PAPER

An Efficient Signature Matching Scheme for Mobile Security

Ruhui ZHANG[†a)], ***Student Member and*** **Makoto IWATA**[†b)], ***Member***

SUMMARY The development of network technology reveals the clear trend that mobile devices will soon be equipped with more and more network-based functions and services. This increase also results in more intrusions and attacks on mobile devices; therefore, mobile security mechanisms are becoming indispensable. In this paper, we propose a novel signature matching scheme for mobile security. This scheme not only emphasizes a small resource requirement and an optimal scan speed, which are both important for resource-limited mobile devices, but also focuses on practical features such as stable performance, fast signature set updates and hardware implementation. This scheme is based on the finite state machine (FSM) approach widely used for string matching. An SRAM-based two-level finite state machine (TFSM) solution is introduced to utilize the unbalanced transition distribution in the original FSM to decrease the memory requirement, and to shorten the critical path of the single-FSM solution. By adjusting the boundary of the two FSMs, optimum memory usage and throughput are obtainable. The hardware circuit of our scheme is designed and evaluated by both FPGA and ASIC technology. The result of FPGA evaluation shows that 2,168 unique patterns with a total of 32,776 characters are stored in 177.75 KB SelectRAM blocks of Xilinx XC4VLX40 FPGA and a 3.0 Gbps throughput is achieved. The result of ASIC evaluation with 180 nm-CMOS library shows a throughput of over 4.5 Gbps with 132 KB of SRAM. Because of the small amount of memory and logic cell requirements, as well as the scalability of our scheme, higher performance is achieved by instantiating several signature matching engines when more resources are provided.

key words: *signature matching, finite state machine, Aho-Corasick algorithm, mobile security*

1. Introduction

With the advent of ubiquitous computing [1], more and more network-based functions and services are being introduced to mobile devices. This development also poses threats to mobile devices, such as Denial of Service (DoS), malware, network viruses, and worm storming. With the increase in threats, mobile security mechanisms are urgently needed.

Signature matching is one way to detect a variety of attacks through discovering the corresponding predefined patterns in the payload of network packets. It has been extensively studied and utilized with network security in personal computers (PCs) and high-end network devices domains. Because of its generality and effectivity, signature matching can also be effectively transplanted into mobile security.

As a content-aware scheme, signature matching scans the entire payload data of every network packet for numerous predefined patterns. As a result, it is computation-intensive and resource-exploitive. The statistic data [2] shows that in Snort [3], a popular open-source network-based intrusion prevention system (NIPS) for PC, up to 70% of the total execution time and 85% of instructions can be dominated by the signature matching routines. Although the present signature set for mobile devices is not as large as the set for high-end network devices, the number of patterns keep expanding rapidly. In addition, it includes some complex pattern types, such as case sensitive/insensitive patterns, very long/short patterns, and enumerative patterns which can only be countered by regular expression matching rather than exact signature matching. Many signature matching schemes have been hitherto proposed, but few of them can be applied to mobile devices mainly because their target platform is assumed to be high-end products with abundant resources.

Our key contribution in this paper is a novel lightweight signature matching scheme which can be embedded as a deep packet filtering in mobile devices with limited resources. Our scheme is an adjustable finite state machine (FSM) scheme which we call two-level finite state machine (TFSM). By dividing the original single FSM into two levels of FSMs, our scheme compacts the required memory space and shortens the critical path of the single-FSM solution. Additionally, it satisfies several essential features for practical mobile security, such as stable performance, fast updates, and scalability. To the best of our knowledge, there is no existing scheme that focuses on hardware-oriented signature matching for mobile security.

The remainder of this paper is organized as follows. Firstly, our motivations and the related works are discussed in Sect. 2. Secondly, the theory of our TFSM scheme and its hardware circuit design are described in Sect. 3. After that, the evaluation results by both Xilinx [4] FPGA and ASIC technology, as well as the comparisons with other algorithms are shown in Sect. 4. Finally, the conclusion and future work are addressed in Sect. 5.

2. Motivations and Related Works

Compared with PCs or high-end network devices that provide fully equipped resources to achieve high throughput, mobile devices are required to obtain an optimal throughput with the lowest cost due to their severe resource constraints. Therefore, signature matching for mobile devices

Manuscript received December 5, 2007.
Manuscript revised April 30, 2008.

[†]The authors are with the Department of Information Systems Engineering, Kochi University of Technology, Kami-shi, 782-8502 Japan.
a) E-mail: 086405j@gs.kochi-tech.ac.jp
b) E-mail: iwata.makoto@kochi-tech.ac.jp

DOI: 10.1093/ietcom/e91–b.10.3251

Copyright © 2008 The Institute of Electronics, Information and Communication Engineers

should be more light-weight with optimal performance-cost ratio. Additionally, there are some requirements confronted by signature matching for mobile security. They are listed as follows:

1. *Stable performance*: Since signature matching is a critical security function, it should be robust and safe itself. This means a stable performance should be maintained in not only average case but also the worst case to avoid intended algorithmic attacks manipulating some craftily-designed payload.
2. *Hardware accelerator*: Signature matching algorithms based on software solutions have been widely studied, and it is generally accepted that they are not powerful enough to support multi-Gbps throughput. In order to achieve a higher throughput, a dedicated hardware accelerator is required for signature matching. Actually, for the devices with a good yield, the ASIC implementation has higher performance-cost ratio per device.
3. *Scalability*: Since the signature set expands rapidly, more patterns will be expressed in the signature matching hardware. This requires the scheme extendable to process more patterns with acceptable additional cost and to maintain its throughput.
4. *Small number of groups*: Signature set grouping is a kind of divide-and-conquer solution. It is denoted as dividing a big signature set into several small groups, and then processing them in parallel. Because the scale of each group becomes smaller, the critical path becomes shorter and the throughput becomes faster. However, the matching results of the independent groups are meaningless, and they should be collected and analyzed to obtain the final matching information. Normally, this further action requires complex logic and has an impact on the total performance. Therefore, the trade off between the number of groups and the total performance should be thoroughly noted.
5. *Fast updates*: In order to detect numerous attacks that emerge continuously and spread fleetly, the signature set needs to be updated frequently and quickly. Since the signature matching function will be interrupted during the updates, a fast update function is required to avoid a long-time inactivation.

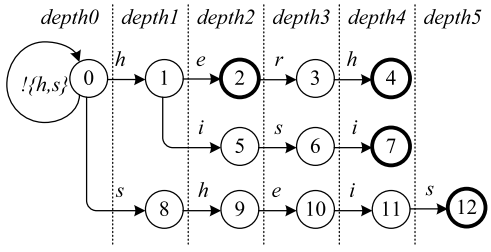
As an extensive research topic of computer science, several signature matching algorithms have been proposed and some of them are being used for network security. They are classified as follows:

a) *Software-based* [5]–[8]: This solution is easy to realize, but it is very slow and resource-exploitive due to the lack of dedicated hardware support.
b) *Logic-cell-based* [9]–[13]: This solution instantiates numerous parallel processing units, such as nondeterministic finite automaton (NFA), deterministic finite automaton (DFA), or pattern tree, by the logic cells, and each unit processes one or several patterns. Therefore, it is restricted to the reconfigurable platforms such as FPGA and suffers from time-consumption, which normally needs tens or hundreds of microseconds, for the reconfiguration of the total system when updating any number of patterns.
c) *B/TCAM-based* [14], [15]: This solution takes advantage of the special storage architecture of content addressable lookup. However, there are some inherent disadvantages of B/TCAM, such as low access speed, inflexible width customizing, high power consumption, high price, and this solution normally aims to high-end network device.
d) *SRAM-based*: Several SRAM blocks and a simple combinational logic are utilized in this solution. It can be realized by FPGA or ASIC chip. Compared with other solutions, SRAM-based solution is platform-flexible, economical, and can easily support fast updates of memory content without reconfiguration of the total system. There are two representative categories within the SRAM-based solution. One is based on hashing and the other uses the Aho-Corasick (AC) algorithm [16]. The hashing-based solution [17], [18] can obtain faster average performance, but it is vulnerable to algorithmic attacks invoking many false-positive events. AC algorithm is a classic FSM-based multi-pattern string matching algorithm (see Sect. 3.1). Because of its stable performance, many researchers choose the AC algorithm as a basic algorithm and use some schemes to improve the throughput and decrease the memory requirement. There are multi-character-input AC (M-AC) algorithms [19], [20] and one-character-input AC (O-AC) algorithms [21]–[23]. M-AC algorithm achieves higher throughput at the cost of larger resources. A moderate scalability of resources can not be guaranteed with the increase of the number of patterns and input characters. When resources are limited, the number of patterns will be compromised to obtain a high throughput or vice versa. On the contrary, O-AC algorithm has the advantage of small resource-consumption suitable for mobile security. It is competitive with the M-AC algorithm if running several instances in parallel. However, none of the existing O-AC algorithms meets all of the requirements listed above.

In the case of mobile security, we first select the SRAM-based hardware implementation, and then propose our TFSM scheme to continue improving O-AC algorithm and satisfy the listed requirements.

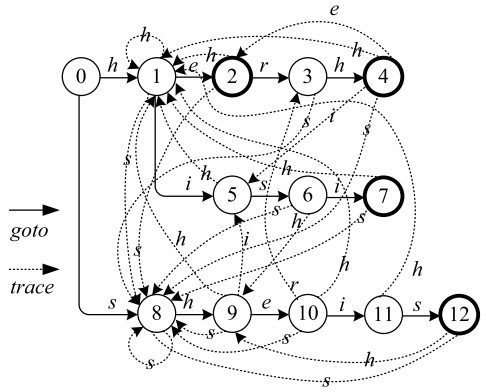
3. Two-Level Finite State Machine Scheme

In this section, we first give the basics of the AC algorithm and its problem, and then introduce our TFSM scheme through explaining its theory and implementation with an example.



s	1	2	3	4	5	6	7	8	9	10	11	12
failure(s)	0	0	0	1	0	8	0	0	1	2	0	8

(a) Trie structure and failure table of AC-FAIL.



(b) DFA of AC-OPT.

Fig. 1 Trie structure of AC-FAIL and DFA of AC-OPT.

3.1 Knowledge of AC Algorithm

The basic theory of the AC algorithm is to process string matching with an FSM to execute a series of state transitions, each of which is determined by the previous state and an input character.

There are two concrete implementations of the AC algorithm, which are the AC-FAIL and the OPT for short [24]. To show the relationship with the AC algorithm, the OPT is called AC-OPT in this paper. The AC-FAIL uses a trie structure and a failure table, while the AC-OPT further preprocesses them into a DFA. For each state in the DFA, the next state by any input character in the alphabet is preprocessed and stored. As a result, the lookup of FSM is required only once to make a state transition. For the AC-FAIL, when the state transition by an input character is failed, the failure table will be referred to, and trace to a new state which denotes the longest prefix by the inputs up to now, and then an additional state transition from this state by this character will be carried out once again. With the depth of the trie structure becomes deeper, the references to the failure table and the additional state transitions may be executed more than once.

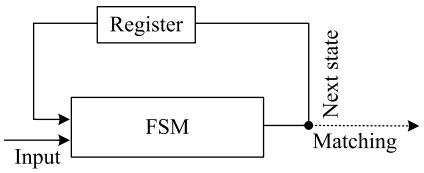
The examples of AC-FAIL and AC-OPT are shown in Fig. 1 based on the signature set $\{he, herh, hisi, sheis\}$. The reduced alphabet is $\{e, h, i, r, s\}$. In Fig. 1(b), all state transitions from a certain state to state 0 are not marked. For example, it is given that the current state is state 9, and character "*i*" is input. In the trie structure of the AC-FAIL, there is no state transition from state 9 by "*i*," so the failure table is referred to and state 1 is obtained. After that, the state transition from state 1 by "*i*" is taken and the state 5 is achieved. On the contrary, in the DFA of the AC-OPT, there is a state transition pointing from state 9 to state 5 by "*i*" already. All the states are grouped by their depths. If the depth of a state is i, the state belongs to the set $depthi$. For instance, state 1 and 8 belong to the set $depth1$ in Fig. 1(a). There are two kinds of state transitions in the DFA. One kind of state transition points from a state in $depthi$ to a state in $depthj$ ($i < j$), and it is called a *goto* transition. The other kind of state transition points from a state in $depthi$ to a state in $depthj$ ($i \geq j$), and it is called a *trace* transition. Actually, *trace* transitions are the results of the further preprocessing by the AC-OPT and they help to achieve a deterministic matching speed.

Because the AC-OPT obtains a thorough stable performance, it is widely adopted. The total DFA of the AC-OPT should be stored in memory to construct a signature matching engine. In the original implementation, it is stored in a two-dimension table for the purpose of random access. However, due to the irregular distribution of characters in a signature set adopting ASCII code, the table will be very sparse. Moreover, as the scale of table grows rapidly with the extension of signature set, it is becoming unacceptable even for high-end network devices. From Fig. 1 we can see that, the *goto* transitions set up the framework of the DFA, and they can not be removed because this operation will disconnect the DFA and make it can not be traveled through. On the other hand, as the results of the further preprocessing by AC-OPT, some *trace* transitions can be eliminated by some schemes [15], [19] without influencing the traveling of the DFA. In addition, more *trace* transition eliminations will save more memory. Based on this observation, we propose an SRAM-based scheme which eliminates the *trace* transitions with a compact storage method. Furthermore, we well design it to better support mobile security.

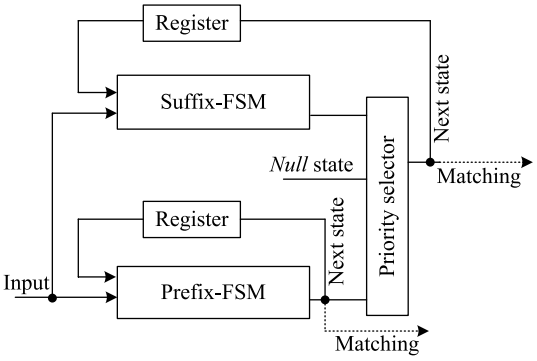
3.2 Theory of TFSM Scheme

In this paper, it is supposed that an input text $T = t_1 t_2 ... t_l$ and a finite set of patterns $P = \{p_1, p_2, \cdots, p_m\}$ over an alphabet Σ ($\Sigma \neq \phi$) are given. The length of a pattern p_m is written as $|p_m|$. A state transition is denoted as $s_k \leftarrow S(s_{k-1}, t_k)$, where s_k is a certain state in the DFA and S can be replaced by a *goto* or *trace* transition. The concatenation of a string u and v is uv. If there is a $p_m = uv$ ($|u| \neq 0$, $|v| \neq 0$), then u is the prefix and v is the suffix of p_m. How to make the division is described below. All the prefixes/suffixes of P construct the prefix/suffix set which we call P_{prefix}/P_{suffix}.

In our SRAM-based two-level finite state machine (TFSM) scheme, an additional FSM for independent P_{prefix} matching is separated from the FSM of the AC-OPT. Its result (state) can be used to substitute the result of some *trace* transitions in P_{suffix} matching, which is taken when a prefix



(a) Structure of AC-OPT.



(b) Structure of TFSM.

Fig. 2 Structure of AC-OPT and TFSM.

appears somewhere in the left FSM of P_{suffix}. In the TFSM scheme, the length of each prefix in the P_{prefix} can be the same or irregular.

As shown in Fig. 2, the unique FSM of the AC-OPT is divided into two levels of FSMs. The FSM separated for P_{prefix} matching is called Prefix-FSM (PFSM), and the FSM left for S_{suffix} matching is called Suffix-FSM (SFSM). In the matching process, these two FSMs will process the input character respectively and concurrently. Because some *trace* transitions in the SFSM have been eliminated by the transition information kept in the PFSM, the next state of the SFSM is determined by the results of both the SFSM and the PFSM. In this case, a priority selector is required for the SFSM to choose the next state with the highest priority. How to set the priority is shown by Table 1. The next state generated by the SFSM, which points to depth > *boundary* (It is a certain depth to separate a pattern into a prefix and a suffix. It can be neatly set to be a fixed value for the whole P, or different values for different patterns.), has the highest priority. The next state generated by the PFSM, which points to 1 < depth ≤ *boundary* or depth 1, is given middle priority or low priority respectively. Actually, next state to the *boundary* depth is the association between the PFSM and the SFSM, whereas next state to 1 < depth < *boundary* is useless for the SFSM. However, we give both of them middle priority in order to simplify the circuit design. For the same purpose, a *null* state is used to substitute the next state to depth 1 generated from the default table [19] (It is introduced in Sect. 3.4.3.) in the PFSM.

The elimination of some *trace* transitions in the SFSM is explained by an example. In Fig. 1(b), state 9 can be reached by the unique *goto* transition from state 8 or two *trace* transitions from state 6 and 12 respectively. The latter two *trace* transitions show that the prefix "*sh*" of pattern "*sheis*" appears from state 6 or 12 to state 9 by character "*h*." Since they finally point to the same destination state, the results of these two *trace* transitions can be substituted by the result of the *goto* transition through matching the prefix "*sh*" independently and adopting the priority selector.

Table 1 Priority setting of the priority selector.

Input	Property of state transitions	Priority
Next state by SFSM	To depth> *boundary*	High
Next state by PFSM	To depth=*boundary*	Middle
	To 1<depth< *boundary* (useless in the SFSM)	
	To depth=1 (useless in the SFSM, substituted by a *null* state)	Low

In [19], *trace* transitions to the states in *depth*1 were eliminated by a default table scheme. It made a partial compression but not a sufficient one. In our TFSM scheme, the default table scheme is adopted in the PFSM level. The advantage of our scheme is that the prefixes matching in the PFSM removes the *trace* transitions pointing to not only depth 1 but also higher depths. In [15], the elimination of *trace* transitions to higher depths was firstly proposed and realized by a solution adopting the content addressable lookup of TCAM. This solution dealt with a single FSM. In order to hold the information to eliminate the *trace* transitions pointing within depth d, $d-1$ characters, which cause *goto* transitions to a node (state), are appended to the identifier of that node in the FSM, and this coding method for the total FSM results in a larger TCAM storage. In our SRAM-based TFSM scheme, the information to eliminate some *trace* transitions is held by extracting the PFSM from the one FSM. As a result, no appended characters are required in the state identifier if the TFSM scheme is realized by B/TCAM, thus the memory requirement is reduced. In addition, the single large FSM is divided into two smaller FSMs working in parallel to shorten the critical path. Moreover, the separation boundary can be not only neatly set along a certain depth d but also irregularly set according to the concrete requirements, e.g. fewer *trace* transitions or the load balance among the memory blocks.

3.3 Algorithm Description of TFSM

The algorithm description of the TFSM scheme is introduced from two aspects: DFA construction and matching process. Because the two FSMs work independently, algorithm descriptions of each are shown respectively if required.

1. *DFA construction algorithm*: We can use the algorithm described in [16] to construct the DFA of the PFSM. The difference is that the signature set of the PFSM is not P but the extracted P_{prefix}. The same algorithm is also adopted in the DFA construction of the SFSM in

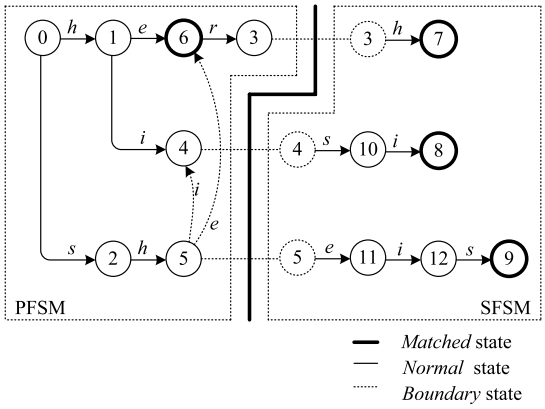


Fig. 3 DFA of TFSM based on set {*he*, *herh*, *hisi*, *sheis*}.

the first step. After that, the states already included in the PFSM and the removable *trace* transitions pointing to the states in the PFSM are deleted. This additional algorithm in the second step is described in Algorithm 1 (see Appendix).

2. *Matching algorithm*: The matching algorithm descriptions of the PFSM and SFSM are summarized in Algorithm 2 and 3 (see Appendix). "Succeed" denotes that the information of a state transition, which is being processed, can be found in its corresponding lookup table. Although we describe the processing by sequential sentences here, there is some parallelism which will be fully utilized by hardware implementation.

3.4 Implementation of TFSM

The detailed implementation of our TFSM scheme is introduced in this section. It includes the utilized data structure, skills for optimization, as well as the hardware circuit design. Examples are adopted to make the explanation clear in some parts.

3.4.1 DFA Construction

At first, the P_{prefix} is exacted. In the example of Fig. 1(b), the *boundary* is set to be depth 3 first, and then, it is found that along the separation boundary, only state 3 is pointed by a *trace* transition while state 6 and state 10 have *trace* transitions pointing out but not pointed by any *trace* transitions. In this case, these two states can be further moved to P_{suffix} to eliminate more *trace* transition. Then, the P_{prefix} becomes {*her*, *sh*, *hi*}, and the DFAs of the PFSM and the SFSM are constructed as the above algorithm descriptions. Compared with Fig. 1(b), we can see that there is no *trace* transition pointing from the SFSM to the PFSM in Fig. 3, and the *trace* transitions to the states in $depth1$ are not shown in the PFSM because they are also eliminated by the default table scheme [19]. Although the number of eliminated *trace* transitions in this example is not so large, it will be considerable for the practical signature set such as the one of Snort or CLAMAV [25]. On the contrary, several new *trace* transitions will be added because the DFA of the PFSM is a closed automaton. For example, the *trace* transition from state 5 to state 6 by character "*e*" is a new *trace* transition compared with Fig. 1(b). Therefore, when these two FSMs are constructed, we should adjust the *boundary* values to obtain an optimal result, which means to eliminate more *trace* transitions in the SFSM, and to add fewer *trace* transitions in the PFSM.

3.4.2 Coding of States

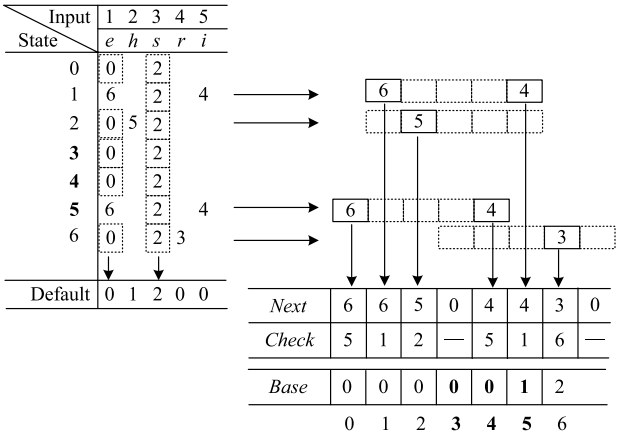
A lookup table is used to store the state transitions in an FSM and each state transition is assigned an entry. Since each FSM only processes its own states, the states of the other FSM can be neglected. However, as the lookup table is a continuous structure, the neglectable entries can not be deleted if they are located among useful entries. Therefore, the state identifiers in the two FSMs should be well coded so as to keep the two lookup tables as compact as possible.

The last state of each pattern is denoted to be a *matched* state, and other states are denoted to be *normal* states. In our solution, the *normal* states in the PFSM are coded continuously at first. Then the *matched* states in both FSMs are coded orderly. Finally, the *normal* states in the SFSM are coded. As a result, only a small number of empty entries are kept in the lookup table of the SFSM, and there is no empty entry in the lookup table of the PFSM. In addition, matched information is attached as [19].

3.4.3 Storage of the Lookup Tables

The lookup tables of the PFSM and the SFSM are shown in Figs. 4(a) and (c). They are constructed in terms of the state transitions in Fig. 3. The values in the default entry (table) in Fig. 4(a) are the states which appear mostly in the column of input characters [19]. Normally they are state 0 or belong to $depth1$. The default entry (table) is saved independently and the corresponding items in the lookup table can be treated as state 0. The useless outputs in Fig. 4(c) denote the eliminated state transitions in the SFSM, and they are set to be state 0.

The data structure, called three-array in this paper, was utilized to compress the sparse lookup table of one FSM in [19]. An improvement, which replaces the ADD operation by an XOR operation to shorten the critical path, was proposed in [20]. The compression of one lookup table is shown in Fig. 4(b). The input characters are recoded from 1 to 5. At first, the entries, which hold items unequal to state 0, are extracted. They are the entries of state 1, 2, 5 and 6. And then, they are interleaved and located to the *Next* and *Check* arrays without overlapping by adjusting the *Base* value. The initial value of *Base* is set to be 0 and can be simply adjusted by an increment 1 until no collision happens. For example, when the entry of state 6 is located, the *Base*[6] = 0 and the "*r*" = 4 are adopted by the formula "*location* = *Base* XOR *input*" to calculate the inserting position, and the *location* = 4 is obtained. However, the space has been occupied by an item of state 5, then the *Base*[6]



State \ Input	1 *e*	2 *h*	3 *s*	4 *r*	5 *i*
0	0		2		
1	6		2		4
2	0	5	2		
3	0		2		
4	0		2		
5	6		2		4
6	0		2	3	
Default	0	1	2	0	0

(a) Lookup table of PFSM.

	0	1	2	**3**	**4**	**5**	6
Next	6	6	5	0	4	4	3	0
Check	5	1	2	—	5	1	6	—
Base	0	0	0	**0**	**0**	**1**	2	

(b) XOR-based three-array data structure of (a).

State \ Input	1 *e*	2 *h*	3 *s*	4 *r*	5 *i*
0-2					
3		7			
4			10		
5	11				
6-9					
10					8
11					12
12			9		
Useless	0	0	0	0	0

(c) Lookup table of SFSM.

	0	1	2	**3**	**4**	**5**	6
Base	0	0	0	**0**	**0**	**1**	0
	7	8	9	10	11	12	13
Base	0	0	0	0	1	1	0
Next	11	9	7	10	12	8	0
Check	5	12	3	4	11	10	—

(d) XOR-based three-array data structure of (c).

Fig. 4 Lookup tables and corresponding data structures of two FSMs.

adds 1 and the *location* is recalculated to be 5 which is also occupied by the item of state 1. Then the *Base*[6] is increased to 2, and the *location* = 6 is achieved without any collision. Then *Next*[6] = 3, which denotes the destination of the state transition, and *Check*[6] = 6, which shows the origin of the state transition, are set.

Here, we extend the XOR-based three-array data structure for the two FSMs. As we know, these two FSMs are not completely independent, and they are associated by the *boundary* states {3, 4, 5}, which are the end states of the PFSM and the root states of the SFSM. This means that the information of the *boundary* states kept by both FSMs respectively should be consistent for the purpose of association. In Fig. 4, the *boundary* states {3, 4, 5} are emphasized in boldface. As we can see, the *Base* values of the *boundary* states of the two FSMs are identical. Additionally, in order to obtain a high compression ratio by three-array, we recode the characters in the signature set by genetic algorithm, and the method in [26], which locates the states in degressive order of their fan-outs, is adopted.

3.4.4 Matching Process

In the matching process of the TFSM scheme, the XOR-based three-array is searched in each FSM respectively. It is supposed that the current states of the PFSM and the SFSM

Input	—	h	e	r	h	a	h	i	s	i
Next state by PFSM	0 (initial value)	1	**6**	3	1	0	1	4	2	0
Next state by SFSM	0 (initial value)	0 (*null* state)	**6**	3	**7**	0 (*null* state)	0 (*null* state)	4	10	**8**
Result combination	0	1	**6**	3	**7**	0	1	4	10	**8**

→ Association by *boundary* state

Fig. 5 Matching process of TFSM.

are the *boundary* state 5, and the input character is "*e*," which is recoded to 1. In the PFSM, the "*location* = *Base*[5] XOR 1" is calculated and obtains 0. Since *Check*[0] equals to 5, this means a state transition from state 5 by character "*e*" exists, and the next state is *Next*[5] = 6. Or else, the next state equals to default["*e*"] which is 0. The same process is carried out in the SFSM, and the next state is 11. Besides the example in each FSM, an input sequence of characters "*herhahisi*" is supposed as *T* to simulate the matching process of both FSMs together. The result combination function is included. Figure 5 gives an intuitional illumination that the PFSM only processes the P_{prefix} matching while the SFSM processes the P_{Suffix} matching, and they are associated by the *boundary* states and the priority selector. The result combination function is also a priority selector. It aims to treat the two FSMs as one FSM, and obtain the next state of the total FSM for further analysis. The priority setting in it is almost the same as Table 1. Only one difference is that the state transitions to the depth = 1 will not be substituted by a *null* state. In Fig. 5, the boldface denotes the *matched* state.

3.5 Hardware Design of TFSM

The hardware circuit of (XOR-based) three-array data structure for one FSM has been introduced in [19], [20]. The matching process mentioned in Sect. 3.4.4 is finished in one clock cycle by setting the *Base* value of the next state in advance as shown in Fig. 6.

Here, we extend the circuit for two FSMs in our TFSM scheme. As shown in Fig. 7, both the PFSM and the SFSM are realized by the circuit in Fig. 6, and they are correlated by multiplexers as a priority selector. The next state of each FSM is generated in one clock cycle respectively, and the critical path is the longest path of these two parts. Because of the parallelism, our scheme does not lengthen the critical path of the single FSM circuit. Actually, the delay time of the critical path is shortened as the size of SRAM is decreased. The circuit of the result combination function is similar to the priority selector, and it is not included in the critical path.

An SRAM update function is integrated into this circuit in order to realize the fast update of the signature set without reconfiguration of the whole circuit. When one or more signatures are deleted from or inserted to a signature group,

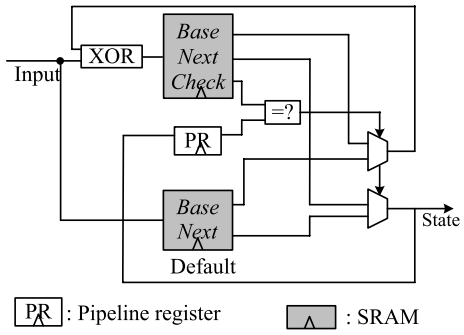


Fig. 6 Circuit of XOR-based three-array data structure based on [19], [20].

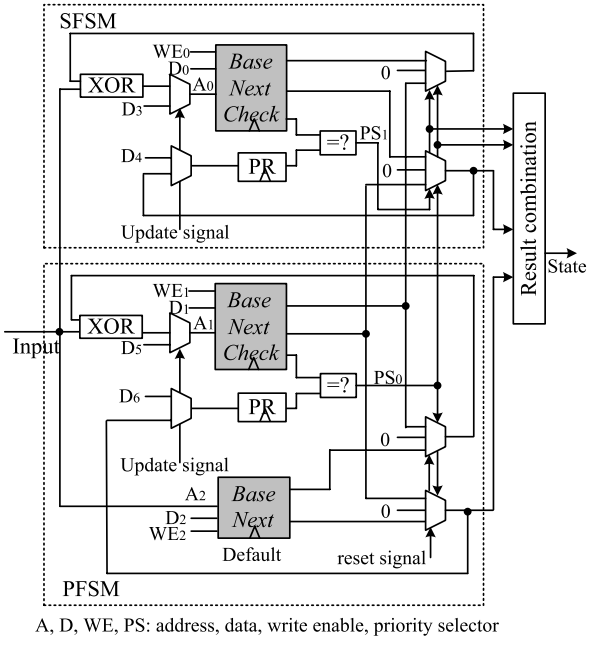


Fig. 7 Abstract circuit diagram of TFSM.

we can compile them and update a part of memory contents rather than the whole contents of memory at the speed of the signature matching. Additionally, this update function is also used in reset function to reset the circuit. No matter what content each SRAM block has, if we activate the update and reset signals, and assign the address lines to be "0" without activating any of the write enable (WE) signals, as well as set the pipeline registers (PRs) to be "0," both the two FSMs will generate *null* state.

In our evaluation, the circuit in Fig. 7 is realized by dual-port SRAM blocks with duplicated sequential logics to double the throughput, because two input payload streams can be processed concurrently.

4. Performance Evaluation

The prototype of our signature matching engine is evaluated by both Xilinx FPGA and ASIC technology. The design environment of FPGA is Xilinx ISE 9.1*i* equipped with both the synthesis tool XST, and the place&route tool. The XC4 VLX40 with speed grade −12 is selected as the target device. The design environment of the ASIC implementation includes a 180 nm-CMOS 6-layered-metal gate library of TSMC and the Cadence EDA tools. The whole rule set of Snort 2.3.3 is used in our evaluation for the purpose of comparison. A total of 2,168 unique patterns are extracted from the rule set. All of these patterns are classified into two groups depending on the case condition. There are 1,328 patterns with 23,804 characters in the case-insensitive group, and the average length is around 18. There are 840 patterns with 8,972 characters in the case-sensitive group, and the average length is around 10.

4.1 Adjustment of *Boundary*

A novel metric of memory efficiency (ME) is introduced first. ME is defined as the ratio between the number of utilized entries and the total memory size. On the one hand, the ME of ASIC chip is high because the memory can be fully customized in terms of the concrete memory usage. On the other hand, a large memory is implemented by cascading some 18k bit memory blocks, each of which is called "block SelectRAM" in the Xilinx FPGA chip. Because one block is the smallest memory unit, it will be occupied even if only one entry is used. In this case, the ME is lower.

Based on the patterns in both case-insensitive and case-sensitive groups, we calculated the number of state transitions in the two FSMs. Here, the *boundary* is neatly set from depth 1 to depth 4 without further adjustment. As can be seen from Fig. 8, with the increase of the depth, the number of state transitions descends drastically to a minimum value from depth 1, and then it begins to ascend gradually. This trend is shown in both groups. The difference is that the minimum value appears at a different depth.

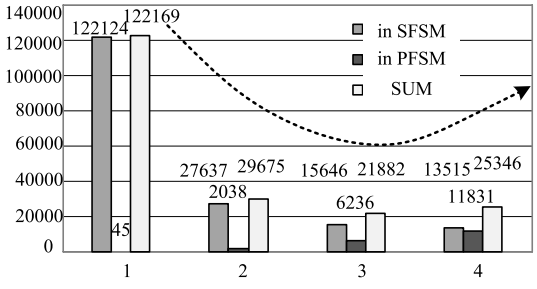
From the results of case sensitive group, we can see that the total numbers of state transitions are very close at depth 2 and 3. However, if we transform the state transitions into an XOR-based three-array data structure and calculate the ME of FPGA implementation, we can see that the ME at depth 2 and 3 are 85% and 75% respectively. In this case, the ME becomes a factor to select *boundary* = 2. That is to say, *boundary* in our scheme is adjusted to not only obtain a small number of state transitions but also a high ME in accordance with the concrete signature set.

4.2 Resource Requirement and Throughput

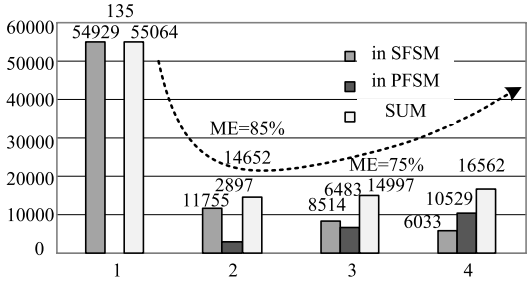
As shown in Fig. 7, an access to SRAM is included in the critical path, and it dominates the critical path delay. In order to shorten the delay time, one solution is to decrease the size of SRAM, since smaller memory has higher access speed. Therefore, we divide the large signature set into several groups as mentioned in Sect. 2 and each group will be processed by a smaller signature matching engine. The fundamental of our division is to consider the trade off between the size of the largest SRAM in the critical path and the

number of groups, as well as the load-balance among the signature matching engines.

Table 2 shows the result by FPGA evaluation. The result is based on the post-place&route timing analyzer. Excluding the first case which has no division, three cases are evaluated. In each case, the signature set is divided into a number of groups, and the memory size in the critical path of each group is guaranteed to be equal by adjusting the boundary depth. We can see that the memory size in the critical path becomes smaller when the signature set is divided into more groups. As a result, the delay of access time will descend. However, the clock cycle does not decline continuously in case 4 due to the increased routing delay. Moreover, many inefficiently-utilized memory blocks in the FPGA implementation increase the memory requirement and decrease the ME in case 4. However, this case does not happen in the ASIC implementation. Another resource requirement of FPGA implementation is the number of logic cells. It is normally approximately calculated by doubling the number of utilized slices in the report of Xilinx ISE tool because two logic cells form one slice. We can see that the value is proportional to the number of groups.



(a) State transitions based on case-insensitive group.



(b) State transitions based on case-sensitive group.

Fig. 8 The number of state transitions with the adjustment of *boundary*.

Because of the better performance-cost ratio, case 3 is selected to be evaluated by ASIC implementation. We achieve a 5.0 Gbps throughput with 11 KB of SRAM for one group without any time-oriented optimization. As for the total groups, if the place&route is processed under the support of the comprehensive time-oriented optimization, because each group works completely independently, over 4.5 Gbps throughput is estimated even considering an overhead of wire delay. The maximum memory requirement is 132 KB SRAM.

4.3 Scalability of Memory Requirement

The concrete memory requirement of the TFSM scheme can not be formulized due to the randomicity of patterns. Here we analyze its scalability of memory requirement with the increase of signature set. The memory is used to store the state transitions in the TFSM scheme. Based on the signature set grouping method and our efforts mentioned in Sect. 3.4.3, there are few empty entries left in the memory by the storage method, which is the XOR-based three-array data structure. Therefore, we can study the scalability of the number of state transitions in the TFSM scheme with the increase of signature set instead.

It is supposed that a random signature set of m patterns with a total of r characters are given. The average length of the pattern is r/m. The numbers of *goto* transitions and *trace* transitions are analyzed respectively. A worst-case assumption is given that there is no common prefix among the m patterns, and then a trie structure with r states is constructed. Therefore, the number of *goto* transitions grows in terms of $O(r)$. The *trace* transitions pointing to state 0 are excluded at first, because they are eliminated by the three-array data structure. A *trace* transition, which does not point to state 0, shows that a prefix appears somewhere in (but not at the beginning of) a certain pattern. It is related with one and only one prefix. Because of this exclusivity, we group the prefixes by length 1, 2, 3, ... r/m, and calculate the possibility of *trace* transitions. For the prefixes with length l, the possibility to cause some *trace* transitions proportionally increases with the scale of signature set r and inversely decreases with the length l. When all prefixes are considered, the possibility of *trace* transitions grows in terms of

$$\begin{aligned} &O[r + r/2 + r/3 + ... + r/(r/m)] \\ &\quad = O[(1 + 1/2 + 1/3 + ... + m/r)r] \\ &\quad \approx O\{[\ln(r/m) + c]r\} \end{aligned} \tag{1}$$

Table 2 Resource requirement and throughput by different divisions (FPGA).

Size of memory in critical path (KB)		Adjustment of *boundary*	Groups case (insensi. + sensi.)	Total memory (KB)	Logic cells	Clock cycle (ns)	Throughput (Gbps)
Case1	29.25	2, 3	2(1 + 1)	222.75	—†	—†	—†
Case2	6.75	1–3	7(4 + 3)	182.25	2242	6.191	2.6
Case3	4.5	1–5	12(7 + 5)	177.75	3294	5.321	3.0
Case4	2.25	1–8	21(12 + 9)	202.5	5346	5.848	2.7

†This case is not supported by XC4 VLX40 which has 96 SelectRAM blocks totally.

Table 3 Memory-time efficiency comparison.

O-AC algorithms	Implementation technology	Char.(B)	Char./Group	Memory (KB)	Throughput (Gbps)	PE (Gbps*char./KB)
Classic AC [21]	ASIC (130 nm)	19124	—	53100	5.9	2.1
Bitmapped-AC [21]	ASIC (130 nm)	19124	—	2800	7.8	53.3
Pathcomp.-AC [21]	ASIC (130 nm)	19124	—	1100	7.8	135.6
Split-AC [22]/[27]	ASIC (130 nm)/ FPGA (Xilinx XC4 VFX100)	(12–19 K)†/ 16715	< 304†/ 356	400/ 769	10/ 1.6	(300–475)†/ 37.8
BFPM [23]	ASIC/ FPGA (Xilinx Virtex-IV)	31.6 K	988	128–191	4/ 2	987.5/ 493.8
TFSM	ASIC (180 nm)/ FPGA (Xilinx XC4 VLX40)	32776††	2731	132/ 178	4.5/ 3.0	1117.4/ 553.2

† These numbers were deduced from the paper but not cited. †† More patterns can be expressed, because there is one group partially occupied.

The c in Eq. (1) approximately equals to 0.577216, and it is called Euler's constant. In our TFSM scheme, most of the *trace* transitions to depth ≤3 or higher depth have been eliminated; therefore, the high-weigh items, such as 1, 1/2, 1/3 or more, can be almost substracted from the coefficient ratio of Eq. (1) as

$$O\{[\ln(r/m) + 0.577216 - 1 - 1/2 - 1/3 - ...]r\} \qquad (2)$$

For the practical signature sets normally with average pattern length 20, we calculate the value of the coefficient of r in Eq. (2) to be around 2. These results show the high scalability with the increase of signature set and the resource-saving feature of our TFSM scheme. Acctually, the practical performance will be better, because the above analysis is based on the worst case.

4.4 Memory-Time Efficiency Comparison

Recently, many algorithms and solutions are proposed in signature matching domain. Each of them has inherent features and particular application environments. Based on our motivations in Sect. 2, we provide a memory-time efficiency comparison of the TFSM scheme with the hardware-implemented AC algorithm and its derived O-AC algorithms [21]–[23]. In [21], Tuck et al. compressed the memory requirement of AC-FAIL by using both bitmap and path compression methods which are widely employed in IP lookup domain. In [22], Tan et al. split the whole DFA of AC-OPT into several DFAs and each one was in charge of certain bits of an input character. Since the fan-out was reduced from 256 to 2 or 4, the memory requirement was decreased. In [23], a kind of hashing-based transition-rule selection scheme was used to compress the memory requirement of AC-OPT, and it was named balanced-routing-table-based FSM pattern matching (BFPM).

Besides the number of characters, characters per group, memory requirement and throughput, another popular metric of performance efficiency (PE), which was defined as Throughput*char/Memory, is adopted to give a general comparison. As shown in Table 3, our TFSM scheme has a remarkable PE compared with the AC, the bitmapped-AC, the path-compressed-AC, and the split-AC algorithms. Supposing that the same resources are provided, through implementing multiple instances of signature matching engine, the performance of our scheme by FPGA implementation has already surpassed the above algorithms even though they are implemented on ASIC chips. Compared with the BFPM scheme, the PEs of the our scheme are 13.2% higher in ASIC evaluation and 12% higher in FPGA evaluation. In addition, the BFPM scheme obtained its best performance by dividing the signature set of Snort into 32 (or more) groups, which is 20 groups more than our scheme. Actually more groups require more workload on matching result analysis (Sect. 2).

5. Conclusion and Future Work

In this paper, we fully customize a novel multi-pattern signature matching scheme called TFSM for mobile security. Our scheme not only shows a high performance-cost ratio but also provides other effective features, such as stable performance, fast updates, hardware implementation, and scalability. The evaluation results by both FPGA and ASIC technology show that our scheme is more memory-time efficient than other existing O-AC algorithms and suitable for mobile security.

Our ongoing work focuses on improve the TFSM scheme by introducing the skills such as signature set updates through minimum operations on the SRAM blocks, optimal *boundary* setting, as well as a more compact storage style. In the future, we aim to set up a comprehensive mobile NIPS which adopts our TFSM scheme as its core component. Other countermeasures against NIPS-evasion tricks such as segment reassembly, protocol analysis will be considered systematically.

Acknowledgements

This work is supported by VLSI Design and Education Center (VDEC), the University of Tokyo in collaboration with Cadence Design Systems, Inc.

References

[1] M. Weiser, "Some computer science problems in ubiquitous computing," Commun. ACM, vol.36, no.7, pp.75–84, July 1993.
[2] S. Antonatos, K.G. Anagnostakis, and E.P. Markatos, "Generating realistic workloads for network intrusion detection systems," Proc. 4th Int. Workshop on Software and Performance, Jan. 2004.

[3] M. Roesch, "Snort: Lightweight intrusion detection for networks," Proc. 13rd Administration Conf., Seattle, WA, USA, Nov. 1999.
[4] Xilinx, Vertex4 datasheet and user guide. (on-line). Available: www.Xilinx.com
[5] M. Fisk and G. Varghese, "Fast content-based packet handling for intrusion detection," UCSD Technical Report CS2001-0670, May 2001.
[6] S. Wu and U. Manber, "A fast algorithm for multi-pattern searching," Technical Report TR94-17, Department of Computer Science, University of Arizona, 1994.
[7] M. Norton, "Optimizing pattern matching for intrusion detection," White paper, (on-line). Available: http://docs.Idsresearc.org/optimizingPatternMatchingForIDS.pdf, July 2004.
[8] R.T. Liu, N.F. Huang, C.H. Chen, and C.N. Kao, "A fast string-matching algorithm for network processor-based network intrusion detection system," ACM Trans. Embedded Computer System, vol.3, pp.614–633, Aug. 2004.
[9] R. Sidhu and V.K. Prasanna, "Fast regular expression matching using FPGAs," Proc. Symp. on Field-Programmable Custom Computing Machines, pp.227–238, Rohnert Partk, CA, USA, 2001.
[10] R. Franklin, D. Carver, and B. Hutchings, "Assisting network intrusion detection with reconfigurable hardware," Proc. IEEE Symp. on Field-Programmable Custom Computing Machines, pp.111–120, Napa, CA, USA, 2002.
[11] C.R. Clark and D.E. Schimmel, "Scalable pattern matching for high speed networks," Proc. IEEE Symp. on Field-Programmable Custom Computing Machines, pp.249–257, Napa, CA, USA, 2004.
[12] I. Sourdis and D. Pnevmatikatos, "Pre-decoded CAMs for efficient and high-speed NIDS pattern matching," Proc. IEEE Symp. on Field-Programmable Custom Computing Machines, pp.258–267, Napa, CA, USA, April 2004.
[13] I. Sourdis, D. Pnevmatikatos, S. Wong, and S. Vassiliadis, "A reconfigurable perfect-hashing scheme for packet inspection," Proc. 15th Int. Conf. on Field Programmable Logic and its Applications, pp.644– 647, Tampere, Finland, Aug. 2005.
[14] F. Yu, R.H. Katz, and T.V. Lakshmana, "Gigabit rate packet pattern-matching using TCAM," Proc. 12th IEEE Int. Conference on Network Protocols, pp.174–183, Berlin, Germany, Oct. 2004.
[15] M. Alicherry, M. Muthuprasanna, and V. Kumar, "High speed pattern matching for network IDS/IPS," Proc. IEEE 14th Int. Conf. on Network Protocols, pp.187–196, USA, 2006.
[16] A.V. Aho and M.J. Corasick, "Efficient string matching: An aid to bibliographic search," Commun. ACM, vol.18, no.6, pp.333–343, June 1975.
[17] S. Dharmapurikar, P. Krishnamurthy, T.S. Sproull, and J.W. Lockwood, "Deep packet inspection using parallel bloom filters," IEEE Micro, vol.24, no.1, pp.52–61, 2004.
[18] Y.H. Cho and W.H. Mangione-Smith, "A pattern matching coprocessor for network security," Proc. 42nd Annual Conf. on Design Automation, pp.234–239, Anaheim, CA, USA, 2005.
[19] Y. Sugawara, M. Inaba, and K. Hiraki, "Over 10 Gbps string matching mechanism for multi-stream packet scanning systems," Proc. 14th Int. Conf. on Field-Programmable Logic Application, pp.484–493, Antwerp, Belgium, Aug. 2004.
[20] T. Gerald, "A parallel string matching engine for use in high speed network intrusion detection systems," J. Comput. Virol., vol.2, no.1, pp.21–34, 2006.
[21] N. Tuck, T. Sherwood, B. Calder, and G. Varghese, "Deterministic memory-efficient string matching algorithms for intrusion detection," Proc. IEEE INFOCOM, pp.333–340, Hong Kong, China, March 2004.
[22] L. Tan and T. Sherwood, "A high throughput string matching architecture for intrusion detection and prevention," Proc. 32nd Int. Symp. on Computer Architecture, pp.112–122, Madison, WI, USA, 2005.
[23] J.V. Lunteran, "High-performance pattern-matching for intrusion detection," Proc. 25th IEEE Int. Conf. on Computer Communication, pp.1–13, Barcelona, Spain, April 2006.
[24] B.W. Watson and G. Zwan, "A taxonomy of keyword pattern matching algorithm," Computing Science Report 92/27, Eindhoven Univ. of Tech., The Netherlands, Dec. 1992.
[25] Clam AntiVirus Toolkit for UNIX. (on-line). Available: www. clamav.net
[26] R.E. Tarjan and A.C. Yao, "Storing a sparse table," Commun. ACM, vol.22, no.11, pp.606–611, 1979.
[27] H.J. Jung, Z.K. Baker, and V.K. Prasanna, "Performance of FPGA implementation of bit-split architecture for intrusion detection systems," Proc. 20th Int. Conf. on Parallel and Distributed Processing Symposium, April 2006.

Appendix: Algorithm Description of TFSM

Algorithm 1 Additional algorithm of DFA construction of SFSM.

Input: DFA_PFSM: the DFA of the PFSM based on P_{prefix}, ini_DFA_SFSM: the initial DFA of the SFSM based on P.
Output: DFA_SFSM: the final DFA of the SFSM.
{ a: $a \in \Sigma$, s: state number. S: state transition in the DFA based on P.}

```
DFA_SFSM ← ini_DFA_SFSM.
for each s ∈ DFA_SFSM do
    if s ∈ DFA_PFSM then
        DFA_SFSM ← DFA_SFSM - {s}.
    else
        for each a ∈ Σ do
            if S(s, a) ∈ DFA_PFSM then
                S(s, a) ← useless_state.
            end if
        end for
    end if
end for
```

Algorithm 2 Matching algorithm of PFSM.

Input: t_i: the *i-th* character of T, Ls_{i-1}: previous state.
Output: PS_0: the LSB of priority selector, Ls_i: next state.
{ S_PFSM: state transition in the PFSM. LMs: state number. Reset function is omitted.}

```
LMs ← S_PFSM(Ls_{i-1}, t_i).
if S_PFSM succeed then
    PS_0 ← 1.
    Ls_i ← LMs.
else
    PS_0 ← 0.
    Ls_i ← default_table(t_i).
end if
```

Algorithm 3 Matching algorithm of SFSM.

Input: t_i: the *i-th* character of the T, Hs_{i-1}: previous state. PS_0: the *LSB* of priority selector.
Output: Hs_i: next state.
{ PS_1: the MSB of priority selector, S_SFSM: state transition in the SFSM. *HMs*: state number.}

```
1: HMs ← S_SFSM(Hs_{i-1}, t_i).
2: if S_PFSM succeed then
3:     PS_1 ← 1.
4: else
5:     PS_1 ← 0.
6: end if
7: if PS[1:0]="1*" then
8:     Hs_i ← HMs.
9: else
10:    if PS[1:0]="01" then
11:        Hs_i ← Ls_i.
12:    else
13:        Hs_i ← null_state.
14:    end if
15: end if
```

Ruhui Zhang was born in 1978. She received the B.S and M.S. degrees in Computer Science from Yanshan University, Hebei, China, in 2001 and 2004, respectively. She has been in the Ph.D. program in Information Systems Engineering Course of Kochi University of Technology, Kochi, Japan. Her research interests are network processor, embedded network security system design and evaluation.

Makoto Iwata received the B.E. and M.E. degrees in Electronic Engineering from Osaka University, Osaka, Japan, in 1986 and 1988, respectively. He received the Dr. Eng. degree in Information Systems Engineering from Osaka University in 1997. Now he is a professor of Department of Information Systems Engineering, Kochi University of Technology, and a visiting professor of Research Institute of Electrical Communication, Tohoku University. His research interests are the software, architectures, and applications of novel massively parallel processing systems.