A High-Speed Programmable Network Intrusion Detection System Based on a Multi-Byte Transition NFA

Tomoaki Hashimoto, Shin'ichi Wakabayashi*, Shinobu Nagayama†, Masato Inagi, Ryohei Koishi, Hiroki Takaguchi

Graduate School of Information Sciences, Hiroshima City University,
Hiroshima 731-3194, Japan
email: {*wakaba, †s_naga}@hiroshima-cu.ac.jp

***Abstract*—To improve the network security, when a virus pattern is updated, an arbitrary updated pattern should be quickly set in a network intrusion detection system (NIDS). This type of NIDS is called "programmable." However, present programmable NIDSs could hardly be applied to a high-speed network with more than 10 Gbps of network transmission speed due to the limitation of clock frequency of the circuit. To overcome this speed limitation, this paper proposes a programmable NIDS based on a multi-byte transition non-deterministic finite automaton (NFA). The proposed NIDS is implemented on an FPGA to evaluate its performance. The FPGA implementation results show that the proposed NIDS can achieve more than 10 Gbps of throughput.**

Keywords-Regular expression matching; non-deterministic finite automaton; programmable hardware; network intrusion detection system; FPGA.

I. INTRODUCTION

Network intrusion detection systems (NIDSs) that can detect network attacks such as computer viruses and worms in real time have become indispensable nowadays to maintain network security. To detect suspicious data patterns included in packets, NIDSs perform *regular expression matching* [14] between packet payloads and data patterns predefined as regular expressions. Since regular expression matching is a time-consuming task, NIDSs tend to be bottleneck in network transmission. In addition, since regular expression patterns are frequently updated, NIDSs tend to be security holes until pattern updating is completed. Thus, 1) fast regular expression matching and 2) quick pattern updating are major requirements for NIDSs.

Software NIDSs that perform regular expression matching by a software program are popular. The Snort system [19] is well-known as an open source software NIDS, and its regular expression patterns are available at the website. Software NIDSs can update regular expression patterns quickly, but speed of regular expression matching is slow. Although various algorithms [1][3][6][7][12][14][18][21][25] have been proposed to perform regular expression matching faster, their software implementation is difficult to achieve enough performance to catch up with the speed of latest Gigabit Ethernet with more than 10Gbps.

On the other hand, hardware NIDSs that perform regular expression matching with FPGAs [4][8][10][13][15][16][17][23][24] can perform regular expression matching much faster, but they require long time for pattern updating because regular expression patterns are embedded as hardware circuits. To update patterns in such *pattern-specific* circuits, a sequence of FPGA design and implementation processes (i.e., generating HDL code, logic synthesis, place and route, etc.) should be performed again. It is well known that those FPGA design processes require a fairly long time, sometimes, a few hours. Since this time is longer than update interval [2], it is hard to keep NIDSs up to date.

To overcome this problem due to architecture of NIDSs, *programmable NIDSs* [5][11][20][22] that perform regular expression matching with *pattern-independent* circuits have been proposed as the third approach. The programmable NIDSs can update patterns more quickly because regular expression patterns are stored in registers. In addition, the programmable NIDSs can perform regular expression matching much faster than software NIDSs. However, since size of programmable NIDSs is larger than that of pattern-specific NIDSs, we use a programmable NIDS with pattern-specific ones, as hybrid NIDSs. In hybrid NIDSs, a pattern-specific NIDS performs regular expression matching for all patterns currently registered in the NIDS, while the programmable NIDS performs regular expression matching for only new patterns. In this way, we can compensate the defect of programmable NIDS in terms of hardware size. However, since throughput of the existing programmable NIDSs is at most a few Gbps, faster programmable NIDSs are still required to achieve more than 10 Gbps of throughput.

The existing programmable NIDSs process one byte of packet payload per one clock, and their throughput has been improved mainly by increasing clock frequency of circuits. But, there is a limitation to increase of clock frequency. Thus, this paper proposes a method to improve throughput of a programmable NIDS by processing k bytes per one clock. We design such a high-speed programmable NIDS using a k-byte transition non-deterministic finite automaton (NFA) that is converted from a one-byte transition NFA.

Although a similar method has been proposed by Yamagaki et al. [23], their method is targeted to their hardware NIDS, and thus, the capability of quick pattern updating

 ISBN: 978-1-61208-496-1

is not considered. Therefore, we propose an architecture of a programmable NIDS considering quick pattern updating. As far as we know, a programmable NIDS based on k-byte transition NFA has not been proposed so far.

The rest of this paper is organized as follows: Section II briefly defines regular expressions and NFAs. Section III introduces k-byte transition NFAs. Section IV presents architecture of a programmable NIDS based on k-byte transition NFAs. Its FPGA implementation results are shown in Section V. Section VI concludes the paper.

II. Preliminaries

A. Regular Expressions

In NIDSs, suspicious data patterns to be found are described by not only simple strings, but also **regular expressions** [9]. This is because its acceptance algorithm is simple, and it has practically enough expression power [13]. A set of strings represented by regular expression is called a **regular set** or regular language. Regular expressions and their regular sets are recursively defined as follows:

Definition 1: Let Σ be a finite set of characters: $\Sigma = \{a_1, a_2, \ldots, a_n\}$, called the **alphabet**, R and S be regular expressions on Σ, and $L(R)$ and $L(S)$ be regular sets denoted by R and S, respectively. Then,

1) $\emptyset$ is a regular expression denoting the regular set $\emptyset$.
2) ε is a regular expression denoting the regular set $\{\varepsilon\}$ (the null character).
3) A character $a_i \in \Sigma$ is a regular expression denoting the regular set $\{a_i\}$.
4) An alternation $R \mid S$ is a regular expression denoting the regular set $L(R) \cup L(S)$.
5) A concatenation $R \cdot S$ is a regular expression denoting the regular set $\{rs | r \in L(R), s \in L(S)\}$. $R^1 = R$ and $R^m = R \cdot R^{m-1}$ for $m \geq 2$. Usually, '$\cdot$' is omitted.
6) A Kleene closure $R*$ is a regular expression $\varepsilon \mid R \mid R^2 \mid \ldots$ denoting the regular set $\{\varepsilon\} \cup L(R) \cup L(R^2) \cup \ldots$, where $L(R), L(R^2), \ldots$ are regular sets denoted by $R, R^2, \ldots$, respectively.

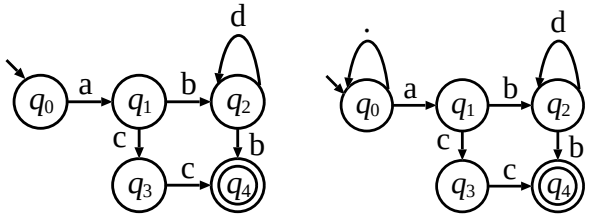
Only expressions obtained by considering the above 1) to 3) as constants, and applying the above operations 4) to 6) finite times are regular expressions on Σ.

For simplicity of expressions, this paper introduces the following operation as well:

7) A dot . denotes the set $L(.) = \Sigma \cup \{\varepsilon\}$ (don't care). □

B. Non-Deterministic Finite Automaton

Definition 2: A **non-deterministic finite automaton (NFA)** [9] is defined by a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ is the alphabet, δ is a state transition function $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to 2^Q$, 2^Q is the power set of Q, $q_0 \in Q$ is an initial state, and $F \subseteq Q$ is a set of accepting states. When $\delta : Q \times \Sigma \to 2^Q$, it is called an **ε-free NFA**. In the following, an NFA means an ε-free NFA, unless otherwise stated. □



(a) NFA for prefix matching. (b) NFA for infix matching.

Figure 1. NFAs for regular expression $a(cc \mid bd*b)$.

An arbitrary regular expression R can be converted into an NFA that accepts its regular language $L(R)$. Usually, NFAs are represented as directed graphs. Since directed graphs can be represented by adjacency matrices, NFAs can also be represented by adjacency matrices as follows:

Example 1: Figure 1(a) shows an NFA for the regular expression $a(cc \mid bd*b)$. In this NFA, q_0 is the initial state, and q_4 is the accepting state. By considering this NFA as a directed graph, the NFA can be represented by the following adjacency matrix M:

$$M = \begin{pmatrix} \emptyset & a & \emptyset & \emptyset & \emptyset \\ \emptyset & \emptyset & b & c & \emptyset \\ \emptyset & \emptyset & d & \emptyset & b \\ \emptyset & \emptyset & \emptyset & \emptyset & c \\ \emptyset & \emptyset & \emptyset & \emptyset & \emptyset \end{pmatrix}$$

In this matrix, an element in i-th row and j-th column of M denotes a set of transition characters from q_i to q_j. When no state transition is defined, $\emptyset$ is entered as the corresponding element. □

C. Regular Expression Matching Based on NFAs

Regular expression matching performed in NIDSs is often formulated as follows:

Problem 1: Given a regular expression R and a text T, detect if T has a string in $L(R)$ as a substring. □

Regular expression matching can be performed by iterating state transitions on an NFA for R according to characters in T. If a substring in T can cause state transitions from the initial state q_0 to an accepting state in F, then it is in $L(R)$ (i.e., a matching is achieved). Otherwise, the substring is not in $L(R)$ (i.e., matching failed).

Although the NFA in Figure 1(a) is equivalent to the regular expression $a(cc \mid bd*b)$, the NFA in Figure 1(b) is used in NIDSs to solve Problem 1. This is because the NFA can discard unmatched prefix characters by the state transition by the don't care character at q_0. In the following, we focus on this type of NFAs.

Regular expression matching hardware based on NFAs realizes the above behavior on a circuit. Since in an NFA, multiple states can be active, state transitions from active states caused by a character are computed in parallel in hardware. Existing programmable NIDSs based on NFAs [5][11][20][22] compute all state transitions caused by a character in a clock.

 ISBN: 978-1-61208-496-1

III. Multi-Byte Transition NFAs

The alphabet Σ, on which regular expressions and NFAs are defined, usually consists of one-byte (8-bit) characters. Since existing programmable NIDSs based on NFAs compute all state transitions for a character in a clock, network transmission speed s that the NIDSs can be applied is $s = 8f$ bits per second, where f is clock frequency of the NIDSs. Thus, network transmission speed s has been increased by increasing clock frequency f of circuits so far. However, there is a limitation to increase of clock frequency, and thus, it is difficult to achieve 10 Gbps of network transmission speed in this way.

To overcome the problem, we introduce multi-byte transition NFAs [23] to design of programmable NIDSs.

A. Definition of Multi-Byte Transition NFAs

Whereas ordinary NFAs require a *one-byte character* for state transitions, multi-byte transition NFAs require a *multi-byte character*. To obtain multi-byte transition NFAs that accept the same regular sets as regular expressions R defined on one-byte characters, we generate k-byte transition NFAs, in which state transitions are caused by k *one-byte characters*, from one-byte transition NFAs for R. Thus, this paper defines multi-byte transition NFAs as follows:

Definition 3: Let a one-byte transition NFA be $(Q, \Sigma, \delta, q_0, F)$. Then, a **multi-byte ($k$-byte) transition NFA** equivalent to it is defined by $(Q_k, \Sigma^{k'}, \delta_k, q_0, F_k)$. $F_k = F \cup F'$ where F' is a set of additional accepting states. $Q_k = F' \cup Q'$ where $Q' \subseteq Q$. $\Sigma^{k'} = \Sigma^k \cup \bigcup_{i=1}^{k-1} \Sigma^i \times \{\varepsilon\}^{k-i}$, where products of sets are Cartesian products. And, $\delta_k : Q_k \times \Sigma^{k'} \to 2^{Q_k}$. □

Since k-byte transition NFAs take in k one-byte characters at a time, null characters ε can be input at the end of a text when length of the text is not a multiple of k. Thus, in this definition, ε is added to $\Sigma^{k'}$. And, for the same reason, F' is added. The next subsection shows how to produce such k-byte transition NFAs.

B. Conversion into Multi-Byte Transition NFA

We can generate k-byte transition NFAs of Definition 3 by converting k state transitions (a path of length k) on one-byte transition NFAs into a state transition (a path of length 1). Such path conversions can be achieved by considering NFAs as directed graphs, and raising adjacency matrices of NFAs to the k-th power [23]. In the following, we introduce the conversion method briefly. For more details, see [23].

Given an adjacency matrix M of a one-byte transition NFA, at first, add a symbol ω to diagonal elements of M that are associated with accepting states. The symbol ω corresponds to ε in Definition 3. Let $M^{(1)}$ be the obtained matrix. Then, obtain $M^{(k)}$ by raising $M^{(1)}$ to the k-th power using the matrix multiplication defined in the following:

Definition 4: Since all elements of $M^{(1)}$ can be considered as regular expressions, multiplications and additions on ordinary scalar matrix multiplication are computed as concatenation $\cdot$ and alternation $|$, respectively, except for the following cases:

- Multiplication with $\emptyset$ results in $\emptyset$.
- $. \times E = .E$ and $E \times . = \emptyset$, where E is a regular expression for an element.
- $\omega \times E = \emptyset$ and $E \times \omega = E\omega$. □

After the matrix multiplications, apply the following operations to $M^{(k)}$ to obtain an adjacency matrix $M_a^{(k)}$ of a k-byte transition NFA.

1) Add $k-1$ columns for new accepting states $q_{f(1)}, q_{f(2)}, \ldots, q_{f(k-1)}$, and redefine strings including i ω's as state transitions to the new accepting state $q_{f(i)}$. Note that no row is added because there is no state transition from the new accepting states.
2) Replace ω with the don't care character. Although ω can be replaced with ε, the don't care character is preferred because it makes hardware implementation simpler.

And, finally, eliminate states without incoming state transition, except for the initial state, to obtain an irredundant k-byte transition NFA.

Example 2: $M_a^{(2)}$ for the NFA in Figure 1(b) is as follows:

$$
\begin{aligned}
M^{(2)} &= M^{(1)} \times M^{(1)} \\
&= \begin{pmatrix} . & a & \emptyset & \emptyset & \emptyset \\ \emptyset & \emptyset & b & c & \emptyset \\ \emptyset & \emptyset & d & \emptyset & b \\ \emptyset & \emptyset & \emptyset & \emptyset & c \\ \emptyset & \emptyset & \emptyset & \emptyset & \omega \end{pmatrix} \begin{pmatrix} . & a & \emptyset & \emptyset & \emptyset \\ \emptyset & \emptyset & b & c & \emptyset \\ \emptyset & \emptyset & d & \emptyset & b \\ \emptyset & \emptyset & \emptyset & \emptyset & c \\ \emptyset & \emptyset & \emptyset & \emptyset & \omega \end{pmatrix} \\
&= \begin{pmatrix} .. & .a & ab & ac & \emptyset \\ \emptyset & \emptyset & bd & \emptyset & bb \mid cc \\ \emptyset & \emptyset & dd & \emptyset & db \mid b\omega \\ \emptyset & \emptyset & \emptyset & \emptyset & c\omega \\ \emptyset & \emptyset & \emptyset & \emptyset & \emptyset \end{pmatrix} \\
M_a^{(2)} &= \begin{pmatrix} .. & .a & ab & ac & \emptyset & \emptyset \\ \emptyset & \emptyset & bd & \emptyset & bb \mid cc & \emptyset \\ \emptyset & \emptyset & dd & \emptyset & db & b. \\ \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & c. \\ \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset \end{pmatrix}
\end{aligned}
$$

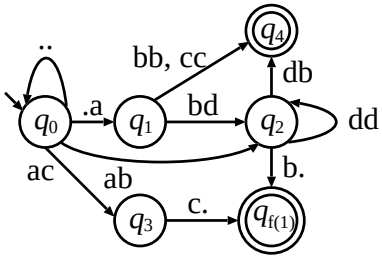


Figure 2. 2-byte transition NFA equivalent to the NFA in Figure 1(b).

Figure 2 shows a 2-byte transition NFA for $M_a^{(2)}$ □

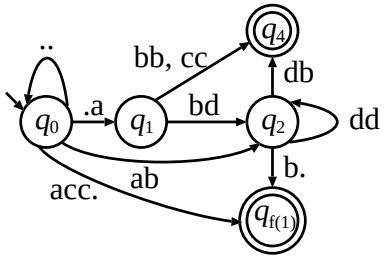


Figure 3. 2-byte STNFA equivalent to the NFA in Figure 1(b).

C. Multi-Byte String Transition NFAs

A **string transition NFA (STNFA)** [20][22] is an extension of an NFA, and it causes state transitions by a string, not a character. By eliminating states connected linearly in an ordinary NFA, and connecting between its beginning state and ending state directly as a *string transition*, an STNFA is obtained. The STNFA obtained by this simple way is equivalent to the original NFA, and has fewer states resulting in a smaller circuit. Since many concatenations are used in regular expression patterns for NIDSs [19], linearly connected states often appear. Thus, this simple conversion method is effective to reduce circuit size [20][22].

We apply the same method to multi-byte transition NFAs to generate **multi-byte STNFAs** with fewer states. In Figure 2, the states q_0, q_3, and $q_{f(1)}$ are linearly connected. By eliminating q_3 and connecting q_0 and $q_{f(1)}$ directly, we obtain a 2-byte STNFA in Figure 3.

IV. Programmable NIDS Based on Multi-Byte String Transition NFAs

This section presents architecture of our programmable NIDS based on multi-byte STNFAs. To realize both fast regular expression matching and quick pattern updating with a compact circuit, the proposed NIDS is designed taking advantages of multi-byte STNFAs shown in Section III.

A. Overall Architecture of Our Programmable NIDS

Since overall architecture of our NIDS follows architecture of the NIDS shown in [20], this subsection shows only an overview of its architecture. For more details, see [20].

Figure 4 shows overall architecture of the proposed programmable NIDS. It has a two-dimensional array structure, consisting of two parts: a matching array (MA) and a feedback array (FA). The MA performs string matching needed to trigger state transitions on an STNFA. The FA is a programmable interconnection network to activate next states according to state transitions triggered by the MA.

B. Matching Array Based on k-Byte String Transition NFAs

The MA is constructed by arranging string matching units (SMUs), shown in Figure 5, in a row. Eash state transition in an STNFA is assigned to an SMU. An SMU is constructed as a one-dimensional array of simple processing

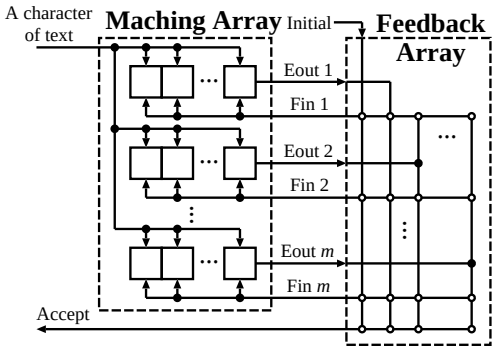


Figure 4. Overall architecture of programmable NIDS [20].

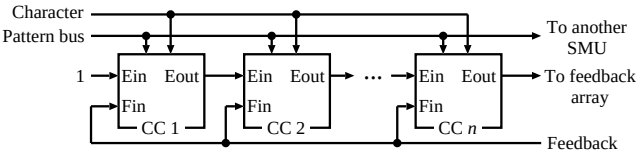


Figure 5. Architecture of string matching unit (SMU) [20].

units, called comparison cells (CCs), shown in Figure 6. Each CC performs one-character matching for a k-byte character fed synchronously with the clock, and transmits its matching result to the right neighbor CC via E_{out}.

By performing one-character matching sequentially using the pipelined CCs, an SMU performs string matching. When an input string matches with a string pattern stored in an SMU, the SMU outputs an enable signal to the FA to trigger state transitions. Then, by transmitting triggered state transitions (the enable signal) to F_{in} of appropriate SMUs via the FA, next states are activated, since CCs perform one-character matching only when an enable signal is fed via E_{in} or F_{in}. Using the pattern bus, we can program which CC receives an enable signal from, E_{in} or F_{in}, to the register connected to the selector.

CCs perform one-character matching using character matching tables, shown in Figure 7, that are new components proposed in this paper. An input k-byte character is divided into each byte, and is fed to addresses of k RAMs in parallel. Word width of a RAM is 1 bit, and it stores 0 or 1. If a byte x given as an address matches with the i-th byte of a k-byte

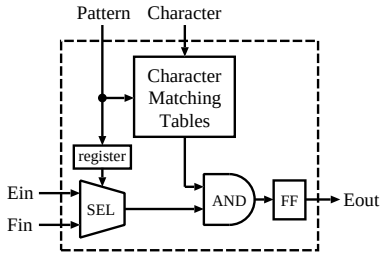


Figure 6. Architecture of comparison cell (CC).

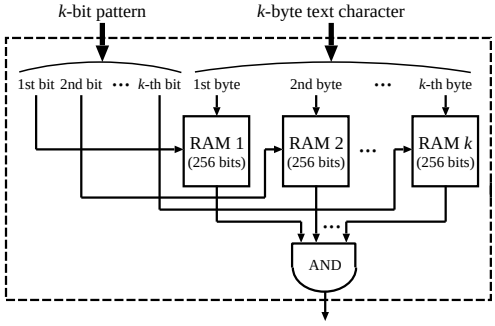


Figure 7. Architecture of character matching tables.

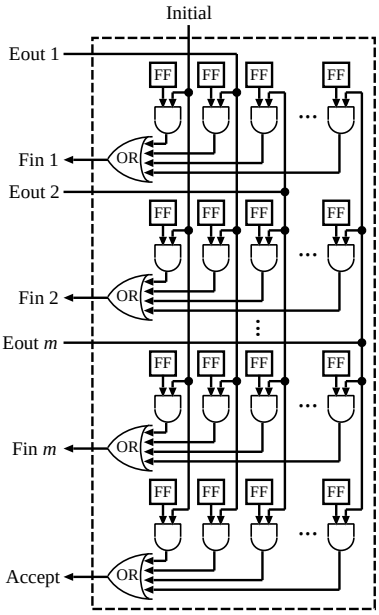


Figure 8. Architecture of feedback array.

pattern character, $\mathrm{RAM}_i[x]$ stores 1. Otherwise, $\mathrm{RAM}_i[x]$ stores 0. If the i-th byte of a pattern character is a don't care character, RAM_i stores 1 for all addresses. Contents of RAMs can be rewritten using the pattern bus.

C. Feedback Array Based on k-Byte String Transition NFAs

The FA is a programmable interconnection network to transmit enable signals fed from the MA to appropriate SMUs. Figure 8 shows its architecture. By setting appropriate bits to the FFs, like crossbar switches, we can program arbitrarily connections. When an FF stores 1, a vertical line and a horizontal line are connected. Otherwise, lines are disconnected.

Since the proposed NIDS based on k-byte STNFA takes in k one-byte characters at a time, network transmission speed s that the NIDS can be applied is improved to $s = 8kf$ bps.

V. Experimental Results

To experimentally evaluate performance of the proposed programmable NIDS, we used the following five regular expressions randomly chosen from rules of the SNORT [19]:

R1: /level/(0|1|2|3|4|5|6|7|8|9)+/(exec|configure)
R2: cookies\s+Monster\s+server\s+engine
R3: template\s*=\s*{$
R4: fn=..(/ |\)
R5: (((\x0*b*dyndns|\x02yi)\x03org)|((\x07dynserv|\x04mooo)\x03com))

where \s denotes a blank character, + is a regular expression operator that means $R+ = R \cdot R*$, \x followed by a two-digit hexadecimal number denotes an ASCII code, and the others are treated as one-byte characters in the alphabet Σ.

A. Results for Multi-Byte Transition NFAs

Table I shows the numbers of states and state transitions in a k-byte transition NFA for each regular expression, where columns of $k = 1$ show results of existing method [20]. From this table, we can see that the numbers of states and state transitions increase as the value of k increases. This is because new accepting states are added, and the number of characters in an alphabet increases as k increases.

By converting into STNFAs, they can be reduced. Since the number of state transitions corresponds to the number of SMUs in the proposed NIDS, we can reduce the circuit size by using STNFAs. When $k = 8$, the number of state transitions is not reduced so much by STNFAs. This is because an 8-byte character consists of 8 one-byte characters, and it already forms a string.

B. FPGA Implementation Results

We designed programmable NIDSs based on the above k-byte STNFAs by setting design parameters as follows: the number of CCs $n = 5$ and the number of SMUs $m = 10, 50, 100, 150, 200$. Since the SNORT rules use 7-bit ASCII characters as one-byte characters, we designed each character matching table in Figure 7 as a 128-bit RAM that is implemented with 2 LUTs in an FPGA. The designed NIDSs were implemented with the Xilinx Virtex-7 XC7VX485T-2FFG1761 FPGA using the Xilinx Vivado Design Suite 2014.2 as a synthesis tool. Table II shows the FPGA implementation results and network transmission speed the proposed NIDSs can achieve.

When $k = 1$, 10 SMUs ($m = 10$) are required in order to realize any rule of the five, since the maximum number of state transitions in an STNFA is 8, as shown in Table I. Although the NIDS with $m = 10$ achieves 429 MHz of operating frequency, its transmission speed is 3.4 Gbps. This corresponds to performance of the existing programmable NIDS [20]. When $k = 2$ and 4, $m = 50$ and 100 are required, respectively. The NIDSs with $m = 50$ and 100 still do not reach 10 Gbps, even though they achieve higher speed than the existing one. When $k = 8$, $m = 150$ is required, but the NIDS can achieve more than 10 Gbps. Even if there is a regular expression that requires $m = 200$, the NIDS with $m = 200$ can achieve more than 10 Gbps when $k = 8$.

 ISBN: 978-1-61208-496-1

TABLE I. NUMBERS OF STATES AND STATE TRANSITIONS IN A k-BYTE TRANSITION NFA.

Rules	Number of states								Number of state transitions							
	Character transition NFAs				String transition NFAs				Character transition NFAs				String transition NFAs			
	$k=1$	$k=2$	$k=4$	$k=8$	$k=1$	$k=2$	$k=4$	$k=8$	$k=1$	$k=2$	$k=4$	$k=8$	$k=1$	$k=2$	$k=4$	$k=8$
R1	22	23	25	29	4	7	13	26	24	38	40	88	6	12	28	85
R2	29	30	32	36	5	12	26	36	32	39	62	144	8	21	56	144
R3	12	13	15	19	4	17	11	19	14	19	39	147	6	13	35	147
R4	7	8	10	14	3	5	9	14	8	10	14	22	4	7	13	22
R5	29	30	32	36	4	7	13	17	32	36	44	60	7	7	25	41

TABLE II. FPGA IMPLEMENTATION RESULTS OF THE PROPOSED NIDSS.

m	Total number of LUTs				Operating frequency [MHz]				Transmission speed [Gbps]			
	$k=1$	$k=2$	$k=4$	$k=8$	$k=1$	$k=2$	$k=4$	$k=8$	$k=1$	$k=2$	$k=4$	$k=8$
10	291	332	662	1,083	429	383	399	405	3.4	6.1	12.8	25.9
50	2,192	2,443	3,743	6,093	351	353	353	312	2.8	5.6	11.3	20.0
100	6,449	6,850	9,550	14,150	297	297	276	255	2.4	4.8	8.8	16.3
150	12,379	14,931	16,957	24,225	241	242	209	212	1.9	3.9	6.7	13.6
200	20,445	22,203	29,429	38,989	216	197	178	182	1.7	3.2	5.7	11.6

*The underlined numbers are results of NIDSs required in order to realize any rule of the five.

In this way, larger k requires larger m (more SMUs), and thus, degrades operating frequency because of longer critical paths in the feedback array. However, it can improve transmission speed significantly since the positive effect due to parallelization by k is larger than its negative effect.

VI. CONCLUSION AND COMMENTS

This paper proposed a programmable NIDS based on a multi-byte STNFA. By using multi-byte STNFAs, the proposed NIDS achieved more than 10 Gbps of network transmission speed that is difficult for existing programmable NIDSs to achieve. Thus, the proposed NIDS can avoid bottleneck even in latest Gigabit network. Since in the proposed NIDS, regular expression patterns can be set by just rewriting contents of memories and registers, the NIDS achieves both fast regular expression matching and quick pattern updating.

In the current design, increasing the number of SMUs makes critical paths in the feedback array longer, resulting in degradation of operating frequency. Thus, improving architecture of the feedback array is one of our future works. We will also study how to minimize the number of state transitions in order to reduce hardware cost.

ACKNOWLEDGMENTS

This research is partly supported by the JSPS Grant-in-Aid for Scientific Research (C), (No. 26330691), 2015.

REFERENCES

[1] J. Aoe, Computer Algorithms: String Pattern Matching Strategies, IEEE Computer Society Press, 1994.

[2] AV-TEST - The Independent IT-Security Institutie, http://www.av-test.org/.

[3] M. Becchi and P. Crowley, “An improved algorithm to accelerate regular expression evaluation,“ Proc. of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems, Dec. 2007, pp. 145–154.

[4] J. Bispo, I. Sourdis, J. M. P. Cardoso, and S. Vassiliadis, “Regular expression matching for reconfigurable packet inspection,” Proc. 2006 IEEE International Conference on Field Programmable Technology, 2006, pp. 119–126.

[5] J. Divyasree, H. Rajashekar, and K. Varghese, “Dynamically reconfigurable regular expression matching architecture,” Proc. International Conference on Application-Specific Systems, Architectures and Processors (ASAP 2008), July 2008, pp. 120–125.

[6] D. Ficara, et al., “An improved DFA for fast regular expression matching,“ ACM SIGCOMM Computer Communication Review, Vol. 38, No. 5, Oct. 2008, pp. 31–40.

[7] D. Ficara, et al., “Differential encoding of DFAs for fast regular expression matching,“ IEEE/ACM Transactions on Networking, Vol. 19, No. 3, June 2011, pp. 683–694.

[8] T. Ganegedara, Y.E. Yang, and V. K. Prasanna, “Automation framework for large-scale regular expression matching on FPGA,“ Proc. 2010 IEEE International Conference on Field Programmable Logic and Applications, 2010, pp. 50–55.

[9] J. E. Hopcroft, J. D. Ullman, and R. Motwani, Introduction to Automata, Theory, Languages and Computation, Second Edition, Addison-Wesley, 2000.

[10] B. L. Hutchings, R. Franklin, and D. Cover, “Assisting network intrusion detection with reconfigurable hardware,” Proc. 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2002, pp. 111–120.

[11] Y. Kaneta, S. Yoshizawa, S. Minato, H. Arimura, and Y. Miyanaga, “Dynamic reconfigurable bit-parallel architecture for large-scale regular expression matching,” Proc. 2010 IEEE International Conference on Field Programmable Technology, Dec. 2010, pp. 21–28.

[12] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner, “Algorithms to accelerate multiple regular expressions matching for deep packet inspection,” Proc. SIGCOMM’06, 2006, pp. 339–350.

[13] A. Mitra, W. Najjar, and L. Bhuyan, “Compiling PCRE to FPGA for accelerating SNORT IDS,” Proc. 2007 ACM/IEEE Symposium on Architecture for Networking and Communications Systems, Dec. 2007, pp. 127–136.

[14] G. Navarro and M. Raffinot, Flexible Pattern Matching in Strings, Cambridge University Press, 2002.

[15] H. C. Roan, W. J. Hwang, and C. T. Dan Lo, “Shift-or circuit for efficient network intrusion detection pattern matching,” Proc. International Conference on Field Programmable Logic and Applications, 2006, pp. 785–790.

[16] Y. SangKyun and L. KyuHee, “Optimization of regular expression pattern matching circuit using at-most two-hot encoding on FPGA,“ Proc. 2010 IEEE International Conference on Field Programmable Logic and Applications, Sept. 2010, pp. 40–43.

[17] R. Sidhu and V. K. Prasanna, “Fast regular expression matching using FPGAs,” Proc. 2001 IEEE International Symposium on Field-Programmable Custom Computing Machines, 2001, pp. 227–238.

[18] R. Smith, C. Estan, S. Jha, and S. Kong, “Deflating the big bang: fast and scalable deep packet inspection with extended finite automata,” Proc. SIGCOMM’08, 2008, pp. 207–218.

[19] Sourcefire Inc., “SNORT network intrusion detection system,” http://www.snort.org/.

 ISBN: 978-1-61208-496-1

[20] H. Takaguchi, Y. Wakaba, S. Wakabayashi, S. Nagayama, and M. Inagi, “An NFA-based programmable regular expression matching engine highly suitable for FPGA implementation,” 18th Workshop on Synthesis and System Integration of Mixed Information Technologies (SASIMI’13), 2013, pp. 231–236.

[21] K. Thompson, “Programming technique: regular expression search algorithm,” Communications of the ACM, Vol. 11, No. 6, June 1968, pp. 419–422.

[22] Y. Wakaba, M. Inagi, S. Wakabayashi, and S. Nagayama, “An efficient hardware matching engine for regular expression with nested Kleene operators,“ Proc. 2011 IEEE International Conference on Field Programmable Logic and Applications, 2011, pp. 157–161.

[23] N. Yamagaki, R. Sidhu, and S. Kamiya, “High-speed regular expression matching engine using multi-character NFA,” Proc. International Conference on Field Programmable Logic and Applications, Aug. 2008, pp. 131–136.

[24] Y.-H. E. Yang and V. Prasanna, “Automatic construction of large-scalce regular expression matching engines on FPGA,” Proc. 2008 International Conference on Reconfigurable Computing and FPGAs, 2008, pp. 73–78.

[25] Y.-H. E. Yang and V. K. Prasanna, “Space-time trade off in regular expression matching with semi-deterministic finite automata,” Proc. IEEE INFOCOM 2011, April 2011, pp. 1853–1861.

 ISBN: 978-1-61208-496-1