Available online at http://www.journalcra.com

International Journal of Current Research
Vol. 4, Issue, 04, pp.165-168, April, 2012

INTERNATIONAL JOURNAL OF CURRENT RESEARCH

ISSN: 0975-833X

RESEARCH ARTICLE

A HIGH THROUGHPUT PATTERN MATCHING USING BYTE FILTERED BIT_SPLIT ALGORITHM

[1]Jennifer Christy, J., [1]Manikandababu, C.S. and [2]Rathish, C.R

[1]M.E. VLSI Design, Sri Ramakrishna Engineering College, Coimbatore, India
[2]M.E VLSI Design, Karpagam University, Coimbatore, India

ARTICLE INFO

Article History:
Received 29th January, 2012
Received in revised form
24th February, 2012
Accepted 07th March, 2012
Published online 30th April, 2012

Key words:
Aho–Corasick (AC) algorithm;
Finite automata;
Pattern matching;
Intrusion detection System.

ABSTRACT

The phenomenal growth of the Internet in the last decade and society's increasing dependence on it has brought along, a flood of security attacks on the networking and computing infrastructure. Intrusion Detection Systems (IDSs) have become widely recognized as powerful tools for identifying, deterring and deflecting malicious attacks over the network. Essential to almost every intrusion detection system is the ability to search through packets and identify content that matches known attacks. Network Intrusion Detection and Prevention Systems have emerged as one of the most effective ways of providing security to those connected to the network, and at the heart of almost every modern intrusion detection system is a pattern matching algorithm. Pattern matching relies on deterministic finite automata (DFA) to search for predefined patterns. Here modifications to the Aho-Corasick pattern-matching algorithm are proposed that drastically reduce the amount of memory required and improve its performance. For Snort rule sets, the new algorithm achieves 30% of memory reduction compared with the traditional Aho–Corasick algorithm. In addition, we can gain further reduction in memory by integrating our approach to the bit-split algorithm which is the state-of-the-art memory-based approach.

INTRODUCTION

Computer networks are vulnerable to enormous break-in attacks, such as worms, spyware, and denial-of-service attacks. These attacks intrude publicly accessible computer systems to eavesdrop and destroy sensitive data, and even hijack a large number of victim computers to launch new attacks. Hence, there has been widespread research interest in combating such attacks at every level, from end hosts to network taps to edge and core routers. Network Intrusion Detection and Prevention Systems (NIDS/ NIPS) have emerged as one of the most promising security components to safeguard computer networks. The worldwide NIDS/NIPS market is expected to grow from $932 million in 2007 to $2.1 billion in the next five years [3]. NIDS/NIPS, such as Snort and Bro, perform real-time traffic monitoring to identify suspicious activities. Deep Packet Inspection (DPI) is the core of NIDS/NIPS, which scans both packet header and payload to search for predefined attack signatures. To define suspicious activities, DPI typically uses a set of rules which are applied to matching packets. A rule consists at least of a type of packet, a string of content called signature string to match.

As link rates and traffic volumes of Internet are constantly growing, DPI will be faced with the high-performance challenges. In high-speed routers, DPI is required to achieve packet processing in line speed with limited memory space. First, string matching is computationally intensive.

**Corresponding author: jenni.jeys@gmail.com*

DPI adopts string matching to compare every byte of every incoming packet for a potential match again hundreds of thousands of rules. Thus, string matching becomes the performance bottleneck of DPI. For example, the routines of Aho–Corasick string matching algorithm account for about 70% of total execution time and 80% of instructions .Since software-based string matching algorithms cannot keep up with 10–40 Gbps 40 Gbps packet processing, hardware-based string matching algorithms have been recently proposed to improve the throughput of DPI. These hardware-based algorithms exploit modern embedded devices, such as Application Specific Integrated Circuit (ASIC), Field Programmable Gate Array (FPGA), Network Processor (NP), and Ternary Content Addressable Memory (TCAM), to perform high-speed packet processing, using their powerful parallelism and fast on-chip memory access.

ARCHITECTURE

Deep Packet Inspection (DPI) is essential for network security to scan both packet header and payload to search for predefined signatures. String matching using Deterministic Finite Automaton (DFA) will be the performance bottleneck of DPI. The recently proposed bit-split string matching algorithm suffers from the unnecessary state transitions problem. The root cause lies in the fact that the bit-split algorithm makes pattern matching in a ‘‘not seeing the forest for trees” approach, where each tiny DFA only processes a b-

bit substring of each input character, but cannot check whether the entire character belongs to the alphabet of original DFA.

A. *String Matching Engine*

At a high level, our algorithm works by breaking the set of strings down into a set of small state machines. Each state machine is in charge of recognizing a subset of the strings from the rule set. Our architecture is built hierarchically around the way that the sets of strings are broken down. At the highest level is the full device. Each device holds the entire set of strings that are to be searched, and each cycle the device reads in a character from an incoming packet, and computes the set of matches. Matches can be reported either after every byte, or can be accumulated and reported on a per-packet basis. When adding rules to modules, there are a several issues that need to be handled. First, care needs to be taken not to overflow a rule module (in terms of either number of rules or number of states allowed The exact nature of the rule update policy is left open to IDS system designer.

B. *Aho-Corasick Algorithm*

The objective of the Aho-Corasick algorithm is to find all substrings of a given input string that matches against some set of previously defined strings. These previously defined strings are called patterns or keywords. The pattern matching machine consists of a set of states that the machine moves through as it reads one character symbol from the input string in each cycle. The movement of the machine is controlled by three types of state transitions: normal transitions (successful character matches), error transitions (when the machine attempts to realign to the next-longest potential match), and acceptance (successful matching of a full string).

BIT-SPLIT STRING MATCHING OVERVIEW

In essence, the bit-split string matching algorithm is based on Aho–Corasick algorithm [8] to construct a set of tiny alphabets and tiny DFAs which process every input character in parallel. For a given set of rules, Aho–Corasick algorithm constructs an original alphabet and original DFA. To clarify this paper and distinguish items, a state of the original DFA is called an original state, whereas a state of the tiny DFA is called a tiny state. Aho–Corasick algorithm [8] is a classic exact string matching algorithm. The key of Aho–Corasick algorithm is to construct an original DFA that encodes all signature strings to be searched for. The original DFA is represented with a tree, which starts with an empty root node that is the initial state. The construction process of an original DFA is as follows. First, each signature string adds states to the tree, starting at the root and going to the end of the string. The tree has a branching factor that is equal to the number of unique characters in an original alphabet. Second, the tree is traversed and failure edges are added to shortcut from a partial suffix match of one string to a partial prefix match of another. The string matching of the original DFA is easy: given a current state and an input character, the DFA makes a normal transition on the character to a next state, or makes a failure transition. Fig. 1 depicts an example original DFA for a set of strings {he, she, his, hers}. In the DFA, the initial state is 0, and accepting states are 2, 5, 7 and 9. As seen in Fig. 1, a real line denotes a normal transition, while a dashed line denotes a failure transition to the initial state.

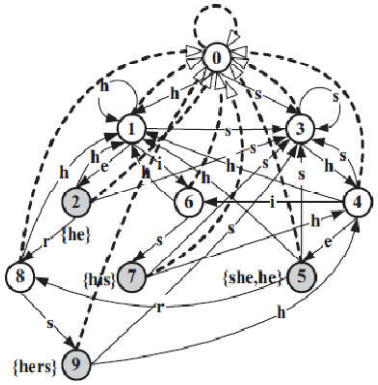

Figure1. Original DFA for a set {he, she, his, hers}

The bit-split algorithm has two stages to construct a set of tiny alphabets and tiny DFAs. The first stage splits an original alphabet of Aho–Corasick algorithm apart into a set of tiny alphabets. A character consists of m bytes (m P 1). For example, an English character has one byte, while a Chinese character has two bytes. Each character is divided into a number of equalized portions called substrings, each with b bits. Thus, each substring of every unique character in the original alphabet is extracted to construct a tiny alphabet. The second stage splits an original DFA of Aho–Corasick algorithm apart into a set of 8m/b tiny DFAs. For a given tiny state and a tuple of tiny alphabet, we search for a subset of possible next original states, labeled as a next tiny state, to construct a tiny DFA. Note that a tiny state may contain a subset of original states and an accepting tiny state is set when it contains any accepting original state. Fig. 2 depicts the original alphabet and two tiny alphabets for a set of strings {he, she, his, hers}. In Fig. 2(a), the original alphabet is R = {h, s, e, i, r}, and each unique character has one 8-bit byte. In Fig. 2(b), the first and second bits are extracted to construct the left tiny alphabet RB_{01}, and the third and fourth bits are extracted to construct the right tiny alphabet RB_{23}. In a tiny alphabet, each tuple (row) consists of a 2-bit substring and a subset of unique characters. For instance, the second tuple in RB_{01} contains a substring 01 and a subset {e, i}. The matching process of bit-split algorithm is as follows. First, an input character is divided into a set of b-bit substrings. Second, each b-bit substring is extracted to feed to the corresponding tinyDFA, and then all tiny DFAs make transitions to next tiny states in parallel.

Char	Decimal	7	6	5	4	3	2	1	0
h	104	0	1	1	0	1	0	0	0
s	115	0	1	1	1	0	0	1	1
e	101	0	1	1	0	0	1	0	1
i	105	0	1	1	0	1	0	0	1
r	114	0	1	1	1	0	0	1	0

(a) Original alphabet

B_{01}

1	0	CharSet
0	0	{h}
0	1	{e, i}
1	0	{r}
1	1	{s}

B_{23}

3	2	CharSet
0	0	{s, r}
0	1	{e}
1	0	{h, i}
1	1	{}

(b) Tiny alphabets

Fig. 2. Original alphabet and two tny alphabets for a set { he, she, his, hers}

BYTE-FILTERED STRING MATCHING

Here, we present a byte-filtered string matching algorithm to overcome the unnecessary state transitions problem. The core idea of the byte-filtered algorithm is that each character of every incoming packet is filtered by checking whether the character belongs to the original alphabet, before performing string matching. If the character is in the original alphabet, the character is divided into a set of substrings, and all tiny DFAs make state transitions on the substrings in parallel for bit-split string matching. If not, each tiny DFA either makes a failure transition or stops any state transition. Furthermore, we propose a novel optimization scheme to speedup the byte-filtered algorithm. In the scheme, given that an input character is not in the original alphabet, each tiny DFA checks to see whether its current state is the initial state before making a state transition. If not, the tiny DFA makes a transition to the initial state. Otherwise, the tiny DFA stops any state transition. The main purpose of the optimization scheme is to further avoid the unnecessary state transitions in hardware to improve the throughput of DPI. In the byte-filtered algorithm, we use Bloom filters as the byte filter to pre-process every input character.

A. *Bloom Filters Overview*

Bloom filters are a simple space-efficient data structure for fast approximate set-membership queries. Bloom filters are widely used in most network applications from Web caching to P2P collaboration to packet processing .Bloom filters represent a set S of n items by an array V of m bits, initially all set to 0. Bloom filters use k independent hash functions h1, h2, . . ., h_k with the range [1, . . .,m]. For each item s 2 S, the bits $V[h_i(s)]$ are set to 1, for i=1…. k. To query if an item u is in S, we check whether all bits $V[h_i(u)]$ are set to 1. If not, u is not in S. If all bits $V[h_i(u)]$ are set to 1, it is assumed that u is in S.

B. *Byte-Filtered String Matching Engine*

We present a novel architecture of byte-filtered string matching engine, suitable for hardware implementation such as FPGA and ASIC. We also describe the optimization schemes for hardware design and implementation of the engine. Fig. 3 depicts a top-level diagram of byte-filtered string matching engine architecture. First, the byte-filtered engine reads as input a data stream with the window size w. Second, each input character of multiple bytes is transferred to the byte filter which checks to see whether the character is in the original alphabet per clock cycle. If not, four tiny DFAs including B_{01}DFA, B_{23}DFA, B_{45}DFA, and B_{67}DFA, either make a transition to the initial tiny state or stop any state transition. If the input character is in the original alphabet, multiple bytes are of the character divided into four parts, each of which is feed to the corresponding tiny DFA. Four tiny DFAs output the partial match vectors including PMV_0, PMV_1, PMV_2, and PMV_3, which are transferred to the logical bitwise AND unit to take an intersection. Finally, the full match vector is outputted and the matched signatures are reported to system administrators.

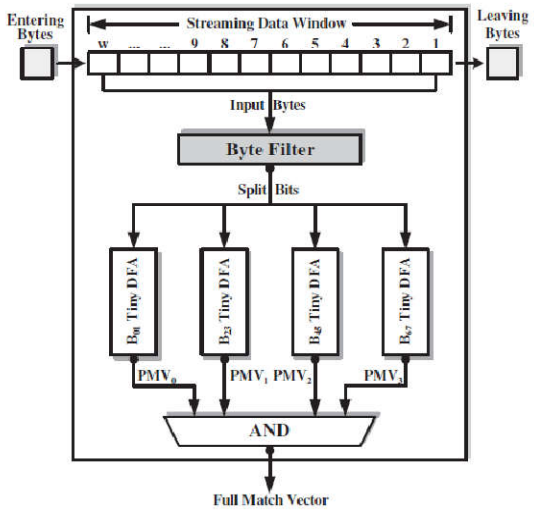

Figure 3. Byte filtered string matching Engine

EXPERIMENTAL EVALUATION

In this section, we present the software simulation experiments to compare the byte-filtered algorithm with the bit_split algorithm. In the experiments, we measure the performance metrics including the string matching time and the area for state transition. We designed and implemented both the bit-split and byte-filtered algorithms with VHDL. We synthesize or choose a set of signature strings, and generate a set of testing strings to evaluate the performance of string matching. The simulation process results is showm below. First, the simulator converts the set of signature strings into one DFA, and then the DFA is split into a set of tiny DFAs for both the bit-split and byte-filtered algorithms.

Logic Utilization	Bit_Split Algorithm	Byte Filtered String Matching Algorithm
No of Slice Flipflops	20	18
No of 4 input LUTs	40	43
Total Equivalent Gate Count	431	418
Peak Memory Usage	147 MB	144 MB

CONCLUSION

Existing bit-split string matching algorithm suffers from the unnecessary state transitions problem. The bit-split algorithm makes pattern matching in a ‘‘not seeing the forest for trees” approach, where each tiny DFA only processes a substring of

each input character, but cannot check whether the entire character belongs to the original alphabet. This paper proposes a byte filtered string matching algorithm using Bloom filters. The byte-filtered algorithm pre-processes each character of every incoming packet by checking whether the character belongs to the original alphabet, before performing bit-split string matching. We further optimize the byte-filtered algorithm for hardware implementation, and theoretically analyze the performance gain in terms of the number of state transitions. Experimental results on synthetic rule set show that compared with the bit split algorithm, the byte-filtered algorithm reduces the string matching time by up to 15.6 times and the number of state transitions of tiny DFAs by about 30%; when the alphabet size increases, the string matching time ratio of the byte-filtered algorithm decreases.

REFERENCES

[1] A. V. Aho and M. J. Corasick, "Efficient string matching: An AID to bibliographic search," CommunACM vol. 18, no. 6, pp. 333– 340, 1975.

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

[3] M. Alicherry, M . Muthuprasanna, and V. Kumar, "High speed pattern Matching for network IDS/IPS," in Proc. IEEE Int. Conf. Netw. Protocols (ICNP) , 2006, pp. 187–196.

[4] B. Brodie, R. Cytron, and D. Taylor, "A scalable architecture for high-throughput regular expression pattern matching," in Proc. 33rd Int. Symp. Comput. Arch. (ISCA) , 2006, pp. 191–122.

[5] Z. K. Baker and V. K. Prasanna, "High-throughput linked-pattern matching for intrusion detection systems," in Proc. Symp. Arch. For Netw. Commun. Syst. (ANCS) , Oct.2005, pp. 193–202

[6] Application layer packet classifier for Linux, 2008. <http://l7-filter.sourceforge.net>.

[7] S. Sen, O. Spatscheck, D. Wang, Accurate, scalable in-network identification of P2P traffic using application signatures, in: Proceedings of WWW 2004,Manhattan, 2004.

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

****