International Journal of Computer Applications Technology and Research
Volume 2– Issue 2, 137 - 140, 2013, ISSN: 2319–8656

Deep Packet Inspection with Regular Expression Matching

T. Nalini
Dept of Computer Science and Engineering
Bharath University
Chennai

M. Padmavathy
Dept of Computer Science and Engineering
Bharath University
Chennai

Abstract: Deep packet inspection directs, persists, filters and logs IP-based applications and Web services traffic based on content encapsulated in a packet's header or payload, regardless of the protocol or application type. In content scanning, the packet payload is compared against a set of patterns specified as regular expressions. With deep packet inspection in place through a single intelligent network device, companies can boost performance without buying expensive servers or additional security products. They are typically matched through deterministic finite automata (DFAs), but large rule sets need a memory amount that turns out to be too large for practical implementation. Many recent works have proposed improvements to address this issue, but they increase the number of transitions (and then of memory accesses) per character. This paper presents a new representation for DFAs, orthogonal to most of the previous solutions, called delta finite automata (δFA), which considerably reduces states and transitions while preserving a transition per character only, thus allowing fast matching. A further optimization exploits Nth order relationships within the DFA by adopting the concept of temporary transitions.

Keywords: Regular Expressions, Deep Packet Inspection, Differential Encoding, Finite Automata (FA), Pattern Matching

1. INTRODUCTION

Nowadays, deep packet inspection is required in an increasing number of network devices (intrusion prevention systems, traffic monitors, application recognition systems). Traditionally, the inspection was done with common multiple-string matching algorithms, but state-of-the-art systems use regular expressions (regexes) [1] to describe signature sets. They are adopted by well-known tools, such as Snort [2] and Bro [3], and in devices by different vendors such as Cisco [4].

Typically, finite automata (FAs) are employed to implement regexes matching. In particular, deterministic FAs (DFAs) allow for fast matching by requiring one state transition per character, while nondeterministic FAs (NFAs) need more transitions per character. The drawback of DFAs is that for the current regex sets they require an excessive amount of memory. Therefore, many works have been recently presented with the goal of memory reduction for DFAs, by exploiting the intrinsic redundancy in regex sets. However, most of these solutions can require more than one memory reference, thus lowering search speed.

This paper introduces a novel compact representation scheme (named δFA), which is based on the observation that since most adjacent states share several common transitions, it is possible to delete most of them by taking into account the different ones only (the δ in δFA just emphasizes that it focuses on the differences between adjacent states). Reducing the redundancy of transitions appears to be very appealing since the recent general trend in the proposals for compact and fast DFAs construction suggests that the information should be moved toward edges rather than states. In particular, our idea comes from D^2FA [5], which introduce default transitions.

We add the concept of "temporary transition," to improve the δFA. Instead of specifying the transition set of a state with respect to its direct parents (also defined 1-step ancestors), relaxing this requirement to the adoption of *N*-step "ancestors" increases the chances of compression. As we will show in the following, the best approach to exploit this *N*th-order dependence is to define the transitions of the states between ancestors and child as "temporary." This, however, introduces a new problem during the construction process. The optimal construction (in terms of memory or transition reduction) appears to be an *NP*-complete problem. Therefore, a direct and oblivious approach is chosen for simplicity. Results on real rule-sets show that our simple approach does not differ significantly from the optimal construction. Since this optimized technique is an extension to δFA that exploits *N*th-order dependence, we name it $\delta^N FA$. While many other proposed algorithms for DFA compression require more transitions per character, δFA and $\delta^N FA$ examine one state per character only, thus reducing the number of memory accesses and speeding up the overall lookup process. This improvement comes at the cost of wider memory accesses (with respect to previous schemes).

The rest of this paper is organized as follows. Section II explains the background of the work. Section III describes the algorithms for the creation of transition and lookup table. Section IV describes the experiments and results. Section V concludes this paper.

2. BACKGROUND WORK

Deep packet inspection has recently gained popularity as it provides the capability to accurately classify and control traffic in terms of content, applications, and individual subscribers. Cisco and others today see deep packet inspection happening in the network and they argue that "Deep packet inspection will happen in the ASICs, and that ASICs need to be modified" [6]. Some applications requiring deep packet inspection are listed below:

- Network intrusion detection and prevention systems (NIDS/NIPS) generally scan the packet header and payload in order to identify a given set of signatures of well known security threats.
- Layer 7 switches and firewalls provide content-based filtering, load-balancing, authentication and monitoring. Application-aware web switches, for example, provide scalable and transparent load balancing in data centers.

Deep packet inspection often involves scanning every byte of the packet payload and identifying a set of matching predefined patterns. Traditionally, rules have been represented as exact match strings consisting of known patterns of interest. Naturally, due to their wide adoption and importance, several high speed and efficient string matching algorithms have been proposed recently. Some of the standard string matching algorithms such as Aho-Corasick [7] Commentz-Walter [8], and Wu-Manber [9], use a preprocessed data-structure to perform high-performance matching. A large body of research literature has concentrated on enhancing these algorithms for use in networking. Tuck et.al presents techniques to enhance the worst-case performance of Aho-Corasick algorithm. Their algorithm was guided by the analogy between IP lookup and string matching and applies bitmap and path compression to Aho-Corasick. Their scheme has been shown to reduce the memory required for the string sets used in NIDS by up to a factor of 50 while improving performance by more than 30%.

3. PROPOSED SYSTEM

As discussed, several works in the recent years have focused on memory reduction of DFAs by trading size for number of memory accesses. The most important and cited example of such a technique is D2FA [6], where an input character (hereafter simply "char") can require a (configurable) number of additional steps through the automaton before reaching the right state.

3.1 Motivating Example

In this section, we introduce δFA, a D^2FA-inspired automaton that preserves the advantages of D^2FA while requiring a single memory access per input char. In order to make clearer the rationale behind δFA construction and the differences with D^2FA, we start by analyzing the example of [6] given below.

Fig. 1(a) represents a DFA on the alphabet {a,b,c,d} that recognizes the regular expressions (a^+), (b^+c), and (c^*d^+). In Fig. 1(b), the D^2FA for the same set of regexes is shown. The main idea is to reduce the memory footprint of states by storing only a limited number of transitions for each state and by taking a default transition for all input chars for which a transition is not defined. When, for example, in Fig. 1(b) the state machine is in state 3 and the input is *d*, the default transition to state 1 is taken. State 1 has a specified transition for char *d*, therefore we jump to state 4 (as in the standard DFA).

In this example, taking a default transition costs one more hop (one more memory access) for a single input char. However, it may happen that also after taking a default transition, the destination state for the input char is not specified and another default transition must be taken, and so on. In the example, the number of transitions was reduced to nine in the D^2FA (while the DFA has 20 edges), thus achieving a remarkable compression.

However, observing the graph in Fig. 1(a), it is evident that most transitions for a given input lead to the same state, regardless of the starting state. In particular, adjacent states share the majority of the next-hop states associated with the same input chars. Then, if we jump from state 1 to state 2 and we "remember" (in a local memory) the entire transition set of 1, we will already know all the transitions defined in 2 (which has the same set of 1). This means that state 2 can be described with a very small amount of bits. Instead, if we jump from state 1 to 3, and the next input char is c, the transition will not be the same as the one that c produces starting from 1. Then, state 3 will have to specify its transition for c.

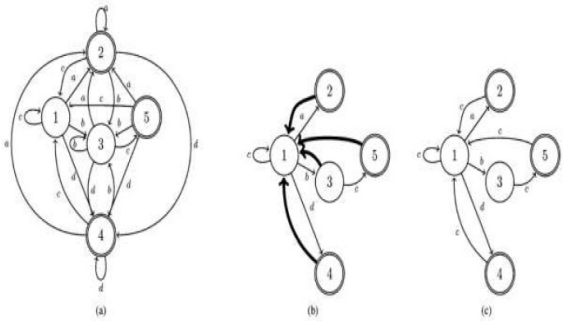

Fig-1: Automata Recognizing (a+), (b+c), and (c*d+) (a) DFA, (b) D2FA and (c) δFA

The result of what we have just described is shown in Fig. 1(c) (except for the local transition set), which is the δFA equivalent to the DFA in Fig. 1(a). We have eight edges only in the graph, and every input char requires a *single state traversal*.

3.2 Main Idea of δFA

The target of δFA is to obtain a similar compression as D^2FA without giving up the *single state traversal per character* of DFA. The idea of δFA comes from the following observations on D^2FAs and DFAs.

- Most default transitions are directed to states closer to the initial state.
- In a DFA, most transitions for a given input char are directed to the same state. Therefore, it becomes evident that most adjacent states share a large part of the same transitions. Thus, we can store only the differences between adjacent states. This requires, however, the introduction of a supplementary structure that locally stores the transition set of the current state. The main idea is to let this local transition set evolve as a new state is reached. If there is no difference with the previous state for a given character, then the corresponding transition defined in the local memory is taken. Otherwise, the transition stored in the state is chosen. In all cases, as a new state is read, the local transition set is updated with all the stored transitions of the state. The δFA in Fig. 1(c) only stores the transitions that *must* be defined for each state in the original DFA.

3.3 Construction

In Algorithm 1, the process for creating a δFA from an *N*-states DFA (for a char set of C elements) is shown. The algorithm works with the *transition tablet[s, c]* of the input DFA (i.e., an $N \times N$ matrix that has a row per state and where the *i*th item in a given row stores the state number to reach upon the reading of input char *i*). The final result is a "compressible" transition table $t_c[s, c]$ that stores the transitions required by the δFA only. All the other cells of the $t_c[s, c]$ matrix are filled with the special LOCAL_TX symbol and can be simply eliminated through a bitmap.

Algorithm 1: Creation of the Transition Table t_c of a δFA

1: for c ← 1, C do

```
2:          t_c[1, c] ← t[1, c]
3:    for s ← 2, N do
4:          for c ← 1, C  do
5:                t_c[s, c] ← EMPTY
6:    for S_parent  ← 1, N do
7:          for c ← 1, C  do
8:             Schild ← t[Sparent, c]
9:            for y ← 1, C do
10:         if t[S_parent, y] ≠ t[S_child, y] then
11:               t_c[S_child, y] ← t[S_child, y]
12.               else
12:          if t_c[S_child, y] == EMPTY then
14:               t_c[S_child, y] ←
LOCAL_TX
```

The construction requires a step for each transition (C) of each pair of adjacent states ($N \times C$) in the input DFA, thus it costs $O(N \times C^2)$ in terms of time complexity. The space complexity is $O(N \times C)$ because the structure upon which the algorithm works is another $N \times C$ matrix. In detail, the construction algorithm first initializes the t_c matrix with EMPTY symbols and copies the first (root) state of the original DFA in the t_c (it will act as base for subsequently storing the differences). Then, the algorithm observes the states in the original DFA one at a time. It refers to the observed state as *parent*. Then, it checks the *children* states (i.e., the states reached in one transition from parent state).

If, for an input char c, the child state stores a different transition than the one associated with any of its parent nodes, we cannot exploit the knowledge we have from the previous state, and this transition must be stored in the t_c table. On the other hand, when all of the states that lead to the child state for a given character share the same transition, then we can omit to store that transition. In Algorithm 1, this is done by using the special symbol LOCAL_ TX.

After the construction, since the number of transitions per state is significantly reduced, it may happen that some of the states have the same identical transition set. If we find j identical states, we can simply store one of them, delete the other j-1, and substitute all the references to those with the single state we left. Notice that this operation again creates the opportunity for a new state-number reduction because the substitution of state references makes it more probable for two or more states to share the same transition set. Hence, we iterate the process until the duplicate states end.

3.4 Lookup

The lookup in a δFA is shown in Algorithm 2. First, the current state must be read with its whole transition set. Then, it is used to update the local transition set t_{loc}: For each transition defined in the set read from the state, we update the corresponding entry in the local storage. This procedure comes at virtually no cost since it requires a number of operations on a fast local memory and its execution is easily masked in threaded systems or hardware implementations. Finally, the next state s_{next} is computed by simply observing the proper entry in the local storage t_{loc} .

Algorithm 2: Pseudocode for the lookup in a δFA. The current state is s and the input char is c

```
1.  read(s)
2.  for i ← 1, C do
3.    if t_c[s,i] ≠ LOCAL_TX then
4.          t_loc[i] ← t_c[s,i]
5.  s_next←t_loc[c]
return s_next
```

4. EXPERIMENTS AND RESULTS

The language used is java sdk2.0. Java is related to C++, which is a direct descendant of C. The trouble with C and C++ is that they are designed to be compiled for a specific target. But Java is a portable, platform-independent language that could be used to produce code that would run on a variety of CPUs under differing environments. Java can be used to create two types of programs: applications and applets. An application is a program that runs on our computer, under the operating system of that computer. Java is Simple, Secure, Portable, Object-oriented, Robust, Multithreaded, Architectural-neutral, Interpreted, High Performance, Distributed, and Dynamic.

As the number of network security threats rises, the NIDS has become one of the most important applications of packet inspection. Therefore, this study demonstrates the feasibility of integrating the proposed as the matching regular expression sets, introduces a novel compact representation scheme (named FA), which is based on the observation that since most adjacent states share several common transitions, it is possible to delete most of them by taking into account the different ones only (the in FA just emphasizes that it focuses on the differences between adjacent states). Reducing the redundancy of transitions appears to be very appealing since the recent general trend in the proposals for compact and fast DFAs construction suggests that the information should be moved toward edges rather than states.

Fig-2: Packet construction

Fig-3 Transmission

Packet Construction:

Fig-2 shows the packet construction, we are constructing the packets with a regular expression. The header information of a data packet is added in this phase. A packet consists of two kinds of data: control information and user data (also known as *payload*). The control information provides data the network needs to deliver the user data, for example: source and destination addresses, error detection codes like checksums, and sequencing information. Typically, control information is found in packet headers and trailers, with user data in between. The algorithms shown in Section II C.1 and D.1 are used to construct the transition table, lookup table respectively. It initializes the number of transitions in the DFA construction.

Transmission:

Fig-3 shows the transmission, the sender transmits the data to the receiver through the intermediate nodes.To find which source data is send into particular destination in the network. Here the data is received in all the intermediate

nodes and it must be checked for the illegitimate packets. The GUI provides the facility to send the packets to be inspected. Here the transitions will be checked and default transition will be added. The result of this will sent to look up table.

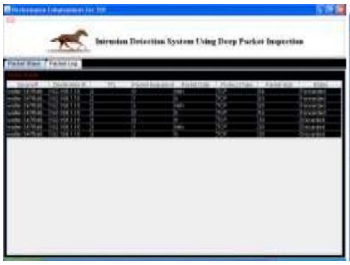

Fig-4: Packet Inspection

Fig-5: Evaluation

Packet Inspection:

Fig-4 shows the packet inspection. This is the phase of classifying intruder packet from the transmission. In the high-layer intrusion detection, expression may appear anywhere in the packet payload and making the attacking packets difficult to recognize. It may not only examine the header information but also the contents of the packet in order to determine more about the packet than just information about its source and destination. If the packets are matched with our expression and the packet is alive then it will be forwarded to the client else that will be discarded.

5. CONCLUSION

In this paper, we considered the implementation of fast regular expression matching for packet payload scanning applications. While NFA based approaches are usually adopted for implementation because naïve DFA implementations can have exponentially growing memory costs, we showed that with our rewriting techniques, memory-efficient DFA-based approaches are possible. While we do not claim to handle all possible cases of dramatic DFA growth (in fact the worse case cannot be improved), rewrite rules do over those patterns present in common payload scanning rule sets thus making fast DFA-based pattern matching feasible for today's payload scanning applications. It is possible that a new type of attack also generates signatures of large DFAs. For those cases, unfortunately, we need to study the signature structures before we can rewrite them.

6. REFERENCES

[1] R. Sommer and V. Paxson, "Enhancing byte level network intrusion detection signatures with context,''in Proc. ACM CCS, 2003, pp. 262 – 271.

[2] "Snort: Lightweight intrusion detection for networks," Sourcefire, Inc., Columbia, MD[Online]. Available: http://www.snort.org/.

[3] "Bro: A system for detecting network intruders in real time," Lawrence Berkeley National Laboratory, Berkeley, CA [Online]. Available: http://www.bro-ids.org

[4] J. William and W. Eatherton, "An encoded version of reg-ex database from Cisco Systems provided for research purposes," 2005.

[5] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner, "Algorithms to accelerate multiple regular expressions matching for deep packet inspection," in Proc. ACM SIGCOMM, 2006, pp. 339–350.

[6] Scott Tyler Shafer, Mark Jones, "Network edge courts apps,"http://infoworld.com/article/02/05/27/020527newebdev_1.html

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

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

[9] S. Wu, U. Manber, "A fast algorithm for multi-pattern searching," Tech. R. TR-94-17, Dept. of Comp. Science, Univ of Arizona, 1994.

[10] D.R. Sidhu and V. K. Prasanna, "Fast regular expression matching using FPGAs," In IEEE Symposium on Field- Programmable Custom Computing Machines, Rohnert Park, CA, USA, April 2001.

[11] Antichi, G. Di Pietro, A. Ficara, D. Giordano, S. Procissi, G. Vitucci, "Second-Order Differential Encoding of Deterministic Finite Automata," Dept. of Inf. Eng., Univ. of Pisa, Pisa, Italy Nov. 30 2009-Dec. 4 2009.