StriD^{2}FA: Scalable Regular Expression Matching for Deep Packet Inspection

Xiaofei Wang[†] Junchen Jiang[‡] Yi Tang[‡] Yi Wang[‡] Bin Liu[‡] Xiaojun Wang[†]
[†]School of Electronic Engineering, Dublin City University, Dublin, Ireland
[‡]Department of Computer Science and Technology, Tsinghua University, Beijing, China

***Abstract*—Deep packet inspection (DPI) has become one of the key components of a Network Intrusion Detection System (NIDS) and it compares packet content against a set of rules written in regular expression. The need to keep up with ever-increasing line speed has forced NIDS designers to move to hardware-based implementation where the memory resources are limited.**

In this paper, we present LBM, a novel accelerating scheme for regular expression matching which converts the original byte stream into much shorter integer stream and then matches it with a variant of DFA, called Stride-DFA(StriD^{2}FA). In the instance of LBM that we realize, a speedup of 10-15 is achievable while the required memory size is much less than that in the traditional DFA.

***Index Terms*—Regular Expression Matching, DPI, DFA**

I. Introduction

DPI technologies have been increasingly deployed in NIDS to detect attacks or viruses. To this end, state-of-the-art systems, including Snort [1], ClamAV [2] and security applications from Cisco Systems [3], compare packet content to a set of rules. Rules written in strings are initially popular, but have limited expressiveness. To support increasingly complex services, regular expression (regex) has been used to replace string by these systems due to its higher expressiveness and flexibility. The need to keep up with ever-increasing line speed has forced NIDS designers to move to hardware or high-speed memory where memory resources are limited. Thus, to design regex matching that achieves both time and space efficiency is a significant challenge.

A novel *length-based matching (LBM)* is presented for accelerating regex matching. Like traditional methods, LBM has a DFA-like matcher called *Stride-DFA (StriD^{2}FA)*. However, LBM differs from traditional methods in two key ways:

- In LBM, a packet as a byte stream is first converted into a much shorter *stride-length (SL)* stream (*i.e.*, integer stream) before sending to StriD^{2}FA. Therefore, the shorter the SL stream is, the higher the speedup can be achieved (in our system, 10 to 15 times speedup is achievable).
- Since it is the SL stream that StriD^{2}FA receives (rather than original byte string as in DFA), StriD^{2}FA is not directly built from regex, but is built according to different kinds of SL streams. Therefore, the fundamental difference between StriD^{2}FA and DFA is that in DFA a transition records a byte while in StriD^{2}FA it records a length (*i.e.*, integer).

This paper is supported by NSFC (60625201, 60873250, 61073171), 973 project (2007CB310702), Tsinghua University Initiative Scientific Research Program, the Specialized Research Fund for the Doctoral Program of Higher Education of China and Dublin City University Research Collaboration Program.

The benefits of LBM are not only limited to increase matching speed. As to memory consumption, StriD^{2}FA also costs less memory than DFA-based accelerating algorithms, for two reasons: 1) it has less states since regexes are stored more compactly in StriD^{2}FA (Section IV), and 2) the upper bound of SL are easily controlled (Subsection III-A) so that each state has less fan-out. Moreover, LBM can be expediently applied on existing hardware/software platform, as StriD^{2}FA share the same I/O interfaces and logic structure with traditional DFA built directly from the regex set.

LBM also leads to two key challenges. First, to preserve the expressiveness of regex,any regex should be able to transform to StriD^{2}FA. This is achieved by a graph algorithm that transform any DFA to a StriD^{2}FA (Section IV). Second, since the SL stream is a compressed representation of the original stream, only part of the original stream is matched by StriD^{2}FA, causing false positive (but no false negative). An algorithm is proposed that ensures the false positive rate is at an acceptable low level (detail in Section V). A verification phase is used for accurate matching if a possible match is found by StriD^{2}FA. Since the majority of the Internet traffic is not malicious so that it is possible to get quite high throughput if the probability of having to execute accurate matching is low [4].

In particular, the contributions are summarized as follows:

- Introduce the concept of LBM, a novel accelerating scheme for regex matching which converts the original byte stream into much shorter integer stream and then matches it with a variant of DFA, called StriD^{2}FA.
- Give the formal construction of StriD^{2}FA that transforms any set of regex to a StriD^{2}FA.
- Describe the method to extract SL stream from input stream so that false positive rate can be reduced to an relative low level.
- Realize an general instance of LBM. It is demonstrated that this instance achieves both space and time efficiency and can be expediently migrated to existing platforms. 10 to 15 times speedup is achievable while the memory cost is smaller than traditional DFA.

The rest of the chapter is organized as follows. In Section II the previous work related to pattern matching is discussed. Section III presents the overall structure of LBM and how it works with an example. Section IV gives the formal construction of a StriD^{2}FA and false positive will be addressed in Section V. Section VI reports and analyzes the performance of LBM and StriD^{2}FA. The paper is finally concluded by Section VII.

II. Related work

Regex matching was initially studied as a topic in automata theory and formal theory in the context of theoretical computer science [5]. To accelerate the regex matching in real-world systems, the problem has been intensively studied in practical scenarios in recent years. Vulnerability signature is recently proposed and applied as an alternative of regex, but it still requires a high speed regex matching subsystem [6].

For multiple characters multi-string matching, the algorithms in [7]–[9] all exploit parallelism, to improve throughput with tradeoffs between memory and bandwidth. With Bloom-filter implemented in on-chip memory, Dharmapurikar et al. presented a scheme [7] that can process multiple characters per clock cycle and attain average throughput up to multigigabit with moderate memory consumption. However, the proposed schemes are vulnerable to malicious attacks since in the worst case they must frequently access the relatively slow off-chip SRAMs to launch exact string comparisons. In [8], Nan et al. introduced a variable-stride method to deal with string matching rule set. The variable-stride can enhance the matching performance. However, the performance of the matching scheme is rule set and input string dependent, and the worst case performance can be pretty bad. Brodie et al. [9] increased the throughput of regex matching by expanding the alphabet set, resulting in an exponentially increased memory requirement in the worst case. A recent method [4] introduced the sampling techniques to accelerate regex matching, but it not all kinds of regex are supported.

In additional to the aforementioned accelerating approaches, DFA-based compression methods also enhance the system performance by using multiple matching engines, or faster memory. Transition compression approaches obtain a high compression rate by greatly reducing per-state-transition table. D^{2}FA [10] acknowledged as the original work in this approach, compress DFA by applying default transitions, at the cost of accessing DFA multiple times per input character. Subsequent works including [11], [12] improved the worst case as well as average performance. State compression technique was first utilized in [13], where patterns are selectively grouped to deflating state explosion. Another work [14] performed a partial NFA-to-DFA conversion to preventing state explosion. The state-of-the-art work XFA [15] uses auxiliary memory to reduce the DFA state explosion and achieves a great reduction ratio. However, it is not suitable for real-time applications on networks due to its significant startup overheads. Our proposed method does not conflict with above works, since the fundamental structures DFA is completely preserved.

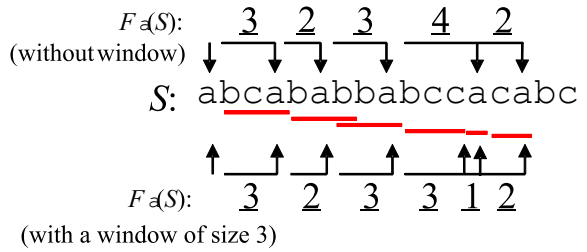


Fig. 1. Use tag and sliding window to convert input stream into SL stream (with tag 'a', window size $w = 3$).

III. System Design Principles and Challenges

A. Converting input stream into SL stream

The convertor works as follows (see Figure 1). When sequentially scanning a byte stream, some specific characters are chosen, called a ***tag***. A tag is used to get the distances between two adjacent tags which are called *stride lengths (SL)*. Let $F_x(S)$ denote the SL stream of S when using x as the tag. The top row of Figure 1[1] shows the result of $F_{\mathtt{a}}(S)$. The SLs of the same tags are calculated by a convertor. A convertor is basically a counter which can be easily implemented in either software or hardware platform.

The convertor leads to a problem that the SL sent to a StriD^{2}FA can be arbitrarily large, forcing StriD^{2}FA to handle an infinite alphabet set. To solve this problem, a fixed size sliding window is adopted (a similar application is found in [8]). The window works in the following way (the bottom row of Figure 1): if a tag is not found within a window, then the last character of the window is marked as a fingerprint anchor, the window size w is sent to StriD^{2}FA and the character following the fingerprint anchor is set to be the beginning of the window. In this manner, any SL sent to a StriD^{2}FA must be in a finite alphabet set $\Sigma = \{1, \ldots, w\}$.

B. An Example of StriD^{2}FA

In this Section, an example is used to explain how input stream is converted, matched by a StriD^{2}FA and verified if a potential match is found by StriD^{2}FA. Suppose the regex rule is `.*abba.{2}caca`[2]. It matches any contiguous part of the input stream that starts with `abba`, followed by two arbitrary characters, and finally ends with `caca`. Here '`a`' is chosen as the *tag* and the window size is 3. Then $F_{\mathtt{a}}$(`.*abba`) is (1 | 2 | 3)+3. Since the length of the first `a` in `caca` relies on two arbitrary characters which could be `[^ab][^ab]`[3], `a[^a], [^a]a,` or `aa`. So possibly the SL stream of F_a(`.{2}caca`) = 3 1 2 | 1 3 2 | 2 2 2 | 1 1 2 2. Finally the regex `.*abba.{2}caca` could be transformed to (1 | 2 | 3)+ 3 (3 1 2 | 1 3 2 | 2 2 2 | 1 1 2 2), where the alphabet set is $\{1, 2, 3\}$ Figure 2 shows the traditional DFA and the StriD^{2}FA for regex `.*abba.{2}caca`.

Given an byte stream $T =$ ''abcababbabccacabc'', it is first converted into SL stream $F_{\mathtt{a}}(T)$ =3 2 3 3 1 2, and then matched by the StriD^{2}FA in Figure 2(b). As 3 2 3 3 1 2 is matched by the StriD^{2}FA, then the input stream is sent the verification module to make an accurate match by using some traditional methods (*e.g.*, reversed DFA in [4]).

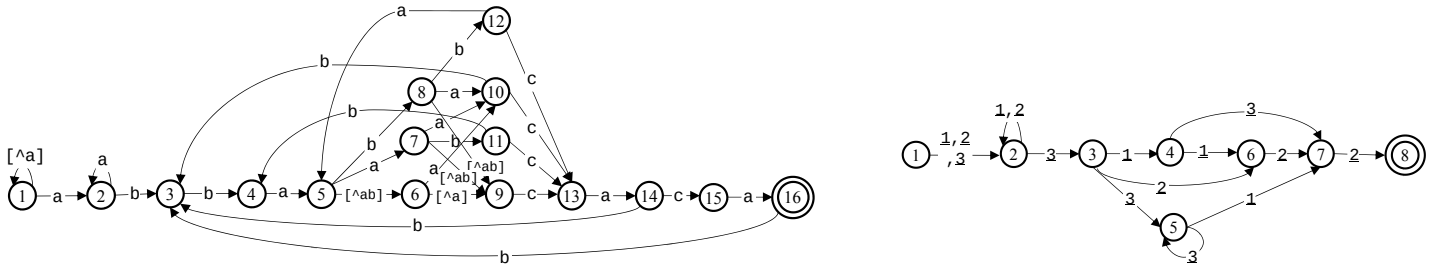
C. Benefits of LBM

1) Increased speed: If the lengths between tags (properly chosen) are relatively long, the method can achieve quite a high speedup. According to the statistics in Section VI, average SLs of some characters are larger than 100. As a result, a high speedup is supported statistically by our experiments.

[1]Underscore is used to indicate an SL, not a character.

[2]JFlex syntax for regex is used in this report, i.e., "." and "*" means the wildcard and any number of matchings respectively. In default, the 8bit input character set is used to analyze.

[3][^ab] matches any character other than 'a' or 'b'.



(a) Traditional DFA (b) StriD^{2}FA using tag 'a'

Fig. 2. Traditional DFA and StriD^{2}FA of regex `.*abba.{2}caca` (transitions back to state 1, 2 and 3 are partly ignored for simplicity).

2) Small memory consumption: Compared with traditional DFA, the StriD^{2}FA is more compact in memory for two reasons. Firstly, the number of states is generally less than traditional DFA (*e.g.*, StriD^{2}FA has 8 less states than the traditional DFA in Figure 2). Secondly, the fanout of each state is controlled by the window size, and which is generally far smaller than the fanout of a traditional DFA (256 in standard ASCII). With less states and a more compact state-transition table, the memory consumption is greatly reduced compared with traditional DFA.

D. Challenges

1) Regex converting: Regex is powerful by using multiple wildcard characters (e.g., '.', '*'), length restrictions ('?', '+') and groups of characters ('[]', '^'). So how to preserve the expressiveness in StriD^{2}FA raises a significant challenge. For example, by applying the LBM method, `.*abba.{2}caca` can be rewritten as a different pattern: ($\underline{1}$ | $\underline{2}$ | $\underline{3}$)+ $\underline{3}$ ($\underline{3}$ $\underline{1}$ $\underline{2}$ | $\underline{1}$ $\underline{3}$ $\underline{2}$ | $\underline{2}$ $\underline{2}$ $\underline{2}$ | $\underline{1}$ $\underline{1}$ $\underline{2}$ $\underline{2}$). In Section IV, a formal method to efficiently construct StriD^{2}FA from any regex is described.

2) False positive rate: Since the SL stream is a highly compressed form of an input stream, part of the information is left out before being sent to StriD^{2}FA. So it is only a potential matching if StriD^{2}FA reports a matching, causing a false positive. For example, if given two strings T="efe" and T'="ere", we have $F_e(T') = \underline{5} = F_e(T)$. If the stride length $\underline{5}$ is matched, it is not sure if it is T or T' that is matched.

IV. Building StriD^{2}FAs from Regex

A formal construction is given about how to construct StriD^{2}FA from a regex. It involves four steps: 1) build a standard DFA by traditional method from a regex; 2) restructure the DFA by classifying all the transitions; 3) transform the restructured DFA to a non-deterministic StriD^{2}FAs by the depth first search (DFS) algorithm; 4) determinize to the final StriD^{2}FA (similar to the determinization in traditional DFA).

Step 1: Compile Regex to standard DFA -The way of compiling a regex to a standard DFA is conventional (intensively studied in [5]). Figure 2(a) is the traditional DFA of regex `.*abba.{2}caca`. Since regex has an equivalent expressiveness with DFA (every regex has an equivalent DFA), so all the things need to do is to convert a DFA into its corresponding StriD^{2}FA that matches the SL stream without giving any false negative rate. That is the task of the next two steps.

Step 2: Restructure DFA by classifying transitions -In this step, all labels are removed on transitions and mark each transition whether its character is c[4] (solid transition if true and dashed transition otherwise). Figure 3(a) shows the output after classification on Figure 2(a) with tag 'a'.

Step 3: Transform to non-deterministic StriD^{2}FA -The input of this step is the restructured DFA (a directed graph consisting of solid and dashed transition), and the output is a non-deterministic StriD^{2}FA, a directed graph where each transition is labeled with a SL value (*i.e.*, integer). Non-deterministic means some states can have more than one outgoing transitions labeled with the same SL (integer). To explain the method, the following steps are processed recursively: starting from any state q,

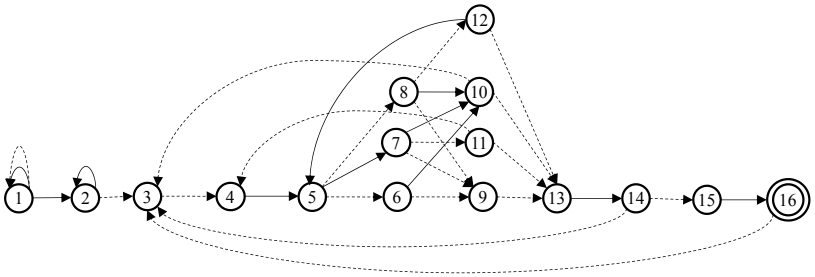
- if a solid transition (pointing to state q') is reachable in l steps where $l \leq w$, add a transition labeled l from q to q';
- otherwise (i.e., there is an all-dashed-transition path of length w to state q'), add a transition labeled w from q to q'.

The first case applies when the convertor extracts SL of $l (l \leq w)$, and the second case applies when the convertor finds no tag in the window and then uses the window size w as the SL. The basic operation can be done by a depth first search with maximum depth w. For the whole graph, the basic operation is processed iteratively, starting from the initial state to find all the reachable

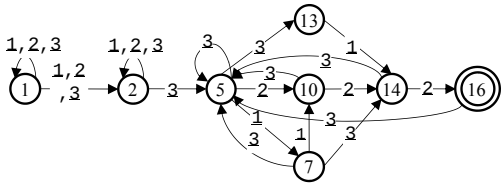
Algorithm 1 Step 3: Transform directed graph to StriD^{2}FA

```
1: Procedure BUILD(G, w)
2: Input: G = (V, E, q0, F), w                          ▷ E
   consists of solid and dashed transitions, q0: initial state and
   F: set of final states, w: window size
3: Output: StriD2FA A = (Q, Σ, δ, q0, F)
4: Q = δ = F = ∅, Σ = {1, ..., w}
5: Explore(q0, q0, w, Q, δ, F)
6: return A = (Q, Σ, δ, q0, F)
7:
8: Procedure Explore(q, p, depth, Q, δ, F)
9: if depth = 0 then
10:     Q ← q ∪ Q
11:     δ ← δ ∪ (p, w, q)
12: else
13:     for e = (q, q') ∈ D+(q) do        ▷ D+(q): the set of
   outgoing transitions of q
14:         if e is dashed then
15:             depth ← depth − 1
16:             Explore(q', p, depth, Q, δ, F)
17:         else
18:             Q ← q' ∪ Q
19:             δ ← δ ∪ (p, w − depth, q')
20:             Explore(q', p, w, Q, δ, F)
21:         end if
22:     end for
23: end if
24: return
```

[4]Let c denote the tag which is selected and windows size is w.



(a) Directed graph after step 2, transitions pointing to state 1 and 2 are partly omitted for clarity).



(b) The non-deterministic StriD^{2}FA after step 3 (for clarity, transitions labeled 2 and 3 from state 7,10,13,14,16 to state 2 and transitions labeled 1 from state 10,14 to state 2 are omitted).

Fig. 3. An example showing the results after step 2 and step 3.

states and build transitions between them. The pseudocode is given in Algorithm 1 (Let $\triangleright$ denote the comment symbol). Figure 3(b) is the non-deterministic StriD^{2}FA transformed from Figure 3(a).

Step 4: Determination to StriD^{2}FA -The method to determinate non-deterministic StriD^{2}FA to a deterministic finite automaton is exactly the same to the procedure of determining traditional NFA to DFA, so we will not present its details. Figure 2(b) shows the final structure of StriD^{2}FA determined from Figure 3(b).

V. Optimization of False Positive

Minimizing false positive rate while preserving other performances (i.e., throughput and memory usage) is analyzed in this section. High false positive rate leads to frequently use of the verification module, degrading the overall throughput. Although the idea and core mechanism is simple and straightforward, StriD^{2}FA is a very complicated system as a whole, so several optimizations have been adopted in LBM system.

Essentially, the optimization of reducing false positive rate is a balance between two extremes. On the one hand, to reduce the false positive rate to zero, all possible characters should be used as tag at the convertor should be checked (that is, 256 in worst case) and build a StriD^{2}FA for each of them; however, this will degrade speedup to one (as every input character invokes an access to some StriD^{2}FAs) and lead to large memory consumption (linear with the number of tags). On the other hand, to achieve a high throughput and low memory usage, it is expected to use as few tags as possible; however, the possibility of false positives can increase significantly since a large portion of the input characters is left out[5].

To strike a balance, a small group of proper tags is selected so that each regex rule can be "covered" by some tags in the group. In the following two subsections, first the definition about how a regex rule is "covered" by some tags is given , and then the choosing of tags is addressed to achieve both space-time efficiency and lower false positive rate.

A. How tags "cover" regex rule

Intuitively, a rule is covered by one or more tags $c_1, \ldots, c_k$ when it is of very high probability that the rule is matched if $StriD^2FA_1, \ldots, StriD^2FA_k$ all report a match. Here $StriD^2FA_i$ is the StriD^{2}FA with respect to tag c_i in rule set($i = 1, \ldots, k$). However, it is impossible to pre-compute this probability since it strongly relies on the input string which is unknown when choosing tags. Fortunately, from the experiments, it is easy to find that choosing "frequent" characters in a rule as tags can greatly reduce false positive rate. The meaning of being "frequent" is quite straightforward: the frequency $Freq(c, r)$ of a character c in a regex r refers to the number of occurrences of c in regex r over the sum of lengths of all fixed substrings in r:

$$Freq(c, r) = \frac{\sum_{s \in \mathcal{S}_{c,r}} (\#c \text{ in } s)}{\sum_{s \in \mathcal{S}_{c,r}} |s|}$$

Here $\mathcal{S}_{c,r}$ is the set of fixed substrings in regex rule r and $|s|$ is the length of string s. Then the definition of how tags "cover" regex rule is given as followings.

Definition 1: A regex rule r is covered by a set of tags, TAG, if the frequency of TAG in r exceeds a predefined *threshold* θ.

The reason of using $Freq(c, r)$ to select tags is twofold. First, $Freq(c, r)$ has the additive property; that is, the frequency of a set of characters in regex r is the sum of $Freq(c, r)$ over all c in the character set. So it is very simple to calculate the frequency of a set of characters in all regex rules. Second, with higher $Freq(c, r)$, the possibility of false positive is lower because more part (*i.e.*, more characters) of the regex rule is checked by the chosen set of tags.

VI. Evaluation

To evaluate the efficiency of LBM, an instance is implemented of it, which uses tags to produce SL stream (as in Section III-A). The StriD^{2}FAs are built in the way described in Section IV.

The performance of LBM system, including throughput and memory size, is influenced by four aspects: (a) input stream (trace), (b) regex rules, (c) window size and (d) selected tags. The memory consumption is influenced by (b)(c)(d); speedup is influenced by (a)(c)(d); and the overall throughput is influenced by (a)(b)(c)(d).

A. Memory consumption

The rule set is used from the latest version of Snort (v2.8.6 as of 03 Aug, 2010) and ClamAV (0.96.2 as of 15 Feb, 2010). Traditionally, combining all (even part of) regex rules into one DFA requires large amount of memory; for example more than 15 GB of memory is needed if 88 Snort rules are combined into one single DFA [15] (the traditional DFA memory usage is in

[5]The last character in the window is marked as a fingerprint anchor if it is no tag in the window.

TABLE I
COMPARISON WITH STRID^{2}FA, K-DFA AND TRADITIONAL DFA

Snort						
Memory factors	**Traditional DFA**	**k-DFA**		**StriD^{2}FA**		
		k=2	k=4	w=10	w=20	w=30
# State	5298	6152	8314	671	447	264
Alphabet set	256	2114	7024	10	20	30
# Transition	1.35M	13.0M	58.39M	6.7k	8.94k	7.92k
ClamAV						
Memory factors	**Traditional DFA**	**k-DFA**		**StriD^{2}FA**		
		k=2	k=4	w=10	w=20	w=30
# State	6721	8362	10051	834	526	392
Alphabet set	256	2673	9532	10	20	30
# Transition	1.72M	8.78M	28.04M	8.3k	10.5k	11.7k

the second column of Table I). For StriD^{2}FA, the memory usage of different rule sets using various window sizes is investigated. Sliding window is adopted to control the largest SL which is sent to the StriD^{2}FAs and reduce the size of state-transition table from 256 to window size w. The fifth to seventh column in Table I show the total number of states and transitions in all StriD^{2}FAs using different window size w. The results are compared with the memory cost of another well-known DFA acceleration matching method k-DFA [16].

B. Speedup

Three real-life are employed network traffic traces to evaluate the system, which are Darpa, Defcon and Tsinghua-trace respectively. Darpa trace is from the DARPA intrusion detection data sets collected by MIT Lincoln Laboratory [17]. Defcon trace is from the Shmoo Group DefCon 9.0 Capture the Flag Contest [18]. Tsinghua-trace was collected in the gateway of Tsinghua University campus network.

In LBM, the speedup over a long time scale equals the average stride length (SL). Figure 4 illustrates the SL's dependency on the window size, confirmed by experiments on 3 different traces. In each of the boxplots, the upper and lower bars mark the 97% and 3% percentiles of the average strides, respectively. A quasi-linear relationship is made evident by the line that is constituted by the mid-points, with the average SL surpassing half the window size. Therefore, a high throughput can be guaranteed with different window sizes and tags.

C. Performance on real trace

The above three traces are used to evaluate the performance of LBM method. With LBM matching engine, only SL stream which is occupies about 0.38% of the original input stream will be sent to StriD^{2}FA to make a match. According to the experimental results of Tsinghua-trace and some groups of ClamAV rules, there are 127 potential matches alarmed by StriD^{2}FA while 13 real matches are confirmed by the verification module.

In addition to the above three different kinds of traces, other traces collected from the World Wide Web should also be used to test the performance and stability of the LBM scheme. WebbSpam [19] consists of nearly 572,292 Web spam pages with the size of 5.54GB, was chosen additionally to demonstrate the performance of the LBM scheme. According to the statistics, about 99.41% of the input stream has been pre-filtered by the architecture which means only 0.58% (263.5M) data needs to be sent to StriD^{2}FA to match instead of matching the whole raw bytes. There are 392 potential matches and 89 matches real matches among of it.

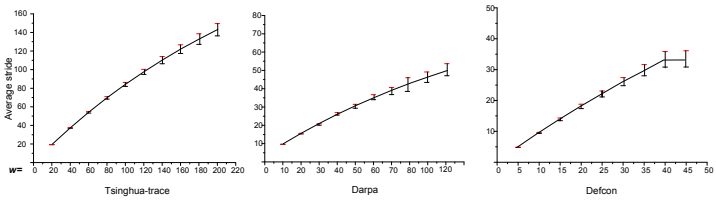


Fig. 4. Average stride length under different window sizes.

VII. CONCLUSION

In this paper, we present LBM, a novel accelerating scheme for regular expression matching which converts the original byte stream into much shorter integer stream and then matches it with a variant of DFA, named StriD^{2}FA. We also give the formal construction of StriD^{2}FA that transforms any set of regex to a StriD^{2}FA and describe the method to produce SL stream so that false positive can be reasonably reduced. Evaluating results show that our architecture can achieve 10 to 15 times speedup with about 30% less memory consumption than the traditional DFA.

REFERENCES

[1] "Snort: Network intrusion detection system," http://www.snort.org/.
[2] "Clam AntiVirus," http://www.clamav.net/.
[3] "Cisco IOS IPS," http://www.cisco.com.
[4] D. Ficara, G. Antichi, A. Pietro, S. Giordano, G. Procissi, and F. Vitucci, "Sampling techniques to accelerate pattern matching in network intrusion detection systems," in *Proc. of IEEE ICC*, 2010.
[5] J. Hopcroft, R. Motwani, and J. Ullman, *Introduction to Antomata Theory, Languages and Computation*. Addison Wesley.
[6] Z. Li, G. Xia, H. Gao, Y. Tang, Y. Chen, B. Liu, J. Jiang, and Y. Lv, "Netshield: Matching with a large vulnerability signature ruleset for high performance network defense," in *Proc. of ACM SIGCOMM*, 2010.
[7] S. Dharmapurikar and J. W. Lockwood, "Fast and scalable pattern matching for network intrusion detection engines," *IEEE JSAC*, vol. 24, no. 10, 2006.
[8] N. Hua, H. Song, and T. Lakshman, "Variable-stride multi-pattern matching for scalable deep packet inspection," in *Proc. of INFOCOM*, 2009.
[9] B. C. Brodie, D. E. Taylor, and R. K. Cytron, "A scalable architecture for high-throughput regular-expression pattern matching," in *Proc. of ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2006.
[10] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner, "Algorithms to accelerate multiple regular expressions matching for deep packet inspection," in *Proc. of ACM SIGCOMM*, 2007.
[11] S. Kumar, J. Turner, and J. Williams, "Advanced algorithms for fast and scalable deep packet inspection," in *Proc. of ACM/IEEE Symposium on Architectures for Networking and Communcations Systems (ANCS)*, 2006.
[12] M. Becchi and P. Crowley, "An improved algorithm to accelerate regular expression evaluation," in *Proc. of ACM/IEEE Symposium on Architectures for Networking and Communcations Systems (ANCS)*, 2007.
[13] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz, "Fast and memory-efficient regular expression matching for deep packet inspection," in *Proc. of ACM/IEEE Symposium on Architectures for Networking and Communcations Systems (ANCS)*, 2006.
[14] M. Becchi and P. Crowley, "A hybrid finite automaton for practical deep packet inspection," in *Proc. of ACM International Conference on emerging Networking EXperiments and Technologies(CoNEXT)*, 2007.
[15] R. Smith, C. Estan, and S. Jha, "XFA: Faster signature matching with extended automata," in *IEEE Symposium on Security and Privacy*, 2008.
[16] M. Becchi and P. Crowley, "Efficient regular expression evaluation: theory to practice," in *ANCS*, 2008, pp. 50–59.
[17] "Mit darpa intrusion detection data sets," http://www.ll.mit.edu/mission/communications/ist/corpora/ideval/data/index.html.
[18] "Shmoo group defcon 9.0 capture the flag contest data sets," http://ictf.cs.ucsb.edu/data/defcon_ctf_09/.
[19] S. Webb, J. Caverlee, and C. Pu, "Introducing the webb spam corpus: Using email spam to identify web spam automatically," in *Proceedings of the 3rd Conference on Email and Anti-Spam (CEAS)*.