This full text paper was peer reviewed at the direction of IEEE Communications Society subject matter experts for publication in the IEEE INFOCOM 2009 proceedings.

Variable-Stride Multi-Pattern Matching For Scalable Deep Packet Inspection

Nan Hua
College of Computing
Georgia Institute of Technology
nanhua@cc.gatech.edu

Haoyu Song
Bell Labs
Alcatel-Lucent
haoyusong@alcatel-lucent.com

T.V. Lakshman
Bell Labs
Alcatel-Lucent
lakshman@alcatel-lucent.com

***Abstract*—Accelerating multi-pattern matching is a critical issue in building high-performance deep packet inspection systems. Achieving high-throughputs while reducing both memory-usage and memory-bandwidth needs is inherently difficult. In this paper, we propose a pattern (string) matching algorithm that achieves high throughput while limiting both memory-usage and memory-bandwidth. We achieve this by moving away from a byte-oriented processing of patterns to a block-oriented scheme. However, different from previous block-oriented approaches, our scheme uses variable-stride blocks. These blocks can be uniquely identified in both the pattern and the input stream, hence avoiding the multiplied memory costs which is intrinsic in previous approaches. We present the algorithm, tradeoffs, optimizations, and implementation details. Performance evaluation is done using the Snort and ClamAV pattern sets. Using our algorithm, the throughput of a single search engine can easily have a many-fold increase at a small storage cost, typically less than three bytes per pattern character.**

I. Introduction

Multi-pattern matching is a central function needed in content inspection systems such as signature-based Network Intrusion Detection and Prevention Systems (NIDS/NIPS). Multi-pattern matching is complex to implement and an important system goal is to achieve fast, deterministic multi-pattern matching performance without prohibitively high memory-usage. A major focus of the research on multi-pattern matching has been on reducing memory usage. However, the continual increase in line rates makes it imperative to achieve high throughputs of 10Gbps+ as well. Predictable line-speed processing is necessary for handling real-time flows and consequently memory bandwidth has become an even more critical factor than memory size in building practical content inspection systems. We develop multi-pattern matching schemes that take into account both these critical resources of memory-bandwidth and memory-usage.

In this paper, we present a variable-stride multi-pattern matching algorithm in which a variable number of bytes from the data stream can be scanned in one step. We use the Winnowing idea [1] developed for document fingerprinting in a novel manner to develop a variable-stride scheme that increases the system throughput considerably while also decreasing memory usage by an order of magnitude over previously proposed algorithms.

The two traditional methods for increasing system throughput are pipelining and parallelism. The existence of back-tracking paths in the finite state automaton (FSA) limits the effectiveness of pipelining. Hence, the only viable option is to exploit inter-stream and intra-stream parallelism.

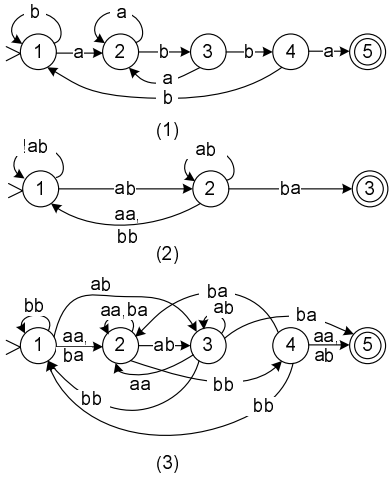


Fig. 1. Finite State Automaton for Pattern $abba$

Inter-stream parallelism uses multiple instances of the basic pattern-matching engine to process independent parts of the input stream in parallel. While this can increase system throughput, the use of multiple pattern-matching engines increases costs and introduces complexity in scheduling, buffering, ordering. Also, pattern matching tables may need to be replicated and hence increasing memory usage or very high memory-bandwidths are needed if tables are to maintained in memory shared by the parallel matching engines. Exploiting intra-stream parallelism judiciously can lead to improved performance at lower costs.

One way of exploiting intra-stream parallelism to increase throughput is to scan multiple bytes (a block) of the input data stream in each matching step. (Since a pattern can start or end at any offset in a block, it must be ensured that no patterns are unmatched because of the block-oriented processing.) For example, assume that the alphabet Σ is $\{a, b\}$ and a pattern $abba$ needs to match against an input stream. The basic one-character DFA is shown in Figure 1-(1). Clearly, if two characters can be scanned in one step (i.e. the stride s is two), the system throughput is doubled. Previous work achieves this goal using two basic schemes.

In the first scheme, patterns are divided into s-byte blocks

978-1-4244-3513-5/09/$25.00 ©2009 IEEE

and the blocks are used to construct the DFA. This results in a DFA with fewer states and transitions. However, for Pattern matching, s instances of the DFA need to run in parallel, each accepting the same input data stream with an one-byte offset (to ensure that no patterns are overlooked). In the above example, one can build a DFA as shown in Figure 1-(2). If the input stream is $babbaba...$, the block sequence $|ba|bb|ab|...$ and its one-byte shifted version $|ab|ba|ba|...$ both need to be scanned in to ensure that no possible match is missed. With this approach, higher throughputs are achieved at the expense of higher memory-bandwidth usage (due to running s instances of the matching engine in parallel) and the memory-bandwidth needs grow in proportion to the block size s.

Alternatively, one can build a single DFA for which the transitions account for all the possible s-byte patterns that can occur in the stream. By using a larger DFA, a single instance of the matching engine can be used to scan the input data stream without the possibility of missed matches. Figure 1-(3) shows the corresponding DFA for the example. The throughput gain is now at the cost of higher memory usage, rather than higher memory bandwidth. The number of transitions from any state can be as large as $|\Sigma|^s$. Indeed, for NIDS signature sets which use the ASCII alphabet, memory usage becomes prohibitively high even for a block size of two bytes.

In sum, the conventional block-oriented approaches that exploits intra-stream parallelism would inevitably incur much higher memory costs. Our goal in this paper is to develop a multi-pattern matching algorithm that can improve system throughput multi-fold without incurring prohibitively high memory-costs.

For better understanding of our algorithm, we draw on an analogy to how text is read. We do not process the text character-by-character but instead process it in larger natural units delimited by punctuation or other markers. For pattern matching, a conventional approach is to view a byte as an atomic unit because of the lack of simple delineation methods that can be used to segment data into coarser-grained multi-character atomic units. However, with some novel preprocessing we can delimit patterns and the input stream uniquely and unambiguously into variable sized multi-byte blocks. Pattern matching can then be done on a DFA that uses these blocks (or atomic units) as new alphabet symbols for state transitions. A greater-than-one average size for these atomic units results in considerable gains in pattern matching speeds. This method also reduce the number of DFA states and state transitions, since each state transition now spans more than one byte.

To generate good variable-sized blocks from the data in an unambiguous manner, we use the Winnowing scheme [1] originally developed for document fingerprinting. Other fingerprinting schemes can be used as well but the Winnowing scheme has properties that are desirable for our purpose. However, since the Winnowing algorithm was proposed for a different application, we need to modify it for our pattern and data-stream block-generation needs. We use the modified scheme to develop a new variable-stride multi-pattern matching algorithm, VS-DFA. Our algorithm can be thought as a method that exploits the intra-stream parallelism to accelerate the pattern matching speed. The VS-DFA algorithm has the advantage of achieving higher speeds without increased memory usage. On the contrary, VS-DFA increases throughput while decreasing memory usage.

II. Related Work

A. DFA-based Pattern Matching

Our algorithm, like many other multi-pattern matching algorithms, is based on the Aho-Corasick algorithm [2]. One or more DFAs are built using the pattern set and pattern matching is done by traversing the DFAs.

A large body of previous work has been focused on reducing memory-usage for pattern matching. For byte-at-a-time processing, there is very little scope for reducing the number of states. However, the number of state-transitions (which can take a lot of memory to store) can be reduced considerably. For the ASCII character set, a naive DFA implementation would allocate 256 transition pointers (one for each character in the alphabet) to each state even though most of the inputs do not have a valid next state other than the initial state. Use of a bitmap plus only the necessary pointers can efficiently remove this redundancy [3]. When the characters actually used are only a subset of the ASCII character set, a translation table can be used to compress the ASCII character table to a smaller table that contains only the characters that are used [4]. Another proposed approach splits each character into multiple slices and builds a DFA for each slice [5]. This can reduce overall memory usage because of a reduction in the fanout in the DFA states. The tradeoff is that multiple DFAs need to be traversed in parallel for pattern matching and this increases the memory bandwidth needs. In [6], [7], it is observed that a large fraction of state transitions are to the initial state or to states which are one-transition away from the initial state. Memory reduction can be achieved by eliminating these transitions from the DFA and instead using a 256-entry on-chip table to keep track of them.

The algorithms in [8], [9] and [4] all exploit intra-stream parallelism, to improve throughput, with different memory-bandwidth storage tradeoffs. Unnecessary off-chip accesses are eliminated in [8] using Bloom Filters. TCAMs are used in [10] and [11] to speed up pattern matching. Interestingly, they achieve high throughputs with the exactly same tradeoffs as the aforementioned SRAM-based algorithms. As an example of inter-stream parallelism application, [12] uses multi-processors to implement parallel searches on multiple input data streams.

For memory-efficiency, the state-transition table is implemented as a hash table rather than as a direct lookup table. This entails some unpredictability in pattern-matching speeds because of the possibility of hash-table collisions. Many recent developments, which permit near perfect hashing using reasonable storage, alleviate this problem [13], [14], [15], [16], [17]. Among these new hash table designs, the hash table most suitable for our application is the Cuckoo hash table [17] which can resolve queries deterministically in one

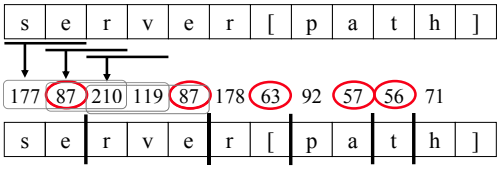


Fig. 2. Winnowing with $k = 2$ and $w = 3$

cycle. Moreover, memory utilizations as high as 89% can be achieved with just two tables and two items per entry [18].

B. Winnowing Algorithm

The Winnowing algorithm [1] is a document fingerprinting scheme that was proposed for detecting copied content. A nice feature of this algorithm is position independence which ensures that adding some new text to a file will still result in all the original matches between two files to be preserved. This property is useful for our pattern matching application as we explain later.

The algorithm works as follows. First, calculate the hash value of every consecutive k characters. A stream of l characters generate $l - k + 1$ hash values. Second, use a sliding window of size w to select the minimum hash value in the window. A tie is broken by selecting the rightmost minimum value. While these selected values are used as the fingerprint of the data stream in the original paper, they are used as delimiters to segment the flow into blocks in our algorithm. Figure 2 shows an example, where the hash value is in the range 0 to 255. We can see that the block size s is bounded by w.

For use in pattern matching, we need to modify the Winnowing algorithm in several ways.

Because patterns can be short, we need to keep the block size s relatively small in order to get meaningful number of blocks from a pattern. We will show that we need to segment a pattern into at least three blocks for our basic algorithm to work. Therefore, k and w cannot be large. On the other hand, a larger block size s is better for the throughput, so we also want to make w as large as possible. As a tradeoff, k is chosen to be between 1 and 2, and w can range from 3 to 8.

Because the hash window is small, we use simple hardware to realize hash functions and do not use the Rabin fingerprinting used in [1]. Moreover, since our goal is to simply segment the data stream rather than to retain the fingerprints we do not need large hash values. Eight bits or even less are sufficient for our purpose. This is an advantage for efficient and fast hardware implementation.

As shown in Figure 2, we align the hash value of a hash window with the first character in the window, and set the delimiters to be after the minimum hash value that is selected. This guarantees that the size of any block is never longer than w, when $k \leq w + 1$.

III. Variable-Stride DFA

The Winnowing algorithm has a "self-synchronization" property that is very useful for our application, which ensures that irrespective of the context in which a pattern appears in the input stream, it is always segmented into the same sequence of blocks (with some head and tail pattern exceptions). The key idea in VS-DFA is to use these unique blocks as atomic units in the construction of a DFA for pattern matching.

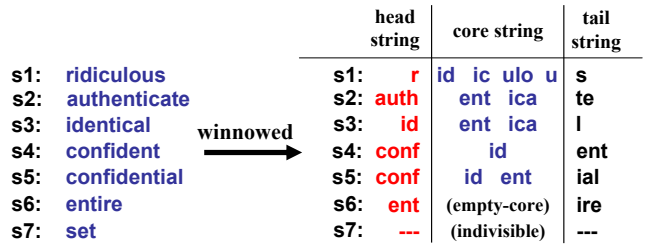


Fig. 3. Segment Pattern into Head/Core/Tail Blocks ($k = 2, w = 4$)

A. Segmentation Scheme Properties

As a pre-processing step, all patterns are first segmented into a sequence of blocks. The procedure for this pre-processing step is described in Section II-B. We call the first block as the head block and the last block as the tail block. All other blocks in between are called core blocks. Pattern segmentation examples are shown in Figure 3.

Some short patterns may have only one delimiter (i.e. there is no core block and the pattern consists of only the head and tail blocks) or no delimiter at all. These two types of patterns are called coreless patterns and indivisible patterns respectively. In Figure 3, *entire* is a coreless pattern and *set* is an indivisible pattern.

Our segmentation scheme has the following properties:

Property 1: The size of any segmented block is in the range $[1, w]$. Tail block sizes are in the range $[k - 1, w + k - 2]$, indivisible pattern sizes are in the range $[1, w + k - 2]$ and coreless pattern sizes are in the range $[w + k - 1, 2w + k - 2]$.

Moreover, the segmentation scheme has the following property which is key to guarantee the correctness of our algorithm.

Property 2: If a pattern appears in a data stream then segmenting the data stream results in exactly the same delimiters for the core blocks of the pattern.

The correctness of these properties can be directly inferred from the segmenting procedure described in Section II-B. Note that the head and tail blocks of a pattern may have extra delimiters when they appear in a data stream, because the head block can be affected by the prefix and the tail block can be affected by the suffix. However, the core blocks are totally confined to the pattern and isolated from the context. These unchanged core blocks can be used to narrow the search in pattern matching.

We illustrate this using an example. Suppose that the pattern S_2 appears in a data stream. Depending on the $w - 1 = 3$ characters that appear immediately before S_2, the head block $auth$ can possibly be segmented into $a|u|t|h$, $au|th$, and so on. Likewise, the segmentation of the tail block te is determined by the following $w - 1 = 3$ characters. However, it is guaranteed that there will be a delimiter after character h, because all the characters that are needed to make the decision

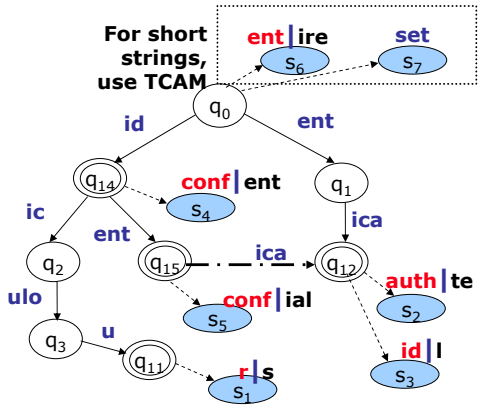


Fig. 4. Example of a VS-DFA

are already known. This is also true for the following two delimiters, so we will still get the two blocks ent and ica.

The core blocks of a pattern are invariant, as noted earlier, and this permits fast pattern matching. For example, even when we have matched only two blocks en and $tica$ consecutively we know that the match to S_2 is impossible without any further investigation. On the other hand, if we have matched two block ent and ica consecutively, there is potentially a match to S_2. To verify the match, all we need to do is to retrieve the w-byte prefix characters and $(w + k - 2)$-byte suffix characters in the data stream to compare against S_2's head block and tail blocks.

B. Finite Automaton Construction

Our VS-DFA construction follows exactly the same procedure as the Aho-Corasick DFA construction algorithm [2]. We extract the core blocks of patterns and use them as alphabet symbols to build the DFA. Hence, the key feature of VS-DFA is that its state transitions are based on core blocks rather than individual characters. When all the core blocks match, we still need to compare the head and tail. The construction of the DFA ensures that all core block sequences can be detected with one state transition per block. The "quasi-match" states, if hit, lead to a final verification step.

Figure 4 presents an example of the state transition graph for the patterns given in Figure 3. The DFA contains seven states. q_0 is the initial state. q_{11}, q_{12}, q_{14}, q_{15} are the quasi-matching states. To avoid clutter, we do not show the failure transitions that point to the initial state.

The correctness of VS-DFA is guaranteed through the Property 2. As long as one pattern appears in the data stream, its core blocks should appear in the segmented data stream and hence the constructed DFA would definitely capture those and then check head and tail. The algorithm to construct VS-DFA is fundamentally the same as all conventional DFAs to detect symbol sequences. The only difference is that some paths could be removed due to the prior knowledge here, which would not harm the correctness. For the sake of length, we only describe the difference in detail as follows .

The constructed DFA works as if the matching is started directly on the pattern body without any knowledge of the head and tail blocks. However, when constructing the DFA, we do have this knowledge and we use it to remove some unnecessary state transitions.

In the classical DFA, if the path $q_i \rightarrow q_j$ consumes the same character string as the path $q_0 \rightarrow q_k$, all the transitions starting from q_k need to be copied to q_j, unless the transition has already been defined on some forwarding path. This is because in this case the string covered by the shorter path is the suffix of the string covered by the longer path, and the longer path can possibly lead to the matches to the same patterns as the shorter path does. The jump transitions are therefore constructed to resolve the overlapping between patterns. This can be seen in Figure 3 where we need a jump transition from q_{15} to q_{12}, because the core block sequence $|id|ent|$ overlaps with core block sequence $|ent|ica|$.

However, in VS-DFA, the transitions are copied only when the head on the path $q_0 \rightarrow q_k$ matches the entry path of q_i. In our example, the head blocks on the path $q_0 \rightarrow q_{12}$ include $id|$ and $auth|$. One of them matches the entry path of q_1 (i.e. $id|$), hence the transition is needed. If we remove the pattern $id|ent|ic|al$ from the DFA, the transition $q_{15} \rightarrow q_{12}$ becomes unnecessary and can be safely removed.

As a special case of the above, in classical DFA construction, all transitions from q_0 should be copied to all other states, unless the transition is defined on some forwarding path. These transitions are also known as restartable transitions. In VS-DFA, the restartable transitions are only generated when necessary. In our particular example, there is no restartable transitions at all.

We should point out that in a large VS-DFA, the number of restartable transitions can also become large and can increase memory needs. Fortunately, the total number of "start transitions" from q_0 is small. As proposed in [6], [15], we could store the "start transitions" in a separate on-chip "start table" which is searched in parallel with the VS-DFA. Then all the restartable transitions can be removed. Here, if we cannot find an outgoing transition from a state, we resort to the search result from the start table to figure out another path.

C. System Design and Basic Data Structure

From the hardware perspective, a VS-DFA pattern matching system consists of two modules: the Winnowing module and the state machine module, as shown in Figure 5. The incoming data stream (packet payload) is first segmented into variable sized blocks by the Winnowing module and pushed into a First-In-First-Out (FIFO) queue. Then the blocks are fetched from the queue and fed into the state machine one by one. The state machine runs the VS-DFA and outputs the matching results.

As discussed in Section II-B, we prefer to use a small hash window size k, winnowing window size w, and a hash range yielding the best performance. The circuit for the winnowing module is very simple and can easily process more than w bytes per clock cycle. Since the state machine may consume

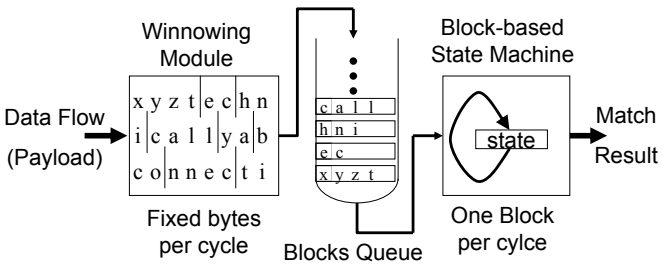


Fig. 5. System Overview

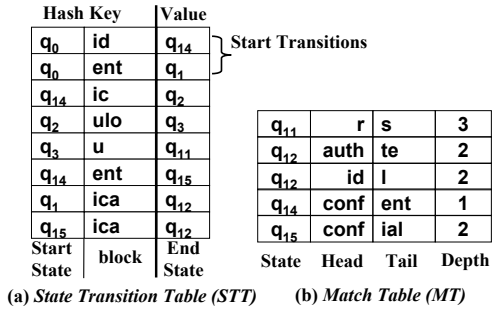


Fig. 6. State Transition Table and Match Table

less than w bytes per clock cycle, we need to use the FIFO as a buffer. The system throughput is hence determined by the state machine.

In the state machine module, in addition to the start table, we need two more tables as support data structures – the State Transition Table (STT) and the Match Table (MT). These are shown in Figure 6.

The STT is essentially a hash table. The hash key is the combination of the current state and the input block. The hash value is the address of the next state. In each cycle, if the queried combination is found, the next state stored in the entry is retrieved; otherwise, the next state is obtained from the start table. As presented in Section II-A and SectionV-A, the newest Cuckoo Hashing result could resolve query in deterministically one cycle while achieving high load factor at the same time. Hence the solution is feasible.

Since our transition table is hash-table based, the allocation of state numbers (also the address) is generally arbitrary. However, for the quasi-match states, we pre-allocate a specific consecutive range for them, since we want to directly calculate the corresponding entry's address in the Matching Table. For example, in Figure 4(b) we pre-allocate range q_{1*} for the matching states. State q_{1a} is corresponding to the a^{th} MT entry. A special case is that q_{12} is associated with two possible head/tail pairs. Hence q_{12} is allocated two MT entries and the state machine needs to check the entries linearly for possible matching. q_{13} is deliberately not allocated to reserve the third MT entry for q_{12}. Section V-D would present more discussion on this type of special case.

To enable match verification on the Quasi-match states, we need to maintain a Head Queue (HQ) that remembers the

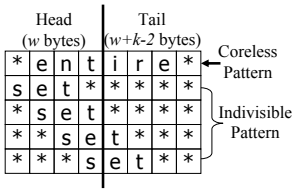


Fig. 7. Using TCAM for Short Pattern Lookups

Block-matching history. The HQ is implemented as a circular buffer with D entries, where D is the length of the longest forwarding path of the VS-DFA. Because the exact length of the head block cannot be known in advance, the size of each queue entry is set to w bytes. If a block is shorter than w bytes, we extend the block with its preceding bytes to make up. Hence the HQ size is $D \times w$ bytes and it guarantees that enough history is held for future reference. The depth field in the MT determines which entry in the HQ is to be retrieved as the head. It can be calculated by subtracting the current HQ index with the depth value.

D. Short Pattern Handling

From Property 2 in Section III-A, for patterns of length $l \geq 2w+k-1$, it is guaranteed that there is one core block for DFA construction. However, when $w+k-1 \leq l \leq 2w+k-2$, the pattern becomes a coreless pattern which has only head and tail blocks. Patterns with length $l \leq w+k-2$ have no delimiter at all and become indivisible patterns. VS-DFA cannot handle these short patterns directly because of the lack of core blocks.

However, these patterns can be handled by a small TCAM efficiently. Ideally, the TCAM should have the same throughput as the state machine (i.e. one block per lookup cycle). To achieve this goal, the conventional method has to duplicate the pattern multiple times, each with one byte shifted. However, Our pattern segmentation scheme allows us to use only one TCAM entry for Storage of a coreless pattern.

The TCAM entry is $2w+k-2$ bytes in width. Each entry is partitioned into two sections, which are w bytes and $w+k-2$ bytes, respectively. To store a coreless pattern, we align its head block to the right of the first section and its tail block to the left of the second section. The unused bytes in the entry are masked as "don't care".

No such optimization can be done for e indivisible patterns and $max\{w, w+k-2\}$ TCAM entries are needed for each indivisible pattern. Each entry covers a possible shift were this pattern can appear in the data stream. The first entry is left aligned with the TCAM entry. Each following entry shifts one byte toward the right side.

An example is shown in Figure 7. In this example, since $w = 4$ and $k = 2$, the TCAM entry is 8-byte wide. The coreless pattern $entire$, segmented into $ent|ire$, is stored in TCAM entry 1 as shown. The indivisible pattern set is spread in 4 TCAM entries.

With this arrangement, to perform matches we still segment the input stream as usual. We add one extra delimiter at the end of the byte stream. At each delimiter in the byte stream,

we extract w bytes before the delimiter and $w+k-2$ bytes after it. We then use the combined string as a key to query the TCAM. We are guaranteed to find the correct matching for any short pattern.

Short pattern are typically only a small fraction of the pattern set. SectionV-B shows that the size of the TCAM for either Snort or the larger ClamAV pattern set is only a few tens of kilobytes. Actually the winnowing method also contributes here since only one entry is enough for coreless pattern.

IV. Algorithm Optimizations

A. Reducing Single-Byte Blocks

VS-DFA's advantage in throughput is due to the large average stride per DFA traversal step. For a random data stream, the expected block size is $\frac{w+1}{2}$ [1]. However, it is possible to generate specific inputs that result in only single-byte streams being produced independent of the chosen hash functions and window parameters. For example, one can simply send a packet with any repeated single character. This can cause reduction in systems throughput and also be an avenue for a Denial-of-Service (DoS) attack to the system.

A low entropy data stream would also be problematic for the Winnowing algorithm. In [1] a method to break the tie among these same hash values was proposed but this is not suitable for our application. Here we generalize the method to alleviate the problem of single-byte blocks appearing in data streams:

Combination Rule 1 (applied on data streams): If more than one consecutive single-byte blocks appear in a data stream, then starting from the first block, we combine every w single-byte blocks to form a new w-byte block. The remaining blocks, if more than one, are also combined. For example, suppose $w = 4$ and the consecutive single-byte blocks are $|c_1|c_2|c_3|c_4|c_5|c_6|$ (the block before c_1 and the block after c_6 are longer than one byte). After applying the rule, we get two combined blocks $|c_1c_2c_3c_4|c_5c_6|$.

Combination Rule 1 eliminates all consecutive single-byte blocks, leaving only some isolated single-byte blocks sitting between multi-byte blocks. In the worst case, where the single-byte blocks and double-byte blocks interleave with each other in a data stream, our algorithm still gives a speedup of 1.5.

Clearly, a similar combination must also be performed on the patterns. However, blindly combining the single-byte blocks in patterns may cause the segmentations on streams and patterns to lose their synchronization. The ambiguity happens when the first or the last several core blocks are single-byte blocks. Given different contexts, i.e. the prefix and suffix bytes, the combination may lead to different results. To ensure algorithm correctness, if the first one or more original core blocks are single-byte blocks, they are no longer regarded as core blocks. Instead, they are pushed out as part of the head block. Likewise, special handling is also needed for the single-byte blocks neighboring the tail. single-byte core The Combination Rule 1 applied on patterns is defined as:

Combination Rule 1 (applied on pattern): Combine all consecutive single-byte core blocks right after the original head block into the new head block. Combine the rest core blocks in the same way as *Combination Rule 1 (applied on data streams)*. After the combination, if the last core block is a combined one and it is shorter than w bytes, combine it into the tail block.

We use an example to illustrate the *Combination Rule 1 (applied on patterns)*. Assume pattern $aaaaacbddddddabc$ is originally segmented as $aaa|a|a|cbd|d|d|d|d|d|abc$ with $w = 3$. The first two single-byte blocks $|a|a|$ should be combined with the head block. For the remaining blocks, the first three consecutive $|d|$s are combined into $|ddd|$ and the last two consecutive $|d|$s are combined into $|dd|$. Since the block $|dd|$'s length is shorter than 3, it is combined with the tail block. Hence the new segmentation after applying the rule becomes $aaaaa|cbd|ddd|ddabc$. If, in another case, the same pattern is originally segmented differently as $aaa|a|a|cbd|d|d|d|d|d|ab|c$. After applying the combination rule, the new segmentation should be $aaaaa|cbd|ddd|dd|ab|c$. The tail block remains the same because the last core block $|ab|$ is not a result of combination.

Combination Rule 1 improves both the worst-case and the expected throughput. The downside of the combination is that patterns may have longer head and tail blocks. Moreover, some patterns may even become coreless patterns if all their core blocks are single-byte blocks before applying the rule. After the combination, the maximum tail block size can be $(w + k - 2) + (w - 1) = 2w + k - 3$ bytes, since at most $w - 1$ single-byte blocks can be merged into the tail block. As for the head block, there is now no upper bound on the size.

Since our algorithm needs to buffer the potential head blocks from the data stream for later matching verification, the size of the head blocks must be constrained. Fortunately, the number of patterns with their head block longer than w bytes is small. We resolve the issue by replicating the pattern w times. Each replicated pattern is given a unique segmentation on the head block. The head block segmentation procedure is as follows: the first delimiter is set after the i-th byte, where $1 \leq i \leq w$; after the first delimiter, a new delimiter is set after every w bytes. When the segmentation is done, the first block is kept as the new head block and all the others are promoted to be core blocks. Each replicated pattern is then programmed in the VS-DFA as a new pattern, although they still lead to matching the same pattern. For example, suppose the pattern $c_1c_2c_3c_4c_5c_6c_7c_8c_9c_{10}c_{11}$ is initially segmented as $c_1|c_2|c_3|c_4|c_5|c_6|c_7|c_8c_9c_{10}|c_{11}$ with $w = 3$. After applying the Combination Rule 1, it becomes $c_1c_2c_3c_4c_5c_6c_7|c_8c_9c_{10}|c_{11}$. The new head block is even longer than $2w$ bytes. To constrain the length of the head block, we replicate the pattern three times and each receives a new segmentation as follows: $c_1c_2c_3c_4|c_5c_6c_7|c_8c_9c_{10}|c_{11}$, $c_1c_2c_3c_4c_5|c_6c_7|c_8c_9c_{10}|c_{11}$, and $c_1c_2c_3c_4c_5c_6|c_7|c_8c_9c_{10}|c_{11}$. Now no matter in which context the pattern appears, it is guaranteed to be caught by one of the w segmentations. In summary, for the patterns with big head block after applying the Combination Rule 1, we constrain the head block within $2w$ bytes at the cost of replicating pattern w times. Actually

we could constrain the maximum head block size even smaller while it will cause more replications.

B. Eliminating Single-Byte Blocks

Combination Rule 1 can eliminate all the consecutive single-byte blocks. However, the remaining isolated single-byte blocks may still slow down the processing. We have shown that the worst-case throughput is 1.5 characters per step. In this section, we propose Combination Rule 2 which can eliminate almost all single-byte blocks.

Combination Rule 2: After applying Combination Rule 1, combine every remaining single-byte block into its preceding block. For patterns, this only applies to core blocks. Also, since the first byte of the tail block might become a single-byte block in a data stream, we need to replicate the pattern once and make a new segmentation to cover this case.

For example, the pattern $c_1c_2c_3|c_4c_5|c_6|c_7c_8|c_9c_{10}c_{11}$ becomes $c_1c_2c_3|c_4c_5c_6|c_7c_8|c_9c_{10}c_{11}$ after applying Combination Rule 2. However, in a data stream the tail block of the pattern might be segmented as $c_9|c_{10}c_{11}$, so the Combination Rule 2, when applied on data stream, can also result in the segmentation $c_1c_2c_3|c_4c_5c_6|c_7c_8c_9|c_{10}c_{11}$. Hence the two pattern segmentations should both be inserted into the VS-DFA. However, we should point out that in some cases, this replication is not necessary. For example, for the above pattern, if $k = 2$ then we can calculate the winnowing hash value at the position of c_9 and c_{10}. If the first value is greater than the second, there will be certainly no delimiter between i and j, regardless what characters appears after the pattern. In this case, we do not make the replication.

Combination Rule 2 prevents the DoS attack problem and potentially increases system throughput. It also has other impacts on the system. The maximum block size now becomes $w + 1$ bytes. The patterns with long head blocks after applying Combination Rule 1 should be replicated $w+1$ times accordingly.

Although the combination rules may lead to some pattern replications, they do not necessarily increase memory consumption of the VS-DFA. On the contrary, they may actually help to lower memory consumption since single-byte state transitions are avoided. The impact of these rules is evaluated in Section V-C.

C. Reducing Memory Space

Similar to [7], the majority of states in VS-DFA have only one next state, which is called "linear trie" in [7]. Here we adopt a similar technique to save memory. We re-order the states and store the states linearly in "Main_STT" table, as shown in Figure 8. Unless the state is a "match state", its naturally next state would be the next entry in "Main_STT" table. All other transitions not covered in "Start_STT" and "Main_STT" table would be placed in the "Jump_STT" table. Since there is no need to store state pointers in 'Main_STT's entry, much memory is saved.

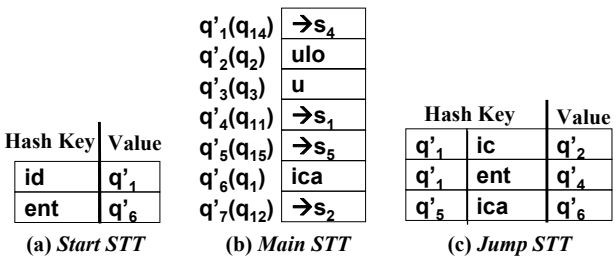


Fig. 8. Three STTs Design

TABLE I
CHARACTERISTICS OF PATTERN SETS

Pattern Set	*# of Patterns*	*# of Characters (KB)*
Snort-fixed	7,524	155
Snort-full	8,444	162
ClamAV-fixed	33,240	2,045
ClamAV-full	94,514	8,726

V. PERFORMANCE EVALUATION

We evaluate the memory-usage and throughput of the proposed algorithm. We take into consideration both the size of the SRAM, used for the VS-DFA data structure, and the size of the TCAM, used for short patterns. The measurement of bytes per pattern character is used to evaluate memory efficiency and scalability. The throughput is measured by the number of bytes per lookup step.

The pattern sets are extracted from the latest Snort [19] and ClamAV [20] datasets, as of July, 2008. Their basic characteristics are shown in Table I. The term "fixed" here refers to patterns extracted from the fixed string rules, and "full" refers to the expanded pattern sets that also include the fixed strings extracted from the regular expression rules.

A. Selection of Data Structures

This section presents the results pertaining to the selection of different data structures, as described in Table II(a). The selection of data structures is orthogonal to selection of other parameters of the algorithm.

In Table III, we set $k = 2$ and the hash value $h \in [0, 255]$. No single-byte combination rule is applied. We choose $w = 4$ for Snort and $w = 6$ for ClamAV, because the patterns in ClamAV are generally longer than the patterns in Snort. The *Compiled Chars* column shows the total bytes of patterns that are compiled in the VS-DFA. About 5% of the characters from the short patterns in Snort are ruled out. Less than 1% is for ClamAV. The *States* column shows the total number of states in the VS-DFA. The *Trans* column shows the total number of

TABLE II
CONFIGURATION OF VS-DFA SETTING

(a) Data Structures

Single_STT	All transitions in one SST
Start_STT	Separate Start Transition Table and Main Transition Table
Three_STTs	Separate Start Transition Table, Main Transition Table and Jump Table

(b) 1-byte-block Combination

No_Combo	Not Apply 1-byte-block Combination Rule
Combo_1	Apply 1-byte-block Combination Rule 1
Combo_2	Apply 1-byte-block Combination Rule 1 and 2

state transitions. The data show that with the help of the *Start STT*, the total number of transitions is reduced 75% to 87.5%.

We store the transition tables in hash tables. The memory size is calculated as the product of the number of transitions, the table entry size, and the hash table load factor. Recent research results show that a collision-free hash table can achieve a high load factor of 89% [18]. The scheme, called Cuckoo Hashing, uses two sub hash tables and packs two items per table entry. The load factor can approach 100% with the tradeoff of higher memory bandwidth. In our evaluation, we assume the load factor to be 89%.

In summary, the *Three_STTs* design achieves very high memory efficiency by consuming only 2.5 Bytes per character. The memory efficiency of the *Two_STTs* design is 3.5B/char, which is also very compelling. In the rest of this section, we only present results for the *Two_STTs* case.

B. Effect of Winnowing Parameters

This subsection investigates the effect of different Winnowing parameter configurations on the algorithm performance. The parameters include the hash window size k, the Winnowing window size w. All our simulation uses 8-bit hash value for winnowing, i.e. hash value range $[0, 255]$.

Generally, a larger Winnowing window w lead to higher throughput and smaller memory consumption. However, the larger Winnowing window also means more patterns would be left out from the VS-DFA as indivisible or coreless patterns. Moreover, larger w also means larger TCAM entry size.

Figure 9 shows that the TCAM size rapidly increases and the SRAM size rapidly decreases, as the value of w increases from 2 to 8. The figure also shows the effect of k. A smaller k can significantly reduce the TCAM size and slightly increase the SRAM size.

Although $k = 1$ can greatly reduce the required TCAM, we prefer to use $k = 2$ because $k = 1$ fails to provide enough randomness for block segmentation.

C. Effect of Single-byte Block Combination

The single-byte block combination technique is a double-edged sword as is evident from Figure 10. On the one hand, it can improve both the average-case and the worst-case throughput, and reduce the VS-DFA states and transitions. On the other hand, it also expands the head/tail block sizes, and also increases the number of patterns that need to be processed by the TCAM.

Next, we present results obtained after applying Single-byte Block Combination.

Table IV and Table V show the comparison of the SRAM and TCAM consumption after applying the single-byte block combination rules. Here Mem_1 denotes the memory consumed by "Start_STT" data structure and Mem_2 denotes that for "Three_STT".

We observe that the memory consumption drops somewhat due to the dual effect of Combination Rules, i.e. increasing entry size while reducing states and transitions. However, the SRAM consumption increases almost fourfold, due to both the doubling of TCAM entry size and the increase in indivisible patterns.

TABLE IV
APPLYING COMBINATION RULES ON SNORT SET

Combo_Rule	*w*	*States*	*Trans*	*TCAM*	Mem_1	Mem_2
No_Combo	3	47K	71K	14KB	611KB	404KB
Combo_1	3	40K	53K	50KB	505KB	327KB
Combo_2	3	32K	50K	73KB	546KB	412KB
No_Combo	4	36K	50K	39KB	514KB	355KB

TABLE V
APPLYING COMBINATION RULES ON CLAMAV SET

Combo_Rule	*w*	*States*	*Trans*	*TCAM*	Mem_1	Mem_2
No_Combo	5	570K	714K	45KB	9.35 MB	5.44 MB
Combo_1	5	527K	622K	120KB	8.54 MB	4.88 MB
Combo_2	5	459K	560K	148KB	8.44 MB	5.36 MB
No_Combo	6	477K	576K	76KB	8.39 MB	5.10 MB

Note that with Combination Rules the expected throughput also increases. From Figure 10 we see that the throughput with $w = 3$ using Combination Rule 2 is even higher than that for the case of $w = 4$ and no Combinations Rules used. Hence for a fair comparison, we present the results for both in Table IV and Table V.

D. Patterns Matching to the Same Quasi-Match State

Another issue with the proposed algorithm is that one quasi-match state might match to multiple patterns. These unwanted cases actually could be classified into two types. The first is that two patterns might have the same core blocks, such as pattern $s|earc|h.pl$ and pattern $?s|earc|h =$. They share the same core block sequence while the two patterns are different. The second is the unlikely possibility that the core block sequence of one pattern is the suffix of another pattern's core block sequence, while the latter pattern could not contain the former one. One example is pattern $co|nnec|tio|n =$ and $f|orco|nnec|tio|n$. doesn't include the former. Hence two comparisons are needed. However, if the former pattern is only $co|nnec|tio|n$, we regard it as a pattern already covered by the latter pattern and no double handling is needed.

Obviously we want both of these two unwanted cases to happen as infrequently as possible. In Table VI, we show the frequency with which these unwanted cases happen. $\# > n$ denotes the number of states that match to more than n patterns, i.e. more than n entries in match table. $max\#$ denotes the maximum number of patterns that match to the same state. We can see that although hundreds of states match to more than one pattern, the number of states that match to more than 4 patterns is small. Since the matched patterns are stored consecutively in the Match Table, 4 entries in one fetch is feasible and in this case the state machine will not halt even for most of these unwanted states.

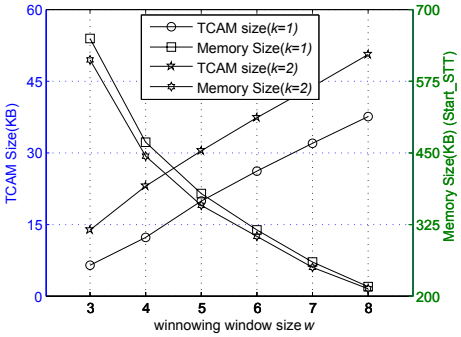
If we need to guarantee performance even for the very few match states that correspond to more than 4 patterns, these few exceptional states can be handled by TCAM too.

VI. CONCLUSIONS

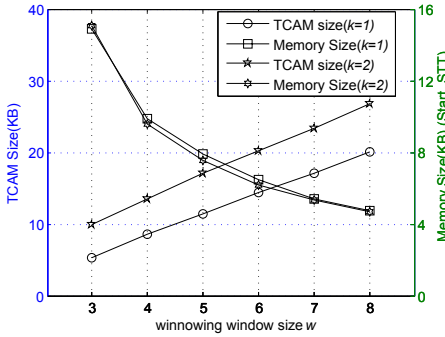
Building fast content inspection systems requires the implementation of complex pattern matching functions at very high

TABLE III
COMPARISON AMONG THREE DATA STRUCTURES

Pattern Set	w	Total State	Match Table	"Single_STT"			"Start_STT" Design					"Three_STTs" Design			
				Trans (K)	Total Mem (KB)	B per char	Trans (K)	Main STT (KB)	Start STT (KB)	Total Mem (KB)	B per char	Main STT (KB)	Jump STT (KB)	Total Mem (KB)	B per char
snort-fixed	4	36K	51KB	192	1,725	11.1	24	446	16	514	2.9	185	102	355	2.3
snort-full	4	36K	55KB	200	1,794	11.0	50	455	17	527	2.8	189	105	366	2.3
clamav-fixed	6	477K	414KB	2,971	40MB	19.6	576	7,760	217	8,391	3.8	3,428	1,040	5,100	2.5
clamav-full	6	2,143K	1.19MB	——	——	—	2,525	34,039	640	35,864	3.9	15,073	4,260	21,158	2.4



(a) SNORT-fixed



(b) ClamAV-fixed

Fig. 9. Effects of Changing w with $k = 1$ and 2

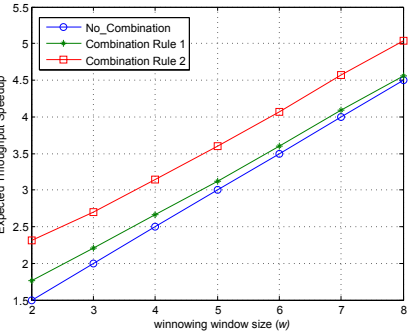


Fig. 10. Average Speedup when Combination Rules Applied

TABLE VI
STATISTICS ON MULTIPLE PATTERNS MATCHING TO THE SAME STATE

Pattern Set	*w*	*Match States*	*#> 1*	*# > 2*	*#> 4*	*#> 8*	*max#*
snort-fixed	4	5,486	416	123	26	1	9
clamav-fixed	6	31,588	407	54	8	1	9

speed. In this paper, we proposed a new variable-stride based pattern matching algorithm (VS-DFA) that is very memory-efficient and has a many-fold increase in system throughput. Compared with the fixed-length-stride approaches, VS-DFA avoids the curse of memory by a "self-synchronized" segmentation method borrowed from document fingerprinting [1]. A single search engine that implements the VS-DFA algorithm can increase the throughput by multiple times (depends on winnowing window size) while using less than three bytes of memory per pattern character. Moreover, the scheme is very scalable. Even for the very large pattern set such as ClamAV, the byte per pattern character is still almost the same, achieved by single engine and no pattern set partitioning. Compared with previous results, our result is very good. For example, the most recent result [7] need to divide pattern set into 32 subsets to achieve 4B/char and 512 subsets to achieve 2B/char, which is a prohibitively huge overhead.

Many interesting problems remain for future investigation: can we find a data segmentation algorithm that is better than the Winnowing that we use? Can we achieve larger stride-lengths than what we get now? Can we extend the algorithm to handle regular expression matching?

REFERENCES

[1] S. Schleimer, D. S. Wilkerson, and A. Aiken, "Winnowing: Local algorithms for document fingerprinting," in *ACM SIGMOD*, 2003.
[2] A. V. Aho and M. J. Corasick, "Efficient string matching: An aid to bibliographic search," *Commun. ACM*, vol. 18, no. 6, 1975.
[3] N. Tuck, T. Sherwood, B. Calder, and G. Varghese, "Deterministic memory-efficient string matching algorithms for intrusion detection," in *IEEE INFOCOM*, 2004.
[4] B. C. Brodie, D. E. Taylor, and R. K. Cytron, "A scalable architecture for high-throughput regular-expression pattern matching," in *ISCA*, 2006.
[5] L. Tan and T. Sherwood, "A high throughput string matching architecture for intrusion detection and prevention," in *ISCA*, 2005.
[6] J. van Lunteren, "High-performance pattern-matching for intrusion detection," in *IEEE INFOCOM*, 2006.
[7] T. Song, W. Zhang, D. Wang, and Y. Xue, "A memory efficient multiple pattern matching architecture for network security," in *IEEE INFOCOM*, 2008.
[8] S. Dharmapurikar and J. W. Lockwood, "Fast and scalable pattern matching for network intrusion detection systems," *IEEE JSAC*, vol. 24, no. 10, 2006.
[9] H. Lu, K. Zheng, B. Liu, X. Zhang, and Y. Liu, "A memory-efficient parallel string matching architecture for high-speed intrusion detection," *IEEE JSAC*, vol. 24, no. 10, 2006.
[10] Y. Fang, R. H. Katz, and T. V. Lakshman, "Gigabit rate packet pattern-matching using tcam," in *IEEE ICNP*, 2004.
[11] M. Alicherry, M. Muthuprasanna, and V. Kumar, "High speed pattern matching for network ids/ips," in *IEEE ICNP*, 2006.
[12] D. P. Scarpazza, O. Villa, and F. Petrini, "Exact multi-pattern string matching on the cell/b.e. processor," in *ACM CF*, 2008.
[13] N. S. Artan and H. J. Chao, "Tribica: Trie bitmap content analyzer for high-speed network intrusion detection," in *IEEE INFOCOM*, 2007.
[14] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese, "Beyond bloom filters: from approximate membership checks to approximate state machines," in *ACM SIGCOMM*, 2006.
[15] S. Kumar, J. Turner, and P. Crowley, "Peacock hashing: Deterministic and updatable hashing for high performance networking," in *IEEE INFOCOM*, 2008.
[16] H. Yu and R. Mahapatra, "A memory-efficient hashing by multi-predicate bloom filters for packet classification," in *IEEE INFOCOM*, 2008.
[17] R. Pagh and F. F. Rodler, "Cuckoo hashing," in *ESA*, 2001.
[18] K. A. Ross, "Efficient hash probes on modern processors," in *ICDE*, 2007.
[19] "Snort rules," at http://www.snort.org/pub-bin/downloads.cgi.
[20] "Clamav virus database," at http://www.clamav.net/download/cvd.