When char 2 also is examined an additional rule pointer is retrieved. For a mismatch, we must retrieve the second failure pointer as well as its failure pointer offset. So, 5 accesses are needed. For a match, 4 accesses are required. So, in case of a mismatch in an $O2$ node, 3 or 5 accesses are needed; otherwise, 4 are needed.

For $O3$ nodes, we place char 3 and its associated failure pointer offset into the memory block of $O2$ that contains the second failure pointer offset. The associated rule and failure pointers are placed in separate memory blocks. When all 3 characters are matched, we need 6 memory accesses. When a mismatch occurs at char 1, there are 3 accesses; at char 2, there are 5 accesses; and at char 3, there are 6 accesses.

An alternative mapping for an $O3$ node places the data fields into memory in the following order: node and first child type fields (1 byte total), pairs of character and rule pointer fields ((char j, rule pointer j), 5 bytes per pair), first child pointer (4 bytes), pairs of failure pointer and failure pointer offsets (5 bytes per pair). When i characters are examined, we retrieve $\lceil (1+5i)/4 \rceil$ blocks to process the characters and their rule pointers. In case of a mismatch at character i, 2 additional accesses are needed to retrieve the corresponding failure pointer and its offset. In case of a match, a single additional memory access gets us the first child pointer. So, the total number of memory accesses is $\lceil (1+5i)/4 \rceil + 2$ when there is a mismatch and $\lceil (1+5i)/4 \rceil + 1$ when all characters in the nodes are matched. When this alternative matching is used, a mismatch at character i, $1 \leq i \leq 3$ takes 4, 5, and 6 memory accesses, respectively. When there is no mismatch, 5 memory accesses are required.

For an $O4$ node, we extend the original $O3$ mapping by placing char 3, char 4, and offset pointers 3 and 4 in one memory block; and offset pointer 2 in another. Rule and failure pointers occupy one block each. When all 4 characters are matched, we need 7 memory

accesses. A mismatch at character i, $1 \leq i \leq 4$, results in 3, 5, 6, and 7 accesses, respectively.

An $O5$ node is mapped with chars 3, 4, 5 and offset pointer 3 in a memory block and offset pointers 2, 4, and 5 in another. When all 5 characters in an $O5$ node are matched, there are 8 memory accesses. When there is a mismatch at character i, $1 \leq i \leq 5$, the number of memory accesses is 3, 5, 6, 8, and 9, respectively.

O Nodes, $W = 32$ and 1024

For simplicity, we extend the alternative mapping described above for $O3$ nodes. Fields are mapped to memory in the order: node type, first child type, and capacity fields (2 bytes total), pairs of character and rule pointer fields ((char j, rule pointer j), 5 bytes per pair), first child pointer (4 bytes), pairs of failure pointer and failure pointer offsets (5 bytes per pair). The memory access analysis is similar to that for $O3$ nodes and the total number of memory accesses, when $W = 32$, is $\lceil (2 + 5i)/4 \rceil + 2$ when there is a mismatch and $\lceil (2 + 5i)/4 \rceil + 1$ when all characters in the nodes are matched.

When $W = 1024$, an O node fits into a single memory block provided its capacity, c, is no more than 12. Hence, for $c \leq 12$, a single memory access suffices to process this node. When $c > 12$, the memory access count using the above mapping is is $\lceil (2 + 5i)/128 \rceil + 1$. Since $i \leq c \leq 256$, at most 12 memory access are need to process an O node when $W = 1024$.

Path Compressed Node of [32], $W = 32$ and 1024

When $W = 32$, the type, size, failure offset 1, and char 1 through 3 fields of the path compressed node of [32] may be mapped into a single memory block. The char 4 and 5 fields together with the 4 remaining failure pointer offset fields may be mapped into another memory block. For a mismatch at char 1, we need to access block 1, rule pointer 1, and failure pointer 1 for a total of 3 memory accesses. For a failure at char i, $2 \leq i \leq size$, we must access also block 2 and an additional $i - 1$ rule pointers. The memory access count is $3 + i$.

Notice that since [32] path compresses end-node sequences only, a failure must occur whenever we process a path compressed node whose size is less than 5 as the last state in

such a node has no success transition (i.e., its degree is 0 in the Aho-Corasick automaton). Hence, for a match at this node, we may assume that the size is 5. The two blocks, 5 rule pointers, and the first child pointer are accessed. The total number of memory accesses is 8.

When $W = 1024$, all 52 bytes of the path compressed node fit in a memory block. So, only 1 memory access is needed to process the node. Note that for an end-node sequence with 256 states, 53 path compressed nodes are used. The worst-case accesses to go through this end-node sequence is 53. Using our O node, 12 memory accesses are made in the worst case.

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. Figures 12 and 13 give 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. [32]. We note that the counts of Figures 12 and 13 are specific to a certain mapping of the fields of a node to memory. Using a different mapping will change the memory access count. However, we believe that the mappings used in our analysis are quite reasonable and that using alternative mappings will not improve these counts in any significant manner.

6.4 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

	W=32		W=64	
	match	mismatch	match	mismatch
B (Type I)	4 to 7	4	3 to 6	3
B (Type II)	4 to 6	4	3 to 5	3
B (Type III)	4 to 5	4	3 to 4	3
L	3 to 5	3 to 5	2 to 3	2 to 3
$O1$	3	3	2	2
$O2$	4	3 or 5	2	2 or 3
$O3$	6	3, 5, or 6	3	2 or 3
$O4$	7	3 or 5 to 7	4	2 to 4
$O5$	8	3, 5, 6, 8, or 9	4	2, 3 to 5
O	3 or $\lceil\frac{2+5i}{4}\rceil$+1	3 or $\lceil\frac{2+5i}{4}\rceil$+2	2 or $\lceil\frac{2+5i}{8}\rceil$+1	2 or $\lceil\frac{2+5i}{8}\rceil$+1
TB ([32])	4 to 5	4	3	3
TO ([32])	$1+i$, 6, or 8	3 or $3+i$	$1+i$ or 7	2+$\lceil\frac{i}{2}\rceil$ or 4

Figure 12: Memory accesses to process a node

	W=128		W=1024	
	match	mismatch	match	mismatch
B (Type I)	2 to 5	2	1	1
B (Type II)	2 to 4	2	1	1
B (Type III)	2 to 3	2	1	1
L	1 to 2	1 to 2	1	1
$O1$	1	1	1	1
$O2$	1	2	1	1
$O3$	2	2	1	1
$O4$	2	2	1	1
$O5$	2	2 or 3	1	1
O	1 or $\lceil\frac{2+5i}{16}\rceil$+1	1 or $\lceil\frac{2+5i}{16}\rceil$+1	1($c \leq 12$), $\lceil\frac{2+5i}{128}\rceil$+1($c > 12$)	1($c \leq 12$), $\lceil\frac{2+5i}{128}\rceil$+1($c > 12$)
TB ([32])	2	3	1	1
TO ([32])	2	2	1	1

Figure 13: Memory accesses to process a node

structure of Figure 10. By this, we mean that if there are (say) 3 LX states that are siblings and there is a bitmap node with at least 75 bytes of unused space, all 3 siblings are packed into this unused space. If there is no bitmap node with this much unutilized space, none of the 3 siblings is packed into a bitmap node. The packing of sibling LX nodes is done in non-increasing order of the number of siblings. Note that

by packing all siblings into a single bitmap node, we make it possible to access any child of a bitmap node using its first child pointer, the child's rank (i.e., index in the layout of contiguous siblings), and the size of the first child (this is determined by the type of the first child). Note that 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, which is the same size as a low degree node.

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.

7 Experimental Results

We benchmarked our compression method of Section 6 against that proposed by Tuck et al. [32] using two data sets of strings extracted from Snort [26] 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.

Number of Nodes

Figures 14 15 give the number of nodes of each type in the compressed Aho-Corasick structure for each of our string sets. The maximum capacity of an allocated O node was 128 for data set 1284 and 256 for data set 2430.

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

Figure 14: Number of nodes of each type, Ol and O counts are for Type I summaries

Node Type	Ol (Type II)	O (Type II)	Ol (Type III)	O (Type III)
DataSet 1284	848	456	851	464
DataSet 2430	938	576	940	578

Figure 15: Number of Ol O nodes for Type II and Type III summaries

Memory Requirement

Although the total number of nodes used by us is less than that used by Tuck et al. [32], our nodes are larger and so the potential remains that we actually use more memory than used by the structure of Tuck et al. [32]. Figures 16 and 17 give the number of bytes of memory used by the structure of [32] as well as that used by our structure for each of the different summary types of Section 5. Recall that the size of a B node depends on the summary type that is used. As stated in Section 6, the B node size is 110 bytes for Type I summaries, 94 bytes for Type II summaries, and 61 bytes for Type III summaries. The memory numbers given in Figures 16 and 17 do not include the 16 bytes (or less) needed for the single $T4$ table used by Type II summaries or the 256KB needed by the $T16$ table used by Type I summaries. In the case of Type II summaries, adding in the 16 bytes needed by $T4$ doesn't materially affect the numbers reported in Figures 16 and 17. For Type III summaries, the 256KB needed for $T16$ is more than what is needed for the rest of the data structure. However, as the data set size increases, this 256KB remains unchanged and fixed at 256KB. The row labeled *Normalized* gives the memory required normalized by that required by the structure of Tuck et al. [32]. The normalized values are plotted in Figure 18. As can be seen, our structures take between 24% and 31% less memory than is required by the structure of [32]. With the 256KB required by $T16$ added in for Type III summaries, the Type III representation takes twice as much memory as does [32] 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

Methods	[32]	Type I	Type II	Type III
Memory (bytes)	208624	157549	155603	152237
Normalized	1	0.76	0.75	0.73

*Excludes memory for $T4$ and $T16$

Figure 16: Memory requirement for data set 1284

Data set 2430)				
Methods	[32]	Type I	Type II	Type III
Memory(bytes)	251524	177061	175511	172523
Normalized	1	0.70	0.70	0.69

*Excludes memory for $T4$ and $T16$

Figure 17: Memory requirement for data set 2430

than [32] on total memory required.

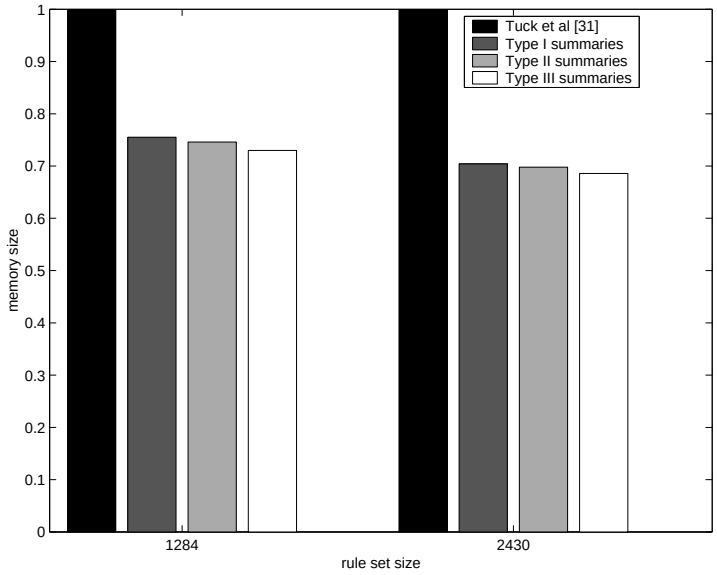

Figure 18: Normalized memory requirement

Popcount

Figures 19 and 20 give the total number of additions required to compute popcounts when using each of the data structures. For this experiment, we used 5 query strings obtained by concatenating a differing number of real emails that were classified as spam by our spam

Methods	[32]	Type I	Type II	Type III
strlen=1002832	10.61M	1.37M	1.25M	0.76M
strlen=2032131	32.21M	4.15M	3.79M	2.29M
strlen=3002665	64.26M	8.25M	7.51M	4.55M
strlen=4006579	107.21M	13.74M	12.49M	7.56M
strlen=5035666	161.76M	20.75M	18.82M	11.37M
Normalized	1	0.128	0.117	0.071

Figure 19: Number of popcount additions, data set 1284

Methods	[32]	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
strlen=4006579	116.11M	14.67M	13.28M	8.00M
strlen=5035666	175.60M	22.25M	20.09M	12.08M
Normalized	1	0.127	0.114	0.069

Figure 20: Number of popcount additions, data set 2430

filter. The string lengths varied from 1MB to 5MB 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 5 query strings normalized by the total for the structure of [32]. The normalized values are plotted in Figure 21. When Type III summaries are used, the number of popcount additions is only 7% that used by the structure of [32]. Type I and Type II summaries require about 13% and 12%, respectively, of the number of additions required by [32].

Memory Accesses

Figures 22 and 23 give the normalized number of memory accesses required to process our query strings. The data is normalized using the total memory access count for the method of [32]. Since the normalized numbers were virtually the same for each of our 5 query strings, we give only the numbers for the first query string. The number of memory accesses using our data structure is generally larger than when the structure of [32] is used. However, as the memory bandwidth increases, the difference between the two schemes becomes very small

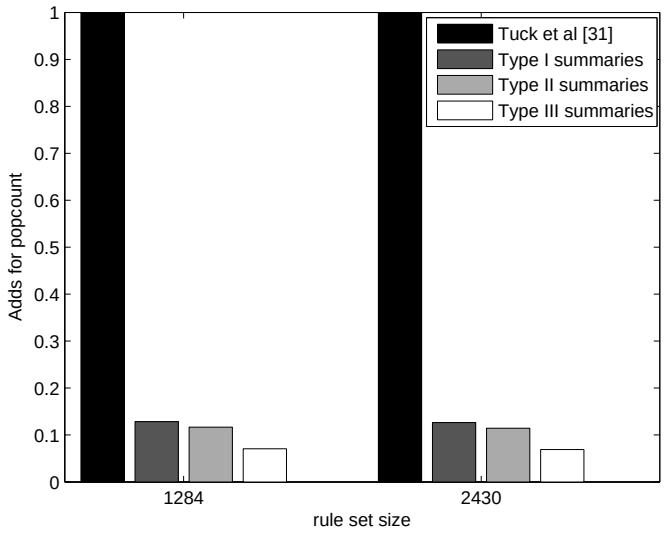

Figure 21: Normalized additions for popcount

Methods	W=32	W=64	W=128	W=1024
[32]	1	1	1	1
Type I	1.108	1.241	1.136	1.004
Type II	1.069	1.184	1.064	1.004
Type III	1.007	1.093	0.977	1.004

Figure 22: Normalized memory accesses to process a query string, data set 1284

Methods	W=32	W=64	W=128	W=1024
[32]	1	1	1	1
Type I	1.115	1.250	1.158	1.0021
Type II	1.076	1.193	1.086	1.0021
Type III	1.013	1.100	0.983	1.0021

Figure 23: Normalized memory accesses to process a query string, data set 2430

(0.1% to 0.4% when $W = 1024$) on this metric.

8 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 [20, 21]. As near as we can tell, we are the first to use more than 1 level of summaries for popcount computation in network applications. 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 [32]. 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 [32] is used. Although a search using our structure makes more memory accesses than when the structure of [32] is used, the two schemes make almost the same number of memory accesses when the memory bandwidth is sufficiently large.

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] F. Baboescu, S. Singh and G. Varghese, Packet Classification for Core Routers: Is there an alternative to CAMs? *INFOCOM*, 2003.

[4] R. Bace and P. Mell, Intrusion detection systems, *NIST Special Publication on IDSs*.

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

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

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

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

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

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

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

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

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

[14] H. Lu and S. Sahni, $O(\log W)$ multidimensional packet classification, *IEEE/ACM Transactions on Networking*, 15,2, 2007, 462–472.

[15] W. Lu and S. Sahni, Packet classification using two-dimensional multi-bit tries, *IEEE Symposium on Computers and Communications*, 2005. *http://www.cise.ufl.edu/∼wlu/papers/2dtries*.

[16] W. Lu and S. Sahni, Packet classification using pipelined two-dimensional multibit tries, *IEEE Symposium on Computers and Communications*, 2006.

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

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

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

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

[21] J. Munro and S. Rao, Succinct representation of data structures, in *Handbook of Data Structures and Applications*, D. Mehta and S. Sahni ed., Chapman & Hall/CRC, 2005.

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

[23] S. Sahni, *Data structures, algorithms, and applications in C++*, Second Edition, Silicon Press, 2005.

[24] S. Singh, F. Baboescu, G. Varghese, and J. Wang, Packet classification using multidimensional cutting, *ACM Sigcomm*, 8, 2003.

[25] Snort users manual 2.6.0, 2006.

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

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

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

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

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

[31] D. Taylor and J. Turner, ClassBench: A packet classification benchmark, *INFOCOM*, 2005.

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

[33] M.Waldvogel, G.Varghese, J.Turner, and B.Plattner, Scalable high-speed prefix matching, *ACM Trans. on Computer Systems*, 19, 4, 440-482, 2001.

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

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

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