An Efficient Memory Architecture For Network Intrusion Detection Systems Using Pattern Partitioning And Parallel String Matching

[1,]T.Aswini Devi, [2,] Mrs. K.Surya Kumari.
[1,2,]M.Tech Project student, Department of ECE, Pragati Engineering College,
Surampalem, Kakinada, A.P, India.

Abstract

Due to the advantages of easy re-configurability and scalability, the memory-based string matching architecture is widely adopted by network intrusion detection systems (NIDS).The ability to inspect both packet headers and payloads to identify attack signatures makes network intrusion detection system (NIDS) a promising approach to protect Internet systems. In this paper, we propose a memory-efficient pattern-matching algorithm which can significantly reduce the memory requirement. Using the pattern dividing, the variety of target pattern lengths can be mitigated, so that memory usage in homogeneous string matchers can be efficient. In order to identify each original long pattern being divided, a two-stage sequential matching scheme is proposed for the successive matches with sub-patterns. We synthesized this design using Quartus II 11.0 version.

Keywords: Computer network security, finite automata,Parallel processing, Deterministic-finite automata (DFA).

1. INTRODUCTION

As Proliferation of Internet applications increases, security becomes a problem within network solutions. Intruders attempt to break into publicly accessible victim systems to misuse the functionality provided. Traditional network based security devices such as firewalls, performing packet filtering on packet headers only, fail to identify attacks that use unsuspicious headers. By inspecting both packet headers and payloads to identify attack signatures, network intrusion detection system (NIDS) is able to discover whether hackers/crackers are attempting to break in or launch a denial of service (DOS) attack.Because most of the known attacks can be represented with strings or combinations of multiple substrings, string matching is one of the key components in NIDS. String matching in NIDS is computationally intensive in that, unlike simple packet classifications, NIDS needs to scan both the headers and the payloads of each incoming packet for thousands of suspicious strings. Worse, the string lengths are variable. As a result, string matching has become the bottleneck in NIDS to address the requirement of constantly increasing capacity. A string matching engine can have multiple string matchers for parallel string matching. Due to the slow speed of the software-based string engine, the hardware-based string matching engine is preferred due to great parallelism for the high-performance IDSs. In particular, the memory-based string matching engine allows on-the-fly update of memory contents for high re-configurability. However, there are several well-known challenges: high throughput, regularity, scalability, and low memory requirements.

Especially, in the memory-based string matching engine, the string matching based on deterministic-finite automaton (DFA) is frequently adopted due to the deterministic transitions between states according to input symbols; state transitions can be performed in a fixed number of cycles, where the throughput can be maintained unchanged. In addition, due to the fixed number of output transitions in a state, regularity can be guaranteed in the DFA-based string matching engine. Scalability can be supported by the homogeneity of multiple string matchers where DFAs are mapped. Because of the deterministic transitions between states, however, memory requirements are proportional to both the number of states and the number of transitions in a state. The total cost of a string matching engine is directly related to memory requirements; therefore, the target pattern information should be compressed. In the traditional Aho-Corasick algorithm and the bit-split DFA-based string matching, common prefixes between target patterns are shared in a DFA.First, the (attack) string patterns are compiled to a finite state machine (FSM) whose output is asserted when any substring of input strings matches the string patterns. Then, the corresponding state table of the FSM is stored in memory. For the pattern identification, a state should contain its own match vector with a set of bits, where each bit represents a matched pattern in the state. Even though the information of shared common infixes was stored in match vectors, the number of shared common infixes was limited by the size of the match vectors. In addition, throughput could decrease due to the modified state transition mechanism. In the memory requirements for match vectors were reduced by relabeling states and eliminating the match vectors of non-output states. By sharing common infixes of target patterns or relabeling states and eliminating the match vectors of non-output states, the memory usage in the match vectors could be efficient The pattern-matching problem considered here is that of searching for occurrences of a pattern string within a larger text string. Stated formally, given a pattern x with length $|x| = m$ and a text y with length $|y| = n$, where $m, n > 0$ and $m \leqslant n$, the task is to determine if x occurs within y. This basic problem is found in many application domains of network intrusion detection systems (NIDS)

2. Pre Design Information

A. Target Patterns

A target pattern and a set of its k sub patterns, which are obtained after dividing the target pattern, are denoted as **P**i and **Q**i={SP$_{i1}$;SP$_{i2}$; . . . ;SP$_{ik}$}respectively. The subscript **i** is the index of the target pattern. However, the variety of target pattern lengths is another serious problem in achieving regularity and scalability with low hardware cost. Each pattern consists of multiple character codes, where the number of character codes is defined as the pattern length. According to the rule sets, the distribution of pattern lengths could be different each other. In addition, the variation of pattern lengths in each rule set is irregular. If target patterns are to be mapped onto multiple homogeneous string matchers, memory usage cannot be balanced without considering different pattern lengths.

B. Pattern Dividing

The target pattern has set of k sub-patterns, which are obtained after dividing the target pattern, are denoted as Pi and Qi respectively. A set of k sub-patterns Qi will be called the quotient vector of Pi. The fixed length of k sub-patterns is denoted as f. If the length of a target pattern is shorter than f, the target pattern does not need to be divided, so the pattern is defined as the short pattern. The remnant pattern Ri represents a suffix or residual sub-pattern of the target pattern Pi that succeeds the quotient vector of Pi. The match in a sub-pattern is encoded with a quotient index, which represents the unique index of the sub-pattern match. The remnant pattern Ri should be matched sequentially after the quotient vector Qi is matched.

C. Identification Of Pattern

Consider four target patterns {“abc,” “abcd,” “ac,” “bcd”} are mapped on a DFA, where target pattern lengths range from 2 to 4. The fourth target pattern is a suffix of the second target pattern. If the second target pattern is matched, the fourth target pattern is always matched, but not vice versa. We let a target pattern Pi be a suffix of another target pattern Pj for Pi is not equal to Pj. If different identification indexes are provided for the matches with Pi and Pj, respectively, Pi is explicitly identified when Pj is matched. If only the identification index for Pj is provided, Pi is implicitly identified. In this case, only the target pattern with the longest prefix from the initial state has its own identification index in the implicit identification; therefore, users could detect matches with Pi after analyzing the identification for Pj with extra effort. In addition, a target patternPi can be identified after its quotient vector Qi and its remnant vector Ri are matched in order.

D. Bit- Split Dfa

DFA is an FSM where there is one and only one transition to a next state according to each pair of state and input symbols. DFA can be represented with a five-tuple: a finite set of states (Q), a finite set of input symbols (P), a transition function, an initial state, and a set of output states. The identification index of a target pattern is an individual keyword used to distinguish the target pattern match. The memory requirements of DFA are proportional to the size of Q and P.

3. Proposed Scheme

A. Deterministic Finite Automaton

In automata theory, a branch of theoretical computer science, a deterministic finite automaton (DFA)—also known as deterministic finite state machine—is a finite state machine that accepts/rejects finite strings of symbols and only produces a unique computation (or run) of the automaton for each input string. 'Deterministic' refers to the uniqueness of the computation. In search of simplest models to capture the finite state machines, McCulloch and Pitts were among the first researchers to introduce a concept similar to finite automaton in 1943. The following figure shows the structure of DFA.

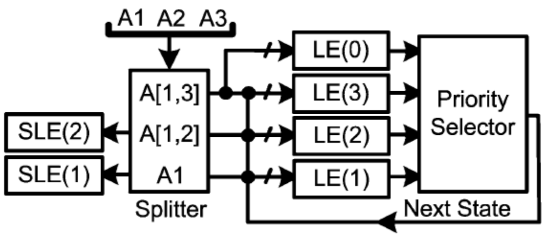


Fig.1. Structure of DFA

In the automaton, there are three states: S0, S1, and S2 (denoted graphically by circles). The automaton takes finite sequence of 0s and 1s as input. For each state, there is a transition arrow leading out to a next state for both 0 and 1. Upon reading a symbol, a DFA jumps deterministically from a state to another by following the transition arrow. For example, if the automaton is currently in state S0 and current input symbol is 1 then it deterministically jumps to state S1. A DFA has a start state (denoted graphically by an arrow coming in from nowhere) where computations begin, and a set of accept states (denoted graphically by a double circle) which help define when a computation is successful.

A DFA representing a regular language can be used either in an accepting mode to validate that an input string is part of the language, or in a generating mode to generate a list of all the strings in the language. In the accept mode an input string is provided which the automaton can read in left to right, one symbol at a time. The computation begins at the start state and proceeds by reading the first symbol from the input string and following the state transition corresponding to that symbol. The system continues reading symbols and following transitions until there are no more symbols in the input, which marks the end of the computation. If after all input symbols have been processed the system is in an accept state then we know that the input string was indeed part of the language, and it is said to be accepted, otherwise it is not part of the language and it is not accepted. The generating mode is similar except that rather than validating an input string its goal is to produce a list of all the strings in the language. Instead of following a single transition out of each state, it follows all of them. In practice this can be accomplished by massive parallelism or through recursion. As before, the computation begins at the start state and then proceeds to follow each available transition, keeping track of which branches it took. Every time the automaton finds itself in an accept state it knows that the sequence of branches it took forms a valid string in the language and it adds that string to the list that it is generating. If the language this automaton describes is infinite (ie contains an infinite number or strings, such as "all the binary string with an even number of 0s) then the computation will never halt.

B. Aho-Corasick Algorithm

Aho-Corasick String Matching Automaton for a given finite set P of patterns is a (deterministic) finite automaton G accepting the set of all words containing a word of P. Transition table is built during the preprocessing part. Where at each state, there is in-formation about where to jump to for each character. It just traverses the string to be matched making transitions, the transition function which tells which state to jump for each character. Whenever we reach a state F, a match is reported by the engine. For simple string matching cases, it does not performs very well but when there are multiple patterns or pattern matching is done at regular expression level, it is one of the best options for pattern matching. Aho-Corasick Matching, implementation first forms a combined DFA for all patterns. Since this is preprocessed during the initialization part, there is no overhead of DFA formation for each pattern and also no (individual or set of) patterns traversal. And for each new character we have to just take one step. But the memory overheads are huge. Also, the state holding at each step is huge because there are multiple copies of active DFA's since a new DFA gets activated at each new character input other than the existing DFA's. Of course some go out also but difference is huge. But power of the algorithm is, it is unaffected by the variance in size of the patterns and worse and average case performance is same.The enhanced design on Aho-Corasick uses an optimized vector storage design for storing the transition table. This memory efficient variant uses sparse matrix storage to reduce the memory requirements and further improve performance on large pattern groups.
Sparse-Row format
Vector: 0 0 0 2 4 0 0 0 6 0 7 0 0 0 0 0 0
Sparse-Row Storage: 8 4 2 5 4 9 7 11 7
Now for each DFA state rather than having a 256-size vector of which most are 0 values, we use sparse matrices to present the transition element and the corresponding value. Cleary since we cannot have O(1) transition time in this implementation, since we need to traverse this new vector to find the transition element. The memory requirements go down by four times which is quite significant.

C. Fsm Tiles

In a string matcher, several homogeneous FSM tiles take n bits as an input at every cycle. In the state of each FSM tile, the pattern identification information is stored as a partial match vector (PMV), where the ith bit indicates whether the ith pattern is matched or not in the state. A pattern can be identified with a full match vector (FMV), which is obtained with the logical AND operation of PMVs in all FSM tiles.

The different types of FSM tiles in can be adopted. The number in the angle brackets describes the field width.

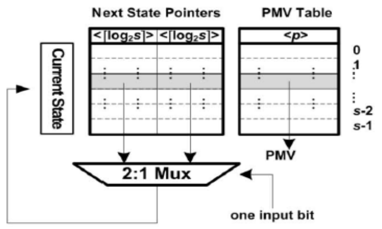


Fig.2a.

Every state can indicate its PMV. A difference of the FSM tile in Fig. 2a is that the FSM memory for storing next-state pointers can be separated from the PMV table. As shown in Fig.2b,

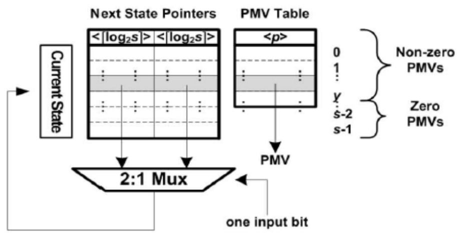


Fig.2b.

If there is no need to have PMVs in several states, the memory allocation for the states is not required; only several PMVs are stored in a PMV table. The stored PMVs are defined as nonzero PMVs; the PMVs to be reduced are defined as zero PMVs. When many PMVs can be shared between multiple states, the FSM tile type in Fig. 2c is beneficial by

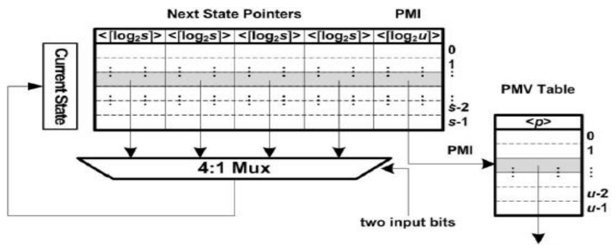


Fig.2

A pattern match index (PMI) in each state indicates a unique PMV for the state. By adopting a separate PMV table, the memory requirements for storing repeated PMVs can be eliminated.

D.Pattern Mapping And Partitioning

The pattern mapping for a string matcher, where sorted patterns ST and string matcher parameter M are given. Initially, for loop checks whether all FSM tiles can be built when the number of mapped patterns is maximum. A variable denotes the number of mapped patterns in each turn. With front patterns from the given sorted patterns, mapped t, a procedure Build tries is called to obtain the tries for FSM tiles in a string matcher. Then, if the largest number of states in tries is greater than the maximum number of states in an FSM tile, next iteration is continued by reducing by one; otherwise, the loop is broken after obtaining the mappable patterns, mapped t. The unmapped patterns are returned by a procedure Remove. After exiting for loop, a procedure Add failing pointer is called to add failing pointers from each state to the longest suffix state. Finally, for the mapped patterns mapped t, the contents of the PMVs are obtained by calling a procedure called Set PMV s.

Pattern Mapping Algorithm:

```
procedure PM(sorted patterns ST, string matcher M)
   t   ST
   for   =p(M) to 1 do
     mapped_t   front(   t)
     tries   Build_tries(mapped_t)
     if max(Num_states(tries) > s(M) then
        continue
     else
        Remove(mapped_t, t)
        break
     end if
   end for
  dfas    Add_failing_pointer(tries,mapped_t)
  fsms    Set_PMV s(dfas, mapped_t)
 return fsms and t
 end procedure
```

Pattern Partitioning Algorithm:

Procedure PP denotes the pattern partitioning with patterns T and string matcher parameter M. First, patterns are sorted lexicographically to increase the number of shared common prefixes. Then, a procedure PM, which denotes the pattern mapping, is called for obtaining the FSM tile contents for a string matcher. Then, the unmapped patterns are returned. The obtained FSM tile contents are stored in *vec_fsms*. The pattern mapping is repeated until there are no unmapped patterns.

```
Procedure PP(patterns T, string matcher M)
    t   Sort(T)
```

```
        While t        do
            fsms, t    PM(t,M)
            vec_ fsms= vec_ fsms+ fsms
        end while
        return vec_fsms
    end procedure
```

The pattern mapping algorithm shows the constant time complexity. Therefore, the time complexity for partitioning total patterns can be O(T), where T denotes the number of patterns. On the other hand, the time complexity of pattern sorting can be O (T.log2T). However, due to the large constant factor of the pattern mapping, if T is not sufficiently large, the pattern sorting will not be dominant.

4. Results

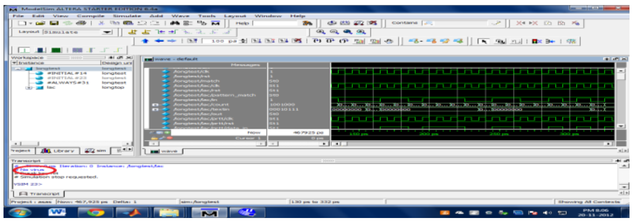


Fig.3.a. Window showing No Virus

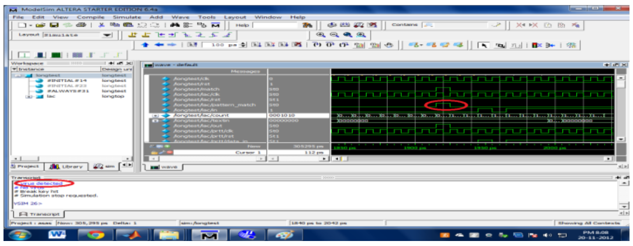
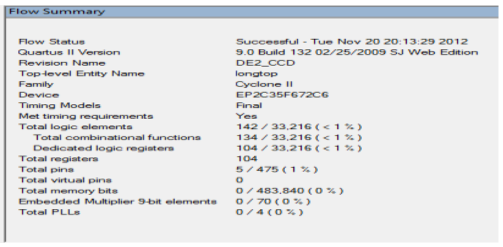


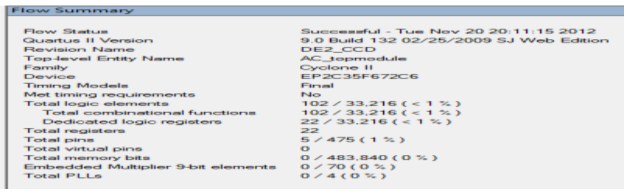
Fig.3.b. Window showing Detected Virus



Flow Summary

Flow Status	Successful - Tue Nov 20 20:13:29 2012
Quartus II Version	9.0 Build 132 02/25/2009 SJ Web Edition
Revision Name	DE2_CCD
Top-level Entity Name	longtop
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Met timing requirements	Yes
Total logic elements	142 / 33,216 (< 1 %)
Total combinational functions	134 / 33,216 (< 1 %)
Dedicated logic registers	104 / 33,216 (< 1 %)
Total registers	104
Total pins	5 / 475 (1 %)
Total virtual pins	0
Total memory bits	0 / 483,840 (0 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

Fig.3.c. Area used by Existing System



Flow Summary

Flow Status	Successful - Tue Nov 20 20:11:15 2012
Quartus II Version	9.0 Build 132 02/25/2009 SJ Web Edition
Revision Name	DE2_CCD
Top-level Entity Name	AC_topmodule
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Met timing requirements	No
Total logic elements	102 / 33,216 (< 1 %)
Total combinational functions	102 / 33,216 (< 1 %)
Dedicated logic registers	22 / 33,216 (< 1 %)
Total registers	22
Total pins	5 / 475 (1 %)
Total virtual pins	0
Total memory bits	0 / 483,840 (0 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

Fig.3.d. Area Used by Proposed System

Extensive verification of the circuit design is performed using the Verilog and then synthesized by the Quatrus II 9.0 to demonstrate the feasibility of the proposed architecture design for Intrusion Detection Systems.

5. Conclusion

The proposed DFA-based parallel string matching scheme minimizes total memory requirements. The problem of various pattern lengths can be mitigated by dividing long target patterns into sub-patterns with a fixed length. The memory-efficient bit-split FSM architectures can reduce the total memory requirements. Considering the reduced memory requirements for the real rule sets, it is concluded that the proposed string matching scheme is useful for reducing total memory requirements of parallel string matching engines.

References:

[1] Snort-the de Facto Standard for Intrusion Detection/Prevention, [Online]. Available: www.snort.org

[2] S. Antonatos, K. G. Anagnostakis, and E. P. Markatos, "Generating realistic workloads for network intrusion detection systems," presented at the Proc. ACM Workshop on Software and Performance, Redwood Shores, CA, 2004.

[3] S. Iyer, A. Awadallah, and N. McKeown, "Analysis of a packet switch with memories running slower than the line rate," in Proc. IEEE INFOCOM, Mar. 2000, pp. 529–537.

[4] Embedded Memory, [Online]. Available: http://www.ti.com/research/ docs/cmosmemory.shtml

[5] G. Navarro and M. Raffinot, Flexible Pattern Matching in Strings-Practical On-Line Search Algorithms for Texts and Biological Sequences. Cambridge, U.K.: Cambridge Univ. Press, 2002.

[6] C. J. Coit, S. Staniford, and J. McAlerney, "Towards faster string matching for intrusion detection or exceeding the speed of snort," in Proc. DARPA Information Survivability Conf. Exposition (DISCEX II'01), 2001, pp. 367–373.

[7] M. Fisk and G. Varghese, "Fast content-based packet handling for intrusion detection," UCSD, UCSD Tech. Rep. CS2001–0670, 2001.

[8] K. G. Anagnostakis, E. P. Markatos, S. Antonatos, and M. Polychronakis, "E XB: A domain-specific string matching algorithm for intrusion detection," presented at the 18th IFIP Int. Information Security Conf., Athens, Greece, 2003.

[9] R. T. Liu, N. F. Huang, C. H. Chen, and C. N. Kao, "A fast string-match algorithm for network processor-based network intrusion detection system," ACM Trans. Embedded Comput. Syst., vol. 3, pp. 614–633, 2004.

[10] J. Moscola, J. Lockwood, R. P. Loui, and M. Pachos, "Implementation of a content-scanning module for an Internet firewall," in Proc. 11th Annu. IEEE Symp. Field-Programmable Custom Comput. Mach., Napa, CA, Apr. 2003, pp. 31–38.