Deterministic Memory-Efficient String Matching Algorithms for Intrusion Detection

Nathan Tuck[†] Timothy Sherwood[‡] Brad Calder[†] George Varghese[†]

[†]Department of Computer Science and Engineering, University of California, San Diego
[‡]Department of Computer Science, University of California, Santa Barbara

Abstract

Intrusion Detection Systems (IDSs) have become widely recognized as powerful tools for identifying, deterring and deflecting malicious attacks over the network. Essential to almost every intrusion detection system is the ability to search through packets and identify content that matches known attacks. Space and time efficient string matching algorithms are therefore important for identifying these packets at line rate.

In this paper we examine string matching algorithms and their use for Intrusion Detection. In particular, we focus our efforts on providing worst-case performance that is amenable to hardware implementation. We contribute modifications to the Aho-Corasick string-matching algorithm that drastically reduce the amount of memory required and improve its performance on hardware implementations. We also show that these modifications do not drastically affect software performance on commodity processors, and therefore may be worth considering in these cases as well.

Keywords: System Design, Network Algorithms

I. Introduction

With each passing day there is more critical data accessible in some form over the network. Any publicly accessible system on the Internet today will be rapidly subjected to break-in attempts. These attacks can range from email viruses, to corporate espionage, to general destruction of data, to attacks that hijack servers from which to spread additional attacks. Even when a system cannot be directly broken into, denial of service attacks can be just as harmful to individuals, and can cause nearly equal damage to the reputations of companies that provide services over the Internet. Because of the increasing stakes held by the various users of the internet, there has been widespread interest in combating these attacks at every level, from end hosts and network taps to edge and core routers.

Intrusion Detection Systems (or IDSs) are emerging as one of the most promising ways of providing protection to systems on the network. The IDS market has been estimated at $100 million by the Aberdeen Group, with expectations that it will double in 2004 and keep growing in future years. By automatically monitoring network traffic in real time, intrusion detection systems can alert administrators of suspicious activities, keep logs to aid in forensics, and assist in the detection of new worms and denial of service attacks.

As with firewalls, intrusion detection systems are growing in popularity because they provide a site resilience to attacks *without* modifying end-node software. While firewalls only limit entry to a network based on packet headers, intrusion detection systems go beyond this by identifying possible attacks that use valid packet headers that pass through firewalls. Intrusion detection systems gain this capability by searching both packet headers and payloads to identify attack signatures.

To define suspicious activities, an IDS makes use of a set of rules which are applied to matching packets. A rule consists at minimum of a type of packet to search, a string of content to match, a location where that string is to be searched for, and an associated action to take if all the conditions of the rule are met. An example rule might match packets that look like a known buffer overflow exploit in a web server; the corresponding action might be to log the packet information and alert the administrator.

Because of the utility of IDSs they are beginning to be deployed in a wide range of operating environments. End-hosts use them to monitor and prevent attacks from incoming traffic. They can be found in network-tap devices that are inserted into key points of the network for diagnostic purposes. They will soon even find their way into edge and core routers to protect the network infrastructure from distributed attacks.

The challenge is that increasing line-rates and an explosion in the number of attacks mounted as well as plummet-

ing unit costs have made cost-effective deployment a serious issue. In addition, as IDSs move from end-hosts into edge and core routers, the needs placed on algorithms for intrusion detection will change. While common-case performance can be an acceptable metric for end-hosts that are based on commodity processors, in order to be successful inside the network infrastructure, algorithms must satisfy stringent worst-case performance bounds and tight constraints on memory.

At the heart of almost every modern intrusion detection system is a string matching algorithm. String matching is crucial because it allows detection systems to base their actions on the *content* that is actually flowing to a machine. From this sea of packets, the string identifies those packets that contain data matching the fingerprint of a known attack. Essentially, the string matching algorithm compares the set of strings in the rule-set to the data seen in the packets that flow across the network.

String matching is computationally intensive. For example, the string matching routines in Snort account for up to 70% of total execution time and 80% of instructions executed on real traces [2]. Because string matching dominates the performance in this and many other IDS, in this paper we concentrate our efforts on building smaller and faster string matching algorithms.

We present optimized techniques for matching large sets of strings in incoming packets in the context of network intrusion detection. Our optimizations draw upon parallels between the well-studied problem of IP lookup and the nascent problem of detecting suspicious strings in packets. We show that most of the memory used by modern string matching algorithms goes towards the storage of pointers, which is similar to IP lookup.

By formulating a novel compressed pointer methodology for string matching data structures, we can reduce the amount of memory required to be 2% of the original (from 53.1 MB down to 1.09 MB) for a current rule set used in a modern IDS. This result is important because we provide this compression while at the same time providing worst case performance guarantees for the string matching algorithm.

We present the results of our techniques as applied to the open-source IDS software *Snort* [14]. We characterize the properties of a real set of IDS string matching rules and examine both how the rules have changed over time, and the effect of those changes on the data structures used. These characteristics are then exploited to produce a new string matching technique within an actual implementation of *Snort*.

We examine the amount of memory saved by our string matching memory optimizations and the improvement in throughput that we obtain for both commodity hardware and for proposed next generation network processors. By addressing worst case performance in both the algorithms and architecture we ensure that it is impossible for an adversary to construct an attack based on flooding the IDS with packets that it performs poorly on. An important contribution of this work is the development of an algorithm that performs well, requires little memory, *and has useful bounds on worst case performance*.

The contributions of this paper can be summarized as:

- **Characterization:** We characterize the need for and use of string matching in intrusion detection systems, and show how certain properties of the data lend themselves well to optimizations somewhat similar to those applied to IP-lookup. We also characterize the growth and properties of the database of known attacks.
- **New Algorithms:** Based on these characterizations, we design two new string matching algorithms that can reduce the memory usage to as low as 2% of that required by existing algorithms while maintaining bounded worst case performance.
- **Evaluation:** We evaluate these new algorithms in two operational contexts, first by examining worst-case performance of hardware implementations and secondly in a commonly used intrusion detection system, *Snort*, running on a commodity processor for an example trace. We show hardware performance more than 30% greater that other algorithms and only slightly degraded software performance.

We begin by characterizing the place of string matching in intrusion detection systems such as *Snort* and discuss relevant prior work in string matching algorithms in Section II. We then discuss our proposed optimizations based on these observations in Section III. A detailed evaluation of the results of our techniques can be found in Section IV. Our contributions are summarized in Section V.

II. String Matching For Intrusion Detection

In the Introduction we motivated the need for string matching in Intrusion Detection Systems. In this section we further demonstrate how string matching is used in an actual intrusion detection system. We also examine the state of the art in string matching as it relates to intrusion detection, and note some interesting parallels between the problem of string matching and the problem of IP-lookup.

A. Quantifying the Use of String Matching

We asserted earlier that string matching is the most critical component of an Intrusion Detection System (IDS). It

is important to further examine exactly *how* the matching is being exercised. To do this analysis we use the freely available and widely used IDS tool, *Snort*.

1) Snort - An Intrusion Detection System: *Snort* uses a set of rules that are derived from known attacks or other suspicious behavior. The rules are generated manually by experts who extract relevant (presumably unusual) *content strings* from the payload and header of known attacks. If all the conditions of the rule are met (which include matching the string, its location within the packet, and several other possible conditions) then the action specified by the rule is applied. This action can include logging the packet, alerting a system administrator via email, ignoring the packet, or dynamically activating other rules.

The distribution of *Snort* includes a set of rules which cover known attacks such as the exploit that allowed CodeRed [5] to spread or buffer overflows in POP3 servers. Rules are usually added to *Snort* as new vulnerabilities are discovered. Each of these rules contains a content string, associated rules for its location, and the type of packet it can appear in. To the best of our knowledge, this default rule set is used in most production uses of *Snort* with minor modifications, as it represents best practices and knowledge of the internet community.

2) Scalability of the Intrusion Detection System Database: An Instruction Detection System (IDS) contains a set of rules with corresponding actions; the set of rules supported by the IDS is called a database. In order to understand the hardware issues behind building an IDS we need to first understand the scalability of the IDS database over time.

We begin by examining the actual data within the strings contained in the rules of a typical IDS. The current standard distribution of *Snort* comes with over 1500 rules enabled by default. Figure 1 shows a histogram of the number of bytes in the character portion of each unique rule in the default database.

Rules can match non-letter characters such as IP addresses; this partially explains the large number of 4-byte rules. As can be seen in Figure 1, the bulk of the rules have length on the order of 15 bytes but there is a large distribution above and below. We also see from this figure that there are many rules with very long lengths. This implies that it is beneficial to avoid any string-matching technique which has run-time proportional to the length of the rules in the database.

New attacks are being created all of the time, and as they are, new rules are being added to the *Snort* database to detect or combat them. Figure 2 shows how the size of the *Snort* rule database has grown over time. We characterize the size of the database in terms of the number

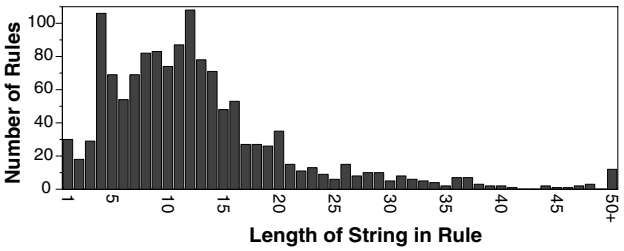

Fig. 1. Distribution of the lengths of the unique strings found in the default Snort database.

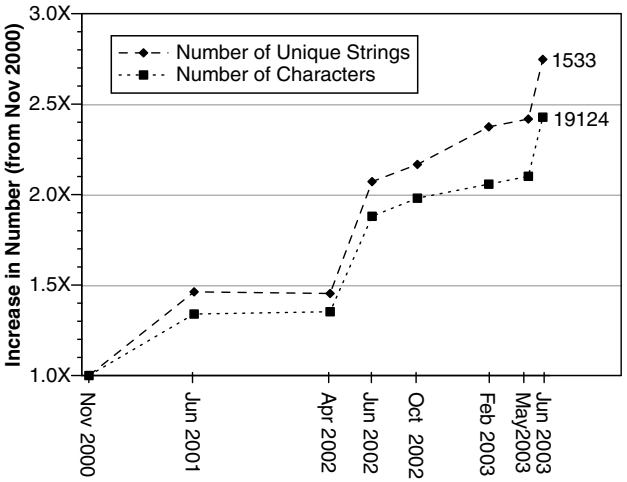

Fig. 2. The growth of the Snort rule database over the last three years.

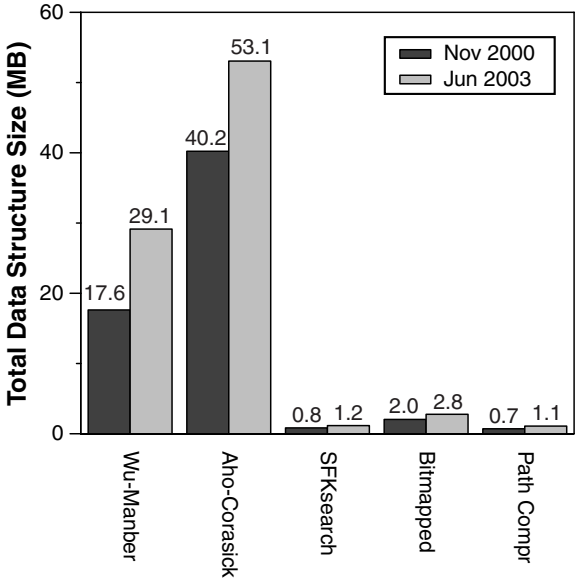

Fig. 3. Sizes of string matching data structures for known algorithms and our work.

of unique strings (multiple rules may use the same string, but we do not include these duplicates as they should have little effect on determining a string match), and the total number of bytes in unique strings. It can be seen that the number of rules and the number of bytes in these rules is increasing quite quickly. Over the last three years the number of rules has increased by over a factor of 2.75 while the total number of bytes in these rules has increased by almost a factor of 2.5.

This data leads to three conclusions. First, the simple technique of linearly searching through the set of rules is becoming increasingly infeasible. Second, the database is growing at a rate that is well within Moore's Law, which implies that the increase in the size of the rule database has thus far been compensated for by increased transistor count on chips and has thus gotten easier to implement in a single chip rather than harder. The third conclusion is that we need a technique with run-time performance, excluding data structure construction costs, that is independent of the size of the rule database. Luckily, such algorithms already exist today, one example being the Aho-Corasick algorithm [1]. We ignore construction costs because building the data structure in all of the algorithms we consider is quite fast and is only performed when loading a new ruleset, restarting the IDS, or for a very few dynamic rules.

B. State of the Art in String Matching

To understand the heart of a modern IDS system, we must now explain the core string matching algorithms. The first thing worth noting is that the relevant body of literature for this problem is the multi-pattern string matching problem, which is somewhat different from the single-pattern string matching solutions that many people are familiar with such as Boyer-Moore [3]. For single-pattern string matching, there is a large body of work in which a single string is to be searched for in the text. This may come up in word processing applications, e.g., in search-and-replace operations.

On the other hand, the multi-pattern string matching problem searches a body of text (in our case a packet stream) for a *set* of strings. One can trivially extend a single pattern string matching algorithm to be a multiple pattern string matching algorithm by applying the single pattern algorithm to the search text for each search pattern. Obviously this does not scale well to larger sets of strings to be matched. Instead, multi-pattern string matching algorithms generally preprocess the set of input strings, and then search all of them together over the body of text. Previous work in precise multi-pattern string matching includes Aho-Corasick [1], Commentz-Walter[6], Wu-Manber [21], and others.

There has also been even more recent work in imprecise string matching algorithms using hashing and signature-based techniques [13], [9]. Although these methods may meet the criteria of having deterministic execution time per packet, there is the problem that positive matches must be reverified using a precise string matching algorithm. Thus, the performance of the underlying precise matching algorithm is still important, albeit at a reduced level.

Imprecise string matching also introduces the possibility that certain innocent data streams may introduce a rate of sequential false positives that overwhelm the exact matching algorithm unless it is capable of processing at line rate. We do not address the open question of whether imprecise methods are completely appropriate for use in situations where worst-case performance is an important metric, but assert that in any case the underlying precise multi-pattern string matcher performance is still important.

We now describe some of the multi-pattern string matching algorithms that appear in the current version of *Snort* and extend this with some further discussion of other string matching algorithms.

1) Bad Character Heuristics: The bad character heuristic should be familiar to those who have seen Boyer-Moore string matching before. Given a single pattern of length n to match, one can look ahead in the input string by n characters. If the character at this position is not a character from our pattern, we can immediately move the search pointer ahead by $n + 1$ characters without examining the characters in between. If the character we look-ahead to does appear in the string, but is not the last character in the search string, we can skip ahead by the largest amount that ensures that we have not missed an instance of our pattern.

This bad character heuristic is adapted in a straightforward manner to most implementations of multi-pattern string matching algorithms. In order to make it work, one must be conservative. The algorithm can only look ahead by the length of the shortest pattern to be matched and the skip ahead values for any character must be the minimum of the skip ahead values for that character in any of the individual truncated patterns.

The basic problem with using the bad character heuristic as part of an algorithm in an IDS is that it yields sub-linear performance only in the average case, a fact that is easily exploitable by attackers. If the underlying pattern matching algorithm is not fast enough to keep up with line rate, it is quite easy for an adversary to hide his attack by first swamping the IDS with packets that defeat the

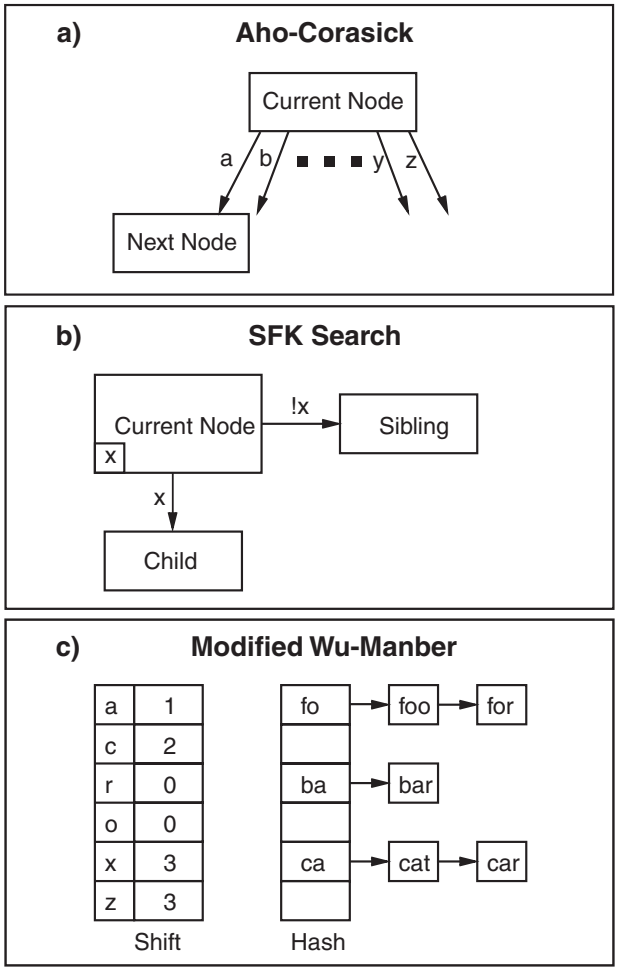

Fig. 4. High Level Data Structures for String Matching Algorithms

bad character heuristic. This is particularly true with the growing size of rule sets, as the adversary has an increasing language of packets to work with that both defeat the bad character heuristic and yet are perhaps valid enough not to be noticed as a deliberate attack.

We mention the bad character heuristic in its own section because it is a very common mechanism in most multi-pattern matching algorithms to achieve performance. Implementors should think very carefully about the extent to which they rely on a heuristic if it affects the security of their network.

2) Aho-Corasick: One of the earliest algorithms in precise multi-pattern string matching is due to Aho-Corasick [1], which is able to match strings in *worst-case* time linear in the size of the input. Aho-Corasick works by constructing a state machine from the strings to be matched. The state machine starts with an empty root node which is the default non-matching state. Each pattern to be matched adds states to the machine, starting at the root and going to the end of the pattern. The state machine is then traversed and failure pointers are added from each node to the longest prefix of that node which also leads to a valid node in the trie. We show a single node of the state machine in Figure 4 (a).

```
struct aho_state {
  struct aho_state * next_state[256];
  struct rule * rule_list;
};
```

Fig. 5. Base Optimized Aho-Corasick Data

```
struct aho_state {
  struct aho_state * next_state[256];
  struct rule * rule_list;
  struct aho_state * fail_ptr;
};
```

Fig. 6. Base Un-optimized Aho-Corasick Data

Beyond this basic notion, there are two choices for the algorithm. We can optimize the data structure further by using the failure pointers to precompute the next state for every character from every state in the machine (see Figure 5), or we can leave these transitions undefined and traverse the failure pointers at run-time (see Figure 6). If the data structure is optimized, then Aho-Corasick requires only a single memory reference (albeit a very wide memory reference) per character in the input. If the data structure is left unoptimized, one can show via amortized analysis that only two (again wide) memory references per character of input string are required to traverse the data structure.

The default implementation of Aho-Corasick in *Snort* uses the optimized data structure. We use the unoptimized data structure because the undefined pointers allow us significant opportunity for space optimizations. Watson [18], [19] previously showed that the unoptimized Aho-Corasick (AC-FAIL in his terminology) could be more space efficient than optimized Aho-Corasick, but he compressed his data into a tabular state machine which we believe is not compatible with our goal of an efficient hardware implementation that can be easily altered as rule sets change.

3) SFKSearch: SFKSearch is the algorithm used for low memory situations in *Snort*. The algorithm builds a trie, a node of which we show in Figure 4 (b). Each level in the trie is a sequential list of sibling nodes that contain a pointer to matching rules, a character that must be matched to traverse to their child node, and a pointer to the (next) sibling node. The algorithm uses a bad character shift table to advance through search text until it encounters a possible start of a match string, at which point it traverses the trie looking for matches. If there is a match

between the character in the current node and the current character in the packet, the algorithm follows the child pointer and increments the character packet pointer. Otherwise, it follows the sibling pointer until it reaches the end of the list, at which point it recognizes that no further matches are possible. In the case that matching fails, the algorithm backtracks to the point at which the match started, and now considers matches starting from the next character in the packet.

In the worst case, this algorithm can make $L \cdot P$ memory references where L is the length of the longest pattern string and P is the length of the packet. This worst-case performance is quite poor compared to algorithms like Aho-Corasick that take roughly P memory references. Indeed, with the default ruleset, we can find character combinations that would require the traversal of 20 nodes in the datastructure per character examined.

4) Wu-Manber: The Wu-Manber algorithm [21] was developed by Wu and Manber for use in agrep [20] and glimpse [12] – two text searching applications. It is also used with small modifications in *Snort* as the default pattern matching algorithm. The algorithm is again an algorithm designed for average-case rather than worst-case. The implementation in *Snort* is slightly less sophisticated that the original paper describing the algorithm.

The algorithm starts by precomputing two tables, a bad character shift table, and a hash table. When the bad character shift fails, the first two characters of the string are indexed into a hash table to find a list of pointers to possible matching patterns. These patterns are compared in order to find any matches and then the input is shifted ahead by one character and the process repeats.

Figure 4 (c) shows an excerpt of the modified Wu-Manber data structure, assuming that the strings to be searched for include *cat, car, bar foo,* and *for*. Note that character such as *x* and *z* which do not appear in any of the strings have the maximum shift values. In comparison, characters in the middle of the strings have reduced shift values, and those that are at the end of the strings, such as *r* and *o* must be resolved by indexing into the hash table.

Overall the algorithm achieves a worst case performance that is no better than naive string matching, but the average case performance is among the best of all multi-pattern string matching algorithms. In the worst case the algorithm requires for every character of input a memory access to the shift and hash table, followed by as many string compares as there are patterns to be matched (this can only happen if the hash fails). The algorithm is widely used and achieves very good average-case performance, but we feel that it is likely not a good candidate for hardware implementations that have more precise worst-case requirements.

C. Applying IP Lookup Techniques to String Matching

While on the surface the problems of IP-lookup and String Matching looks very distinct, as has been noted previously [15], [4] they are very much analogs of one another in that both are longest prefix matching problems. IP-lookup takes as input a set of patterns to match and is tasked with finding the longest possible match for a set of IP address that are streaming by. String matching takes as input a set of strings to match and is tasked with finding all of the places in the input stream where there is a match. Modern algorithms for both problems rely on limited preprocessing of the set of input patterns/strings, build large tree-like data structures for traversal at run-time, and suffer from a high degree of fanout from each edge.

We claim that many of the optimizations that were responsible for the speeding up of IP-lookup have analogs in the string matching world. Our optimizations take a cue from these optimizations; thus it is worth describing the IP-lookup optimizations that we draw inspiration from.

1) Unibit and Multibit Tries: Unibit and multibit trie schemes improve on linear search by placing data in a trie. Each node in an n-bit trie consists of a descriptor of any rules that are matched at this point, and 2^n pointers to next nodes. Traversal of the trie requires starting at the root and at each node considering the next n bits of the trie to select a pointer to the next trie node, while remembering the longest node matched thus far. Using this scheme, one can perform IPv4 lookup in $\log_n(32)$ steps. To save space and avoid building a full trie, only nodes that match rules or which have children that match rules must be populated. Multibit tries have similar structure to the Aho-Corasick data structure except that they have no failure nodes. When multibit tries increase in fanout by attempting to traverse more bits at a time, they waste a great deal of space with pointers to non-existent destinations, just as Aho-Corasick wastes space with null pointers or pointers that are optimized failure paths. The two following algorithms were adopted to address this shortcoming; our analogy suggests that we can use similar techniques to improve Aho-Corasick.

2) Lulea Algorithm: The Lulea algorithm [8] uses the concepts of leaf pushing and bitmaps to compress the size of the IP lookup database. By pushing routing information down to the leaves of the trie, Lulea prevents internal nodes from having to contain routing information. Traditional leaf-pushing creates a great deal of duplication in the leaf nodes, which is reduced by using a bitmap that compresses subsequent redundant entries. In order

to figuring out which routing information corresponds to a given leaf in the trie amounts to counting the bits set prior to that index in the bitmap. In order to add efficiency to the counting of what can be very large bitmaps, Lulea adds occasional summaries of the count up to a certain bit position to the bitmap data.

3) Eatherton Algorithm: The Eatherton tree-bitmap algorithm [11] improves upon Lulea by creating a data structure which does not require leaf-pushing and which can be traversed with a single wide memory access per node. For a tree-bitmap implementation that attempts to traverse n bits at a time; the nodes contain two bitmaps, the *internal bitmap* which is size 2^n and the *external bitmap* which is size $2^n - 1$, a pointer to an array of children and a pointer to an array of matching rules.

Traversing the datastructure is performed by checking for the most specific matching rule in the internal bitmap and if there is one, noting the offset that it corresponds to, and then checking to see whether there is a matching part of the tree in the external bitmap. We do not detail the correspondence between the internal bitmap and offsets into the routing array per node (see the paper [10] for a good explanation). The external bitmap correspondence is more straightforward though. One considers the next n bits to be traversed in the IP address as an integer and determines whether that integer's bit position in the external bitmap is set to one. If it is not set to one, then there is no corresponding entry in the child array. If it is set to one, then the offset of that entry from the child pointer is given by the sum of all bits prior to that bit in the external bitmap.

The Eatherton tree-bitmap algorithm is now reaching wide deployment in industry routers, indicating that the methods it employs are significant wins in hardware and achieve good compression on real IP lookup databases. More significantly for our paper, this means that the computations required for it, such as performing a population count, are likely to be made fast by network processor vendors, so there is significant incentive for us to also use such computations to compress string matching data structures.

III. Optimizations for String Matching

Having now discussed previous work on string matching and the relationship to existing algorithms from the domain of IP-lookup, we now propose two new data storage methods for string matching and describe the complexities and tradeoffs involved.

A. Bitmap Compression

Bitmap compression applied to Aho-Corasick uses a bitmap to achieve the same functionality that the external bitmap achieves in the Eatherton algorithm. To perform this optimization, we start with the unoptimized version of the Aho-Corasick data structure with failure pointers described earlier. Instead of every state in the Aho-Corasick state machine having 256 next state pointers, we now use a single pointer to point to the first valid next state and maintain a 256 bit bitmap indicating whether a traversal with a given character is valid or requires traversing along the failure pointer path. C-like pseudo code for our bitmap data structure can be seen in Figure 7 and a corresponding diagram of the datastructure can be seen in Figure 8. If the next state is valid, then the pointer to it is obtained in similar fashion to the Eatherton algorithm, by summing all the bits prior to that bit number and adding them to the base next node pointer.

Thus, to give an example from Figure 8, if we assume an alphabet where A=1, B=2, ... Z=26, if the next character in question is "D", then we transition from the current state by first checking to see if the fourth bit from the left in the bitmap is set. Finding that it is, we know that there is a valid transition at some offset from the next pointer. We then count all the set bits prior to bit four in the bitmap, and find that there is only one of them and therefore our offset from the next pointer is one. We add the size of one node to our next state pointer, jump to that datastructure which is the correct node for our "D" transition, and examine the next character in our packet. If on the other hand, our next character was "C", we would look in the bitmap and see that "C" was disabled. We would then follow the failure pointer and repeat the check with "C" on whatever node it pointed to.

On a machine with 32-bit pointers, the original optimized Aho-Corasick implementation creates a data structure with 1028 byte nodes, while our bitmapped version requires only 44 bytes. This decreased memory requirement improves cache-behavior of software implementations and makes it possible for hardware implementations to store the entire Aho-Corasick data structure in SRAM.

Using bitmaps incurs two costs though. First, the compression requires the unoptimized data structure, which as mentioned earlier doubles the worst case amount of work per character in the input. The second main cost is that traversal from node to node requires checking a bit in a bitmap and then performing a sum (or popcount) up to 256 prior bits in the bitmap. We attempt to minimize this in our software implementation by maintaining running sums of every 32 bits in the bitmap similar to Lulea, but that does not save us from doing at least a single 32-bit wide popcount per successful node traversed. On the x86 based systems that we tested on, performing a 32-bit popcount is very expensive, requiring more cycles than an L2

```
struct bitmap_state {
  struct bitmap_state * next_state;
  bitmap next_state_valid : 256;
  struct bitmap_state * failure_state;
  struct rule * rule_list;
};
```

Fig. 7. Bitmap Compressed Aho-Corasick Data Structure

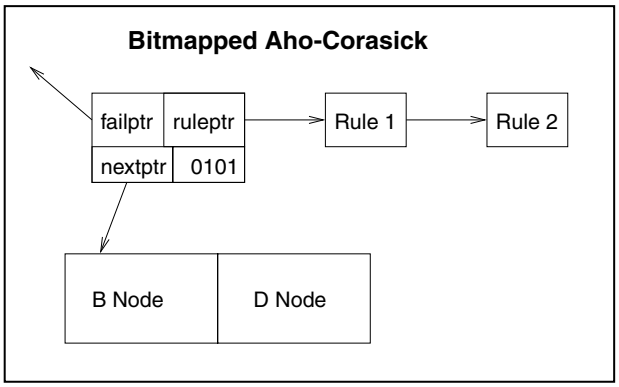

Fig. 8. Bitmap Compressed Data Structure Illustration

cache hit. We expect that it is much less expensive in most network processors, which as noted earlier are attempting to optimize the Lulea and Eatherton algorithms, as well as other networking algorithms which require similar bit manipulation operations.

B. Path Compression

Our next data optimization does not decrease the size per node or improve worst-case performance, but it does again decrease the total space required by the Snort database. In spirit, this optimization can be likened to the end-node optimizations performed in Eatherton's tree-bitmap or other IP lookup algorithms. We note that the Aho-Corasick state machine applied to the *Snort* database constructs a data structure with two regimes. Near the root of the state machine, the nodes are very dense, whereas further away they become long sequential strings with only one valid next state each. Bitmap compression works well for the top of the state machine, but even the bitmap itself is largely wasted information at the bottom nodes. To address this, we introduce path compression to handle nodes lower in the tree by squeezing as many of these sequential nodes as possible into a single node.

First, we recognize that bitmap compression fundamentally works because the elements represented by the bitmap are all equal size. Thus any path compressed nodes that are to be slipped into our bitmap compressed structure must also be equal in size to bitmapped nodes. In order to distinguish between the two nodes, we must add an extra bit to both data structures that differentiates between the two. The path compressed nodes consist of an array of characters, matching rules, and failure pointers which are traversed sequentially until either the end of the node is reached (in which case the next_state pointer is taken) or a mismatch is found (in which case the appropriate failure pointer is taken). We give pseudo code for our data structure in Figure 9 and show graphically the relevant structure in Figure 10. In Figure 10 the current node is path compressed and recognizes the sequence **ABCD** as being necessary to traverse to the **D** node. In the interest of drawing simplicity, we have omitted the fact that multiple failure pointers and rule pointers that correspond to each character are within the node.

As previously shown in Figure 3, this optimization decreases the size of the Aho-Corasick data structure significantly from the already reduced size achieved by bitmap compression. However, as with bitmap compression, it comes with additional costs. In addition to the single bit to differentiate between path compressed and non-path compressed nodes, the size of bitmapped nodes increases slightly because the failure pointer becomes more complex. Rather than simply being a pointer to the node to fail to, failure pointers must also include an offset within that node to account for the possibility that the failure transitions into the middle of a path compressed node. In a software implementation these two additions to the data structures increase the size of the underlying structure by approximately two bytes, depending upon how the compiler aligns structures. In a hardware implementation, the overhead would be three bits.

A second penalty for software implementations is the increased complexity of the search algorithm. Rather than traversing across well-known data structures, there are now a number highly unpredictable, data dependent branches in the code which execute one set of code or another based on whether the current node under examination is a bitmap node or a path compressed node.

On a 32-bit pointer machine, a single path compressed node can contain data equivalent to 4 bitmap compressed nodes, giving us a maximum compression ratio of 4:1. In practice, we find that we achieve a 2.54:1 compression ratio on the snort ruleset versus our bitmapped implementation.

IV. Results

In this section we examine the area and performance of the different algorithms in two operational environments. First we consider the speed of the various algorithms as applied to specialized network hardware for which worst-case bounds are the best metric of performance. We fur-

```
struct path_state_ptr {
  struct state_ptr *state;
  unsigned char offset;
};

struct path_state {
  struct rule *rule_array[PATH_NODE_SIZE];
  struct path_state_ptr
         failure_array[PATH_NODE_SIZE];
  unsigned char array_length;
  unsigned char carray[PATH_NODE_SIZE];
};

struct node {
  bool is_bitmap;
  union {
    path_state p;
    bitmap_state b;
  };
};
```

Fig. 9. Path Compressed Aho-Corasick Data Structure

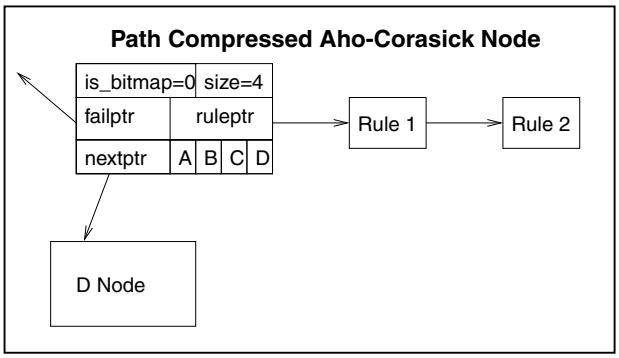

Fig. 10. Path Compressed Data Structure Illustration

ther examine how these algorithms fare on modern commodity hardware for both real packet traces and rule sets and for carefully crafted examples that demonstrate worst case behavior.

A. Instruction Detection in Hardware

The first constraint on any algorithm that is supposed to find it's way into a router is that it have a small memory footprint. The required amount of memory needs to be as small as possible to fit on-chip in fast memory (e.g., SRAM) and to reduce chip costs. If the algorithm cannot fit completely into on-chip memory it greatly complicates the timing analysis when it comes time to analyze its worst case performance.

Figure 3 shows the area used by Wu-Manber, SFK Search, default Aho-Corasick and our two optimized derivatives of Aho-Corasick (Bitmapped and Path Compressed). The sizes shown are generated by applying the different algorithms to the default Snort rule sets provided. Data is shown for both the oldest/smallest dataset (November 2000) and the most current and largest dataset (June 2003). The results show that our compression optimizations resulted in a 20 times savings in database size over Wu-Manber, and 50 times reduction in database size over the default Aho-Corasick. For current routers, it would be feasible to store the database for SFK Search, Bitmapped, and Path Compressed on-chip, but not the other two algorithms. In addition, over the past 3 years the number of rules has gone up by over a factor of 2.5, whereas the size of the memory footprint for our algorithms has only gone up by 30% giving us confidence in the continued scaling of our techniques.

We now turn our attention towards the performance of our algorithms. As intrusion detection algorithms find their way into routers, it is important that they perform well, even in the face of worst case situations. If the worst case is not considered, the router can be attacked with a worst case packet stream allowing a crafty adversary to slip the real data through while the IDS is overloaded.

In terms of performance, we examine the bandwidths achievable from the different algorithms assuming both an ASIC and Programmable router design. Both of these designs assume an accessible memory width of 128 bytes, which is easily implementable on-chip in modern ASICs. We assume that the full rules database, can be stored on-chip even though this is not the case for Wu-Manber and default Aho-Corasick and we do not assess any latency penalty for the memory shuffling that these two algorithms would inevitably be forced to do with off-chip SRAMs.

The reason for leveling the playing field and giving other algorithms the benefit of the doubt is that we want to focus our attention on the worst-case performance differences between algorithms that are purely a function of data structure width and arrangement. These differences are fundamental to efficient hardware implementations and to penalize the other algorithms for their lack of space efficiency would only obscure the fact that their width and number of independent accesses required are more important limiters to their performance.

Again using the Snort 2003 rule set, we analyzed the data structures of each algorithm and derived a worst-case sequence of characters from that rule set that triggers a maximal number of memory references for each algorithm. As mentioned earlier, we were able to find a character sequence that traverses 20 nodes per character in SFKsearch and over 80 rules every other character for Wu-Manber. For Wu-Manber, we gave the algorithm the benefit of the doubt and assumed that the 80 rules could

be compressed into one long sequential memory access, rather than charging for 80 random accesses. The worst case for all variations of Aho-Corasick is the default case, one or two nodes traversed per character of input.

We then explore tradeoffs in SRAM memory widths, sizes and numbers of ports using a version of CACTI 3.0 [17] modified to correlate closely with the results generated by 130 nm memory generators. We use the methodology of [16] to find a Pareto optimal design for a pipelined wide-word unified memory subsystem for each algorithm. This subsystem would be representative of what an ASIC designer would attempt to include in a piece of dedicated hardware that executed each algorithm. We also find a design that achieves the highest geometric mean of performance on this algorithm and other networking algorithms. We hold that this tradeoff subsystem would be representative of what would be desired in a general purpose programmable network processor.

As shown in Figure 11, even giving Wu-Manber and default Aho-Corasick the unfair advantage of assuming perfect cost-free on-chip memory, we find that there is a substantial advantage to the bitmapped and path compressed versions of Aho-Corasick. For the ASIC design, the decreased width of the data structure allows us a 31% performance advantage in the most advantageous design examined. If we create a Programmable router that can be used to implement many network-related algorithms (IP lookup, Classification, etc.), we find that performance of our bitmap compressed Aho-Corasick improves to twice the performance of of the non-bitmap compressed Aho-Corasick. In comparing Wu-Manber and SFK Search to Bitmapped, the Bitmapped algorithm is able to achieve 8 times more throughput than SFK Search and over 3.25 times more throughput than Wu-Manber.

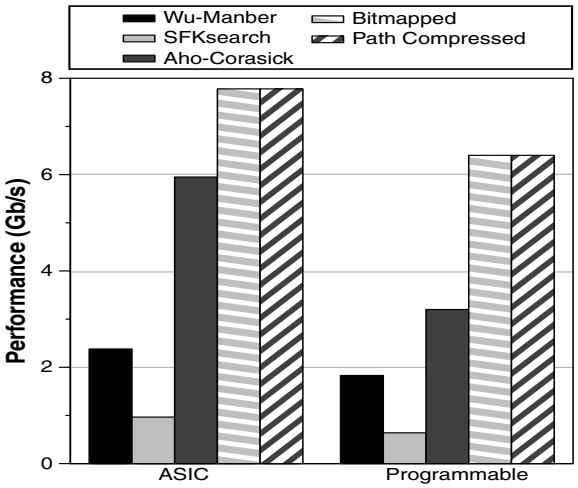

Fig. 11. Hardware Performance of the Algorithms in Terms of the network bandwidth that it will be able to handle in Gigabits per Second. The ASIC design shows the string matching throughput tailoring the implementation to only string matching. Programmable shows the performance assuming an implementation that can be used for many different type of router applications.

B. Intrusion Detection in Software

In this section we measure the performance of an implementation of *Snort* that has been augmented with the various string matching algorithms discussed. We examine the results of running this program on a variety of different real machines, including a 1 GHz Pentium 3, a 2.5 GHz Pentium 4 and a 1.3 GHz Pentium M. For all results, we show the execution time (in seconds) of two separate packet streams. We examine both average-case and worst-case performance through use of two different traces.

For the average case results, we use the most current (June 2003) *Snort* default database with a trace from the Capture the Flag game held at the Defcon9 conference. The Capture the Flag game has the objective of trying to break into the computers of other teams while protecting your own and running a server with several security holes [7]. Therefore the trace includes both innocuous activity and attempts to break in. Figure 12 shows performance results when using this trace.

Figure 13 shows results for a synthetic trace consisting of a worse case packet stream. The worse case packet stream exercises the weaknesses in both the Wu-Manber and SFK Search algorithms. We did this by adding a rule to the snort database which consists of merely a long string of the same character with a single different character in the middle of the string (either towards the beginning of the string for Wu-Manber or towards the end for SFKsearch). We then construct a stream of "attack" packets that contain a payload of the starting/ending character. Because the packets defeat the bad character heuristic, the algorithms are forced into their worst-case of performing a string-compare every character. Although it may seem somewhat contrived to add a rule to the database, there are already several rules like this in it that possess the same property. For instance, there is an icmp packet rule in snort for the "ICMP PING Cyberkit" that matches content 'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa'. This is a rule that is trivial to exploit nearly as effectively as our synthetic rule on both algorithms.

The results show that worst-case execution time is 3 to 4 times faster using Aho-Corasick, Bitmapped, and Path Compressed over Wu-Manber and SFK Search. However, for average-case execution in Figure 12, Wu-Manber is about twice as fast as Aho-Corasick and our Bitmapped algorithms. This is because the bad character heuristics in Wu-Manber usually work quite well and on the occasions

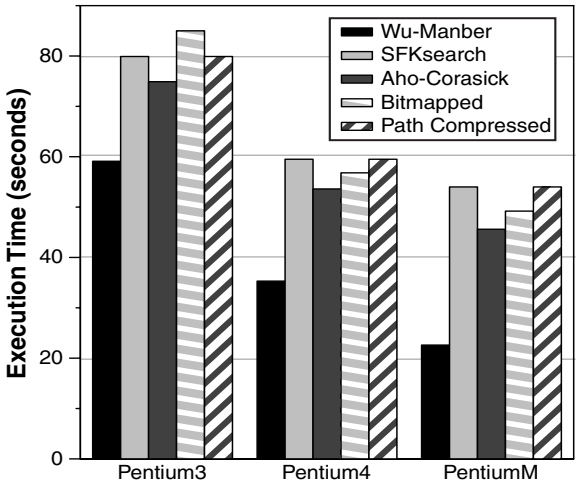

Fig. 12. Average-Case Software Performance of Algorithms on Defcon9 Trace.

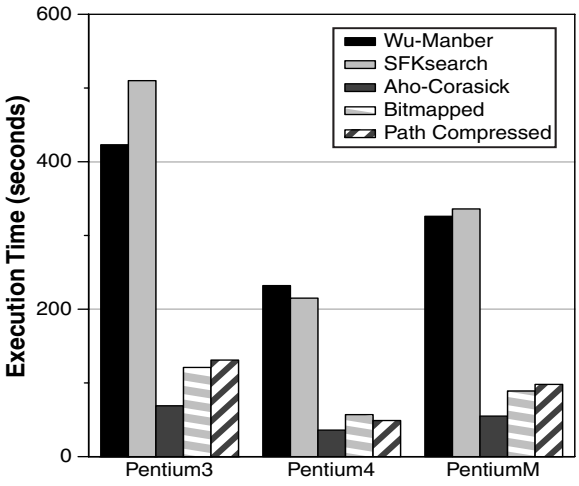

Fig. 13. Software Performance of Algorithms on Synthetic Worst-Case.

when they fail, the hash lookup is actually quite fast in software. Our algorithms require variable size shifts, perform a population count, and generally have bad branch behavior; all of which contribute to poor average case performance on modern general purpose processors. However, we are optimistic that our average-case performance would be better on a processor that was more tailored towards networking (e.g., including popcount instructions), as these are all well understood problems with many networking codes.

V. Summary

Up until recently, most IDSs were deployed using commodity processors. However, as link speeds increase and rule sets become larger, there is increasing pressure to place IDS systems in hardware so that they can run on routers, thereby catching attacks as close to the edge of the network as possible. Current software IDSs largely rely on common-case optimizations (such as bad character heuristics) to gain speed. However, as IDSs migrate to hardware with a useful life of 2 to 5 years, with corresponding growth in uncertainty as to the characteristics of future strings that will be added, there is considerable incentive to design string matching engines that can enable real-time intrusion detection with guaranteed worst-case performance.

Guided by the analogy between IP lookup and string matching, our paper builds on the worst-case guarantees of the classical Aho-Corasick string matching algorithm. As with multibit tries, Aho-Corasick is the only string matching algorithm we know of that has *deterministic* worst-case lookup times and a data structure friendly enough to use for wire speed hardware matching. Unfortunately, the classical Aho-Corasick data structure takes more storage than is likely to fit in on-chip SRAM or the cache of a commodity processor.

The principal contribution of our paper is to apply bitmap node compression and path compression to Aho-Corasick to gain both compact storage and worst-case performance. In particular, we show that the use of such compression gains factors of almost 50 times in database size reductions on current rule sets. While the case is less clear for software implementations unless more predictable performance is desired; we believe that our compressed Aho-Corasick algorithms are the best choice for hardware implementations of string matching for IDS using an FPGA, ASIC or network processor designs of the future.

Acknowledgments

We would like to thank the anonymous reviewers for providing useful comments on this paper. We would like to thank employees of Sourcefire for their comments on drafts of the paper after its acceptance. We would also like to thank Kostas Anagnostakis for enlightening discussions on this paper and other issues associated with worst-case IDS performance. This work was funded in part by NSF grant CRR-0311712, and a grant from Intel Corporation.

References

[1] A. V. Aho and M. J. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

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

[3] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):761–772, 1977.

[4] A. L. Buchsbaum, G. S. Fowler, B. Krishnamurthy, K. Vo, and J. Wang. Fast prefix matching of bounded strings. In *Fifth Workshop on Algorithm Engineering and Experiments (ALENEX03)*, 2003.

[5] CERT/CC. Code Red worm exploiting buffer overflow in IIS indexing service DLL. CERT Advisory CA-2001-19, Jan 2002.

[6] B. Commentz-Walter. A string matching algorithm fast on the average. *Proceedings of ICALP*, pages 118–132, July 1979.

[7] C. Cowan, S. Arnold, S. Beattie, C. Wright, and J. Viega. Defcon capture the flag: Defending vulnerable code from intense attack. In *DARPA DISCEX III Conference, Washington DC*, April 2003.

[8] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink. Small forwarding tables for fast routing lookups. In *Proceedings of SIGCOMM*, pages 3–14, 1997.

[9] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood. Deep packet inspection using parallel bloom filters. In *11th Symposium on High Performance Interconnects*, August 2003.

[10] W. Eatherton. Hardware-based internet protocol prefix lookups. Washington University Electrical Engineering Department (MS Thesis), May 1999.

[11] W. Eatherton, Z. Dittia, and G. Varghese. Tree bitmap : Hardware/software ip lookups with incremental updates. Unpublished, 2002.

[12] U. Manber and S. Wu. GLIMPSE: A tool to search through entire file systems. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 23–32, San Fransisco, CA, USA, 17–21 1994.

[13] E.P. Markatos, S. Antonatos, M. Polychronakis, and K.G. Anagnostakis. Exclusion-based signature matching for intrusion detection. In *Proceedings of the IASTED International Conference on Communications and Computer Networks (CCN)*, pages 146–152, November 2002.

[14] M. Roesch. Snort – lightweight intrusion detection for networks. In *Proceedings of LISA'99: 13th Systems Administration Conference*, pages 229–238, November 1999.

[15] M. Sanchez, E. Biersack, and W. Dabbous. Survey and taxonomy of IP address lookup algorithms. *IEEE Network Magazine*, 15(2):8–23, 2001.

[16] T. Sherwood, G. Varghese, and B. Calder. A pipelined memory architecture for high throughput network processors. In *30th International Symposium on Computer Architecture*, June 2003.

[17] P. Shivakumar and N. P. Jouppi. Cacti 3.0: An integrated cache timing, power, and area model. Technical Report 2001/2, Compaq Western Research Laboratory, August 2001.

[18] B. W. Watson. The performance of single-keyword and multiple-keyword pattern matching algorithms. Technical Report 94/19, Eindhoven University of Technology, 1994.

[19] B. W. Watson and G. Zwaan. A taxonomy of keyword pattern matching algorithms. In H. A. Wijshoff, editor, *Proceedings Computing Science in the Netherlands 93*, pages 25–39, SION, Stichting Mathematish Centrum, 1993.

[20] S. Wu and U. Manber. Agrep – a fast approximate pattern-matching tool. In *Proceedings USENIX Winter 1992 Technical Conference*, pages 153–162, San Francisco, CA, 1992.

[21] S. Wu and U. Manber. A fast algorithm for multi-pattern searching. Technical Report TR-94-17, Department of Computer Science, University of Arizona, 1994.