A Regular Expression Matching Using Non-Deterministic Finite Automaton

Hiroki Nakahara*, Tsutomu Sasao*, and Munehiro Matsuura*
*Kyushu Institute of Technology, Iizuka, Japan

***Abstract*—This paper shows an implementation of CANSCID (Combined Architecture for Stream Categorization and Intrusion Detection). To satisfy the required system throughput, the packet assembler and the regular expression matching are implemented by the dedicated hardware. On the other hand, the counting of matching results and the system control are implemented by a microprocessor. A regular expression matching circuit is performed as follows: First, the given regular expressions are converted into a non-deterministic finite automaton (NFA). Then, to reduce the number of states, the NFA is converted to a modular non-deterministic finite automaton (MNFA(p)) with p-character-consuming transition. Finally, a finite-input memory machine (FIMM) to detect p-characters is generated, and the matching elements (MEs) realizing the states for the MNFA(p) are generated. We loaded 140 regular expressions of the MEMOCODE 2010 design contest on Terasic Corp. DE3 prototyping board (FPGA: Altera's Stratix III). The maximum throughput of our implementation was 798 mega bits per second (Mbps).**

I. Introduction

This paper shows an implementation of CANSCID (Combined Architecture for Stream Categorization and Intrusion Detection) [7] deep packet inspector performing two different functions: stream categorization (such as L7-filter [5]) and intrusion detection (such as snort [9]).

The metric judging the design are:

1. The number of the category patterns and the intrusion patterns represented by regular expressions.
2. The system throughput that must be higher than the line rate of 500 mega bit per second.

II. Overview of Implemented System

A. Simulated Ethernet

Since the actual ethernet is complex, MEMOCODE2010 design contest uses **simulated ethernet** shown in Fig. 1. An original data is partitioned into **payloads**. A packet consists of a **header** and the payload. The header consists of the source address (SA) flit, the destination (DA) flit, the port number (PORT) flit, the end of data (EOD) flit, and the end of packet (EOP) flit. On the other hand, the payload consists of one or more DATA flits. The simulated ethernet handles up to 64 original data at the same time. For packets of the same original data, they are sent in-order. On the other hand, for packets of different original data, they are sent out-of-order. In the simulated ethernet, the calculation of checksum and the packet loss are ignored.

B. Strategy of Our Implementation

The overview of CANSCID is described in [7]. It performs the following operations:

Job. 1 Assembles the original data from the packets.
Job. 2 Performs stream categorization.
Job. 3 Performs intrusion detection.
Job. 4 Counts matching results, and printout them.

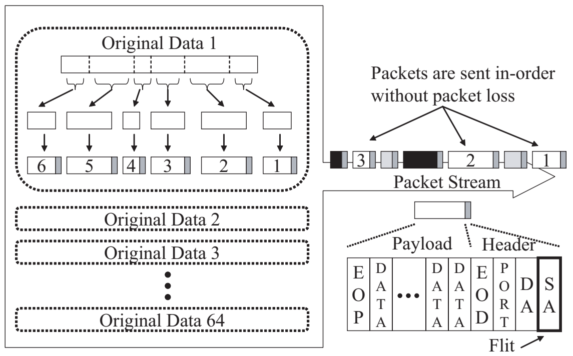


Fig. 1. Simulated Ethernet.

TABLE I
Profile Analysis of CANSCID.

Job	Ratio of CPU Time
Packet Assemble	9.8%
Stream Categorization	20.7%
Intrusion Detection	68.7%
Counts Results and Printout	0.8%

In the design contest, the required throughput must be higher than 500 Mbps. Note that, throughput of any software implementation is at most 10 Mbps. We analyzed a profile of the software implementation of CANSCID. Table I shows that Jobs. 1-3 occupy 99.2% of the CPU time, while the Job. 4 occupies only 0.8%. Hence, by implementing Jobs. 1-3 by hardware, the system can be up to 125 times faster. On the other hand, the control of the system and the counting the matching results are implemented by a microprocessor.

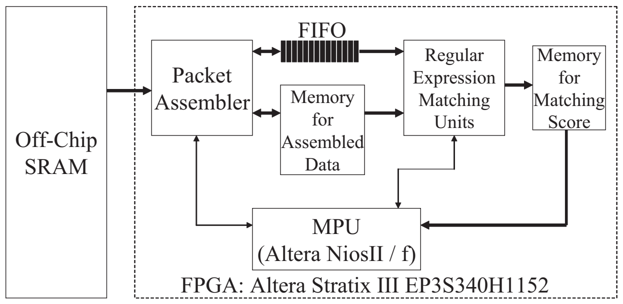

FIFO
Packet Assembler
Memory for Assembled Data
Regular Expression Matching Units
Memory for Matching Score
Off-Chip SRAM
MPU (Altera NiosII / f)
FPGA: Altera Stratix III EP3S340H1152

Fig. 2. Implemented System.

Fig. 2 shows the overview of the implemented system. Initially, the system stores the packets into the SRAM [1]. First, **the packet assembler** reads the packets from the SRAM, and reconstructs the original data, and sends the reconstructed data to the **memory for assembled data**. At the same time, it also sends the start address stored in the memory for assembled data and the length of the data to the FIFO. **A regular expression matching units** scans the assembled data

[1]For performance evaluation, this processing time is ignored.

to perform stream categorization and intrusion detection. **A memory for matching score** stores the matching results produced by the regular expression matching units.

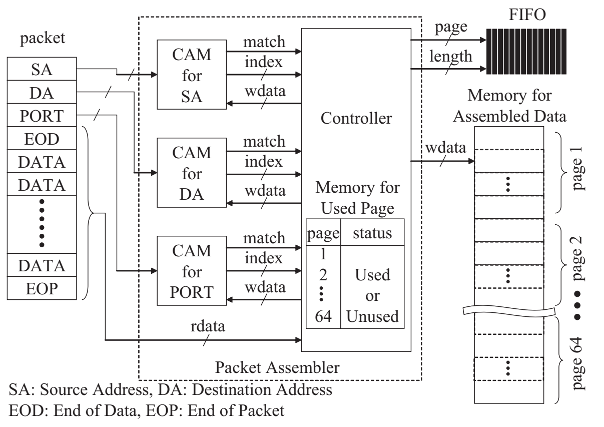


Fig. 3. Packet Assembler.

C. Packet Assembler

Fig. 3 shows the packet assembler that reconstructs the original data from the packets. For packets of different original data, the simulated ethernet sends the packets out of order. On the other hand, for packets of the same original data, it sends in order. The packet assembler performs **a packet classification** to reconstruct the original data. If the headers (consisting of SA, DA, and PORT) of the packets are the same, then the packets come from the same original data. To perform the packet classification, content addressable memories (CAMs) are used. The packet assembler sends the assembled data to the memory for assembled data. The simulated ethernet sends up to 64 data at the same time. Thus, the memory for assembled data has 64 **pages** to retain 64 assembling data. **A memory for used page** stores the page status (used or unused). In our implementation, one page stores up to 2,040 flits.

Algorithm 2.1: The packet assembler reconstructs the original data as follows:

- Alg.1 Clears the CAMs and the memory for used page (Fig. 4(1)).
- Alg.2 Reads the header, and checks to see if it matches a stored header in the CAMs (Fig. 4(2)).
- Alg.2.1 If the header does not match, then the packet assembler stores the header into the CAMs (Fig. 4(3)). In this case, the packet assembler assigns an index corresponding to unused page number showing the memory for unused page.
- Alg.2.2 If the header matches, then the packet assembler read an index stored in the CAMs, and assigns it to the page number.
- Alg.3 Reads the payload from the packet, and stores it in the assigned page in the memory for assembled data (Fig. 4(4)).
- Alg.4 Continues Alg.3 until it reads the EOP.
- Alg.5 If $EOD = 1$, then the packet assembler send the page number and the data length to the FIFO (Fig. 4(5)). When the FIFO is full, the packet assembler waits until the space is available in the FIFO.
- Alg.6 Terminate.

The packet assembler handles the one flit (32 bits) for each clock. On the other hand, the regular expression matching units matches one character (8 bits) for each clock. Since both use

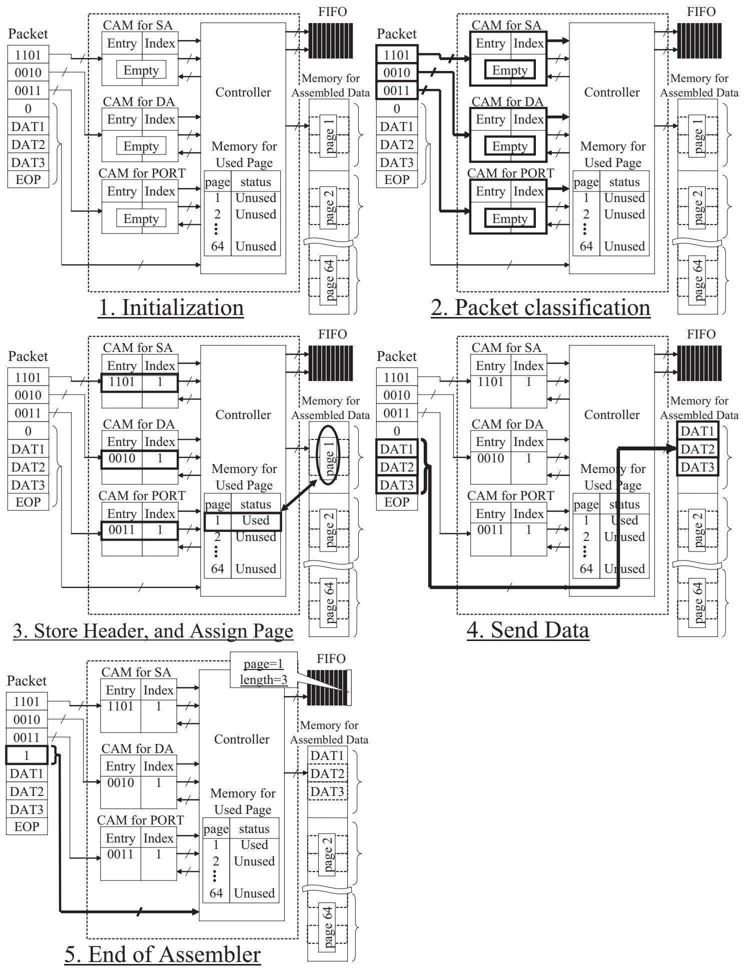


Fig. 4. Operations of Packet Assembler.

the same clocks and they can be processed in parallel, the regular expression matching units becomes the bottleneck.

III. Regular Expression Matching Using NFA

A. Non-deterministic Finite Automaton (NFA)

The regular expressions can be detected by finite automata. In a **deterministic finite automaton (DFA)**, for each state, there is a unique transition to a state for an input, while in a **non-deterministic finite automaton (NFA)**, for each state, there is a multiple transitions to states for an input. In an NFA, there exist transition to other states with **ε-transition** regardless of the inputs. Sidhu-Prasanna's [8] realized the regular expression by an NFA with one-character-consuming transition [1]. Each state for the NFA was implemented by a single character detector and the AND gate. Also, an ε-transition was realized by OR gates and routing on the FPGA. Although the modern FPGA consists of the LUT and the embedded memory, Sidhu-Prasanna's method failed to utilize the embedded memory[2]. So, their method is inefficient with respect to the resource utilization of FPGA. In contrast, our method implements an NFA with p-character-consuming transition by embedded memories and LUTs to utilize the resources of the FPGA efficiently.

[2]Their method uses single character detectors (comparators) instead of the memory shown in Fig. 7

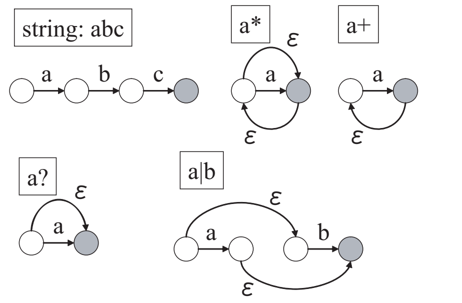
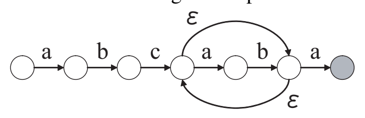


Fig. 5. Conversion of Regular Expression into NFA.



initial	(1, 0, 0, 0, 0, 0, 0)
input 'a'	(1, 1, 0, 0, 0, 0, 0)
ε -transition	(1, 1, 0, 0, 0, 0, 0)
input 'b'	(1, 0, 1, 0, 0, 0, 0)
ε -transition	(1, 0, 1, 0, 0, 0, 0)
input 'c'	(1, 0, 0, 1, 0, 0, 0)
ε -transition	(1, 0, 0, 1, 0, 1, 0)
input 'a'	(1, 1, 0, 0, 1, 0, 1)
ε -transition	(1, 1, 0, 1, 1, 1, $\underline{1}$) Accept 'abca'

Fig. 6. NFA for Regular Expression 'abc(ab)*a'.

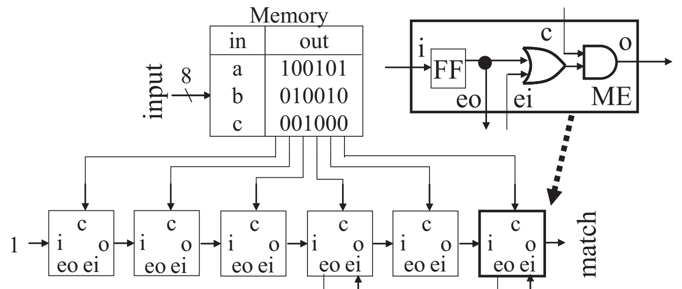


Fig. 7. Realization of NFA [8].

B. Conversion of Regular Expression into NFA

A regular expression consists of **characters** and **meta characters**. Our implementation accepts the following meta characters: '*' (repetition of more than zero character); '?' (zero or one character); '.' (an arbitrary character); '+' (more than one repetition of character); '()' (specify the priority of the operation); '|' (logical OR). Fig. 5 shows examples of conversions of regular expressions into NFAs, where 'ε' denotes an ε-transition, and a gray state denotes an accept state. Fig. 6 shows the NFA accepting the regular expression 'abc(ab)*a', and state transitions with the input string 'abca'. In Fig. 6, each element of the vector corresponds to a state of the NFA, and '1' denotes an active state. Fig. 7 shows the circuit realizing the NFA in Fig. 6. To realize the NFA, first, the memory detects the character for the state transition, and then it sends the character detection signal to the **matching element (ME)**. Each ME corresponds to a state of the NFA, and the ME for the accepted state generates the match signal. In Fig. 7, in each ME, the *FF* corresponds to the element of the vector shown in Fig. 6; *i* denotes the matching signal from the previous state; *o* denotes the matching signal to the next state; *c* denotes the character detection signal; *ei (eo)* denotes the input (output) signal for the ε-transition.

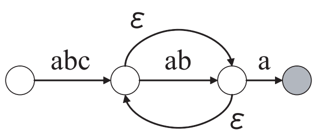


Fig. 10. MNFA(3) Equivalent to NFA Shown in Fig. 6.

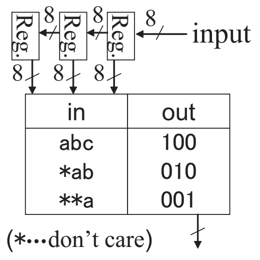


Fig. 11. Finite Input Memory Machine (FIMM).

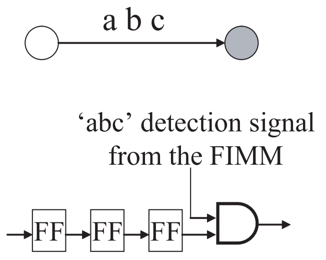


Fig. 12. Circuit for Multi-character-consuming Transition.

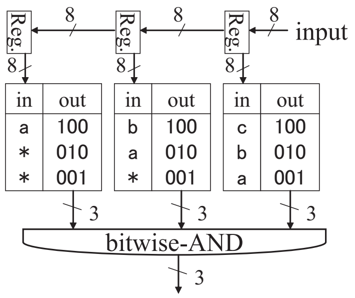


Fig. 13. Partition of FIMM.

C. Realization of NFA Using Memories and Shift Registers

In the circuit for the NFA, each state is implemented by an LUT of an FPGA. Thus, the necessary number of LUTs is equal to the number of states. To reduce the number of states, we use the **NFA with p-character-consuming transition modular non-deterministic finite automaton: MNFA(p)**. Note that, MNFA(1) is simply written by 'NFA'. To reduce an NFA into an MNFA(p), we concatenate characters for sequence of the states. However, to retain the ε-transition, we apply the following restriction: For any state between concatenated characters, no edge is allowed for the ε-transition inputs and outputs. Fig. 10 shows the MNFA(3) that is derived from the NFA shown in Fig. 6.

In an MNFA(p), for each state, there exist transition to other states by consuming string with up to p characters. For the MNFA(3) shown in Fig. 10, the set of transition strings is $\{$abc,ab,a$\}$. To detect the transition strings, we use the **finite input memory machine (FIMM)**. Fig. 11 illustrates the FIMM that detects the strings $\{$abc,ab,a$\}$. When the FIMM detects a string, it generates a detection signal. Fig. 12 illustrates the circuit for three-character-consuming transition. To synchronize the detection signal from the FIMM and the matching signal from the preceding ME, we insert shift registers. Let p be the maximum number of characters for the transition strings of the MNFA(p). The single-memory realization of the FIMM requires $p2^{8p}$ bits. In Fig. 11, since $p = 3$, the necessary memory size is 48 mega bits, which is impractical. Our method decomposes the memory of the FIMM into p parts and uses the bitwise-AND [6]. Each part of the memory is implemented by embedded memories of the FPGA. Fig. 8 shows the circuit for the MNFA(3) in Fig. 10. To realize the multi-character-consuming transition, we insert shift registers into MEs. When the FIMM detects a transition string, it sends the detection signals to the corresponding ME. Then, the ME performs the multi-character-consuming transition.

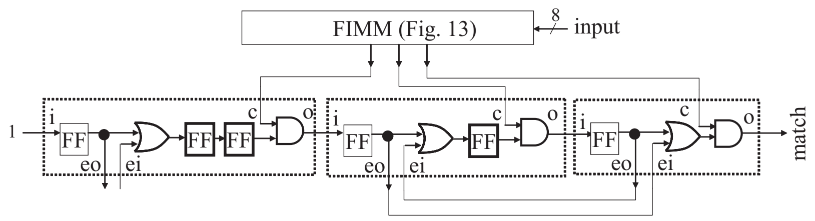


Fig. 8. Circuit for MNFA(3).

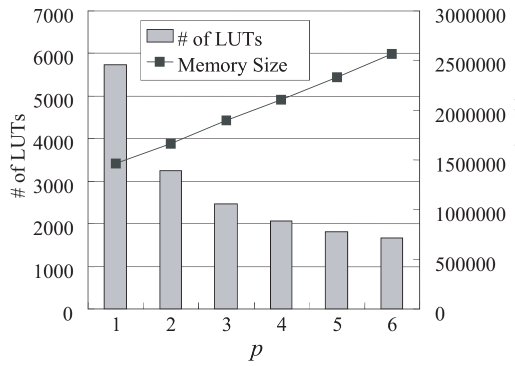


Fig. 9. Number of LUTs and Memory Size for Different Value of p.

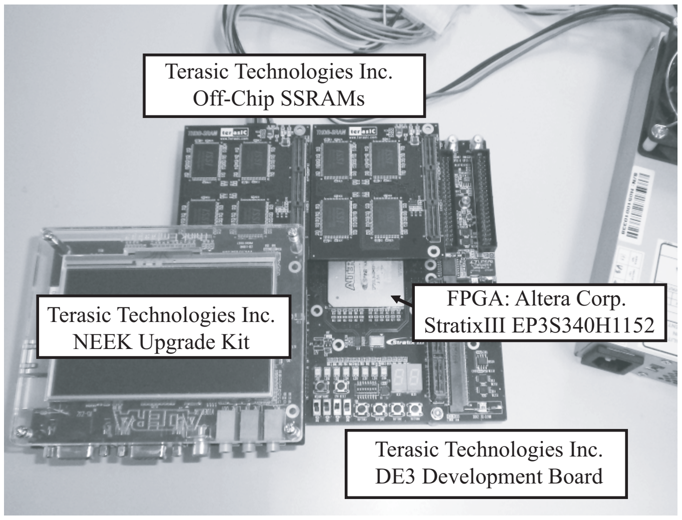


Fig. 14. Photograph of the System.

IV. Implementation Results

A. Environment

We implemented CANSCID on a Terasic Technologies Inc. DE3 development board utilizing Altera Stratix III FPGA (EP3S340H1152C3N4, 270,400 ALUTs and 16,662,528 bits embedded memory). The synthesis tool was Altera Corp. Quartus II version 9.1. We loaded 140 regular expression patterns of MEMOCODE2010 design contest. The embedded processor was Nios II/f. To store the original packet data, we attached two off-chip SSRAMs to the board. Also, to read the packets from the SD-Card, we attached NEEK update kit to the board. Fig. 14 is the photograph of the system.

B. Optimal Value p for MNFA(p)

Let p be the number of characters for the transition strings of NFA. Then, the number of states for the MNFA(p) decreases with p. To implement the MNFA(p), the required number of LUTs is proportional to the number of states. Also, the memory size of the FIMM increases with p. To obtain the value p that reduces both the memory size and the number of states, we realized MNFA(p) for different p. Fig. 9 shows the number of LUTs and the memory size for different values of p of MEMOCODE2010 design contest 140 regular expressions. Fig. 9 shows that an increase of p from 1 to 2 reduces the LUTs by 43.3%. The increase of p from 2 to 3 reduces the LUTs by 23.9%. When p is further increased, the ratios of reduction are 16,5% (p=3), 11,6% (p=4), and 8.3% (p=5), respectively. On the other hand, an increase of p by 1 increases the memory by 11.8%. Thus, in our implementation, we chose $p = 2$.

C. Implementation Results

Our FPGA implementation used 11,218 ALUTs and 4,827,008 bits embedded memory. We set the maximum system clock frequency to 100 MHz. The maximum throughput was 798 Mbps. This outperforms required specification (500 Mbps) of the design contest.

V. Conclusion

We implemented CANSCID on a DE3 development board. To improve the performance, the packet assembler and the regular expression matching units are implemented by dedicated hardware. 140 regular expressions of MEMOCODE 2010 design contest were loaded on an Altera's FPGA. Our regular expression matching circuit is based on the MNFA(p). To detect p characters, an FIMM is used.

Other reduction methods for the NFA-based regular expression circuit include: sharing a part of regular expression circuit [2]; and using the shift register to realize the repeated pattern [3]. In the implementation, the memory of FIMM are decomposed into smaller ones to be implemented by an embedded memory of FPGA. Our method efficiently uses both LUTs and the embedded memory of the FPGA.

VI. Acknowledgments

This research is supported in part by the grant of Regional Innovation Cluster Program (Global Type, 2nd Stage). Discussion with Prof. J. T. Butler was quite useful. Dr. Hiroaki Yoshida encouraged us to participate the design contest. The authors appreciate the organizer's hard work to prepare for the contest.

References

[1] R. Baeza-Yates and G. H. Gonnet, "A new approach to text searching," *COMMUNICATION of the ACM*, , Vol.35, No.10, pp. 74-82, Oct., 1992.
[2] J. C. Bispo, I. Sourdis, J. M.P. Cardoso, and S. Vassiliadis, "Regular expression matching for reconfigurable packet inspection," *Proc. IEEE Int'l Conf. on Field Programmable Technology (FPT 2006)*, pp. 119-126, 2006.
[3] I. Sourdis, J. Bispo, J. M. P. Cardoso and S. Vassiliadis, "Regular expression matching in reconfigurable hardware," *Int. Journal of VLSI Signal Processing Systems*, Vol. 51, Issue 1, pp. 99 - 121, 2008.
[4] Z. Kohavi, *Switching and Finite Automata Theory, McGraw-Hill Inc.*, 1979.
[5] L7 filter official web site, "http://l7-filter.sourceforge.net/".
[6] H. Nakahara, T. Sasao, M. Matsuura, and Y. Kawamura, "A virus scanning engine using a parallel finite-input memory machines and MPUs," *Proc. Int'l Conf. on Field Programmable Logic and Applications (FPL 2009)* Aug. 31- Sept. 2, 2009.
[7] M. Pellauer, A. Agarwal, A. Khan, M. C. Ng, M. Vijayaraghavan, F. Brewer, and J. Emer, "Design contest overview: Combined architecture for network stream categorization and intrusion detection (CANSCID)," *Proc. of Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010)*, Grenoble, France, July 26-28, 2010.
[8] R. Sidhu and V. K. Prasanna, "Fast regular expression matching using FPGAs," *FCCM 2001,* pp. 227-238, 2001.
[9] SNORT official web site, "http://www.snort.org".