Optimizing Pattern Matching for Intrusion Detection

Marc Norton

Abstract – **This paper presents an optimized version of the Aho-Corasick [1] algorithm. This design represents a significant enhancement to the author's original implementation released in 2002 as part of an update to the Snort Intrusion Detection System. The enhanced design uses an optimized vector implementation of the Aho-Corasick state table that significantly improves performance. A memory efficient variant uses sparse matrix storage to reduce memory requirements and further improve performance on large pattern groups.**

Intrusion Detection Systems are very specialized applications that require real-time pattern matching capabilities at very high network speeds, and in hostile environments. Several of the major issues that must be considered in pattern matching and Intrusion Detection are discussed to establish a framework for the use of the Aho-Corasick algorithm as implemented in the Snort Intrusion Detection System.

The performance results comparing the original, optimized, and sparse storage versions of the authors Aho-Corasick algorithm are presented. Tests were conducted using several dictionary tests and a Snort based Intrusion Detection performance test. The impact of pattern group sizes and compiler selection on performance is also demonstrated using several popular compilers.

Index Terms – **pattern matching, Aho-Corasick, Intrusion Detection, IDS, Snort**

I. Introduction

SNORT is an open source Intrusion Detection System that relies heavily on the Aho-Corasick multi-pattern search engine. The performance characteristics of the Aho-Corasick algorithm implemented in Snort have a significant impact on the overall performance of Snort. Snort scans network traffic packets searching for intruders by looking for specific values in the network headers and by performing a search for known patterns in the application layer data.

Snort has utilized a high-speed multi-pattern search engine since the release of version 2.0 in 2002. The introduction of a multi-pattern search engine in Snort was part of a larger enhancement to the detection engine to improve performance, allow for larger rule sets, and achieve gigabit network performance levels [2].

The Sourcefire Intrusion Detection System uses Snort and has been tested and certified via the OSEC [19] test program to perform at 750 Mbits/sec in the OSEC test environment. Although Snort has been shown to perform well in gigabit network environments in the lab and at customer sites the quest for better performance and deeper inspection capabilities persists.

This paper presents an optimized Aho-Corasick implementation that improves on the design of the state table found in the author's original version. It is faster than the previous version, supports an optimized full matrix state table, and a memory efficient state table using a sparse matrix based storage method to minimize memory requirements.

Many other methods have been proposed for managing the storage and access of sparse state tables. Johnson [20] and Yao [21] detail table-compression using multiple vectors. Regular expression based search engines such as Yacc and Lex use variants of these methods. There are also hashing, trie, tree, and bitmap [10] methods. All of these methods attempt to provide an optimal minimum memory representation of the sparse state table, but usually with a measurable loss of performance compared to full matrix representations.

This paper presents some basic sparse matrix and vector storage formats and applies one to the Aho-Corasick state table. These particular methods of sparse storage, while not new, do not appear to have been used before to represent the Aho-Corasick state table. This may be due in part to the fact that these methods do not provide optimal solutions to minimizing memory. The storage savings is also pattern dependent and may provide no savings at all. Sparse storage does however offer significant storage savings for the pattern groups used by Snort to detect network attackers.

The selected sparse storage method requires only minor changes to the optimized algorithm, offers better performance than the author's original version, and is faster than the optimized version on large pattern groups in the dictionary tests.

The full matrix version is 1.5 to 2.5 times faster than the original version and the sparse matrix version is 1.2 to 1.7 times faster, depending on compiler, pattern group size, search frequency, and search text size.

Marc Norton is the Snort IDS Team Lead at Sourcefire ,Inc
marc.norton@sourcefire.com; www.idsresearch.org

II. PATTERN MATCHING AND INTRUSION DETECTION

The pattern search problem in Intrusion Detection Systems is a specialized problem in it's own right. It requires consideration of many issues associated with pattern searches. The following considerations should be accommodated by any pattern search engine used for real-time Intrusion Detection.

a) Multi-pattern search algorithms.
b) Pattern character case sensitivity.
c) Pattern sizes.
d) Pattern group size.
e) Alphabet size.
f) Algorithmic attacks.
g) Search text size.
h) Frequency of searches.

These design issues are discussed below to establish a framework for pattern matching in Intrusion Detection Systems and the enhancements provided by the Aho-Corasick search engine in this paper.

a) The real-time nature of inspecting network packets requires the use of a pattern search engine that can keep up with the speeds of modern networks. There are two types of data we use in this type of search, the patterns and the search text. The patterns are pre-defined and static. This allows them to be pre-processed into the best from suitable for any given pattern match algorithm. The search text is dynamically changing as each network packet is received. This prohibits us from pre-processing the search text prior to performing the pattern search. This type of pattern search problem is defined as a Serial pattern search [6]. The Aho-Corasick algorithm is a classic serial search algorithm, and was first introduced in 1975 [1]. Many variations have been inspired by this algorithm, and more recently Intrusion Detection Systems have generated a renewed interest in the algorithm due to some of its' unique properties.

b) Intrusion Detection Systems search for patterns that can be either case sensitive or case insensitive. The Aho-Corasick algorithm as originally described is a case sensitive pattern search engine. This version of the Aho-Corasick algorithm supports case sensitive and case insensitive patterns. This is accomplished by converting all patterns to upper case prior to insertion in the state machine, and than converting each character of the search stream to upper case for comparison as the stream is processed. This effectively produces a case independent search and guarantees no patterns are missed. If a pattern match is found and it is case sensitive, it is re-tested against the exact search text characters in a case sensitive manner to verify the match.

c) The size of the patterns used in a search can significantly affect the performance characteristics of the search algorithm. State machine based algorithms such as the Aho-Corasick algorithm are not affected by the size of the smallest or largest pattern in a group. Skip based methods such as the Wu-Manber algorithm and others that utilize character skipping features are very sensitive to the size of the smallest pattern. The ability to skip portions of a search text can greatly accelerate the search engines performance. However, skip distance is limited by the smallest pattern. Search patterns in Intrusion Detection represent portions of known attack patterns and can vary in size from 1 to 30 or more characters but are usually small.

d) The pattern group size usually affect IDS performance because the IDS pattern search problem usually benefits from processor memory caching. Small pattern groups can fit within the cache and benefit most form a high performance cache. As pattern groups grow larger, less of the pattern group fits in the cache, and there are more cache misses which reduces performance. Most search algorithms will perform faster with 10 patterns than they do with 1000 patterns, for instance. The performance degradation of each algorithm as pattern group size increases varies from algorithm to algorithm. It is desirable that this degradation be sub-linear in order to maintain scalability. The Aho-Corasick algorithm generally provides good scalability for the pattern groups used in Intrusion Detection, and as demonstrated in the test data shown below.

e) The alphabet size used in current Intrusion Detection Systems is defined by the size of a byte. An 8-bit byte value is in the range 0 to 255, providing Intrusion Detection Systems with a 256-character alphabet. These byte values represent the ASCII and control characters seen on standard computer keyboards and other non-printable values. For instance, the letter 'A' has a byte value of 65. Extremely large alphabets such as Unicode can be represented using pairs of byte values, so the alphabet the pattern search engine deals with is still 256 characters. This is a large alphabet by pattern matching standards. The English dictionary alphabet is 52 characters for upper and lower case characters, and DNA research uses a four character alphabet in gene sequencing. The size of the alphabet has a significant impact on which search algorithms are the most efficient and the quickest. Navarro [4] covers this topic and presents charts showing the best choice of search algorithms based on the alphabet size and minimum patterns sizes with data for patterns of five bytes or more. This paper extends that that pattern range to include pattern matching issues relevant to large alphabets and small pattern sizes of less than five characters.

f) Algorithmic attacks attempt to use the properties and behaviors of a search algorithm against itself to reduce the algorithm's performance. The performance behavior of an algorithm should be evaluated by considering it's average-case and worst-case performance. Algorithms that exhibit significantly different worst-case and average-case performance are susceptible to these attacks. Skip based algorithms , such as the Wu-Manber algorithm [3], utilize a character skipping feature similar to the bad character shift in the Boyer-Moore algorithm. Skip based algorithms are sensitive to the size of the smallest pattern since they can be shown to be limited to skip sizes smaller than the smallest pattern. A pattern group with a single byte pattern cannot skip over even one character or it might not find the single byte pattern.

The performance characteristics of the Wu-Manber algorithm can be attacked, as shown by Tuck et al. [10], and significantly reduced by malicious network traffic resulting in a Denial of Service attack. This type of attack requires degenerate traffic of small repeated patterns. This problem does not exist for text searches where algorithmic attacks are not intentional and are usually very rare. It is a significant issue in Intrusion Detection Systems where algorithmic attacks are prevalent and intentional. The Wu-Manber algorithm does not achieve its best performance levels in Snort because Snort rules often include very small search patterns of one to three characters, eliminating most of the opportunities to skip sequences of characters. In practice, Snort exhibits little differences in performance between the Wu-Manber and the author's original Aho-Corasick algorithm in the average-case, but Wu-Manber can be significantly slower in algorithmically designed attack traffic.

The strength of skip-based algorithms is evident when all of the patterns in a group are large, they can skip many characters at a time and are among the fastest average-case search algorithms available.

The Aho-Corasick algorithm is unaffected by small patterns, and its' worst-case and average-case performance are the same. This makes it a very robust algorithm for Intrusion Detection.

g) The size of a search text in Intrusion Detection Systems is usually less than a few thousand bytes. In general when searching a text, the expense of setting up the search and filling the cache with the necessary search pattern information is usually fixed. If the search is performed over just a few bytes, than spreading the setup costs over those few byte results in a high cost per byte. Whereas , if the search text is very large spreading the setup cost over the larger search text results in very little overhead cost added to searching each byte of text.

h) The frequency of searching in an Intrusion Detection System is dependent on the network bandwidth, the volume of traffic on the network, and the size of the network packets. This implies that the frequency of text searches and the size of each search text are related due to the nature of the network traffic being searched. Again, as with search text size, a high frequency of searching in an Intrusion Detection System will cause the search setup costs to be significant compared to doing fewer larger text searches.

III. The Aho-Corasick State Machine

The Aho-Corasick state machine is a specialized finite state machine. A finite state machine is a representation of all of the possible states of a system, along with information about the acceptable state transitions of the system. The processing action of a state machine is to start in an initial state, accept an input event, and move the current state to the next correct state based on the input event. It is easy to visualize a state machine as a matrix where the rows represent states and the columns represent events. The matrix elements provide the next correct state to move to based on the input event and possibly some specific action to be done or information to be processed when the state is entered or exited.

A simple example may clarify the state transition process. If the current state is state 10 and the next input event is event 6, than to perform a state transition you would find the matrix element at row 10 and column 6 in the matrix and change the current state to the state indicated by the value of this matrix element.

The Aho-Corasick state machine as implemented in Snort uses a Deterministic Finite Automata also called a DFA. A unique property of a DFA is that upon examining an input character in the text stream it requires exactly one transition of the state machine to find the correct next state. This is in contrast to a Non-Deterministic Finite Automata, also called a NFA, which can require more than one state transition to find the correct next state. A DFA can process data faster than an NFA since it requires fewer transition steps. Aho-Corasick [1] indicates that an NFA can require up to 2 times the transitions of the DFA to search a stream of data. The construction and structure of the state table matrix of a DFA is also more complicated than that of a NFA. Aho-Corasick [1] also shows how a DFA can be constructed from an NFA by pre-processing all of the possible state transitions in the NFA until it can be determined how to perform all state transitions in a single step. This procedure has the tendency to fill in more elements of the state table matrix. The physical representation of the Aho-Corasick state transition table varies from one implementation to the next. The choice of the state table representation determines the memory and performance tradeoffs of the search algorithm.

IV. Sparse Storage Formats

Sparse matrices and vectors contain a significant number of zero elements and only a few non-zero elements. Methods and storage formats used for operating on sparse structures efficiently are well developed in the field of Linear Algebra and utilized in many branches of science. There are many books and publicly available codes for operating on sparse matrices and structures. The structures of most interest to this paper are those representing sparse vectors. It is not sufficient to store sparse data efficiently. The storage format must also allow for fast random access to the data.

A sample sparse matrix is shown in figure 1 containing six rows and four columns.

0 0 0 3
0 4 0 1
0 0 0 6
1 0 0 0
0 2 0 0
0 0 5 0

Figure 1

This matrix is a general rectangular matrix with only a few non-zero elements. Using the Compressed Row Storage CSR format [14] it can be stored as shown in figure 2.

CSR Matrix Format

Value	: 3 4 1 6 1 2 5
Column	: 4 2 4 4 1 2 3
Row	: 1 2 4 5 6 7

Figure 2

This format requires three vectors to hold only the non-zero entries. They are a value vector, a column vector and a row vector. The Column array indicates the column the corresponding value in the Value array belongs to in the matrix. The Row index indicates the index in the Column and Value arrays where each row starts. There are seven entries in each of the value and column vectors, since there are seven non-zero entries in the matrix. There are six row entries, since there are six rows in the matrix. For instance, the third Row entry indicates the third row starts at the fourth entry in the Column and Value arrays. This is one of the simplest sparse storage schemes for a matrix and can be broken down to handle vectors by considering a single row to be a vector as shown Figure 3. We will call this the compressed sparse vector format.

Compressed Sparse Vector Format

Vector:	0 0 0 2 4 0 0 0 6 0 7 0 0 0 0 0 0 0 0 0
Values:	2 4 6 7
Index:	4 5 9 11

Figure 3

The Index array indicates the vector array index of the corresponding Value. There are four non-zero values, hence we have four entries in the Index and Value arrays. We can merge our Value and Index arrays together into one integer array as shown in Figure 4. We will call this the sparse-row format.

Sparse-Row Format

Sparse-Row Storage: 8 4 2 5 4 9 7 11 7

Figure 4

Here we have nine entries, the first is the total number of words that follows, followed by eight numbers representing four pairs of numbers in index-value order. Now we have a single array of nine entries, eight in index-value order. This could also be represented as an array of four C language structures each having an Index and a Value entry. In either case, we only need an array of the non-zero entries, and we need to track how many entries we keep. This storage scheme works well at minimizing the total storage of a sparse vector and works well in applications where we have to sequentially touch every element of the vector, such as in matrix-vector multiplies. However, for randomly looking up an individual entry this format requires us to search through the array to find the correct index.

Another representation of the vector in Figure 3 uses the banded nature of the vector to store the elements efficiently and allows us to maintain random access to its elements. We call this the banded-row format, and it is shown in Figure 5.

Banded-Row Format

Num Items:	8
Start Index:	4
Values:	2 4 0 0 0 6 0 7
Band Array:	8 4 2 4 0 0 0 6 0 7

Figure 5

The Banded-Row format stores elements from the first non-zero value to the last non-zero value, the number of terms stored is known as the bandwidth of the vector. Small bandwidth corresponds to large storage savings. To manage access to the data we only need to track the number of data elements and the starting index of the data. This format reduces the storage requirements, and still provides fast random access to the data. Many problems express themselves as banded matrices, in which case the banded-row format can be used for each row in the matrix. This type of banded storage makes no requirements of how the banded-ness of one row will correspond to banded-ness in another row.

The literature on storing and operating on Sparse Matrices and Banded Matrices are quite extensive. We have only scratched the surface of sparse matrix technology but it will suffice to improve the Aho-Corasick state table storage requirements.

V. THE OPTIMIZED AHO-CORASICK

The optimized Aho-Corasick is a refinement of the author's original version released in Snort in 2002. The state table is managed somewhat differently and this allows the search routine to be compiled to a more optimal instruction mix. Previously each entry in the state table had a vector of transitions for the state, a failure pointer for the NFA version of the table, and a list of matching patterns for the state, all contained in one structure. The previous state table is now broken into a state transition table, an array of per state matching pattern lists, and a separate failure pointer list for each state for the NFA.

The state transition table is a list of pointers to state vectors. Each state vector represents the valid state transitions for that state. Additionally, each vector also includes a small header that indicates the storage format of the row, and a Boolean flag indicating if any pattern matches occur in this state. The state transition data follows the header. Another change to the code was to convert each input character to uppercase as it is processed in the search. Previously the entire input text was converted prior to the search. Together these changes are shown in the tests below to significantly improve the performance of the previous algorithm.

The search routine is a small piece of code that essentially jumps from state to state based on the input characters in the text stream. A state machine operates by maintaining a current state, accepting an input event (a character in our case), and using a table or matrix to lookup the correct next allowed state based on the current state and input event as follows:

```
while ( input = next -input )
{
   state = state-transition-table[ state, input ]
   if( patterns matched in this state ) process  patterns....
}
```

It would appear that there is not a lot of opportunity for optimizing such a simple state machine. Yet optimizing this functionality of this state machine for both maximum speed and minimum storage has been the basis of extensive research for many years.

The code below shows the basic algorithm for the search used by the optimized Aho-Corasick algorithm.

```
for ( state = 0; T < Tend; T++ )
{
    ps      = NextState[  state  ];
    sindex = xlatcase[  T[0]  ];
    if(  ps[1]  )
    {
        for( mlist  = MatchList[state];
             mlist != NULL;
             mlist  = mlist->next )
        {
                /* process the pattern match */
        }
    }
    state = ps[  2u + sindex  ];
 }
```

The NextState array is the array of pointers to the row vectors of the state table. The 'ps' variable is used to point to the current state's transition vector. The first two words of the 'ps' array are the storage format and the boolean match flag. The 'T' parameter is the text being searched, and 'xlatcase' converts the text one byte at a time to upper case for a case independent search. Once a match is found the pattern is processed.

A check is made using ps[1] to determine if any patterns are matched in the current state, if not the state machine cycles to the next state. If there is a match all of the patterns that match in the current state are processed in the 'for' loop.

All of the compilers tested produced significantly faster code with this version than they did with the original version.

VI. THE SPARSE STORAGE AHO-CORASICK

The sparse storage format used in this version is the banded-row format previously described. The vectors of the state transition table are somewhat different for the banded-row format. The first word indicates the storage format and the second word indicates if any patterns match in this state as in the optimized full vector version. The third word indicates the number of terms that are stored for the row, and the fourth word indicates the index of the first term. The banded-row format allows us to directly index into the entry we wish to access. However, we must be careful now to do a bounds check prior to each indexing operation.

```
for ( state = 0; T < Tend; T++ )
{
   ps      = NextState[ state ];
   sindex = xlatcase[  T[0]  ];
    if(   ps[1]   )
    {
       for( mlist  = MatchList[ state ];
             mlist != NULL;
```

```
          mlist  = mlist->next )
        {
          /* process the pattern match */
         }
      }
     /* Bounds check  & state transition */
     if(      sindex <    ps[3]           )  state = 0;
     else if( sindex >= (ps[3] + ps[2]) )  state = 0;
     else   state = ps[  4u + sindex - ps[3]  ];
}
```

The bounds check is likely to reduce performance compared to the optimized version. Testing below presents a somewhat different picture and shows that with large pattern groups this algorithm out performs the optimized version.

VII. PERFORMANCE METRICS

The performance metrics for evaluating a multi-pattern search engine will be broken into three categories; algorithmic, computational, and problem domain metrics.

Algorithmic or theoretical metrics are based on consideration of the algorithm independent of the hardware or software. Worst-case behavior is an example of an algorithmic metric. Typically, the worst-case performance of a multi-pattern search engine is proportional to the size of the patterns and the length of the data being searched. The Aho-Corasick algorithm implemented by the author is an O(n) algorithm, indicating the search speed is proportional to n, the length of the data being searched.

Computational metrics are based on examining how an algorithm interacts with the computer hardware it runs on. The significant metrics we will consider are instruction mix, caching properties and pattern group size, and the search text length.

The instruction mix refers to the type of hardware instructions required by the algorithm. A sophisticated instruction mix might be indicative of algorithms that require special purpose machine instructions or hardware to work efficiently. This Aho-Corasick algorithm has no special instruction mix requirements and can run well on most general-purpose computers.

The caching properties of an algorithm can significantly affect performance. The strongest indicators of caching performance are cache size and data locality. Data locality can be defined by the relative amount of sequential versus random memory access the algorithm performs in and out of the cache. The Aho-Corasick algorithm jumps from state to state in the state table. These jumps are data driven, and are essentially random. If only a small part of the state table can fit in the computer's cache, there are likely to be many cache misses, and a cache miss may cost 10 times that of a cache hit. Therefore, the performance of the Aho-Corasick algorithm is very sensitive to the size of the cache, and whether the state table can fit in the cache. Ideally, the cache should be large enough to hold the entire state table and have room left to bring the search data into the cache as needed. This only happens for small state tables, or on systems with very large caches. The state table size is proportional to the total number of characters in all of the patterns included in the search.

The last issue we'll consider in measuring pattern matching performance is the problem domain of the search. The particular problem we apply a pattern matcher to defines the size of the search text, the number of patterns, and the frequency of searches. The problem domain that Snort is used in requires searching network traffic for known attack patterns. The pattern group sizes used in the Snort IDS are up to a thousand or more patterns. Snort searches network packets that average 600-800 bytes in each search, and Snort does this up to 200,000 times a second. In between pattern searches, other operations occur in Snort that potentially flush the cache. Each time a pattern search is started, the cache has to be reloaded which can represent a noticeable setup cost for each search. In direct contrast to this type of search is the sequential search of a very large data stream. This type of search allows the higher performance cached data accesses to greatly accelerate the overall search performance. The dictionary test below demonstrates this behavior.

VIII. TEST RESULTS

The test results for the new Aho-Corasick implementation includes benchmark data for the standard Aho-Corasick previously used in Snort, the new optimized version using full matrix storage, and the optimized version using banded-row storage.

The testing was divided into two types of pattern search tests. Dictionary tests are often used to demonstrate the relative merits of pattern search engines. The dictionary test selected was used to provide a long stream of text to search providing the software an opportunity to achieve the best possible caching performance. A network traffic capture file was also used and demonstrates the relative performance of all three versions of the Aho-Corasick algorithm as implemented in Snort.

Dictionary Test

The dictionary test selected 1000 patterns from throughout an online King-James bible. These patterns were used in groups of 10 to 1000 to search through the 1903 Webster's unabridged dictionary, about 2.3 megabytes of data. This ensured that there were plenty of successful pattern matches, causing the test to provide complete coverage of the search routine by executing both byte testing and matching pattern code.

The three versions of the Aho-Corasick algorithm tested included the standard version already in use in Snort, which treats the state table as a full matrix, the new optimized version again using full matrix storage, and the new optimized version using banded-row storage.

A test bed was developed that included several different compiled versions of each of the Aho-Corasick routines to be tested. The compilers used include Microsoft VC 6.0, Intel C 6.0, and the Cygwin gcc 3.3.1 compiler. Using three different compilers provided broader insight into the performance and caching behavior of the search algorithm and storage methods .

All results are for the DFA version of the Aho-Corasick algorithm. The exact results for the dictionary tests are shown in Charts 1, 2 and 3, results normalized against the standard Aho-Corasick algorithm are shown in Charts 4, 5, and 6. These tests were conducted using 16 bit state values on a Dell 8100 1.7 GHz system with 1 G-bytes of RAM.

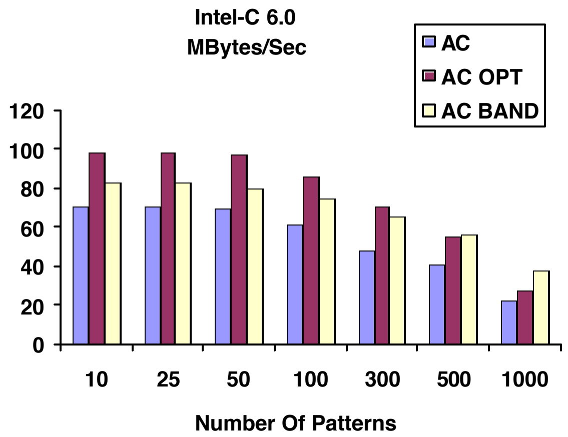

Chart 1

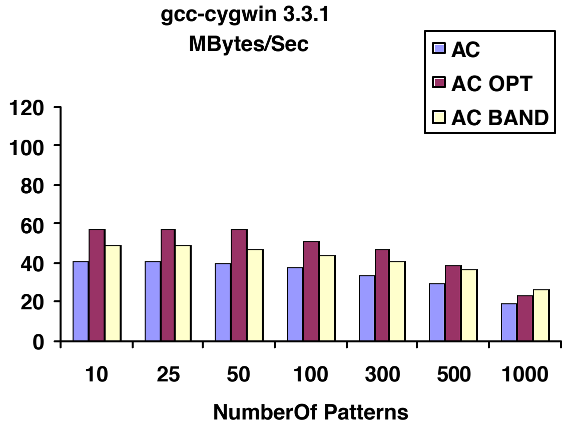

Chart 2

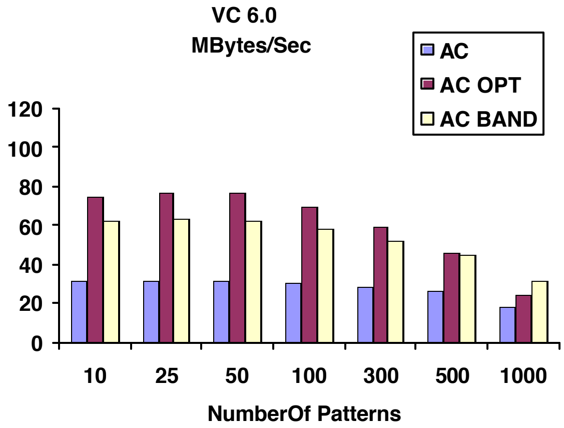

Chart 3

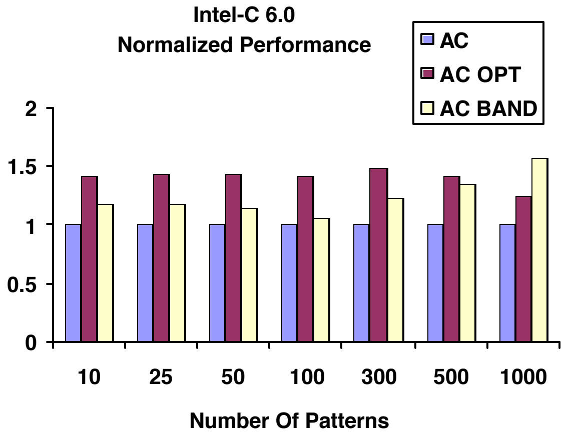

Chart 4

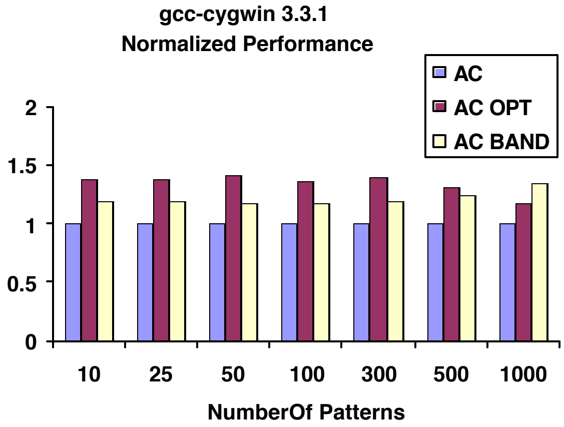

Chart 5

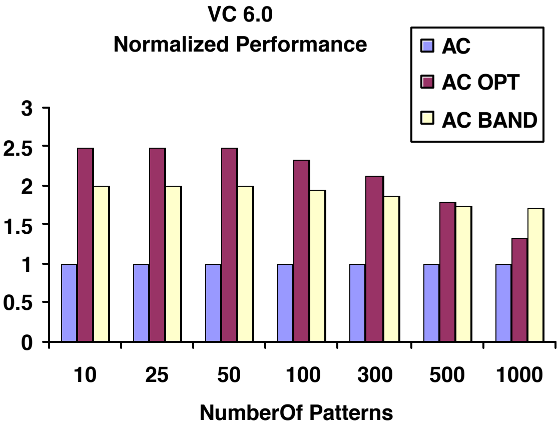

Chart 6

The optimized versions perform significantly better than the standard version in the dictionary tests with all three compilers. Comparing the compiler results in Charts 1, 2, and 3 shows the Intel compiler ranked best in raw performance. The gcc compiler performed better than the VC60 compiler with the standard Aho-Corasick, but the VC60 compiler performed better than the gcc compiler with both the optimized and banded Aho-Corasick.

All three compilers produced similar performance trends. We see at least a 1.4 times speed improvement of the optimized version over the standard version, except for Microsoft where we see as much as a 2.5 times improvement on smaller pattern groups. The optimized banded-row storage version also produces a significant speedup. In fact, it performed the best on the largest pattern group. All three compilers show the optimized full matrix version to be faster than the banded-row version for all pattern group sizes up to 500 patterns. At 1000 patterns, the banded-row version outperforms the optimized full matrix version for all three compilers.

The storage requirements for the different pattern group sizes are shown in Chart 7, with the tabular results shown in Table 1 for brevity. These results reflect a 16 bit state size.

Patterns vs Kbytes

	AC STD	AC OPT	AC BAND
10	58	30	3
25	182	93	9
50	370	188	19
100	729	371	41
300	2333	1160	130
500	3884	1930	239
1000	7567	3760	527

Table 1

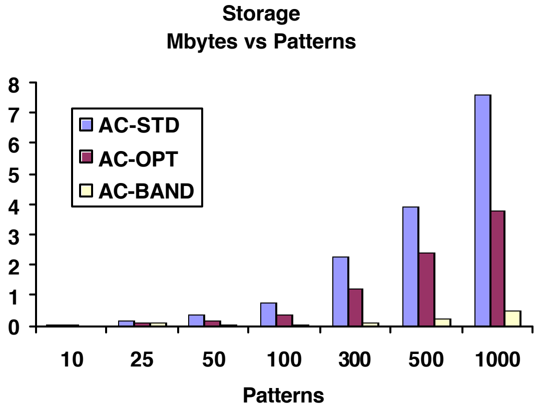

Chart 7

The optimized version uses one-half the memory of the standard version, due to its use of a 16-bit word size for state values. The banded-row storage uses about one-fifteenth the storage of the standard version, and about one-seventh the storage of the optimized full matrix version.

State Size Tests

The optimized and banded Aho-Corasick can use either 16-bit state or 32-bit state values. The 16-bit state values are limited to 2^16 or 65K states. The 32-bit state values are limited to 2^32 or 4 billion states. The merit of using 16-bit states is reduced memory consumption. Charts 8, 9, and 10 compare the performance of the 16-bit (AC OPT) and the 32-bit (AC OPT 32) optimized Aho-Corasick for the three compilers. Charts 11-13 compare the performance of the 16-bit (AC BANDED) and the 32-bit (AC BANDED 32) banded-row Aho-Corasick for the three compilers.

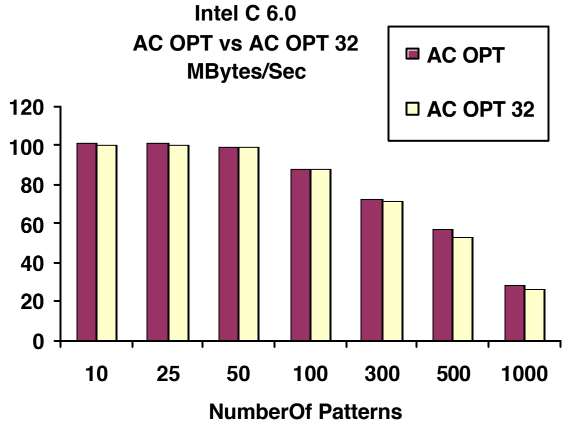

Chart 8

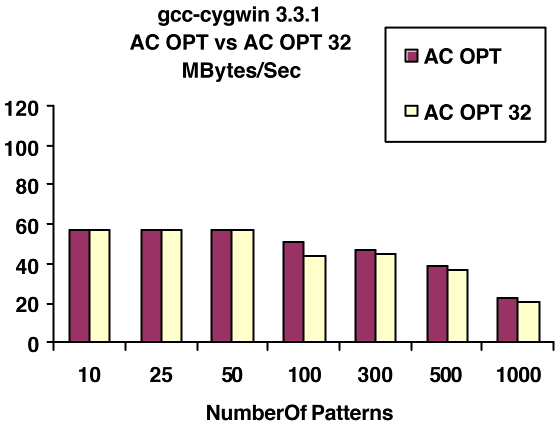

Chart 9

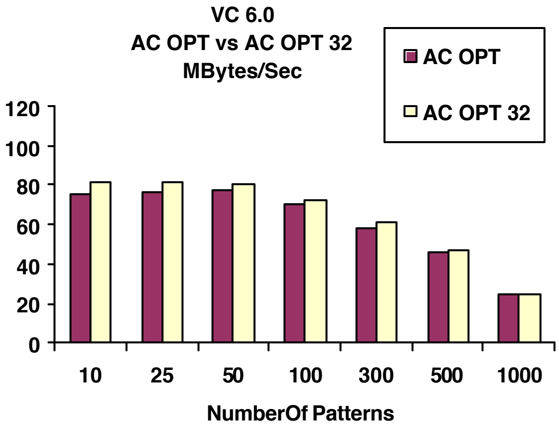

Chart 10

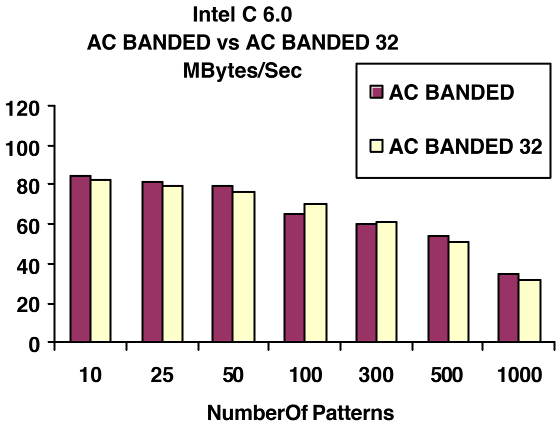

Chart 11

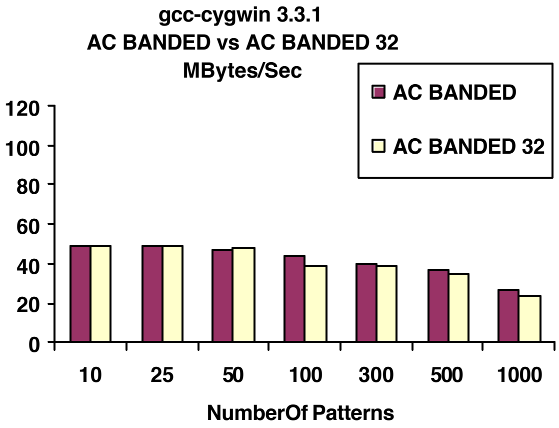

Chart 12

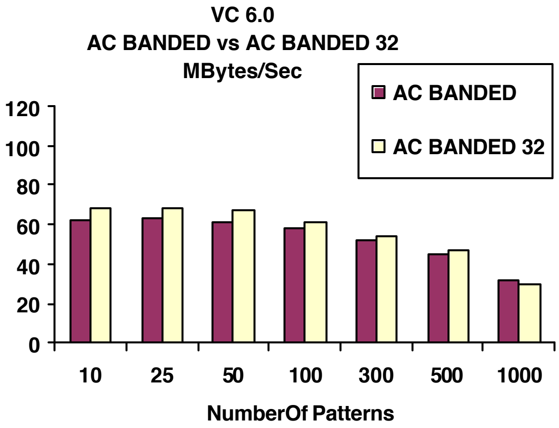

Chart 13

There are only minor differences between the 16 and 32 bit versions of both the optimized and banded versions of the algorithm. The VC60 compiler demonstrates the largest difference across all pattern sizes and favors the 32 bit state size.

Snort Test

Snort processes packets on standard Ethernet based networks that are up to 1460 bytes of data per packet. This type of data tests search performance and search setup costs on smaller search texts, and in a relatively cache unfriendly environment. The tests were performed by replaying previously captured traffic directly into Snort. The compiler used was the Intel C 6.0 compiler on Linux. The test file selected represents about 2 G-bytes of web centric traffic. It was selected since Snort allows us to select how much of the traffic is inspected in each web request. The test works as follows; Snort was configured to inspect the first 300 bytes of each web page request. This is typically how Snort might be configured. The Unix time command was used and the file was processed as usual, noting the user processing time.

Snort then was configured to inspect each of the web pages completely as requested from the server. Once again, Snort was run and the user processing time was noted. The difference in times represents the pure pattern matching time required to pattern match the extra data. This test did not measure absolute performance. It does show the relative performance differences of each search engine in processing the additional data.

This test was run with each version of the Aho-Corasick algorithm and the Wu-Manber algorithm. The Wu-Manber algorithm in Snort is generally a very fast algorithm on average, but does not have a good worst-case scenario. It is included here to show the relative merits of its good average case performance compared to the Aho-Corasick algorithms.

The time differences, computed speed, and performance increase over the standard Aho-Corasick is shown in table 2 and chart 14. These tests used a 16 bit state size, were run on a system with dual 2.4 GHz Xeon cpus, and 2 G-bytes of RAM.

Snort Web Test

	Time Diff	Speed Increase	Performance Increase	Memory M-bytes
AC_STD	12.4	0%	0%	56
AC_OPT	8.5	31%	46%	28
AC_BAND	10.3	17%	20%	14
Wu-Manber	12.1	2%	2%	26

Table2

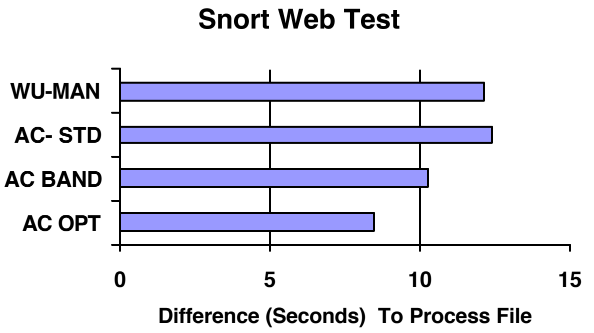

Chart 14

The optimized and banded Aho-Corasick algorithms are significantly faster than the standard Aho-Corasick and the Wu-Manber algorithm. The web rules in Snort contain many small two byte patterns. This prevents the Wu-Manber algorithm from benefiting from a bad character shift strategy.

The optimized algorithm is 31 percent (8.5/12.4) faster, this means the engine can perform 46 percent (12.4/8.5) more pattern matching in the same time, and requires one-half the memory of original algorithm. The banded algorithm is 17 percent faster and yields a 20 percent performance gain, and uses about one-fourth of the memory of the standard Aho-Corasick algorithm.

IX. Summary

The Aho-Corasick search algorithm has been studied extensively since its inception in 1975. Intrusion Detection Systems operate in a real-time domain where searches are serial, patterns are often very small, pattern group sizes are often over a thousand patterns, searches occur up to 200,000 times a second, and attackers exploit the worst-case properties of these algorithms.

The Aho-Corasick algorithm when implemented using a DFA provides a strong basis for dealing with Intrusion Detection pattern searches. The enhancements made in this paper are designed to maintain the best properties of the Aho-Corasick algorithm and improve its performance.

Sparse storage using banded-rows was introduced to reduce cache misses and improve performance. This storage format requires a bounds check for each state transition, this adds a fixed cost test. As pattern groups grow larger the fixed cost of bounds checking is mitigated by better caching behavior as shown in the dictionary tests and the Snort test.

The tests used demonstrate that there are significant differences in the quality of the machine code generated by different compilers for the same source code. In spite of advances in modern code optimizers, an algorithm's performance is still very dependent on the developer crafting source code to best match the hardware's processing characteristics.

Dictionary testing shows the optimized Aho-Corasick implementation is 1.5 to 2.5 times faster than the current version, depending on the compiler selected and pattern group size. Storage was reduced by as much as 93 percent. These tests results show that the optimized sparse storage version demonstrates better performance for large pattern groups than the optimized full matrix version.

Snort testing shows the optimized version is 31 percent faster, yielding 46 percent more pattern matching capacity, and uses one-half the memory of the author's original version. The optimized sparse storage version is 17 percent faster than the original, yielding 20 percent more pattern matching capacity, and uses less than one-fourth the memory of the original version.

This paper has presented an updated version of the author's original Aho-Corasick algorithm and demonstrated that it provides a significant improvement in pattern matching performance when used with an Intrusion Detection System, and in general dictionary testing.

The optimized Aho-Corasick source code can be found in the Snort source code at www.snort.org or along with this paper at www.idsresearch.org.

ACKNOWLEDGEMENTS

Thanks to Dan Roelker at Sourcefire for the many discussions on pattern matching and Intrusion Detection Systems, and your significant editorial suggestions.

Thanks to Marty Roesch and Sourcefire for supporting this research.

REFERENCES

1. Alfred V. Aho and Margaret J. Corasik. Efficient String Matching: An Aid to Bibliographic. Search: Bell Labs, Communications of the ACM Jun 1975 Volume 18 Number 6.
2. Marc Norton and Daniel Roelker. Snort Multi-Rule Inspection Engine. www.idsreseach.org/papers.html
3. Sun Wu and Udi Manber. A Fast Algorithm For Multi-Pattern Searching. May 1994.
4. Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman. Compilers Principles, Techniques, and Tools. Addison Wesley 1986 ISBN 0-201-10088-6, I.S. Duff, A.M. Erishman, J.K. Reid. Direct Methods for Sparse Matrices. Oxford University Press 1986, ISBN 0-19-85342103
5. Gonzalo Navarro and Mathieu Raffinot. Flexible Pattern Matching in Strings. Cambridge University Press 2002, ISBN 0-521-81307-7
6. Alberto Apostolico and Zvi Galil. Pattern Matching Algorithms. Oxford University Press, 1997, ISBN 0-19-511367-5
7. Dan Gusfield. Algorithms on Strings, Trees, and Sequences. Cambride University Press 1997, ISBN 0-521-58519-8
8. C. Jason Coit, Stuart Staniford, Joseph McAlerney. Towards faster String Matching for Intrusion Detection or Exceeding the Speed of Snort. Silicon Defense.
9. Mike Fisk, George Varghese. Fast Content-Based Packet Handling for Intrusion Detection. 2002.
10. Stephen Gossen, Neil Jones, Neil McCurdy, Ryan Persaud. Pattern Matching in SNORT. 2002
11. Nathan Tuck, Timothy Sherwood, Brad Calder, George Varghese. Deterministic Memory Efficient String matching Algorithms for Intrusion Detection. 2003.
12. Robert S. Boyer., J. Strother Moore. A fast string-searching algorithm. Commun. ACM, 20, #10, Oct 1997, pp.776-772
13. R.N.Horspool. Practical fast searching in strings. Softw Pract Exper., 10, #3, 1980, pp 501-506.
14. Beate Commentz-Walter. A string-matching algorithm fast on the average. Automata, Languages, and Programming, Levture notes in Computer Science #71, ed. By H.A. Maurer, Springer-Verlag, Berlin, 1979, pp 118-132
15. Donald E. Knuth, James H. Morris Jr, Vaugh R. Pratt. Fast pattern matching in strings. SIAM J. Comput., 6, #2, June 1997, pp. 323-350
16. I.S.Duff, A.M.Erisman, J.K.Reid. Direct methods for Sparse Matrices, Oxford University Press, 1986, ISBN 0-19-853421-3
17. Sparse matrix related resources can be found at www.vlsicad.cs.ucla.edu/sparse.html .
18. Andrew Binstock, John Rex. Practical Algorithms for Programmers, Addison-Wesley, 1995. ISBN 0-201-63208-X
19. OSEC - http://osec.neohapsis.com
20. Johnson, S.C. [1975]. Yacc – Yet another compiler compiler, Computing science technology report 32, AT&T Bell Laboratories, Murray Hill, N.J.
21. Tarjan, R.E., and Yao, A.C.-C. Storing a sparse table. Commun. ACM 21, 11 (Nov 1979)