Towards Faster String Matching for Intrusion Detection
or Exceeding the Speed of Snort

C. Jason Coit
Silicon Defense
jasonc@silicondefense.com

Stuart Staniford
Silicon Defense
stuart@silicondefense.com

Joseph McAlerney
Silicon Defense
joey@silicondefense.com

Abstract

Network Intrusion Detection Systems (NIDS) often rely on exact string matching techniques. Depending on the choice of algorithm, implementation, and the frequency with which it is applied, this pattern matching may become a performance bottleneck. To keep up with increasing network speeds and traffic, NIDS can take advantage of advanced string matching algorithms. In this paper we describe the effectiveness of a significantly faster approach to pattern matching in the open source NIDS Snort.

1. Introduction

Network Intrusion Detection systems (NIDS) have relied on exact string matching from the very earliest days of the field; the UC Davis Network Security Monitor [1,2] made extensive use of string matching. More recently, a number of commercial NIDS including Dragon [3] rely heavily on exact string matching strategies, as do several free NIDS; Snort [4,5], and Bro [6] both have rule options to do exact matches of content strings in packets.

Exact string matching is somewhat problematic as a NIDS strategy all by itself, given the broad range of tactics available to an attacker for de-synching the IDS [6,7,8]. Nonetheless, its popularity suggests that it is worth studying the most efficient way to carry out the task.

In this paper, we look at the way the popular Snort NIDS performs string matching. To the best of our knowledge, other NIDS's that rely heavily on exact string matching use similar strategies; however this is often a trade secret for commercial systems so we cannot be sure. We show that there is an algorithm that is significantly faster in practice, and that can be expected to scale better as more rules requiring content searches are added to the Snort ruleset. The algorithm we have used is a minor variation of algorithms known in the string matching research community [9,10,11], but not widely used. We have implemented this algorithm, and we provide results demonstrating the improvements in speed. Since Snort is widely used, and since, like all network intrusion detection systems, Snort cannot keep up with heavily loaded fast networks, this is of immediate practical value.

The basic string matching task that must be performed by a NIDS is to match a number of patterns drawn from the NIDS rules to each packet or reconstructed TCP stream that the NIDS is analyzing. In Snort, the total number of rules available has become quite large, and continues to grow rapidly. As of 10/10/2000 there were 854 rules included in the "10102kany.rules" ruleset file [5]. 68 of these rules did not require content matching while 786 relied on content matching to identify harmful packets. Thus, even though not every pattern string is applied to every stream, there are a large number of patterns being applied to some streams. For example, in traffic inbound to a web server, Snort v 1.6.3 with the snort.org ruleset, "10102kany.rules", checks up to 315 pattern strings against each packet. At the moment, it checks each pattern in turn using the Boyer-Moore algorithm. Since the patterns often have something in common, it seemed likely that there is considerable scope for efficiency improvements here, and so it has proved.

In the remainder of this paper, we first describe in detail the way Snort organizes the process of checking the content in each packet. Then in section 3 we describe the faster algorithm that we have applied, together with various implementation details, and minor changes in the operation of Snort that result. In section 4 we record our experimental results from applying the original Snort v 1.6.3 to actual network traffic, together with our improved algorithm. Finally, we conclude and suggest directions for further research in section 5.

2. Description of Existing Algorithm

Snort uses its ruleset to create a two-dimensional linked list structure consisting of Rule Tree Nodes (RTN's) and Option Tree Nodes (OTN's) [5]. The RTN's hold many common properties that must be included in each rule, such as the source and destination addresses, source and destination ports, and protocol type (TCP, ICMP, UDP). The OTN's hold the information for the various options that can be added to

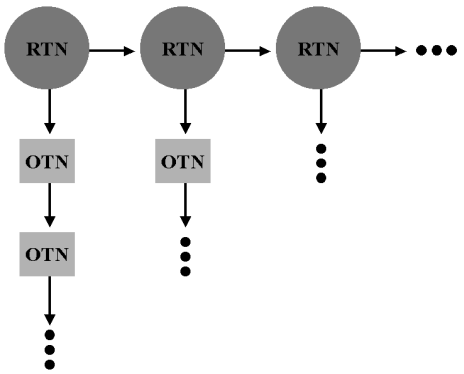


Figure 1. A generic example of the RTN and OTN linked list structure.

each rule, such as TCP flags, ICMP codes and types, packet payload size and a major bottleneck for efficiency, packet content. These structures are organized into chains which can be conceptualized with the RTN's strung from left to right as chain headers and the OTN's hanging down from the individual RTN's that each is associated with (see Figure 1).

In Snort 1.6.3, when packets are being examined against a given ruleset, the packet is first compared along the RTN list from left to right until the packet matches a particular RTN. Only if such a match occurs is the packet then compared down the OTN list of the matching RTN. Snort proceeds to check the packet against each OTN in the chain until a match is found. The options held in each OTN are checked using plugin functions that are called along a linked list as well. When an option matches the packet, the current plugin function calls the next option checking function in the list. If any of the option checks fail, the packet is then checked against the next OTN in the list. For the sake of efficiency, Snort currently checks all other options before checking the packet for content matches on the assumption that pattern matching is most time intensive option to check.

If a content check is required, Snort uses a Boyer-Moore pattern matching algorithm to check the content string held in the OTN against the entire packet payload. If no match exists, Snort will proceed to the next OTN in the list, which could have all options identical to the previous OTN save for a slightly different content string. For example, one OTN represents the rule looking for the content **scripts/CGImail.exe** while the next OTN requires a content search for the string **scripts/fpcount.exe**. So if an exhaustive search of the entire packet content does not reveal **scripts** to exist in the packet, the second search is guaranteed to fail yet is performed regardless.

Boyer-Moore is a rather famous pattern matching algorithm that is quite fast in practice. It uses heuristics to reduce the number of comparisons needed to determine if a given text string matches a particular pattern, i.e. it uses knowledge of the keyword to search for to skip over unnecessary comparisons against the text being searched. The algorithm typically aligns the text and the keyword to search for so that the keyword can be checked from left to right along the text string beginning with the last character of the keyword and ending with the first.

The first heuristic it uses is commonly referred to as a *bad character* heuristic. If a character is seen that does not exist in the keyword to search for, the keyword can be shifted forward N characters where N is the length of the given keyword (see Figure 2). The second heuristic uses knowledge of *repeated substrings* in the keyword. Thus if a mismatch occurs and repeated patterns exist in a given keyword, it is able to shift the keyword to the next occurrence of a substring that matches what has already been successfully matched (see Figure 3).

Boyer-Moore was designed for exact string matching of many strings against a single keyword. While the algorithm is quite efficient at performing this operation, its current implementation in Snort does not take advantage of the similarities of the multiple keywords that are held in the OTN's. We noticed the similar prefixes of many of the rules in a commonly used snort.org ruleset library [5]. We theorized that if we could reduce the redundant pattern matches by using some other pattern matching algorithm, Snort would perform much faster. In the next section we will discuss the concept and implementation of an algorithm that uses elements of Boyer-Moore to search for multiple patterns at the same time.

Before Shift:

```
pattern ->    one plus two
                      *
   text ->    two plus three equals five
```

After Shift:

```
 pattern ->                one plus two
                                    *
    text ->    two plus three equals five
```

The characters are examined starting at * and compared right to the left while the whole pattern moves along the text to search from left to right. The first comparison fails on the character r. Since no r exists in the pattern, it can be shifted by 12 characters as shown above. The next comparison begins at the second *.

Figure 2. Standard Boyer-Moore bad character shift

Before Shift:

```
pattern ->   two plus two
                        *
   text ->   count to two hundred thirty
```

After Shift:

```
pattern ->           two plus two
                                 *
text ->     count to two hundred thirty
```

The comparison begins at * and continues examining characters right to left. It fails on the second o read from the text. Since two exists as a repeated substring in the pattern, the pattern can be shifted 9 characters to line the two of that pattern up with the matching part of the text string as shown above. The next comparison begins at the second *.

Figure 3. Standard Boyer-Moore repeated substring shift

3. Description/Implementation

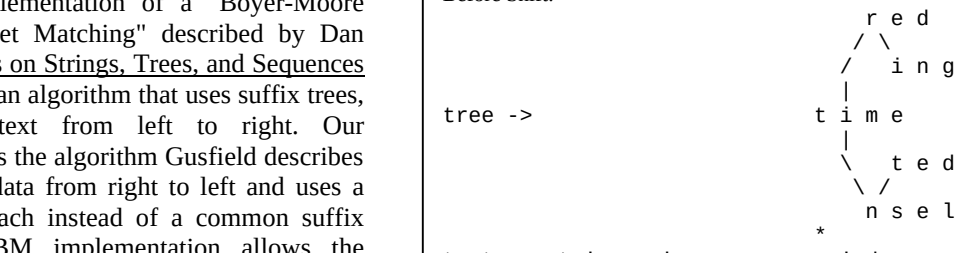
The algorithm is a Boyer-Moore like algorithm applied to a set of keywords held in an Aho-Corassick like keyword tree that overlays common prefixes of the keywords. We designate it as the AC_BM algorithm in Snort. Though we refer to the algorithm as AC_BM, it is essentially an implementation of a "Boyer-Moore Approach to Exact Set Matching" described by Dan Gusfield in Algorithms on Strings, Trees, and Sequences [9]. Gusfield outlines an algorithm that uses suffix trees, and examines the text from left to right. Our implementation mirrors the algorithm Gusfield describes - it examines packet data from right to left and uses a common prefix approach instead of a common suffix approach. The AC_BM implementation allows the various rules that require content searches to be placed in a tree that can be searched using elements of Boyer-Moore.

Like the Boyer-Moore approach previously explained, the algorithm we implemented aligns the text to search with the patterns to search for and performs shifts to eliminate unnecessary comparisons. The keyword tree moves from the right end of the packet payload to the left while the character comparisons are performed from left to right once the keyword tree is in position. The algorithm relies on derivatives of the same heuristics used by standard Boyer-Moore. Instead of sliding a single pattern along the text string to be searched, the AC_BM algorithm slides a tree of patterns along using its *bad character* and *good prefix* shifts. The *bad character* shift is similar to the first heuristic of Boyer-Moore; if a mismatch occurs, it recommends shifting the tree to line up with the next occurrence of the character in some other keyword in the pattern tree. If the character does not exist in any keyword past its current depth, it recommends a shift of the length of the smallest pattern in the tree. The *good prefix* shift recommends a shift to the next occurrence of a complete prefix that has already been seen as a substring of another pattern, or shift to the next occurrence of some prefix of the correctly matched text as the suffix of another pattern in the tree. We always need to be sure not to shift farther than the length of the smallest pattern in the tree so we never skip past a matching pattern that is closer than our heuristic might suggest (see Figure 4 & Figure 5).

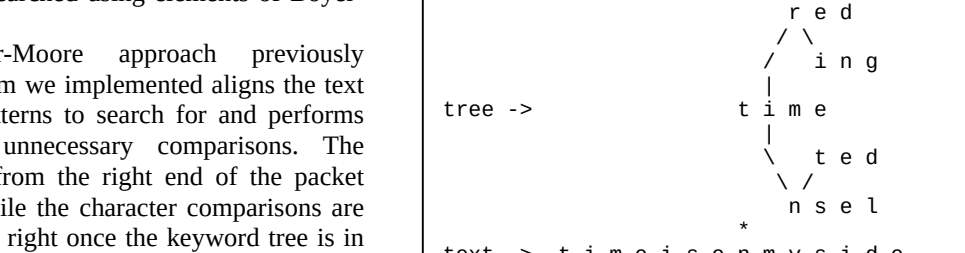
The AC_BM algorithm allows for the content searching of many OTN's to be combined into a single tree that can be searched quite quickly and greatly reduce the many unnecessary comparisons that Snort currently performs when searching its ruleset. One tree is used for each RTN. The keyword tree for a given RTN holds the content strings for all of the OTN's that require a pattern match for that RTN. The keyword tree information is attached, as a pointer to a PatternTreeData struct, to each RTN. If the OTN List for a given RTN does not contain any content rules, this pointer is set to NULL. The patterns for the keyword tree are collected during the construction of the RTN and OTN lists then

Patterns: time, tired, tiring, tinted, tinsel

Before Shift:



After Shift:

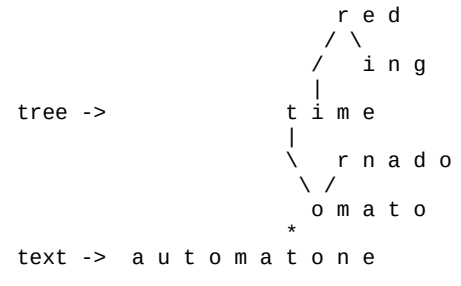


The comparison begins by aligning the smallest pattern in the tree, time, with the last four letters of the string to search. Then the characters are checked from left to right starting from * and in this case failing on char s. The next s occurs in the pattern *tinsel*, which calls for a shift of 3 by the adapted bad character rule.

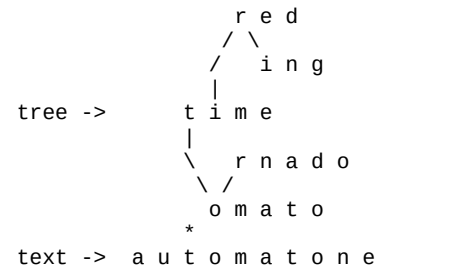
Figure 4. Modified bad character shift

Patterns: time, tired, tiring, tomato, tornado

Before Shift:



After Shift:



The comparison begins by aligning the smallest pattern in the tree, *time*, with the last four letters of the string to search. Then the characters are checked from left to right starting from * and failing on character *n*. Of the characters successfully matched, to appears as a suffix of a pattern, *tomato*, in the tree. This calls for a shift of 4 characters to align last two letters of the pattern with the to already seen.

Figure 5. Modified repeated substring (good prefix) shift

the trees are preprocessed to set up the proper shift information. Since the keyword tree groups many rules together and overlaps the matching prefixes, the various rule options are meaningless to the keyword tree. Thus checking the rule's individual non-content options posed a problem. To allow most of the rule options available to still be used with the AC_BM algorithm, we needed to change how Snort performed options checking without drastically changing the structure or organization of the RTN and OTN data structures. We decided the best way to enable the other rule options and not significantly reorganize how the RTN and OTN lists were structured was to separate the content from non-content rules and handle option checking for each type of rule separately.

During preprocessing we organize the OTN list so that rules without content are checked before rules with content. This is accomplished by adding each new OTN immediately before the first OTN encountered that requires a content search. Whether or not the new OTN requires a content search, it will preserve the separation of non-content and content rules. Since content searching should be the most computationally intensive of the option checking steps, we quickly check all the rules that do not have content first, then move on to the content searching. Thus we proceed down the OTN list as usual and check the OTN's using their aforementioned linked list of option checking functions until a content rule is encountered. Once a content rule is seen, the pattern matching algorithm is called to search the packet's content. If the packet matches a pattern in the keyword tree, the other options are checked, if these are successful, the pattern matching algorithm reports a success and the standard procedures of Snort are undertaken. The content matching for a rule is done prior to the other option checking since many of the rules have similar options set, such as checking for the TCP flags Push and Ack.

The only option in Snort 1.6.3 that our implementation does not support is case sensitive searching. We decided not to implement both case sensitive and non case sensitive searching since this would require two keyword trees - one for case and one for non case. Since the vast majority of the rules in the snort.org ruleset are not case sensitive, we chose to support only these rules to keep the implementation relatively simple.

Multiple content rules, rules that require two or more patterns to be matched in a packet, are handled by searching for the first pattern using our keyword tree, and any additional patterns using Snort's implementation of a standard Boyer-Moore algorithm. This is done to allow for multiple content rules to be supported in our updated version of Snort while still keeping the implementation relatively simple. Our goal was to develop a "quick and dirty", proof of concept implementation to test if a new approach to pattern matching would be at all advantageous. Thus we opted to support those rules that required more than one content search by using our method followed by the standard Boyer-Moore search already implemented in Snort. This also fit well into the option checking functions that Snort uses since any additional content matches are checked after the initial match when traversing the op_func chain previously mentioned.

It is worthwhile to note that the AC_BM algorithm is highly dependent on the length of the shortest pattern being search for, since we can never shift the keyword tree more than this value. So the maximum shift value for a keyword tree under any particular RTN is directly determined by the length of the shortest pattern in that tree. The effectiveness of the standard Boyer-Moore's approach is also limited by the pattern's size when checking any particular pattern, yet one short pattern under a given RTN does not affect the maximum safe distance to shift for all other patterns under that same RTN.

Since we did not support case sensitive searching in our implementation, we disabled the case sensitive

searches during the testing of both algorithms. In the next section we will discuss the results of these tests and the potential differences our implementation could create in Snort's behavior.

4. Results

To examine how the two algorithms affect the performance of Snort, we used actual network traffic from the Capture the Flag game at Defcon 8, which we acquired from www.shmoo.org [12]. We then used the Linux time command to calculate the amount of time in seconds it took both versions of Snort to run the data sets collected. Then we checked the output from each version to ensure they were identical. To easily perform these tests and ensure that each algorithm performed only non-case sensitive tests, we modified Snort v 1.6.3 slightly. We included a command line argument to switch between the algorithms and disabled case sensitive searching in the standard algorithm. We ran Snort on a half-hour data set (77 MB) with AC_BM and standard Boyer-Moore algorithms. We used the 10102kany.rules ruleset from snort.org [5], which at the time held 854 rules, 68 non-content and 786 content.

We hypothesized that our algorithm would scale better as the number of content rules increased than the standard Snort approach. Our reasoning was based on how Snort currently performs content searching: by traversing down its RTN/OTN list and repeated applying Boyer-Moore. Thus the running time for standard Snort to match a packet should increase linearly with the number of content rules in the ruleset used. Since our implementation groups the content rules by taking advantage of the many similar prefixes, and applies Boyer-Moore tactics for skipping over many unnecessary comparisons, its running time to match a packet should increase far less drastically as the number of content rules increase. In order to test this we parsed the ruleset to separate the non-content and content rules.

We ran one series of tests including the 68 non-content rules and another series of tests on the content rules only. The timing results for the first series of tests are represented in Table 1. We performed the first test of this series using the DEFCON 8 data on the 68 non-content rules. Each subsequent test incremented the count of content rules by 200 until we exhausted our supply of 786 content rules. While our implementation consistently ran in less time than the original snort, from 1.02 times faster in the first case to 1.18 times faster in the last test, this is not the margin of improvement we suspected. Noting that Amdahl's Law [13] states that the overall speedup is gained only for the percentage of time our improvement is used, we decided to test the same data set using only content rules in order determine the speedup our algorithm could provide during intensive content matching.

In the second run of tests we began with 200 content rules and incremented each subsequent test in the same fashion as the previous series. The results of these tests demonstrate that our implementation can perform quite well on repetitive pattern matching compared to the standard Boyer-Moore approach. While these tests are skewed by the elimination of the non-content rules, they still illustrate that a keyword tree approach to Boyer-Moore pattern matching is superior to repetitively applying a single pattern Boyer-Moore algorithm, from 1.31 times faster in the first test to 3.32 times faster in the last test. These results are represented in Table 2 below.

Table 3 and Table 4 represent the memory usage of both algorithms during our first and second series of tests respectively. As expected, the tradeoff for higher speed is higher memory use. Our algorithm uses on roughly 3 times the memory of the standard version. This could most likely be improved by using more efficient methods to traverse the keyword tree. Currently the transition matrix for each keyword node is a 256 char array. Assuming that each node does not

Table 1. Time vs. number of rules (non-content and content)

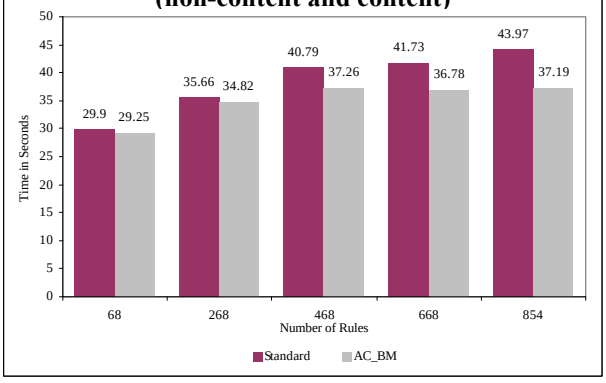


Table 2. Time vs. number of content rules

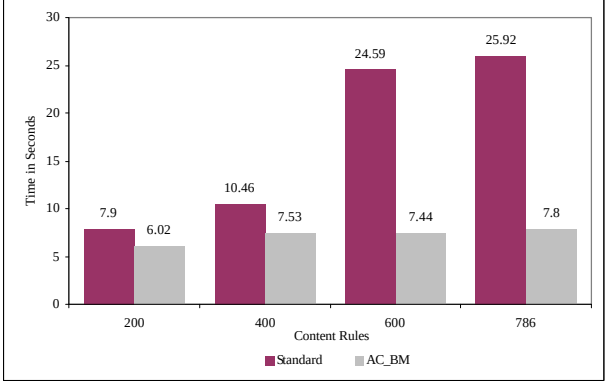


Table 3. Memory vs. number of rules (non-content and content)

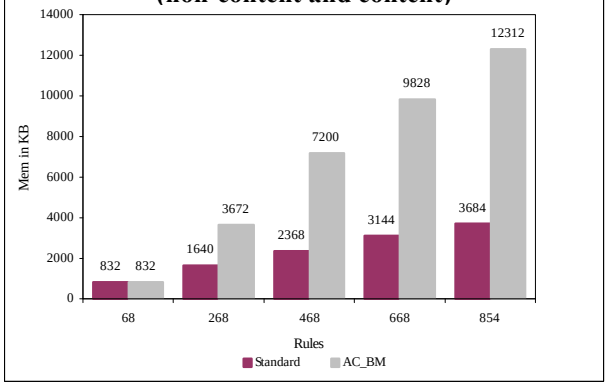
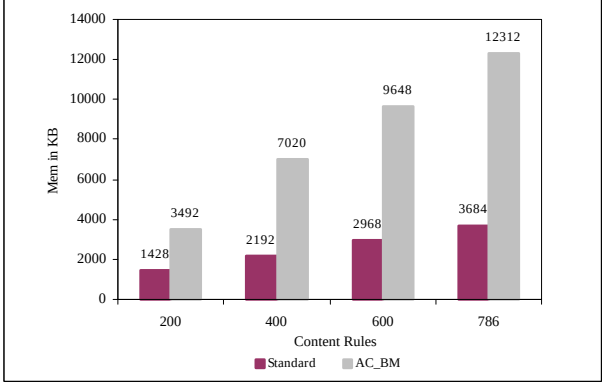


Table 4. Memory vs. number of content rules



have most of these transitions in use, this uses much more memory than necessary. The current transition matrix of each node could be replaced with a structure such as a splay tree to reduce the space needed for each node for an increase in running time. For each test we compared the alert files generated to determine if our implementation produced the same output as the original. There were several alerts that exhibited one of the behavior changes we can expect from our version of Snort.

Our data set produced inconsistent alerts due to the direction each algorithm examines packet payload. Snort's standard Boyer-Moore algorithm examines packet data from the left end of packet data to the right. Our implementation examines packet data from the right end of a packet to the left end. A packet containing the a content string for one rule beginning at its right end and another beginning at its left end could result in different rules being reported by the two algorithms. For example suppose there is a Rule_A that contains a content search for `toomanyexamples` and another almost identical rule that had all of the same options as Rule_A except it checked for the content `tcpforever`. Now suppose there is a packet that has a payload of `toomanyexamplespapertcpforever`. Thus the current version of Snort would report Rule_A while our implementation would report Rule_B for the same packet.

The inconsistencies in our alert files were produced by the following two rules:

alert icmp any any -> any any (msg:"PING *NIX Type";content:"|10111213141516171819 1a1b1c1d1e 1f|";itype:8;depth:"32";)

alert icmp any any -> any any (msg:"IDS152 – PING BSD"; content: "|08 09 0a 0b 0c 0d 0e 0f 10 11 12 13 14 15 16 17|"; itype: 8; depth: "32";)

To test whether these rules would produce identical alerts, we ran the dataset using each rule individually. Both algorithms generated the same alerts.

Another way that the two algorithms could result in different results is that rules attempting to use ordering and generalities to sift packets would always flag the most general rule using our new implementation. For example, suppose there are two rules Last_1 and Last_2. Let Last_1 and Last_2 have all options identical except for their contents to search for. Suppose Last_1 requires a search for `lastexample` and Last_2 requires a search for `lastexamplereallynomore`. Now if they are ordered in the rule set as their names suggest (Last_1, then Last_2), and there is a packet that contains `lastexamplereallynomore`, then the current version of Snort would report on Last_2 while our implementation would report on Last_1. This is due to the way the patterns are stored in the keyword tree. The prefixes are overlapped so that the rules lose their ordering they had in the rule set. The algorithm will report success when the first pattern that matches is found in the tree. Thus the less specific rules will always be reported over the longer more specific rules.

5. Conclusion

We discussed the importance of pattern matching for Intrusion Detection Systems (IDS) and discussed the approach the open-source IDS Snort uses to match packet content. We verified that a "Boyer-Moore approach to exact set-matching" [9] works well in practice to significantly reduce the computation time needed to match packet content to our ever-increasing ruleset [5]. We demonstrated that our version of Snort can operate 1.02 times up to 3.32 times as fast as the current version depending on the number and type of content rules used while noting the day to day speedup will depend on the network traffic and ruleset used. The

higher the percentage of content matching performed, the more speedup our algorithm will contribute. The cost of this speed up is increased memory use (3 times as much as original). Our implementation was shown to scale better than the standard version of Snort as the number of content patterns in the ruleset increases.

Future work could include a more thorough implementation of our algorithm into Snort. The current RTN/OTN structure of Snort does not lend itself well to our approach to pattern matching. To allow the AC_BM implementation to perform better, it might be beneficial to restructure the RTN/OTN list to further group similar non-content options, thus eliminating redundant option checking. Another direction to pursue would implement both suffix and prefix tree approaches and then dynamically choosing which to use for each RTN. In this fashion the content rules under a specific RTN would be grouped to take advantage of either common suffixes or prefixes depending on which gives the greatest value. While the complexity of Snort would most likely increase, so would its effectiveness and efficiency in pattern matching.

References:

[1] L.T. Habergeon, G.V. Dias, K.N. Levitt, B. Mukherjee, (with J. Wood, D.Wolber), "A Network Security Monitor". Proceedings of the 1990 IEEE Symposium on Research in Security and Privacy. Oakland, CA, 7-9 May 1990, pp. 296-304.

[2] L.T. Heberlein , "Network Security Monitor (NSM) - Final Report". Lawrence Livermore National Laboratory project deliverable, http://seclab.ucdavis.edu/papers/NSM-final.pdf.

[3] http://www.securitywizards.com

[4] Martin Roesch, "Snort - Lightweight Intrusion Detection for Networks" USENIX LISA Conference November 1999.

[5] http://www.snort.org/

[6] Paxson, V., Bro: A System for Detecting Network Intruders in Real-Time. Proceedings of the 7th USENIX Security Symposium, San Antonio, TX, January 1998.

[7] T. Ptacek and T. Newsham, "*Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection*," Secure Networks, Inc., http://www.aciri.org/vern/Ptacek-NewshamEvasion -98.ps, Jan. 1998.

[8] Greg Hoglund and Jon Gary, "Multiple Levels of De-synchronization and other Concerns with Testing an IDS System", SecurityFocus.com, http://www.securityfocus.com /focus/ids/articles/desynch.html, August 2000.

[9] Dan Gusfield , *Algorithms on Strings, Trees, and Sequences:Computer Science and Computational Biology,* University of California Press, CA, 1997.

[10] A. Aho and M. Corasick. "Efficient string matching: an aid to biliographic search", *Comm. ACM.* 18:333-40, 1975.

[11] B. Commentz-Walter, "A string matching algorithm fast on average", *Proc. Of the 6th Int. Colloq. On Automata, Languages, and Programming*, pages 118-32, 1979.

[12] http:// www.shmoo.org/

[13] John L. Hennessy and David A. Patterson, *Computer Architecture: A Quantitative Approach Second Edition*, Morgan Kaufmann Publishers, CA, 1996.