JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 26, 1563-1582 (2010)

Two-phase Pattern Matching for Regular Expressions in Intrusion Detection Systems

CHANG-CHING YANG, CHEN-MOU CHENG AND SHENG-DE WANG
Department of Electrical Engineering
National Taiwan University
Taipei, 106 Taiwan

Regular expressions are used to describe security threats' signatures in network intrusion detection (NID) systems. To identify suspicious packets using regular expression matching, many NID systems use memory-based deterministic finite-state automata (DFA) with one-pass-scanning model, which is fast and allows dynamic updates. However, a number of practical signature patterns commonly found in a variety of NID systems, *e.g.*, "`.*A.{N}B`", can cause a state-explosion problem in such a model. In this paper, we propose a two-phase pattern matching engine (TPME) to solve this problem. In our proposed approach, the state storage cost is reduced to linearly dependent on the number of repetitions `N` in the patterns. With the new approach, we are now able to handle those practical patterns that would have caused the state-explosion problem in memory-based DFA. We report our implementation of TPME on a field programmable gate array (FPGA). With our prototype implementation, we can achieve a throughput of more than 1.86 gigabits per second for pattern matching in a practical NID system.

Keywords: network intrusion detection, pattern matching, regular expressions, deterministic finite-state automata, two-phase matching engine

1. INTRODUCTION

In recent years, several types of approaches are proposed and applied to detect attacks against network systems. Their aim is to monitor and filter data traffic arriving in a network to avoid exposing host computers to security threats, in which a third party may, *e.g.*, try to connect to a host and take advantage of the weaknesses in the operating systems or applications.

For detecting malicious attacks, signature matching is the most well-known method in state-of-the-art *network intrusion detection* (*NID*) *systems*. A simpler form is a screening firewall, which can look up header fields of a packet and filter the packet according to a set of rules consisting of Internet Protocol (IP) addresses for source and destination, source and destination port numbers, as well as protocol number.

For deep packet scanning, NID systems not only focus on the header fields but also have to check signatures in the data payload portion of a packet. These signatures are usually a set of strings described by regular expressions. The most widely used NID systems like Snort [1, 2] and Bro [3, 4] use regular expressions to describe attack signatures in their rule sets.

Because of the large traffic volume and complexity of the process, signature matching can easily become the performance bottleneck in deep packet inspection NID sys-

Received March 5, 2009; revised June 22, 2009; accepted August 13, 2009.
Communicated by Tzong-Chen Wu.

tems. It has been shown that in a typical NID system, 40-70% of the total processing time and 60-85% of the operations are spent in string pattern matching [5].

Many NID systems are purely software-based, which are flexible but can not keep up with the traffic of heavily loaded high-speed networks. It is possible to distribute the work to the host computers in a network to offload the computation, but this approach will impose extra load on the host computers, not to mention that the installation and maintenance of software is difficult to manage. For this reason, it is often preferable to have a set of dedicated NID systems with hardware accelerators to achieve high throughput.

In a typical NID system, the most performance-critical part is the pattern matching process. There are two kinds of mechanisms to accelerate this process: circuit-based and memory-based. In the former approach, the rules are translated into a set of comparators, and the input is processed by each comparator in parallel. The advantage is the high-speed parallel comparison, but the circuit-based approach has a fatal drawback, namely, the comparator circuit is fixed and can not be changed easily. In other words, if we want to change the rule sets dynamically, we would have to redesign the comparators. Memory-based approaches, on the other hand, employ a controller to process the input data with high-speed memory look-ups, allowing rule sets to be updated at run time.

Due to the blooming variations of security threats, being able to update the rule sets on line is of crucial importance to commercial NID systems. In this paper, we will present a low storage-cost solution to memory-based pattern matching engine with on-line update capability. We will also report our implementation and performance analysis of the proposed approach on a field programmable gate array (FPGA) evaluation board.

The rest of this paper is organized as follows. In section 2, we will first review necessary background information and present the problems that may be encountered by a regular-expression recognizer for deep packet inspection. In section 3, we will review relevant works in the literature. In sections 4 and 5, we will discuss our proposed algorithms, accelerator architecture, and implementation details. We then present the result of our prototype implementation in section 6 and conclude this paper in section 7.

2. BACKGROUND AND PROBLEM STATEMENT

2.1 Regular Expressions

A regular expression consists of constants and operators denoting the sets of strings and the operations over these sets, respectively. Given a finite alphabet set Σ and subsets R, S of Σ, the following three basic operations are defined.

- **Concatenation:** RS denoting the set $\{\alpha\beta | \alpha \in R, \beta \in S\}$.
- **Alternation:** $R | S$ denoting the set union of R and S.
- **Kleene star:** R^* denoting the smallest superset of R that contains the empty string and is closed under string concatenation. This is the set of all strings that can be made by concatenating zero or more strings in R.

There is a straightforward mapping between regular expressions and finite-state

automata that recognize whether the input data match certain regular expressions [6]. We can build finite-state automata to identify the patterns described using regular expressions. Finite-state automata are 5-tuples: $M = (Q, \Sigma, q_0, F, \delta(q, i))$, in which Q is a finite set of non-terminals or states, Σ is a finite set of terminals, q_0 is the start state, F is the set formed by final states, and $\delta(q, i)$ is the transition function given the current state q and the input i, which is often expressed as a transition table.

There are two kinds of finite-state automata: the *Deterministic Finite-state Automata* (*DFA*), for which there is only one next state given the current state and input, and the *Non-deterministic Finite-state Automata* (*NFA*), for which there can be more than one possible next state given the current state and input.

In both DFA and NFA, the matching process using finite-state automata can be illustrated by the example shown in Fig. 1. If arriving at an accepted state, portrayed as the double circle in Fig. 1, we say that the string in the input tape is *accepted* by the finite-state automata. That is, this string *matches* some regular expression patterns which are equivalent to this finite-state automaton.

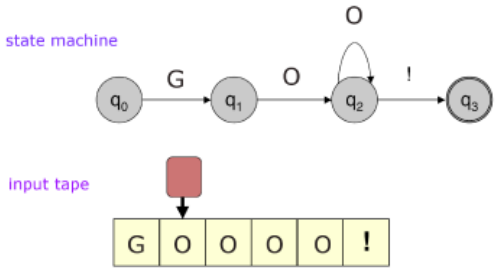


Fig. 1. An example of matching a pattern using FSA.

Characters in an alphabet Σ play two roles: one is *literals*, and the other *meta-characters*, *i.e.*, characters representing extra operations to combine regular expressions in various ways. We will use the POSIX extended regular expression syntax with the GNU flex [7] extension, in which a backslash is used to *suppress* the meaning of meta-characters, as shown in Table 1.

We note that there are other regular expression flavors, such as the Perl Compatible Regular Expressions (PCRE) [8], which wedo not consider in this paper.

We typically build FSA as the language recognizer to check if a fixed-length string belongs to the language defined by the regular expressions to recognize. When the string is matched from start to end by an FSA corresponding to the regular expressions, we say this string is in the language, or simply a *match*. But there could be other scenarios in which we never know what the start or end position is, *e.g.*, when we want to check whether there is any sub-string of the input string that belongs to the language. In this situation, we could separate FSA into two types according to the execution model: *one-pass-scan matching* and *repeated-scan matching* [9].

In the former type, we could find all sub-strings starting from any position when the input is scanned from start to end. In order to achieve this goal, we should prepend a regular expression "`.*`" to each pattern without "^". The advantage of this approach is that the computation cost is constant, independent of the length of the input string. How-

Table 1. List of literals and meta-characters in GNU flex regular expressions.

Regex	Characters should be matched
x	Match the character 'x'
.	Any character except to newline.
[xyz]	A "character class"; matches either an 'x', a 'y', or a 'z'.
[a-z]	A "character class with a range"; any letter from 'a' through 'z'
[^A-Z]	A "negated character class"; any letter but those in the class.
"[xyz]\"foo"	Matches the literal string "[xyz]"foo".
{name}	Matches the character in the user defined class "name".
\0	Matches a NUL character (ASCII code 0).
\123	Matches the character with octal value '123'.
\x2a	Matches the character with hexadecimal value '2a'.
Regex	**Matching condition**
(r)	Matches an r; parentheses are used to override precedence
rs	Matches the regex r followed by the regex s.
r|s	Matches either an r or an s.
r*	Matches zero or more r's, where r is any regular expression.
r+	Matches one or more r's.
r?	Matches zero or one r's, i.e, an optional r.
r{2,5}	Matches anywhere from two to five r's.
r{2, }	Matches two or more r's.
r{4}	Matches exactly 4 r's.
Regex	**Matching anchoring**
^r	Matches an r, but only at the starting of scan, or after a newline.
r$	Matches an r, but only at the end of a line, or before a newline.

ever, it could invoke an exponentially-sized state transition table.

In the latter type, we apply the recognizing process to any start position to find all possible matches. In other words, if we find all matches and reach the end of the input from start position P_i, we need to start the next recognizing process from the next character position in input P_{i+1}. This approach is commonly used in language parser but is not suitable for packet payload scanning because of the inefficiency and low matching probability.

2.2 A Motivating Example and its Analysis

We have the following regular expression from the rule set in Bro for detecting IMAP login buffer-overflow attempt: "`.*LOGIN[^\x0a]{100}`". This matches packet payload that starts with `LOGIN`, followed by 100 non-newline characters from any position. The recognizing process is shown in Fig. 2.

Fig. 2. The recognizing process for pattern "`LOGIN[^\x0a]{100}`".

For the general case, the worst case in terms of number of states is that a regular expression of length n can be recognized by an NFA of $O(n)$ states, constructed by the *Thompson's Construction Algorithm*. If the NFA is converted to a DFA by *Subset Construction Algorithm*, then the number of states will be bounded by $O(\Sigma^n)$ [6]. On the other hand, the processing time for each input character is $O(1)$ in the DFA approach, whereas the processing complexity of an NFA is $O(n^2)$ if all states can be activated when transferring to the next state.

When we want to handle a group of k regular expressions, we can compile all regular expressions to one composite automaton, or we can build k individual automata. For DFA-based systems, the processing complexity is $O(1)$ when compiled into a composite automaton but $O(k)$ when built as k individual automata. The number of states will grow to $O(\Sigma^{kn})$ instead of $O(k\Sigma^n)$ for building k individual automata. On the other hand, for NFA-based systems, if we compile k regular expressions into a composite NFA, we get the result $O((kn)^2)$ for processing and $O(kn)$ for number of states. We can also choose to build k individual automata, resulting in $O(kn^2)$ for processing and $O(kn)$ for the number of states.

From the discussion above, we can see that DFA with all signature patterns compiled together is the best solution when considering processing cost first. A low processing cost is invaluable for NID systems. However, the storage requirement may be exponential on the pattern size n in the worst case. In fact, there exist patterns that make the number of states grow exponentially in the rule sets of practical NID systems. The common feature is that the pattern starts with "`.*`", and there is a class or a wildcard metacharacter such as "`[^A]`" or "`.`" in the middle of the pattern. Moreover, the interaction amongst signature patterns also leads to an exponentially-sized number of states if multiple signature patterns are compiled into one composite DFA. For instance, if we compile two signature patterns "`.*A.*B`" and "`.*C.*D`" into one DFA, there will be a lot of states that do not exist in the individual DFA. In the following, we will analyze why these two types of DFA with exponential sizes occur.

Type 0: Interactions inside a Single Signature Pattern

The common features of these patterns are that they have length restriction to a class of characters that could overlap with the prefix. For example, the signature pattern "`.*A.B`" and its corresponding DFA are shown in Fig. 3.

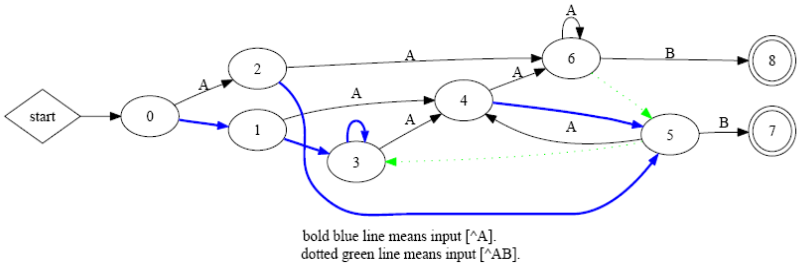


Fig. 3. The corresponding DFA for signature pattern "`.*A.B`".

In order to maintain the deterministic property of DFA, it has to remember all pos-

sibilities that A combined with the subsequent characters before B. Therefore, the number of total states is 9, as opposed to 4: we will need to match some arbitrary characters, the character A, an arbitrary character, and finally the character B. Therefore, when we expand the pattern to "`.*A.{N}B`", the number of states grows like $O(2^N + c)$, where c is the string length.

In real-world NID systems, there are a significant number of signature patterns with features of type 0. A majority of them are for detecting buffer-overflow attempts. In Bro 1.3.2, there are 105 such patterns out of 1278 regular expression signature patterns, whereas in Snort 2.4, there are 123 out of 385.

Moreover, the problem becomes more serious as a result of applying the long length restriction to this type of patterns. The length usually ranges from 0 to 100, but some of them can be as large as 512 or 1024, suggesting that the number of states could be incredibly large, like 2^{512} or 2^{1024}, making them impossible to implement in practice.

Type 1: Interactions among Multiple Signature Patterns

This type of interaction exists when there is a partial match to one signature pattern, which also belongs to another signature pattern. Because of the deterministic property, we need to add extra states to record all possible transitions if we adopt the DFA approach.

As an example, consider two patterns "`.*A.B`" and "`.*C.D`". The sub-pattern "`A.B`" from the former matches the sub-pattern "`.*`" from the latter. If we want to determine which pattern we have matched, we need to introduce some extra states to the automaton. Furthermore, if we have "`.*A.{N}B`" and "`.*C{N}.D`", we would have $O((k + 1)^N)$ states when we compile k signature patterns of type 0 into one DFA. As mentioned previously, there are a large number of patterns of type 0. When they are compiled together, the type 1 interaction will occur, resulting in explosion in the number of states.

3. RELATED WORK

3.1 String Pattern Matching

Traditionally, the signatures in NID systems have been specified as pre-defined strings corresponding to a set of well-known issues. Many efficient algorithms have been proposed, *e.g.*, Aho-Corasick [10] and Wu-Manber [11]. They use a pre-processed structure to parse the input data. There are many enhancements based on their works. For example, Tuck *et al.* have introduced the use of bitmap and path compression to reduce memory requirement [12].

Tan *et al.* present the bit-split algorithm to split an Aho-Corasick automaton into binary state machines to reduce the memory requirement [13]. Dharmapurikar *et al.* propose a hardware architecture based on parallel Bloom filters [14]. A Bloom filter is a space-efficient probabilistic data structure used to test set membership. Aldwairi *et al.* suggest a reconfigurable memory-based accelerator [15]. The accelerator is part of a configurable network-processor architecture. The software running on a 2-wide multiple-issue VLIW processor generates a finite-state machine and creates the state tables, with string matching based on the Aho-Corasick algorithm. Baker *et al.* use a set of comparators pipelined with output flip-flops to identify data bytes [16].

3.2 Pattern Matching using Regular Expressions

Most works about string matching can not be applied directly to regular expression matching because the latter allows a variable amount of repetition. Instead, people use automata-based approaches to develop matching engines for regular expressions. The NFA approach is not efficient in sequential computing, while the DFA approach generally has a high space cost.

In recent years, researchers are interested in hardware-based architecture to accelerate the matching process. When doing so, they mostly implement the finite-state automata using standard logic gates. As pioneers in the field, Floyd *et al.* show that an NFA can be efficiently implemented in programmable logic array [17]. Because a set of logic gates can be triggered simultaneously, the NFA could go through multiple next states at the same time. More recently, Sidhu *et al.* show that an NFA can be implemented efficiently in hardware as flip-flops and comparators [18]. Clark *et al.* improve the system by replacing the comparators with a global decoder [19]. In the same time, Moscola *et al.* use a similar method to implement DFA and demonstrate the improvement in terms of throughput [20].

This kind of approach is efficient in terms of logic resources and successfully delivers high processing throughput. However, they are inflexible because of the fixed interconnections among the logic gates and flip-flops. In other words, changing any pattern in the signature set will require changing the connections on the chip. The FPGA technology is helpful, but it still requires a relatively long configuration time, and the systems will need to be shut down during the reconfiguration.

Other researchers focus on memory-based hardware architectures. Yu *et al.* propose an algorithm to partition a large set of patterns into multiple groups to reduce the memory requirement dramatically [9]. The key point is to partition the set to reduce or even to eliminate the interactions among patterns. However, each packet may need to traverse more than one automaton. They also propose new algorithms for selecting patterns in general-purpose processor architectures to solve this problem.

Kumar *et al.* introduce a new representation, called the Delayed Input DFA, which considerably compresses the memory usage [21]. The main idea is to replace some common transitions with a default transition while guaranteeing the same accepted situation at the same time. As a side effect, it will result in more memory look-ups. Kumar *et al.* propose the Content Addressed Delayed Input DFA, which mitigates the problem by replacing the state numbers with content labels [22].

Brodie *et al.* use a novel pipelining strategy that defers state-dependent logic to the last stage, enabling multiple transitions in a single cycle. At the same time, a regular-expressions compiler is used to encode contiguous strings of input characters and compress the transition table through the introduction of an indirection table [23].

Unfortunately, many NID systems in the real world contain complex regular expressions that can result in an exponentially-sized DFA. Without modifying the signature patterns, Becchi *et al.* propose a hybrid automata solution, which involves a head-DFA and tailed-FAs [24]. They are able to successfully reduce the total number of states by replacing the part of DFA that may result in an exponential size by an equivalent NFA. But the processing speed highly depends on the memory bandwidth because it still needs more than one concurrent memory operations. Finally, it also possible to rewrite the rules to avoid these complex regular expressions altogether, as in, *e.g.*, Yu *et al.* [9].

4. PROPOSED APPROACHES OF MATCHING

The basic idea of our proposed approach is to partition signature patterns into groups, *e.g.*, partitioning each signature pattern of type 0 into two parts. Then, by applying hash algorithms and building the grouped DFA approach to the modified repeated-scan matching model, we can solve the state-explosion problem discussed in section 3.

4.1 Motivation of Hybrid Matching

As we have mentioned in section 3, if we choose the one-pass-scan matching as our search model, the meta-character "`.*`" should be prepended to the original pattern. Then, if there are sub-patterns with a wildcard character and a repetition constraint to this wildcard character in any rule, the exponentially-sized DFA corresponding to the type 0 regular expression is generated.

Hence, the question of concern is: what problems the other search model "repeated-scan matching" has on the search process? The consensus is that this kind of model is inefficient to packet payload scanning processes because the chance of the packet payload matching a particular pattern is considerably low [9]. In other words, if the matching probability is low, then most state transitions are in vain, resulting in a low throughput. If we were to try a hybrid system, the repeated-scan matching model would not be able to keep up with the throughput of the one-pass-scan matching model.

However, if we can apply the "repeated-scan matching" model to these DFA that correspond to the regular expressions of type 0, then we would be able to avoid prepending the meta-character "`.*`" and break the structure of type 0. As a result, we can avoid having a large amount of extra states and reduce the memory requirement directly.

4.2 Multi-staged Partition

We observe that these exponentially-sized DFA corresponding to the regular expressions always contain one character string, which we shall call the *prefix string*, before the sub-pattern with a wildcard character and its repetition constraint, *e.g.*, the character '`A`' in the prototype of the exponentially-sized DFA with type 0, "`.*A.{N}B`". It is a key observation that if there is a string sub-pattern in the original pattern, we can partition it into two parts and match the two parts sequentially. Furthermore, if the first string matching fails, we can skip the second string matching. Because the probability that a packet matches a particular pattern is low, most of the work will be in the first part.

The analysis above gives us a hint to modify the repeated-scan matching model that would allow us to apply existing algorithms to the string matching process. The traditional repeated-scan matching model processes one character and transit the DFA to the next state each cycle. When transiting the DFA to a failed state, the DFA will return to the position next to the starting position. However, our modified repeated-scan-matching processes a string from the starting position and jumps to a position next to the starting position or starts the following DFA matching process, depending on whether the string matching process fails or succeeds.

This modification gives a chance to the repeated-scan matching model. If we guarantee that we can process one character in the input packet each cycle, then we can meet

the processing throughput of the DFA using the one-pass-scan matching model.

We choose a hash function to achieve this goal. Applying a hash function to this problem, we can calculate the digest of the string and use it to address the entry in the memory. If the entry is empty, then there is no associated string in the pattern set.

In the case of collisions, we can use the standard methods of resolution, *e.g.*, linear probing or chaining [25]. In any case, collisions are not fatal in string-matching problem because in the face of a false positive, the following input data still need to match the remaining part of the rule, resulting in a much lower probability of misidentification. In the extreme case, we can build an extra DFA to check the result in the string hash process if we need to completely eliminate false positives. This DFA traversal could be executed in parallel to the second stage.

To summarize, we propose to partition the type 0 regular expressions into two parts and apply a hash process to the first string matching process as a filter. Only when the result of this filter is matched, we trigger the remaining DFA matching process.

4.3 Grouping

When applying multi-staged structure to the matching process, only one DFA should be triggered in the second stage (or two in the situation where there are two patterns with the same prefix). It is because every complete match should pass the first stage, and the number of entry points of the second stage DFA corresponding to the partial match in the first stage is less than two mostly. So we do not have to run the DFA k times. The process is shown in Fig. 4.

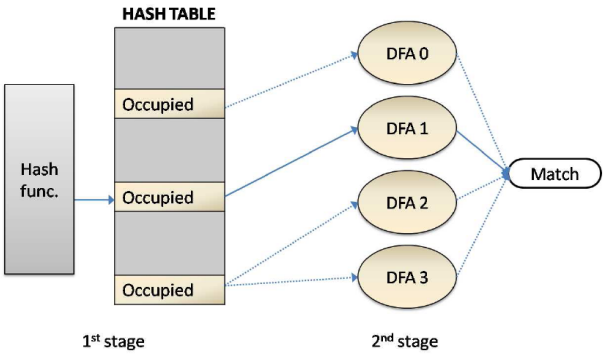


Fig. 4. An example of separate DFA built from multi-staged partitioning.

On the other hand, after analyzing the rule sets of a few popular NID systems [1, 3], we have found that some signature patterns corresponding to type 0 have the same kind of sub-patterns in the remaining part. We view this situation as *having the same suffix*. In the normal DFA construction, we could not share these suffixes, because DFA have to maintain all the state information to distinguish different signature patterns. In our proposed multi-staged partition, on the other hand, we could separate the signature patterns with the same suffix part into the same group and then construct one DFA for each group.

For instance, if there are two patterns, namely, "`ABC.{10}T`", where T means the trailing sub-pattern, and "`DEFG.{10}T`", then we could let the remaining part "`.{10}T`" use the same DFA as the recognizer.

In this way, we can further reduce the memory cost and keep the same ability to recognize signature patterns in rule sets. Finally, the matching process and the possible memory look-up method are illustrated in Fig. 5.

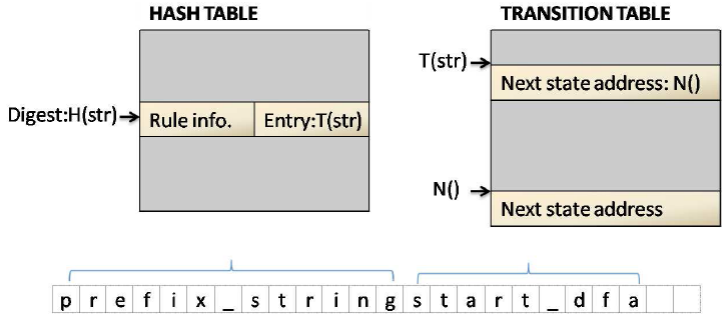


Fig. 5. The proposed two-phase matching process.

5. IMPLEMENTATION

We describe our TPME implementation that works side by side with a traditional DFA recognizer. Our implementation uses software-based contents pre-processing with hardware-based run-time matching circuit. The maximum throughput could be up to 8*h* characters per cycle, where *h* is the number of hash engines. This result can easily keep up with the performance of the state-of-the-art DFA matching engines.

5.1 Choice of Hash Function

We consider three classes of hash algorithms that can be easily implemented in hardware: (1) addition-multiplication-based methods, (2) bitwise operation-based methods, and (3) cyclic redundancy check (CRC) code-based methods.

Some hash algorithms discard position information, whereas CRC algorithms, which are widely used in error detecting and correcting context, preserves position information as a hash function [26].

We experimentally compare a few commonly used hash algorithms for string hashing. The first *basic-hash algorithm* uses character-wise addition and multiplication operations. The second *lh-strhash algorithm* is from the openSSL library, which uses exclusive or (XOR) and shift operations. The third *32bit-FNV algorithm* uses XORs and multiplications modulo a special prime number 16777619. The last one is *CRC-CCITT algorithm* with polynomial $X^{16} + X^{12} + X^5 + 1$.

The comparison will focus on the collision rate, which is defined as the probability that more than one string map to the same entry of the hash table. Our simulation generates a large number of different strings composed of common characters with the ASCII

code ranging from 0x30 to 0x7E, *i.e.*, `0..9:;<=>?..Z[\]`a..z{|}`, and then calculate hash values. The simulation result is shown in Fig. 6.

Fig. 6 shows that the difference in collision rates for these four algorithms is almost negligible across a wide variety of different scenarios, compared with say table size. When we enlarge the table, the collision rate decreases.

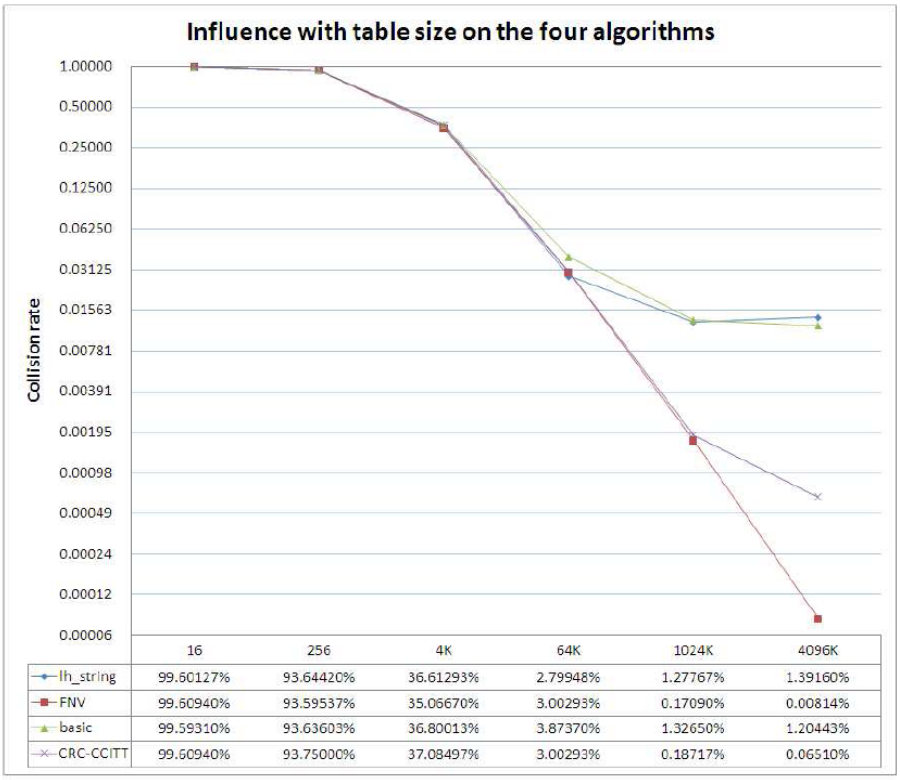


Fig. 6. Collision rates of four hash algorithms under consideration.

Table 2. Hardware utilization of the four hash algorithms under consideration.

Resource type	*lh-strhash*	*FNV*	*basic*	*CRC-CCITT*
Flip Flops	325	319	319	324
LUTs	483	432	255	552
DSP48s(multipliers)	0	3	7	0
path delay (ns)	3.739	34.657	19.429	3.731

We also take resource utilization into account. We implement these four algorithms in the same environment and synthesize them using Xilinx's development tool ISE 9.2i. Table 2 compares the hardware utilization of the four hash functions under consideration.

Based on Table 2, we can say that "CRC-CCITT" and "lh-strhash" are good candidates as our first-stage hash function because they can be implemented by simple XOR

operations, resulting in lower hardware cost and shorter delay on the critical path. The FNV algorithm could also be a decent choice if hardware cost is not a concern.

5.2 Pre-processing

Since our proposed approach is memory-based, we have to prepare the contents stored in the memory before we can run our algorithm. We do not include this cost in this paper because it does not contribute significantly to the total run-time cost.

5.2.1 The format of memory contents

We need two tables: one for hash algorithm look-up and the other for recording the state transitions. The format of these two tables is shown in Fig. 7. The layout of the first-stage is 24-bit wide, including 8 bits for recording the pattern information and 16 bits for storing the address of the corresponding entry in the state transition table. Note that the entry address could be the same as any other rules when they have the same starting state in the state transition table.

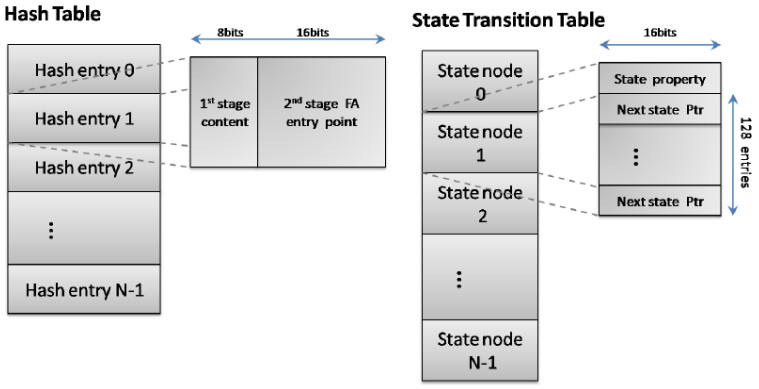


Fig. 7. The memory content layout.

Meanwhile, the content format of the second stage is 16-bit wide for each entry. Every state has a "state property" for recording state-related information, *e.g.*, that this state is an accepted state or an intermediary state. Followed the state property field, there are 128 or 256 records to store the "next state" information. The size of the records should be determined by the character encoding, *e.g.*, 128 for 7-bit ASCII code and 256 for 8-bit ASCII code.

This layout is bounded by at most 256 prefix strings and up to 65536 states in the transition table. Currently these bounds appear to be sufficient but can be adjusted accordingly as the rule sets change in the future.

5.2.2 Translator

After determining the format of the memory content, we now turn to the content it-

self. Here we develop a translator that records regular expression patterns into the two tables. Then, we check the match by looking up in the tables at run time. We list the processing steps in Algorithm 1.

Algorithm: Memory content pre-processing

Input: Regular expression pattern sets
Output: Memory content of first-stage and second-stage

```
1  begin
2      Find similar suffix sequence from the input rule sets;
3      Partition these rules into k groups;
4      for i = 0 to k − 1 do
5          Split every rule in group_i into a prefix-string and
           the remaining part;
6          if stringlength > 8 then
7              Cut the string and store the extra part into a
               temp queue;
8          Calculate the Hash digest of the prefix-string;
9          Generate the state transition table of the
           remaining part;
10         Merge the temp queue into the state transition
           table;
11 end
```

Algorithm 1: Pre-processing Algorithm

First of all, we should partition the signature patterns of type 0 into groups according to the suffixes of the patterns. For example, there are 82 signature patterns in the Snort 2.4 rule set (out of 385 distinguished regular expression patterns) with the same suffix "`[ \n]{N}`" and 10 rule patterns with suffix "`[ \n\t]{N}`".

After partitioning signature patterns into groups, we need to partition each pattern in the same group into two parts: prefix string and the remaining part. We developed a regular expression parser to parse the input regular expressions into a string with a length of less than eight characters and the remaining part. The bounds of the length are from analyzing the signature patterns in real-world rule sets, from which we have found that the sizes of these prefix strings are less than eight in general. For long prefix strings, we should cut it and merge the extra characters into the remaining part. We call these characters as extra characters. Note that all the remaining parts have the common suffix sequence, so we could construct one DFA with multiple entries that accept states to recognize them.

Next, we calculate the hash digests from the set of prefix strings and fill in the hash table with the number of the corresponding patterns. At the same time, we build the state transition table using the GNU flex. We then insert the extra characters to the original state transition table and add the entry state to each entry in the hash table. We collect the extra characters using breadth-first search.

We then push all the extra characters into a temporary queue so that we can pop them up and insert them in front of the transition table easily. The detail of the insertion process is shown in Algorithm 2. Note that some of the steps in this algorithm are limited for the state transition tables generated by GNU flex.

Algorithm: Table modification

Input: Temporary hash table, temporary state transition table from flex, temp queue with extra characters
Output: Memory contents of first-stage and second-stage

```
1  begin
2      f ⟵ max number of state;
3      s ⟵ size of temp queue;
4      t ⟵ size of hash table;
5      transitions(State_0) ⟵ 0;
6      for i = f to 5 do
7          transitions(State_{i+(s+6)}) ⟵
           transitions(State_i);
8      for i = 6 to s + 6 do
9          transitions(State_i) ⟵
           (i − 6)^th element of temp queue;
10     transitions(State_{s+5}) ⟵ transitions(State_1);
11     transitions(State_1) ⟵ 5;
12     for k = 1 to t do
13         assign the transition table entry point of k;
14 end
```

Algorithm 2: Tables-modification Algorithm

5.3 Run-time Matcher

After preparing the memory contents, we now need to build a run-time matching circuit. We use Xilinx's ML405 FPGA evaluation board and map the two required memory blocks to an on-chip block RAM and an off-chip SRAM. There is a hardcore PowerPC 405 processor on this FPGA chip. We run test applications on this processor and get the required information on the host PC via RS-232 port.

We note that our goal is to verify our proposed approach and maximize the throughput of the matching process. The complete system architecture is illustrated in Fig. 8. The dotted fields in Fig. 8 are not necessary for verifying our approach, so they are not included in our prototype implementation.

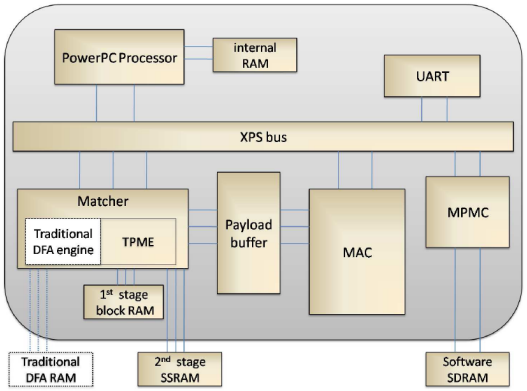


Fig. 8. Complete system architecture.

There are five modules in our implementation: main, data fetch, hash engine, FA engine, and register queue. When the hash engine obtains a matched hash value, it pushes the prerequisite data for the FA engine to the register queue and then continues the first stage matching without halting. The prerequisite data consist of the rule information and the matched position. When there are data in the register queue, the FA engine will be triggered to read the required information and then start the state traversal. At the same time, the first-stage hash process could continue processing data in parallel to the FA engine.

To further improve the throughput of the hash engine, we implement a four-stage pipeline: calculating the hash value, fetching the hash table, checking the table content, and pushing the information into the register queue. The result is that we can process one payload data in the packet buffer memory each cycle, resulting in a throughput of eight bits per cycle.

We also separate the data-fetch module aside, which is responsible for fetching the next data for both hash and FA engines. This module has two internal buffers for hash and FA engines, respectively, since the two engines run in parallel. The data-fetch module is responsible for feeding the two engines while keeping the two internal buffers from being depleted. The design could also minimize the probability that the data-fetch module has to fetch the payload data twice in a single cycle, as the requests from the hash and FA engines are forced to be separated.

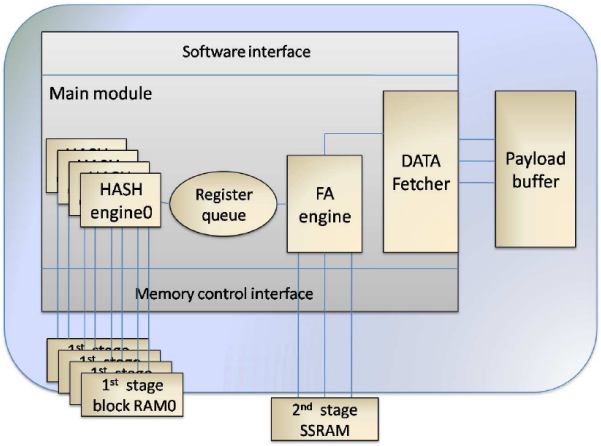


Fig. 9. The TPME architecture.

Our design also allows for multiple hash engines, as depicted in Fig. 9. Note that all hash engines can run in parallel. When any of them gets a hash match, it will push the related information into the register queue and continue the processing. However, because of the duplicated engines, we can consume more than one character in one clock cycle, resulting in a throughput of $8hf$ bits per second, where h is the number of hash engines, and f is the maximum frequency at which the circuit can operate.

If there are extra memory resources, we can have multiple FA engines in a similar way. If the processing bottleneck is the second stage, duplicating the FA engines can relax the size of the register queue and decrease the queuing time of the partial matches from the first stage.

6. EXPERIMENT RESULTS

6.1 Storage-cost Comparison

As mentioned in section 3, the complexity of the number of states of the traditional DFA approach is $O((k + 1)^N)$ when processing type 0 regular expressions. Here we choose four signature patterns in the Bro and Snort NID systems as the benchmark. They are "`.*SSH-[ \t\n][^\n]{200}`", "`.*rename][^\x0a]{1024}`", "`.*http:\\[ \t\n][^\n]{400} `", and "`.*apop[^\x0a]{256} `".

Because of the limitation in GNU flex, we have to shorten the repetition constraint of signature patterns so that GNU flex can generate the corresponding state transition tables. We list the results of various repetition constraints in Table 3.

Table 3. Comparison of number of states for four benchmark patterns.

Repetition Constraint	State Number / Storage cost of Traditional DFA	State Number / Storage cost of proposed TPME
1	56 / 27,848	6 / 2768
2	94 / 47,342	7 / 3155
4	218 / 121,171	9 / 3929
8	1266 / 791,793	13 / 5477
10	3182 / 2,172,049	15 / 6251
16	51522 / 42,277,597	21 / 9329

Table 4. Comparison of number of states for eight benchmark patterns.

Repetition Constraint	State Number / Storage cost of Traditional DFA	State Number / Storage cost of proposed TPME
1	141 / 82,347	10 / 4003
2	288 / 154,971	11 / 4390
4	871 / 502,442	13 / 5164
8	9273 / 6,430,092	17 / 6838
10	30613 / 24,141,899	19 / 7864
16	1124348 / 998,421,024	25 / 10942

On other hand, when we increase the number of signature patterns by adding four extra signature patterns: "`.*HELO\s[^\n]{500}`", "`.*PUT[^\n]{432}`", "`.*file \x3a\x2f\x2f[^\n]{400}`", "`.*User-Agent\x3a[^\n]{216}`", the traditional DFA approach will suffer the state-explosion problem. The result of the eight signature patterns is shown in Table 4.

We observe that our proposed approach can reduce the number of required states for signature patterns. As Fig. 10 shows, the storage cost of our approach is linear to the repetition constraint. Furthermore, the storage cost will increase slightly when adding extra patterns with the same suffix. This shows that our proposed grouping has taken effect, so we can share the suffix between the different signature patterns.

Fig. 10. Storage costs of traditional DFA and TPME for 4 and 8 signature patterns.

6.2 Throughput

We use Xilinx's ISE 9.2i and EDK 9.2i development tools to synthesize our RTL implementation with hash table size of 4096 entries and hash algorithm "lh-strhash".

In the implementation, we introduce the pipelined architecture and a bitmap register array to reduce the look-up latency when visiting the block RAM so that the TPME unit can process one character in each cycle, as predicted in section 5. At the same time, we also compare the implementations with different hash table sizes in Table 5. We find that the required resource is reasonable in most cases, but the required block RAM will become a barrier when we increase the hash table size. For instance, when we increase the size to 65536 entries, the required block RAM will exceed what is available on ML405.

The post synthesis frequency of the TPME unit could reach 233 MHz on ML405. This translates to a throughput of 1.864 gigabits per second.

Table 5. Resource utilization comparison between different hash table sizes.

Resource \ Table size	*256*	*1024*	*4096*	*Available*
Slice Flip Flops	1265	1324	1704	17088
4 input LUTs	1679	2364	5877	17088
FIFO16/RAMB16s	3	4	8	68

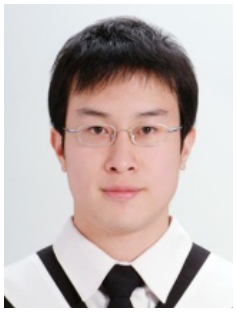
7. CONCLUDING REMARKS

We have considered a class of signature patterns that could cause the state-explosion problem using DFAs to match regular expressions. For these signature patterns, we have shown that they all have a common pattern "`.*A.{N}B`". When using the traditional DFA solution, we might intend to use the one-pass-scan matching model for high-speed matching processes. But for signature patterns belonging to this common type, this model could have an exponentially growing memory cost. We have proposed a partition method to cut a pattern into two matching stages to prevent state explosion from happening. By adopting the partition method, we could use the modified repeated-scan matching model to scan the prefix string repeatedly. Based on the low matching probability, the remaining part matching process would not be triggered frequently. At the same time, by adopting the partition method, we can also share the common suffix from different signature patterns by grouping them appropriately. Finally, we construct DFA with multiple entry points in order to retain the information about the partial matches from the first stage. Our implementation result shows that we can achieve a throughput of 1.864 gigabits per second by choosing the appropriate hash functions to match the prefix string in the first stage. In additions, previous works on compressing the DFA transition tables and speeding up the table look-ups can be applied to the second stage directly. All in all, using TPME as a co-processing unit with the traditional DFA engine could be a sound solution to implement a regular expression matching engine.

REFERENCES

1. "Snort – The de facto standard for intrusion detection/prevention," http://www.snort.org/.
2. M. Roesch, "Snort-lightweight intrusion detection for networks," in *Proceedings of the 13th USENIX Conference on System Administration*, 1999, pp. 229-238.
3. "Bro intrusion detection system," http://www.bro-ids.org/.
4. V. Paxson, "Bro: A system for detecting network intruders in real-time," *Computer Networks*, Vol. 31, 1999, pp. 2435-2463.
5. S. Antonatos, K. G. Anagnostakis, and E. P. Markatos, "Generating realistic workloads for network intrusion detection systems," *SIGSOFT Software Engineering Notes*, Vol. 29, 2004, pp. 2070-215.
6. J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages and Computation*, 2nd ed., Addison-Wesley, MA, 2001.
7. "flex: The fast lexical analyzer," http://flex.sourceforge.net/.
8. "PCRE – Perl compatible regular expressions," http://www.pcre.org/.
9. 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 *Proceedings of the ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, 2006, pp. 93-102.
10. A. V. Aho and M. J. Corasick, "Efficient string matching: An aid to bibliographic search," *Communications of the ACM*, 1975, pp. 333-340.
11. S. Wu and U. Manber, "A fast algorithm for multi-pattern searching," Technical Report TR-94-17, Department of Computer Science, University of Arizona, 1994.

12. N. Tuck, T. Sherwood, B. Calder, and G. Varghese, "Deterministic memory-efficient string matching algorithms for intrusion detection," in *Proceedings of the 23rd Annual Joint Conference of the IEEE Computer and Communications Societies*, Vol. 4, 2004, pp. 2628-2639.
13. L. Tan and T. Sherwood, "A high throughput string matching architecture for intrusion detection and prevention," in *Proceedings of the 32nd Annual IEEE International Symposium on Computer Architecture*, 2005, pp. 112-122.
14. S. Dharmapurikar and J. W. Lockwood, "Fast and scalable pattern matching for network intrusion detection systems," *IEEE Journal on Selected Areas in Communications*, Vol. 24, 2006, pp. 1781-1792.
15. M. Aldwairi, T. Conte, and P. Franzon, "Configurable string matching hardware for speeding up intrusion detection," *SIGARCH Computer Architecture News*, Vol. 33, 2005, pp. 99-107.
16. Z. K. Baker and V. K. Prasanna, "A methodology for synthesis of efficient intrusion detection systems on fpgas," in *Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 2004, pp. 135-144.
17. R. W. Floyd and J. D. Ullman, "The compilation of regular expressions into integrated circuits," *Journal of the ACM*, Vol. 29, 1982, pp. 603-622.
18. R. Sidhu and V. K. Prasanna, "Fast regular expression matching using fpgas," in *Proceedings of the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 2001, pp. 227-238.
19. C. R. Clark and D. E. Schimmel, "Efficient reconfigurable logic circuits for matching complex network intrusion detection patterns," in *Proceedings of the 13th International Conference on Field-Programmable Logic and Applications*, 2003, pp. 956-959.
20. J. Moscola, J. Lockwood, R. P. Loui, and M. Pachos, "Implementation of a content-scanning module for an internet firewall," in *Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 2003, p. 31.
21. S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner, "Algorithms to accelerate multiple regular expressions matching for deep packet inspection," in *Proceedings of the ACM Conference on Applications*, *Technologies*, *Architectures*, *and protocols for Computer Communications*, 2006, pp. 339-350.
22. S. Kumar, J. Turner, and J. Williams, "Advanced algorithms for fast and scalable deep packet inspection," in *Proceedings of the ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, 2006, pp. 81-92.
23. B. C. Brodie, D. E. Taylor, and R. K. Cytron, "A scalable architecture for high-throughput regular-expression pattern matching," *SIGARCH Computer Architecture News*, Vol. 34, 2006, pp. 191-202.
24. M. Becchi and P. Crowley, "A hybrid finite automaton for practical deep packet inspection," in *Proceedings of the ACM CoNEXT Conference*, 2007, pp. 1-12.
25. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed., The MIT Press, Cambridge, MA, 2001.
26. T. Ramabadran and S. Gaitonde, "A tutorial on crc computations," *IEEE Micro*, Vol. 8, 1988, pp. 62-75.
27. Xilinx, "Ml405 evaluation platform user guide," uG210 (v1.5.1).



Chang-Ching Yang (楊長青) was born in Taiwan in 1984. He received the B.S. degree from National Chiao Tung University, Hsinchu, Taiwan, in 2006, and the M.S. degrees in Electrical Engineering from National Taiwan University, Taipei, Taiwan, in 2008. His research interest includes embedded computing and network systems.

Chen-Mou Cheng (鄭振牟) received his B.S. and M.S. in Electrical Engineering from National Taiwan University in 1996 and 1998, respectively, and Ph.D. in Computer Science from Harvard University in 2007. He joined the Department of Electrical Engineering at National Taiwan University in 2007, where he is currently an Assistant Professor. His research interest spans cryptology and cryptanalysis, information security and privacy enhancement technologies, computer and wireless communication networks, and high-performance embedded computing. He currently works in the area of high-performance cryptographic computing.

Sheng-De Wang (王勝德) was born in Taiwan in 1957. He received the B.S. degree from National Tsing Hua University, Hsinchu, Taiwan, in 1980, and the M.S. and the Ph.D. degrees in Electrical Engineering from National Taiwan University, Taipei, Taiwan, in 1982 and 1986, respectively. Since 1986 he has been on the faculty of the Department of Electrical Engineering at National Taiwan University, Taipei, Taiwan, where he is currently a Professor. From 1995 to 2001, he also served as the director of Computer Operating Group of Computer and Information Network center, National Taiwan University. He was a visiting scholar in Department of Electrical Engineering, University of Washington, Seattle during the academic year of 1998-1999. From 2001 to 2003, He has been served as the Department Chair of Department of Electrical Engineering, National Chi Nan University, Puli, Taiwan for the 2-year appointment. His research interests include embedded systems, reconfigurable computing, and intelligent systems. Dr. Wang is a member of the Association for Computing Machinery and IEEE computer societies. He is also a member of Phi Tau Phi Honor society.