High Speed Pattern Matching for Network IDS/IPS

Mansoor Alicherry
Lucent Bell Laboratories
Murray Hill, NJ, USA - 07974
Email: mansoor@bell-labs.com

M. Muthuprasanna
Iowa State University
Ames, IA, USA - 50011
Email: muthu@iastate.edu

Vijay Kumar
Lucent Bell Laboratories
Murray Hill, NJ, USA - 07974
Email: vijay@bell-labs.com

Abstract— **The phenomenal growth of the Internet in the last decade and society's increasing dependence on it has brought along, a flood of security attacks on the networking and computing infrastructure. Intrusion detection/prevention systems provide defenses against these attacks by monitoring headers and payload of packets flowing through the network. Multiple string matching that can compare hundreds of string patterns simultaneously is a critical component of these systems, and is a well-studied problem. Most of the string matching solutions today are based on the classic Aho-Corasick algorithm, which has an inherent limitation; they can process only one input character in one cycle. As memory speed is not growing at the same pace as network speed, this limitation has become a bottleneck in the current network, having speeds of tens of gigabits per second.**

In this paper, we propose a novel multiple string matching algorithm that can process multiple characters at a time thus achieving multi-gigabit rate search speeds. We also propose an architecture for an efficient implementation on TCAM-based hardware. We additionally propose novel optimizations by making use of the properties of TCAMs to significantly reduce the memory requirements of the proposed algorithm. We finally present extensive simulation results of network-based virus/worm detection using real signature databases to illustrate the effectiveness of the proposed scheme. [1]

I. Introduction

The past few years have witnessed a tremendous increase in the frequency and sophistication of attacks on the Internet. There have been notorious viruses/worms like Code Red, Nimda, Slammer etc. [1] making the news. By exploiting the security flaws in operating systems, underlying network protocols and different software implementations, attackers bring down significant parts of the Internet in a matter of hours using distributed co-ordinated attacks aided with an ever-increasing population of zombie machines. The combination of widespread software homogeneity and Internet's unrestricted communication model creates an ideal climate for launching such attacks. With the Internet becoming the hub for global commerce and communications, such attacks can have a devastating effect on global economy.

The most commonly employed defense today is to use end-host based solutions that rely on security service tools, anti-virus software, personal and enterprise firewalls etc. These approaches have drawbacks in being insufficiently fast to meet new virus threats, and the inability to respond fast is increasingly being exploited by new worms designed to contaminate tens of thousands of hosts in less than an hour.

[1]Work done while M. Muthuprasanna was visiting Lucent Bell Labs.

It is hard to install security upgrades in a large number of enterprise network clients within such a short duration of time. This was clearly evident in the inability to defend against the DDoS attacks on SCO Group and others, in spite of prior information about the nature, date and time of the impending attack [2]. A more effective approach would be to use network-based defenses to stop worm propagation in the network before they reach a large number of end users. Previous studies [3] have shown that 90% of transit paths pass through top 10 ISPs, so interdicting attack traffic in the core network is a highly effective approach, if the search speeds needed at the core can be achieved.

Although various network-based defense techniques such as statistical filtering [4], honeypots [5], network telescopes [6] etc. have been proposed, the most common defense in use today is intrusion detection and prevention using signature matching. Signature matching techniques are widely deployed due to their high levels of precision and accuracy. Various network IDS/IPS such as SNORT [7] employ packet filtering of incoming data packets based on the occurrence of known signature patterns in the packet headers or payload. However, most current network-based security devices can perform only layer 3 or layer 4 packet filtering with the packet header. Line speed packet filtering based on bit-patterns in packet payload (layer 7 filtering) is a big challenge especially when scanning for thousands of patterns. Additionally, it is now increasingly common to have upto 10Gbps Ethernet speeds in metro/enterprise networks and upto 40Gbps speeds at the core.

Additionally, high-speed pattern matching is required for a wide variety of other equally critical applications, including scanning through large data-sets (logs) for data mining operations, low latency XML switching, DNA sequence matching etc. Of greater interest is the use of pattern matching in next-generation network monitoring applications, including but not limited to, stateful packet inspection for QoS management, VOIP filtering, bandwidth metering, optimal cache replication etc. However, we limit our focus here to network-based IDS/IPS for virus/worm detection.

Motivated by these problems, we looked at the issue of multiple string matching at line speeds of 10Gbps and above. Most of the string matching solutions today derive from the classic Aho-Corasick algorithm [8]. We identify two main drawbacks of this approach: 1) The state machine transitions

are single byte transitions, which makes it hard to scale to current network speeds. 2) The state machine is large, and highly memory intensive. We also note that current network architectures process the data packet by packet rather than byte by byte, and hence multiple input bytes are available at a time for content inspection. The TCAM memory chips can handle inputs which are hundreds of bits long (upto 576 bits as of today). TCAMs also support a ternary "don't care" state that can be used to greatly scale down memory requirements by aggregating the different rules together. Our proposed algorithm take advantage of all these different features.

Our main contributions in this paper are:
1. We develop a novel multi-character multi-pattern string matching technique to achieve faster search speeds.
2. We propose a novel transition optimization to significantly reduce the memory requirements of the algorithm.
3. Finally, we evaluate the proposed techniques using real-life virus/worm signatures to study their effectiveness.

II. Related Work

The string matching algorithms have been traditionally used in many applications like word processing, search-and-replace operations, bibliographic search etc. Additionally, they are also increasingly being used for IP lookup in routers, virus/worm detection using signature matching, network monitoring for stateful packet filtering etc. Researchers have proposed various software and hardware based solutions in efficiently tackling the underlying string matching problem.

Software-based: The Knuth-Morris-Pratt[9] and Boyer-Moore[10] are classic software-based string matching algorithms. The substring matching[11], two-way string matching[12], Rabin-Karp randomized algorithms[13] and others propose various novel string matching solutions. The real-time algorithm in [14] is highly suited for hardware implementations. The Aho-Corasick (AC) algorithm[8] is the seminal work in the field of multiple string matching algorithms. Tuck et al. proposed novel techniques[15] to reduce the memory requirements of the AC algorithm by using Bitmap Compression and Path Compression. Various other algorithms[16][17][18][19], that improve on the average running time of the AC algorithm have been proposed.

Hardware-based: Recently, in addition to software-based techniques, efforts have been directed towards achieving high speed implementations of the multiple string matching algorithms in hardware. Research has been directed along two directions: firstly towards design of efficient data structures and other optimizations for faster memory access rates, and secondly design of higher throughput algorithms to augment the well-known algorithms in literature. In [20], the authors adapt the AC algorithm to process multiple bytes at a time on an FPGA, leading to much better average throughput. A deterministic finite state machine is used in [21] for performing multi-byte high-speed string matching on an FPGA. In [22], the authors propose a SRAM based AC implementation that also gives higher average case throughput. In [23], the authors propose a TCAM-based multi-byte multiple string matching algorithm with limited support for wildcards. In [24], the authors propose a bit-split finite state machine to achieve higher search speeds on specialized hardware. In [25], the author proposes balanced routing table search based algorithm, for FPGA and ASIC implementations.

There has also been extensive work on imprecise string matching algorithms using hashing techniques and bloom filters as proposed in [26]. The use of non-deterministic finite automata and other FPGA optimizations has been proposed in [27],[28]. In addition, various hardware-based techniques for intrusion detection have been proposed in[29][30][31][32][33].

III. Proposed Solution

The classic Aho-Corasick algorithm was the first technique proposed to do multiple string matching in linear time, and it constructs a finite state machine to do so. Like many subsequent solutions to the multiple string matching problem, our proposed solution is also based on this state machine. In our solution, we describe a TCAM-based architecture that can efficiently implement this state machine, where the number of TCAM entries is equal to number of transitions in the state machine and is independent of the number of states therein. We then propose a state encoding scheme using certain properties of the TCAM, which can implement the state machine with only a fraction of the existing TCAM entries. We finally propose techniques to increase the pattern matching speeds, scalable to multi-gigabit network speeds today, by implementing an equivalent state machine making state transitions on multiple characters.

A. Aho-Corasick Algorithm

The Aho-Corasick (AC) multiple string matching algorithm [8] builds a deterministic finite state automaton (DFA) that encodes all the strings to be searched, in multiple stages. The first stage called the *goto* function constructs a trie of the patterns, where the root of the trie represents the state where no strings have been partially matched. All the strings are extended from the root node adding one state per character. The strings that share a common prefix also share a corresponding set of parent nodes in the trie. We call the distance of a state from the root in the *goto* trie as the *depth* of that state. To match a string, we start from the root node and traverse the edges according to the specified input characters. The second stage involves the insertion of the *next* transitions. When a string match is not found, it is possible that the suffix of the previously matched string is the prefix of another string in the trie, and hence we use the *next* transitions to slide to a different string (branch) in the trie.

Example: Figure 1 gives an example of the *goto* trie for the patterns ABCDEFGHIJK, WXYZABCDIJ and WXYZABPQ. If the input is WXYZABCDEFGHIJK, we start with the root

node and look for the pattern WXYZABCDIJ. After input D (state 19), there is no transition on input E. But it is a partial match for the pattern ABCDEFGHIJK and hence we put a *next* transition from state 19 to state 5. Note that all states have *next* transitions to states 1 and 12 on characters A and W respectively, if they do not have a corresponding *goto* transition on these characters.

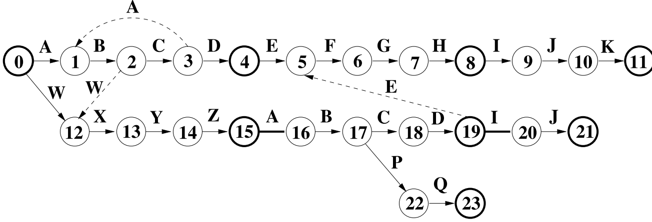

Fig. 1. Aho-Corasick DFA (Not all *next* transitions shown)

B. Hardware Architecture

We now describe the hardware architecture we assume for implementing the state machine. The architecture consists of a TCAM, static RAM (SRAM) and a logic. Each TCAM entry represents a certain transition in the state machine, and has a corresponding memory block (structure) in the SRAM whose address can be computed from the TCAM index. We logically partition the TCAM entries into two fields: *current state* and *input*. If the state machine transitions from state s_1 to state s_2 on input a, then the TCAM contains an entry (s_1, a) and the corresponding entry in SRAM contains s_2. If the state s_2 corresponds to one or more keywords, then the SRAM entry also contains pointers to those keywords.

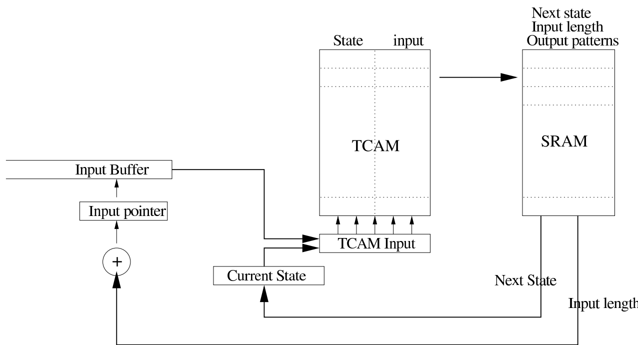

Fig. 2. Hardware Architecture

Figure 2 presents the details of the hardware architecture. In this paper, we assume the existence of a standard flow classification hardware module that takes care of identifying the packets of a flow, sequencing etc. The pattern matching hardware module runs a unique state machine instance for each flow. This is essential to detect patterns spread across multiple packets in a flow. The incoming packets for the current flow are stored in the input buffer. For each flow, we store two pieces of information, namely the current state in the state machine and a pointer to the next input (character) to be fed to the state machine. These are stored in registers marked *current state* and *input pointer*, initialized to state zero and the start of the buffer respectively. The hardware logic concatenates the current state and next character from the input buffer and feeds them to the TCAM. If there is a matching entry in the TCAM, the index of the (first) matching entry gives a location in the SRAM memory which contains the next state information. The input pointer is now advanced and the next state value is stored in the current state register. This process repeats. If there is no match in the TCAM, the current state is set to the start state and the input pointer is correspondingly advanced. The input length field in SRAM is used for multiple character transitions on a compressed state machine, and is described in Section IV.

Example: Consider the state machine in Figure 1. There are 24 states and 69 transitions (23 goto transitions and 46 next transitions). The state can hence be represented in 5 bits and the input character is 8 bits long. Hence each TCAM entry would be 13 bits long, and the TCAM would have 69 entries corresponding to each of the 69 state machine transitions. There would be 69 SRAM structures correspondingly representing the information about the next state and the matched keywords. Figure 3 shows a few sample TCAM and SRAM entries for the state machine in Figure 1.

	TCAM Entries		SRAM Entries	
Transition from Root node	0	A	1	—
Transition from Root node	0	W	12	—
NEXT Transition (eliminated below)	1	A	1	—
NEXT Transition (eliminated below)	1	W	12	—
Sample GOTO Transition	1	B	2	—
	. . .	. . .	. . .	. . .
	15	A	16	—
Sample NEXT Transition	19	E	5	—
Sample Transition with Output Keyword	20	J	21	Pattern 2

(Current State, Input) —> (Next State, Matched Keyword)

Fig. 3. TCAM and SRAM entries (no optimization)

C. Memory Optimization

Note that in the above example, out of 46 $next$ transition entries, 22 are transitions on character A to state 1, and 23 are transitions on character W to state 12. Also note that the $goto$ transition from state 0 on A and W also reach the states 1 and 12 respectively. Hence, if we replace the state 0 with ????? in the TCAM, then all these redundant $next$ transitions are implicitly taken care of, without requiring the TCAM entries. However, we need to ensure that this entry comes after the entry corresponding to the transition from state 15 on input character A. As the TCAM returns the first matching entry, the state machine would otherwise reach state 0 from state 15 on input A. Figure 4 shows a few sample TCAM and SRAM entries for the state machine in Figure 1. Note that the redundant $next$ transitions to states at depth one in the state machine have now been eliminated.

Additionally, we could eliminate $next$ transitions to states at higher depths in the state machine. If we suffix the state id representation of every state by the last character that caused a $goto$ transition into that state, then we can remove the $next$ transitions to states at depth two in the state machine.

Hence, a state at depth one is represented as "?????$i_1, \ldots, i_8$", where $i_1, \ldots, i_8$ is the binary representation of the input character that causes a $goto$ transition from the root node to that state. This state id representation would then match all those states in the state machine that is reached by a $goto$ transition on character $i_1, \ldots, i_8$ from some other state. Thus all $next$ transitions to states at depth two in the state machine are subsumed by the $goto$ transition to depth two state, along similar lines of argument as above, and hence can be eliminated from the TCAM.

	TCAM Entries		SRAM Entries	
Sample GOTO Transition	1	B	2	—
Precedence for non−default Transitions	15	A	16	—
Sample NEXT Transition	19	E	5	—
	⋮	⋮	⋮	⋮
Sample Transition with Output Keyword	20	J	21	Pattern 2
Default Transition to Depth 1 State	?????	A	1	—
Default Transition to Depth 1 State	?????	W	12	—

(Current State, Input) —> (Next State, Matched Keyword)

Fig. 4. TCAM and SRAM entries (with memory optimization)

Along similar lines, if we can dedicate the last $8(m\text{-}1)$ bits of the state id of a state to encode the last m-1 bytes that cause the state machine to reach that state using $goto$ transitions alone, we can ignore the $next$ transitions to all states at depth $\leq m$. For example, to eliminate $next$ transitions to states at depth 5 in the AC DFA in Figure 1, we suffix the state ids with the last four characters, or $goto$ transitions leading to that state. Hence state 4 will be represented as "??..??ABCD" and last 32 bits of state 19 will be ascii value of "ABCD". Now the $next$ transition from state 19 to state 5 on character E is subsumed by the $goto$ transition from state 4 to state 5 on character E. Clearly, there is a trade-off here since the size of state identifier increases with m.

Depth (d)	1	2	3	4	5	≥ 6	Total
States	56	89	100	102	102	7248	7698
Transitions	421243	7576	551	165	121	7382	437038

TABLE I

STATES AND TRANSITIONS IN AHO-CORASICK DFA

Example: We now analyze the AC DFA constructed for a real virus signature database for the Internet (See Section V for details). Table I summarizes the number of states at a certain depth, and transitions to states at those depths in the Aho-Corasick DFA. We see that the number of transitions to states at depth 1 in the AC DFA for the sample database, is as high as 96.4% of the total transitions. Note that most of the transitions to smaller depths are mainly *next* transitions, as the number of *goto* transitions to any state is atmost one. Thus, the proposed optimization (with m=3) reduces the size of the TCAM state machine representation to 1.75% of its original size, yielding memory savings of 98.25%. For the sample database, we see the law of diminishing returns at play with respect to increased memory savings. This is partly due to fewer $next$ transitions to states at higher depths in the AC DFA, and partly due to the increased bit-space needed to represent the state ids. Hence it is advisable to restrict m to smaller values (see Section V).

D. Search Speed Enhancement

The techniques described thus far implement the Aho-Corasick state machine, while processing only one input character per TCAM lookup. As this does not scale to high speeds required today, we now propose techniques to achieve greater speed-up using the same architecture.

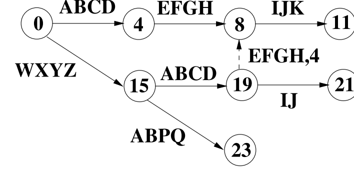

Fig. 5. Compressed AC DFA (Not all *next* transitions shown)

Consider the state machine in Figure 5. This is functionally similar to the state machine in Figure 1, except that the transitions are now on four characters each. We call this state machine, a *multi-character (compressed) state machine*. This state machine can be implemented using our proposed architecture with the following changes: the TCAM entries now contain four characters in the input field requiring 32 bits for their representation, and the input pointer is now incremented by 4 for every state transition (i.e every TCAM lookup). Hence we get a speedup of upto four, provided the input bus to the TCAM is wide enough (which is achievable when implemented in custom hardware).

However, it might not always be possible to make state transitions on the same number of characters (eg. 4 in the above example), and hence we have another field called *length*, in the SRAM corresponding to each transition. The hardware logic increments the input pointer appropriately by the corresponding value in SRAM after every TCAM lookup. Henceforth, we refer to the maximum number of input characters that are placed in a single TCAM entry as *transition width* and denote it by k. Additionally, we need to "synchronize" the input with the state machine to account for the offset at which a pattern might occur in the input stream, and use additional $shallow$ states and transitions to ensure correctness. In the above description we have left out exact details, which are explained in detail in next section.

IV. MULTI-CHARACTER STATE MACHINE

We propose constructing a compressed finite state machine that encodes all the strings and makes state transitions on multiple (at most k) input characters. We start with the Aho-Corasick DFA and create an equivalent state machine called the *compressed DFA* that has transitions on multiple input characters, by combining k consecutive states of Aho-Corasick DFA. The *goto* transitions are created by combining k consecutive states of the Aho-Corasick DFA, while the *next* transitions are created as in the AC DFA, but with variable lengths. Figure 5 shows compressed AC DFA for $k = 4$.

The compressed DFA differs from AC DFA as follows:

1) The state transition is done on multiple characters (between 1 and k characters).
2) *Rollback Feature:* Once a transition is made, not all the input in "consumed" by the state machine. Each transition will indicate the number of characters by which to advance the input pointer. In rare cases this could be negative, which means that we need to appropriately move the input pointer backwards.
3) *Longest Input Match:* From the same state, the compressed DFA may contain two transitions, where one input is a subset of the other. In those cases the DFA takes the transition on the longer input.
4) The strings matched (*output*) in the DFA are associated with the transitions; whereas they are associated with the states in the Aho-Corasick algorithm.

We now present the details of the compressed AC DFA construction here. We use the subscripts c and o to denote the compressed and Aho-Corasick DFA respectively. Hence, $goto_c$ and $goto_o$ represent the goto transitions on the compressed and Aho-Corasick DFA respectively. We denote by $string(s)$ and $dist(string(s))$, the characters consumed in reaching the state s from the root, and length of that string respectively.

Compressed DFA Goto Algorithm: We create states in the compressed DFA corresponding to those states in the Aho-Corasick DFA that are at depths that are multiples of k, or that are leaf nodes in the $goto$ trie. We call these states in the Aho-Corasick DFA as *core states*. The subset of core states which corresponds to pattern-end states in the Aho-Corasick DFA are specifically called *end core states*. We say that two core states are *adjacent* if the corresponding states in the compressed $goto$ trie are adjacent.

We also create a many-to-many mapping (called *skick or sidekick*) from the Aho-Corasick DFA to the compressed DFA. Every core state uniquely maps to its corresponding compressed state. All the non-core states in the path between two adjacent core states $s1_o$ and $s2_o$ ($depth(s1_o) < depth(s2_o)$) map to the sidekick of $s2_o$. There may be multiple elements in the sidekick of a non-core state, if that state is part of multiple patterns. We also define a function *cParent* for states of the Aho-Corasick DFA as follows: If the state is a core state, then *cParent* of the state is its sidekick state; if the state is a non-core state then *cParent* is the parent of any of its sidekicks. Note that the parent of all the sidekick states of a non-core state are the same. Intuitively, if the transitions are only *goto* transitions, the compressed DFA would be on $cParent(state_o)$ with a $goto$ transition to $skick(state_o)$, if the AC DFA reaches $state_o$.

Example: In Figure 1, $0, 4, 8, 11, 15, 19, 21, 23$ are *core* states, while $11, 21, 23$ are *end core* states. Also states $16, 17$ have states $19, 23$ as *sidekicks* and state 15 as *cParent*.

We create a *compressed goto trie* by doing a depth first search (DFS) on the Aho-Corasick goto graph. During the depth first search, we keep track of the compressed state corresponding to the preceding core state ($parent_c$), and the part of the pattern starting from that state ($input$). If the current state is a core state (i.e. depth is a multiple of k or it is a leaf node) then we create a new compressed state. A $goto_c$ transition on the string $input$ is created from the parent to the new state. The new state becomes the new $parent_c$ and the input is reset to $null$. *Sidekick* and the *cParent* of the core state will be the new state. The algorithm is recursively called for all the children in the Aho-Corasick goto graph, with the transition character appended to the $input$ string. If the state is a non-core state, sidekick information is populated from its child states. A formal description is given in Algorithm 1. The algorithm has the same complexity as the depth first search of the flow graph which is linear in the number of states. As the number of states is at most the total number of characters in all the input patterns, the complexity of our algorithm is $O(n)$.

Algorithm 1: Compressed DFA Goto Function

```
cGoto(st_o, parent_c, depth, inp, out)
{
    if (st_o is a core state)
    {
        st_c = Create a new compressed state
        goto_c(parent_c, inp) = st_c
        parent(st_c) = parent_c
        parent_c = st_c
        skick(st_o) = st_c

        output_c(parent_c, st_c, inp) = out
        if (st_o is not an end core state or is a leaf state)
            inp = out = null
    }
    cParent(st_o) = parent_c
    for all a such that goto_o(st_o, a) ≠ null
    {
        out = out ∪ output_o(goto_o(st_o, a))
        cGoto(goto_o(st_o, a), parent_c, depth + 1, inp + a, out)
        if (st_o is not core state)
            skick(st_o) = skick(st_o) ∪ skick(goto_o(st_o, a))
    }
}
```

Compressed DFA Next Algorithm: We present an algorithm here to map the Aho-Corasick DFA $next_o$ transitions to one or more compressed DFA $next_c$ transitions. For a next transition $next_o(s_o, a) = d_o$, we create a $next_c$ transition from $cParent(s_o)$ either to $cParent(d_o)$ (called *backward next transition*) or to all the states in $skick(d_o)$ (called *forward next transition*), based on the length of the transition and $safe(d_o)$, as explained below. Note that forward transitions consume more input characters and hence have a better throughput than the backward transitions. But it is not always possible (or easy) to create forward transitions if d_o is not a safe state or input length $k > TW$ (the TCAM width).

Definition: A state in the Aho-Corasick DFA is *safe* if it is either a *core* state, or all its children (states) are *safe* and there are no $next$ transitions to any state at depth $> m$.

Safe State computation: Let $next_o(s_o, a)$ be d_o^i. Let d_o^1 and d_o^k be the adjacent core states such that d_o^i is in the path

between d_o^1 and d_o^k. Note that there can be multiple such d_o^k. Let $d_o^1, d_o^2, \ldots d_o^i, \ldots d_o^k$ be the path between d_o^1 and d_o^k in the goto graph. If there is no next transition, to a state of depth more than m, from any of $d_o^i, \ldots d_o^{k-1}$, then we can create a next transition from $cParent(s_o)$ to $skick(d_o^i)$. We call such a d_o^i a *safe state*. If the next transition is to a state that is not safe, then we cannot just create such a transition and guarantee correctness. We use the boolean function $safe$ to indicate whether a state is safe or not. Safe state function for all the states can be computed by doing a depth first search on the Aho-Corasick DFA. A formal description is presented in [34]. We use safe states to avoid the possibility of missing certain $next$ transitions, during a cascade of multiple $next$ transitions in the goto graph.

Forward next transitions: Figure 6(b) shows a *forward next* transition. A forward transition for $next_o(s_o, a) = d_o$ is formed when d_o is a safe state and length of the transition from $cParent(s_o)$ to all the nodes in $skick(d_o)$ is less than or equal to k. The transition is on input $string(s_o) - string(cParent(s_o)) + a + (string(skick(d_o)) - string(d_o))$ and consumes the input, $dist(s_o, cParent(s_o)) + dist(skick(d_o), d_o) + 1$ characters.

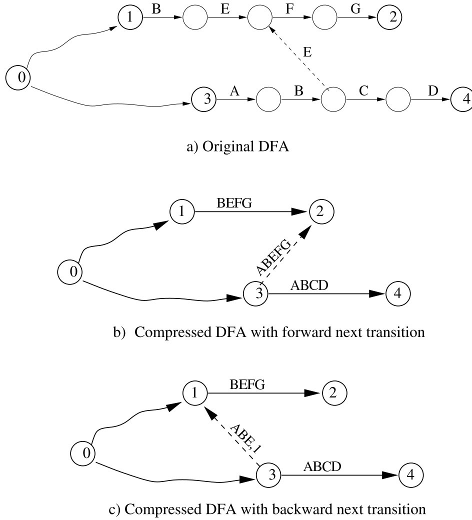

Fig. 6. Creating $next$ transitions in the compressed AC DFA

Backward next transitions: Figure 6(c) shows a *backward next* transition. A backward transition for $next_o(s_o, a) = d_o$ is formed when d_o is not a safe state or length of the transition from $cParent(s_o)$ to any of the states in $skick(d_o)$ is more than k. The transition is on input $string(s_o) - string(cParent(s_o)) + a$ and consumes $dist(s_o, cParent(s_o)) - dist(d_o, cParent(d_o)) + 1$ characters. The *backward next* transition $(ABE, 1)$ indicates that a next transition on input characters ABE needs to be made, while consuming only one character A. The input characters consumed can be negative here; in which case we move the input pointer backwards (*Rollback Feature*). A formal proof of the worst case performance analysis is given in [34].

A formal description is given in Algorithm 2. Here again, we do a depth first search on the Aho-Corasick goto graph and create compressed DFA *next* transitions corresponding to all the *next* transitions of the visited state in the Aho-Corasick DFA. The runtime complexity of the algorithm is also $O(n)$.

Algorithm 2: Compressed DFA Next Function

```
dfsCompressedNext(st_o)
{
        parent_c = cParent(st_o)
        forall (a such that depth(next_o(st_o, a)) > m)
        {
                nxt = next_o(st_o, a)
                pNxt_c = cParent(nxt)
                backneed = 0
                for all sNxt_c ∈ skick(nxt)
                        backneed = cFwd(st_o, parent_c, nxt, sNxt_c)
                if (backneed == 1)
                        cBkwd(st_o, parent_c, nxt, pNxt_c)
        }
        for all a such that goto_o(st_o, a) ≠ null
                dfsCompressedNext(goto_o(st_o, a))
}
/* Forward next transition */
cFwd(st_o, parent_c, nxt, sNxt_c)
{
        len = dist(st_o, parent_c) + 1 + dist(nxt, sNxt_c)
        inp = (string(st_o) − string(parent_c)) + a
        inp = inp + (string(sNxt_c) − string(nxt))
        if (len ≤ TW and safe(nxt) == true)
        {
                next_c(parent_c, inp, len) = sNxt_c
                out = cout(parent_c, st_o, ) ∪ output_o(nxt)
                out = out ∪ cout(nxt, sNxt_c, )
                output_c(parent_c, sNxt_c, inp) = out

        } else
                backneed = 1

        return backneed

}
/* Backward next transition */
cBkwd(st_o, parent_c, nxt, pNxt_c)
{
        len = dist(st_o, parent_c) + 1 − dist(pNxt_c, nxt)
        inp = (string(st_o) − string(parent_c)) + a
        next_c(parent_c, inp, len) = pNxt_c
        if (len > 0)
        {
                out = cout(parent_c, st_o, len)
                if (len == depth(parent_c) − depth(st_o) + 1)
                        out = out ∪ output_o(nxt)
        } else
                out = null
        output_c(parent_c, pNxt_c, inp) = out

}
```

Example: Figure 1 and Figure 5 show the state transitions (both *goto* and *next*) for the Aho-Corasick DFA and compressed DFA respectively. Many *next* transitions to smaller depth states are eliminated by clever TCAM optimizations as explained previously. Thus, compressed DFA with TCAM optimization has only one *next* transition: to state 8 on input $EFGH$ consuming 4 characters, while the AC DFA originally had 46 *next* transitions.

Shallow State Computation: One problem with the compressed DFA we have constructed so far is that we may miss a pattern if the pattern does not start at a position of the input that is a multiple of k. For example, if $k = 4$ and the pattern is $ABCDEFGHIJKL$, there will be a transition

from state 0 to state 1 on the input $ABCD$. But if the input string is $XXABCDEFGHIJKL$, the compressed DFA will not recognize the pattern, as neither $XXAB$ nor $CDEF$ (i.e next set of four characters) have any transition from the state 0. Hence for the compressed DFA to work correctly, we need to "synchronize" the input and the compressed DFA.

Corresponding to each of the states in the Aho-Corasick DFA whose depth is less than k, we create a state in the compressed DFA. We call such states in the Aho-Corasick DFA *original shallow states* and in the compressed DFA *compressed shallow states*. In the compressed DFA, we put a transition from the root state to each of the compressed shallow state s_c as follows: Let i be the length of $string(s_c)$, Then $goto_c(0, ?^{k-i} + string(s_c)) = s_c$, where $?^{k-i}$ represents any $k-i$ characters, and $+$ symbol the concatenation operator. Also, we will add $goto$ transitions from s_c to the compressed states d_c corresponding to its nearest core states (i.e $goto(s_c, string(d_c) - string(s_c)) = d_c$). Figure 7 shows an example of the compressed DFA with new *shallow* states for the string $ABCDEFGH$ for $k = 4$.

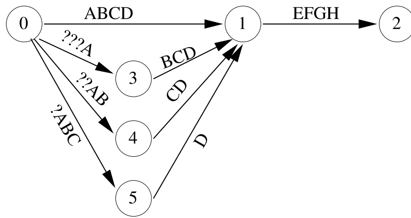

Fig. 7. Adding *shallow states* to the compressed AC DFA

Longest Input Match: Due to addition of shallow states, there might exist multiple transitions on the same input from a given state. For example if the pattern strings are $ABCDEFGH$ and $CDEFGHIJ$, on input $ABCDEFGH$, there are two matching transitions from the state 0: one for state $ABCD$ and the other for $??CD$. In those cases, transition to a state whose $string$ is of longer length takes the precedence. This can be implemented in hardware by placing higher precedence rules before the lower precedence rules, as explained later.

To reach the state corresponding to the first core state, from the root state, the compressed DFA will take either one direct transition or two transitions through an intermediate shallow state. In the former case k characters are consumed in one cycle and in the later case at least $k+1$ characters are consumed in two cycles. To increase the throughput, along the similar lines, we can create shallow states for each of the states of depth less than rk ($r \geq 1$) in the Aho-Corasick DFA. In that case we can get a throughput of $rk+1$ characters in $r+1$ cycles, if only the $goto$ transitions occur.

Compressed DFA Output Function: The Aho-Corasick DFA declares the occurrence of a pattern when it reaches certain states in the DFA. Hence, the matching patterns (output) is associated with the DFA states. Since we have only a subset of the original states in the compressed DFA, it is not always possible to associate the output with the states. Hence we associate the output function with the DFA transitions instead. Each transition in the compressed DFA corresponds to a path in AC DFA. If a pattern is output with any of the states in the path (excluding the start node, but including the end node) in the AC DFA, then we output that pattern with the corresponding transitions. Algorithms 1 and 2, along with creating the state machine and *goto*, *next* transitions, also compute the output functions simultaneously.

Precedence Rules for TCAM Entries: When there are multiple matches on an input, the TCAM reports only the first matching occurrence. Since we use the ternary (don't care state) of the TCAM for our (smaller depth) next transition memory optimizations and shallow states, the order in which the entries are stored in the TCAM contributes to the correctness of the execution of the pattern matching engine. The following order needs to be maintained. Entries for transitions from states that have fewer '?' should precede the ones having more '?'. Within entries having identical state identifiers, $goto$ transitions should precede the $next$ transitions. Also within both the $goto$ and $next$ transitions, the entries that have fewer number of '?' in input should precede the ones having more '?' in input.

Correctness Argument: We prove the correctness of our approach by showing the equivalence between the AC and compressed DFA. In the correctness proof, we show that only the strings that are reachable in the AC DFA, are reachable in the proposed compressed DFA, and there exists an one-to-one equivalence between the two DFAs. An outline of the correctness proof is presented in Theorem 1 (see Appendix). See [34] for the complete proofs of the same.

Regular Expressions: One of the most important applications of pattern matching can be found in virus signature matching in hardware for high speed applications. However, not all virus signatures are simple patterns, and are hence represented using regular expressions. A case-sensitive pattern matching can be implemented as discussed in [23]. Any regular expression S with wildcards, given by $s1???s2*s3$ (say), where $s1$, $s2$ and $s3$ are simple patterns, can be decomposed into three individual simple patterns. These simple patterns can then be implemented in the compressed AC DFA as if they were three different patterns. We now keep track of the different simple patterns and detect the existence of that regular expression, when these simple patterns occur at specific offsets as in the regular expression itself. The exact implementation details are presented in [34], based on efficient post processing stage implementation in [23], [25].

V. Simulation Results

In this section, we compare the Aho-Corasick algorithm with our proposed multiple string matching algorithm. We

evaluate the state space and transition space for the DFA in the two algorithms. We also evaluate the speed-up achieved and the total memory requirements for a TCAM implementation of the original and the proposed algorithms.

Virus/Worm Name	Virus/Worm Signature
Worm.CodeRed	8bf450ff9590feffff3bf490434b434b898534feffffeb2a8bf48b8 8d68feffff518b9534feffff52ff9570feffff3bf490434b434b8b8d 8d4cfeffff89848d8cfeffffeb0f8b9568feffff83c201899568feff f8b8568feffff0fbe0885c97402ebe28b9568feffff83c2018995
Trojan.URLspoof.gen	6c6f636174696f6e2e687265663d756e6573636170652827 *3a2f2f*25303140*2729
DOS.Aardwolf.446	0e1fb82135cd21891e????8c06????b821250e07babc00cd 21b44abb3c008e06????cd21e8

TABLE II

SAMPLE VIRUS/WORM SIGNATURES

We obtained a list of the 100 most widely reported viruses/worms in the Internet, as of April 2005, from *http://www.wildlist.org*. We then obtained signatures for these viruses/worms from CLAMAV, the widely used open-source anti-virus toolkit [35]. A few sample signatures are listed in Table II. These virus signatures show a great variation in the total length of patterns and also the number and type of wildcards found in them. We split the signatures with wildcards into simple patterns as explained previously. Table III and Figure 8 show the distribution of the individual lengths of the simple patterns in the signature database.

No. of Virus Signatures	95	No. of Simple patterns	117
Min. Pattern Size	19	Max. Pattern Size	150
Mean/Median Pattern Size	67.73 (64.00)	Standard Deviation	33.93

TABLE III

VIRUS SIGNATURE DATABASE STATISTICS

Aho-Corasick DFA: Table I summarizes the number of states and transitions to the states at different depths in the Aho-Corasick DFA. Figure 9 shows that the number of states are high closer to the root, and decrease as the depth increases. This is the result of the length distribution of the patterns. Figure 10 shows the number of transitions to the states at each depth in the DFA on a logarithmic scale. Note that most of the transitions are to smaller depths, and are mainly *next* transitions as noted previously in Section III.

Compressed AC DFA State Space: In Figure 11, we plot the number of states in the compressed DFA, as a function of the number of characters allowed in one transition (k). Let N and L represent the total number of simple patterns and their average length respectively. The first term in Equation 1 accounts for the *shallow states*, while the second term accounts for the remaining states in the compressed DFA. We see that k=1 is same as the Aho-Corasick DFA. For higher values of k, we find that the number of states required reduces initially before it raises again. This happens as one term increases with k, while the other decreases; and optimal value of k for any signature database can be trivially found.

$$States\ in\ compressed\ DFA \approx N \times (k-1) + \frac{N \times L}{k} \quad (1)$$

Compressed AC DFA Transition Space: Figure 12 shows the number of transitions in the DFA as a function of the number of characters allowed in one transition (k) for various values of m. We see a significant decrease in the number of transition even for m=1. For the Aho-Corasick DFA, the number of transitions were 437038, where as in our solution it is 15851 with k=1, m=1. There is significant reduction in the transition space for m=2 and 3. After a point (m=4) there is not enough reduction since the number of next transitions to states of higher depths are small. As m increases, we pay penalty in terms of bits needed to represent a state. So m=2 or 3 is ideal. As k increases, for the same m, the number of transitions first decreases and then increases. This can be explained with the same argument as that for number of states.

TCAM Memory Requirements: In Figure 13, we plot total TCAM memory requirements for all the transitions in the compressed DFA, for different values of k and m. Recall that TCAM entries consists of state id and the input for each transition. If we denote the number of states and transitions in the DFA by S and T respectively, then total TCAM memory requirements is given by Equation 2; the 3 terms corresponding to a unique state-id prefix, last m-1 byte suffix for the state-id, and input characters for that transition respectively.

$$TCAM\ Memory = T \times (log_2 S + 8(m-1) + 8k)\ bits \quad (2)$$

The commercially available TCAMs do not have a customizable width in order to allow for flexible usage. The TCAMs that are available in the market are 36, 72, 144, 288 or 576 bits wide. In Figure 14, the same graph is redrawn taking into consideration the available TCAM widths, by using a higher-width TCAM as and when required. We see that current memory technology does not permit us to operate this algorithm at really large values of the transition width. The total TCAM memory requirements of the Aho-Corasick algorithm for the given signature database assuming a TCAM implementation, is represented in the graphs as a horizontal line for comparison. Thus we see that a reduction in the number of states and transitions, not only provides for higher throughput but also lesser TCAM memory requirements.

Throughput (Search Speeds): We now focus on the most crucial aspect of our algorithm, the high throughput it can achieve. The Aho-Corasick algorithm can give only one character throughput per transition or clock cycle, roughly translating to 1 Gbps search speeds on a 250MHz TCAM. Comparatively, the proposed algorithm can achieve a significantly higher average case throughput. For our analysis, we generated multiple packet payload streams by randomly inserting the virus signatures into the test payload.

In Figure 15, we show the average number of transitions required for scanning all the streams generated, for varying values of the transition width in the DFA. The flat line on the top indicates the Aho-Corasick algorithm. In Figure 16,

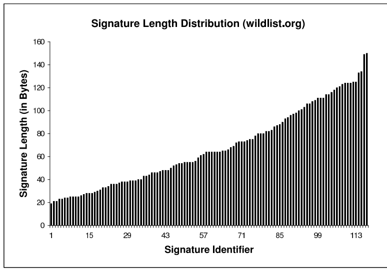

Fig. 8. Virus Signature Database Statistics

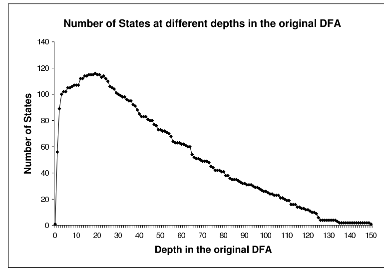

Fig. 9. No. of States (Aho-Corasick DFA)

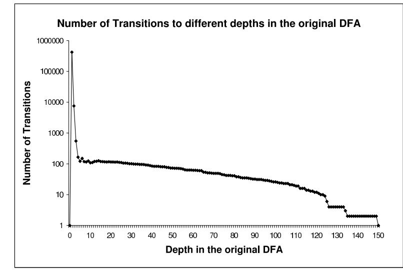

Fig. 10. No. of Transitions (Aho-Corasick DFA)

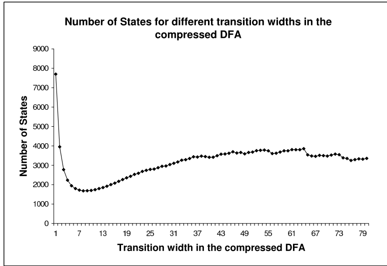

Fig. 11. No. of States (compressed DFA)

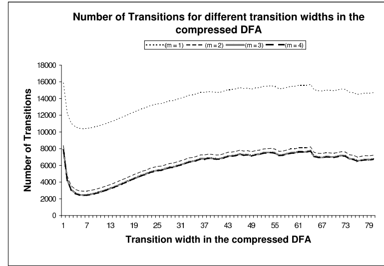

Fig. 12. No. of Transitions (compressed DFA)

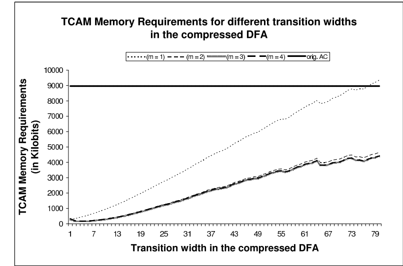

Fig. 13. Theoretical TCAM Memory

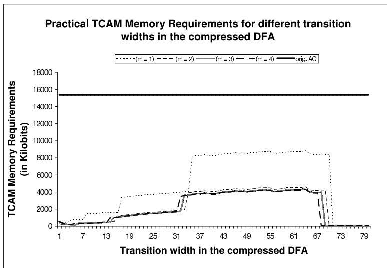

Fig. 14. Practical TCAM Memory

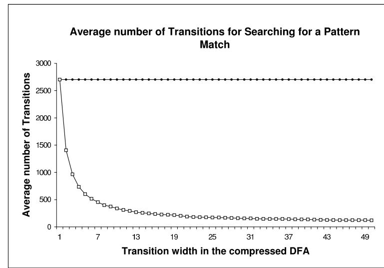

Fig. 15. Average No. of Search Transitions

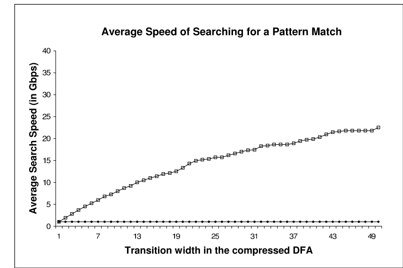

Fig. 16. Average Search Speed

the corresponding search speeds are displayed assuming a 250MHz TCAM implementation. The streams are not evaluated for higher transition widths because it would be practically impossible to achieve such a performance firstly due to limited memory sizes and access speeds today (see Figure 14) and also partly due to a large number of shorter patterns limiting the overall throughput of the system. Also note that the value of m does not matter in this case, as it achieves only memory optimization and does not provide any additional speed improvements. We thus see a significant reduction in the number of transitions or clock cycles, and hence higher search speeds when scanning packet payloads.

Additionally, we could use a parallel implementation processing multiple packet streams, and/or use packet sampling techniques to increase the cumulative throughput achieved. The performance evaluation done in this section did not assume the existence of any such performance enhancing techniques, although they could very well be used in conjunction with our proposed algorithm, to achieve even faster search speeds and scale beyond the immediate needs of 10Gbps in enterprise networks, and 40Gbps at the network core.

Comparison: We now compare our proposed algorithm with other known techniques in literature. Although it may not be possible to make a fair comparison of the different techniques without accounting for the various algorithmic and hardware-implementation aspects, we present these here to illustrate the bare performance of the proposed algorithm when compared with other known techniques.

Table IV shows the throughput of the different algorithms known, while Table V shows the memory utilization per signature database character of the different algorithms. The comparison methods used in [15] and [25] are used here for benchmark evaluation. We thus see that the proposed algorithm has a great potential to meet current day requirements of line speed network-based virus/worm detection.

VI. CONCLUSIONS

As virus/worm spreads and other network intrusion attacks increase in frequency and sophistication, the need for effective attack detection and prevention has increased enormously. The current state of the art in discovering and stopping these attacks is by inspecting packets being transported for known virus patterns or signatures, in packet payloads. The existing software and hardware solutions for deep packet content inspection however do not scale with data rates approaching 10Gbps and beyond. Additionally, packet inspection in the network is essential for various applications including QoS monitoring, bandwidth metering, stateful packet filtering etc.

Algorithm	Max. Speed	Comments
Aho-Corasick [8]	1 Gbps	Earliest known algorithm
Yu et. al. [23]	2 Gbps	TCAM implementation
Tuck et. al. [15]	8 Gbps	Memory optimized Aho-Corasick
Tripp et. al. [21]	10 Gbps	FPGA implementation
Aldwairi et. al. [22]	14 Gbps	SRAM implementation
Sugawara et. al. [20]	14 Gbps	Selective multi-character transitions
BFPM [25]	10 Gbps	20 Gbps ASIC implementations
Proposed Algorithm	1-20 Gbps	Faster FPGA/ASIC implementations

TABLE IV

THROUGHPUT COMPARISON

Algorithm	Memory/Char.
Aho-Corasick [8]	2.8 KB
Wu-Manber [19]	1.6 KB
Aldwairi et. al. [22]	126 B
Bitmap compressed Aho-Corasick [15]	154 B
Path compressed Aho-Corasick [15]	60 B
BFPM [25]	3-8 B
Yu et. al. [23]	3-4 B
Proposed Algorithm	4-48 B (varying with k,m)

TABLE V

COMPARISON OF MEMORY UTILIZATION PER CHARACTER

In this paper, we propose a novel multiple string matching algorithm that uses multi-character transitions on a finite state automata to increase the throughput, and also leverages a clever transition optimization technique to reduce the memory requirements. We describe a TCAM-based hardware architecture to realistically achieve these higher data rates for virus/worm detection employing signature matching. Our simulation results demonstrate that the proposed algorithm indeed scales well in practice to meet the current day requirements, as tested on real virus signature databases. We are currently developing a prototype of the proposed algorithm using network processors. We plan to evaluate the higher data rates that the proposed algorithm can achieve, in real hardware employing hardware threading and a pipelined architecture, and when deployed in a real network, as part of future work.

REFERENCES

[1] *http://www.wildlist.org*. The Wildlist Organization Intl., April 2005.

[2] R. Lemos, *http://www.alertsite.com/articles_cnet-feb-2004.shtml*. Performance Testing and Monitoring, Feb 2004.

[3] D. M. et. al., "Internet quarantine: Requirements for containing self-propagating code," in *INFOCOM*, 2003.

[4] L. Feinstein, D. Schnackenberg, R. Balupari, and D. Kindred, "Statistical approaches to ddos attack detection and response," in *DISCEX*, 2003.

[5] L. Spitzner, *Honeypots: Tracking Attackers*. Addison-Wesley, 2002.

[6] C. Morrow, *http://www.secsup.org/Tracking*. BlackHole Route Server and Tracking Traffic on an IP Network.

[7] *http://www.snort.org*. SNORT: Open-Source Network IDS/IPS.

[8] A. V. Aho and M. J. Corasick, "Efficient string matching: an aid to bibliographic search," *Commun. ACM*, vol. 18, no. 6, pp. 333–340, 1975.

[9] D. E. Knuth, J. H. M. Jr., and V. R. Pratt, "Fast pattern matching in strings," *SIAM J. Comput.*, vol. 6, no. 2, pp. 323–350, June 1977.

[10] R. S. Boyer and J. S. Moore, "A fast string matching algorithm," *Commun. ACM*, vol. 20, no. 10, pp. 762–772, October 1977.

[11] D. M. Sunday, "A very fast substring search algorithm," *Commun. ACM*, vol. 33, no. 8, pp. 132–142, August 1990.

[12] M. Crochemore and D. Perrin, "Two-way string matching," *J. ACM*, vol. 38, no. 3, pp. 650–674, 1991.

[13] R. M. Karp and M. O. Rabin, "Efficient randomized pattern-matching algorithms," *IBM J. Res. Dev.*, vol. 31, no. 2, pp. 249–260, 1987.

[14] Z. Galil and J. Seiferas, "Time-space optimal string matching (preliminary report)," in *STOC*, 1981.

[15] N. T. et.al., "Deterministic memory-efficient string matching algorithms for intrusion detection," *INFOCOM*, 2004.

[16] J. J. Fan and K. Y. Su, "An efficient algorithm for matching multiple patterns," *IEEE Trans. on Knowledge and Data Engineering*, vol. 5, no. 2, pp. 339–351, 1993.

[17] B. Commentz-Walter, "A string matching algorithm fast on the average," in *ICALP*, 1979.

[18] M. Fish and G. Verghese, "Fast content-based packet handling for intrusion detection," in *UCSD Technical Report CS2001-0670*, 2001.

[19] U. Manber and S. Wu, "A fast algorithm for multi-pattern searching," in *Tech.Report TR-94-17, CS Dept., University of Arizona*, 1994.

[20] Y. Sugawara, M. Inaba, and K. Hiraki, "Over 10gbps string matching mechanism for multi-stream packet scanning systems," in *FPL*, 2004.

[21] G. Tripp, "A finite-state machine based string matching system for intrusion detection on high-speed networks," in *EICAR*, 2005.

[22] M. Aldwairi, T. Conte, and P. Franzon, "Configurable string matching hardware for speeding up intrusion detection," *SIGARCH. Comput. Archit. News*, vol. 33, no. 1, pp. 99–107, 2005.

[23] F. Yu, R. Katz, and T. V. Lakshman, "Gigabit rate packet pattern matching using tcam," in *ICNP*, 2004.

[24] L. Tan and T. Sherwood, "A high throughput string matching architecture for intrusion detection and prevention," in *ISCA*, 2005.

[25] J. V. Lunteran, "High-performance pattern-matching for intrusion detection," *IEEE Infocom*, 2006.

[26] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. W. Lockwood, "Deep packet inspection using parallel bloom filters," in *Micro*, 2004.

[27] R. Sidhu and V. K. Prasanna, "Fast regular expression matching using fpgas," in *FCCM*, 2001.

[28] Z. K. Barker and V. K. Prasanna, "Time and area efficient pattern matching on fpgas," in *FPGA*, 2004.

[29] C. R. Clark and D. E. Schimmel, "Efficient reconfigurable logic circuits for matching complex intrusion detection patterns," in *FPL*, 2003.

[30] B. L. Hutchings, R. Franklin, and D. Carver, "Assisting network intrusion detection with reconfigurable hardware," in *FCCM*, 2002.

[31] I. Sourdis and D. Pnevmatikatos, "Pre-decoded cams for efficient and high speed nids pattern matching," in *FCCM*, 2004.

[32] M. Gokhale, D. Dubois, A. Dubois, M. Boorman, S. Poole, and V. Hogsett, "Towards gigabit rate network intrusion detection technology," in *FPL*, 2002.

[33] Y. H. Cho, S. Navab, and W. H. Mangione-Smith, "Specialized hardware for deep network packet filtering," in *FPL*, 2002.

[34] M. Alicherry, M. Muthuprasanna, V. Kumar, and V. Poosala, "Multi-character multi-pattern matching for high speed applications," in *Lucent Bell Labs Technical Report*, 2005.

[35] *http://www.clamav.net*. Clam AntiVirus toolkit for UNIX.

VII. APPENDIX

Theorem 1: The Aho-Corasick and compressed DFA are equivalent.

Proof: First, let us assume that there are no forward next transitions in the compressed DFA, i.e. we replace all the forward transition with backward transitions. Let $0, s_o^1, s_o^2, \ldots, s_o^n$ be the states visited in the Aho-Corasick DFA for the input $a_1 a_2 \ldots a_n$. For simplicity, we assume that s_o^n is a core state, but the claims hold even if s_o^n is not a core state. We claim that the same input in the compressed DFA will visit states $0, cParent(s_o^1), cParent(s_o^2), \ldots, cParent(s_o^n)$, uniquely in order. The claim is plainly evident to be true if there were only goto transitions in the state sequence. However, if there is any next transition in the state sequence, let s_o^i be the first state that has a next transition, i.e. $next_o(s_o^i, a_{i+1}) = s_o^{i+1}$. Let s_o^j $(i - k < j \le i)$ be the nearest core state of s_o^i. Then $cParent(s_o^j)$ does not have a goto transition on $a_{j+1} a_{j+2} \ldots a_i a_{i+1} \ldots$ and will take the backward next transition to $cParent(s_o^{i+1})$. Now continuing the argument from $cParent(s_o^{i+1})$, we can prove that the compressed DFA will take the state sequence $0, cParent(s_o^1), cParent(s_o^2), \ldots, cParent(s_o^n)$.

Now we prove that, the above claim is *true* even if there are forward next transitions, i.e. both forward and backward next transitions will take the compressed DFA to the same state, but with backward next transitions going through an intermediate state. Since the output patterns are associated with transitions and not the states, correctness is not compromised. Let s_o^i be the first state that has a next transition $(next_o(s_o^i, a_{i+1}) = s_o^{i+1})$ which corresponds to a forward next transition in the compressed DFA. s_o^{i+1} has to be a safe state, by definition of forward next transition. Hence in the original state sequence, there exists a state s_o^j $(i + 1 \le j < i + 1 + k)$ such that s_o^j is a core state and all the transitions from s_o^{i+1} to s_o^j are goto transitions. Hence, if s_o^l is the core state in the sequence preceding s_o^i, then for the input $a_{l+1} \ldots a_j$ the compressed DFA with forward next transition will take transition $cParent(s_o^l) \rightarrow cParent(s_o^j)$, whereas the DFA with backward next transition will take the transitions $cParent(s_o^l) \rightarrow cParent(s_o^{i+1}) \rightarrow cParent(s_o^j)$. Continuing the argument for all the next transitions, we prove the claim. Thus our proposed algorithm ensures correctness, if Aho-Corasick algorithm is correct, and this has been proved in [8].