Implementation of Simple SNORT Processor for Efficient Intrusion Detection Systems

Ehsan Azimi
Dept. of CE. School of Engineering
Islamic Azad University, Science and Research Branch
Tehran, Iran
azimi@itrc.ac.ir

M.B. Ghaznavi-Ghoushchi
Dept. of EE. School of Engineering
Shahed University
Tehran, Iran
ghaznavi AT shahed.ac.ir

Amir Masoud Rahmani
Dept. of CE. School of Engineering
Islamic Azad University, Science and Research Branch
Tehran, Iran
Rahmani74@yahoo.com

Abstract— **In this paper, a method is proposed for hardware implementation of basic instructions of SNORT software for using in hardware accelerator systems relating to Network Intrusion Detection (NID). The design is implemented in Verilog hardware description language. The design is synthesized both in FPGA on Virtex5 and ASIC (with CMOS Technology 90nm and 65nm). Initial results of hardware description simulation agree with those of SNORT. Similarity between these two results shows that the proposed hardware infrastructure can be exploited as a hardware accelerator in different applications of Intrusion Detection (ID) within networks where needs high processing rate for multi Gb/s data rates. The ASIC synthesis results indicate that the proposed hardware can process payload section of a TCP/IP stream with the rate of 1 Gb/s and 2 Gb/s in TSMC 90nm and TSMC 65nm respectively.**

Keywords-network intrusion detection systems; hardware accelerators; snort; hardware description languges; FPGA; ASIC

I. INTRODUCTION

With daily increase in computer network traffic and network based applications, intrusion detection has gained special importance and network security is more necessary than before. To reach to this goal, one can use a combination of intrusion detection, encryption and access control systems. Intrusion detection systems have enhanced network security much more by checking payload and header in a data packet. Today most of ID systems are software based and are implemented on a multipurpose processor running Microsoft windows or Linux operating systems. SNORT tool is an open source ID that is used on above mentioned OS's. This software is free and this caused it be very popular in computer networks [1].

ID systems hardly adopt with rate increase in networks. Today any of ID systems cannot neutralize attacks as fast as data rate. This shows the need to fast and flexible ID and denying systems [1].

Moving toward hardware systems provides us with more parallel processing capability than software based systems. Hardware based designs may be the only practical solutions that bring as much flexibility and effectiveness as needed in current and future modern networks. An ID hardware accelerator is able to detect intrusion and do pattern matching in Giga-bit rates, checking packets online. To sustain against novel intrusion techniques and attacks, someone can reconfigure these accelerator. In addition, configuration can be done according to specific security requirements and also some other tasks in network may be included in it [1].

As an example, invader detection system in [2] uses N parallel comparator to process N input character simultaneously. Using pipeline causes that this design to have high frequency. Some of its disadvantages are high consumed area, occupied by comparators and much delay in pipeline.

In [3] SNORT rules are implemented with Non-deterministic Finite Automata (NFA). In proposed design in [3], a content decoder is used instead of distributed comparators to reduce consumed area of FPGA.

In [4], string pattern matching in ID systems is implemented using Finite State Machine (FSM) and bit separating technique of FSM. To gain high rates, patterns are broken to sub-patterns and matching of these sub-patterns is checked separately. The result of all sub-pattern matching modules is combined to make total decision. In this outline, pattern updating is possible when the matching algorithm is running without reconfiguring matching engine.

The main problem with software ID systems is their inadaptability with rate increase in computer networks. To resolve this problem, hardware ID systems are considered nowadays. SNORT is a software ID system which is open source and is very common in computer networks. This software has some rules that are constructed from a limited set of instructions [1].

In this paper, we proposed new hardware architecture for implementing SNORT instructions. The remaining part of paper is as follow: in section II, fundamentals of SNORT as a basic tool in ID systems are discussed. In section III, proposed hardware architecture for equivalent implementation of SNORT instructions is explained. Simulation and synthesis results are given in section IV that leads to conclusion remarks in section V.

II. SNORT SOFTWARE

SNORT tool is open source ID software which runs on Windows or Linux operating systems. Being free and having complete set of capabilities and the possibility to be installed

978-1-4244-4738-1/09/$25.00 ©2009 IEEE

on different machine and operating systems caused that SNORT be popular in ID systems in computer networks.

SNORT in complete version is a kind of Network Intrusion Detection System (NIDS). This software is configurable in three modes: sniffer mode, packet recording mode and ID system. In sniffer mode, as its name indicates, SNORT is just a simple sniffer and displaying the content of transmitted packet within network [5]. In recording mode, SNORT stores data of packets in a specified file. In ID mode, SNORT searches and analyzes packet based on two previous modes and determined rules. After ID, required actions are taken [5].

In ID mode, we can determine set of rules as criteria for detecting invading programs. SNORT checks network traffic based on a characteristic database of invading programs. For example someone can adjust SNORT by a rule to make a warning message or to take proper action whenever an access in a defined protocol from/to a specific port and from/to specific destination with a content containing a specific string happens. Each rule of SNORT has two parts: header and content of data packet. In the rule header, protocol type, IP address and source and destination port numbers of invading packet is located. Content part contains a string pattern in ASCII, Hex format or combination of two formats. In SNORT rule, Hex part is embraced between two '|' sign. A sample SNORT rule is shown in Fig. 1.

```
alert tcp any any ->192.168.1.0/32 111(content:
                    "idc|3a3b|"; msg: "mountd access";)
```

Figure 1. Sample of a SNORT rule

This rule is due to an invading program that is within a TCP protocol and its source has unknown IP address and port number. The IP address of destination is *192.168.1.0* and its port number is *111*. Many of packets in the network have these attributes. Therefore to identify packets of an invader program, data packet content should contain pattern "idc|3a3b|". This pattern includes characters *c, d* and *i* and bytes *3a* and *3b* [5].

A. Byte_test Instruction

Each SNORT rule is a combination of a few different instructions, each of which has its own format and adds a specific ability to SNORT rule. One of the time consuming and major instructions of SNORT is *byte_test*. This instruction can compare some bytes by a specific binary value (or binary equivalent of a byte string). Fig. 2 shows its format.

```
byte_test:  <bytes to convert>, [!]<operator>, <value>,
            <offset>, [,relative] [,<endian>] [,<number type>, string];
```

Figure 2. SNORT's *byte_test* format [5]

"bytes to convert" means number of bytes in data part of network packet that must be chosen for comparison. According to *"offset"* parameter, a few bytes from beginning of packet are ignored and remaining bytes are considered. In case of *"relative"* parameter, *"offset"* is considered in relation with previous matched pattern. In *"value"* parameter, the value that intended bytes are compared by, is determined. *"string"* parameter describes that intended bytes are from string type. In this case *"number type"* parameter describes that the converted string to be displayed as Hex, Dec or Oct format. *"operator"* parameter defines the way of comparison which can be one of the *'<'*, *'>'*, *'='*, bitwise logical and (*'&'*) or bitwise logical or (*'^'*). *'!'* can be used before each of mentioned operators (e.g. *!<*). Using *'!'* solely means *'≠'*. *"endian"* parameter determines that data is little endian or big endian. The default is big endian [5].

B. Byte_jump Instruction

The next instruction from SNORT which is implemented in this paper is *byte_jump* . By using this instruction someone can ignore several bytes in payload and take a new location in payload as start point of data for executing SNORT instructions. Fig. 3 shows *byte_jump* instruction format.

```
byte_jump:  <bytes to convert>, <offset>, [,relative] [,multiplier
            <multiplier value>] [,big] [,little][,string]
            [,hex] [,dec] [,oct] [,align] [,from_beginning];
```

Figure 3. SNORT's *byte_jump* format [5]

Based on an *"offset"* parameter, *byte_jump* instruction overlooks some bytes from the beginning of the payload, reading number of *"bytes to convert"* bytes from packet. If parameter *"relative"* exists in the *"byte_jump"* instruction, this offset is started from the last matched pattern. The *"string"* parameter determines that whether the data that is read from payload is in string format. Parameters *"big"* and *"little"* clarify the method of evaluating the data; the default value is big endian. With *"align"* parameter someone can round up numerical value of converted bytes to 32 bits. Parameter *"from_beginning"* is used for skip forward from the beginning of payload instead of current position of pointer in the packet. Data which is read from payload is multiplied by parameter *"multiplier <value>"* after being converted to its numerical equivalent [5].

III. PROPOSED ARCHITECTURE

The processor introduced in this paper is named *"SMIP*[1]*"*. This processor implements instructions *byte_test* and *byte_jump* of SNORT. Common parts of these instructions are implemented in hardware modules and in executing period are used by other parts of processor leads to reducing hardware and area consumption.

We selected *byte_test* and *byte_jump* instructions because of their high usage percent in SNORT rules. We have written software with *Perl* language and counted the number of *byte_test* and *byte_jump* in SNORT rules. Our investigations show that *byte_test* and *byte_jump* are used at least in %45 and %39 of SNORT rules.

"SMIP" processor has three instructions namely *b_test*, *b_jump* and *halt*. Instruction *b_test* is a three word instruction that introduces SNORT's *byte_test* instruction with all its parameters to the processor. Any of words constituting *b_test* instruction has 32 bits. In the first word of *b_test*, its opcode and value of *"bytes to convert"*, *"relative"*, *"endian"*, *"string"* and *"operator"* parameters in *byte_test* instruction is brought. Table I explains how these parameters are placed in *b_test* instruction.

[1] Snort Minimal Instruction Processor

TABLE I. PLACEMENT OF PARAMETERS OF SNORT'S *BYTE_TEST* INSTRUCTION IN FIRST WORD OF *SMIP*'S *B_TEST* INSTRUCTION

Parameter Name	***Bit/bits Number***
Opcode of *b_test*	15-0
"bytes to convert" parameter value in SNORT's *byte_test*	19-16
Existance/nonexistance of *"relative"* parameter in SNORT's *byte_test*	20
"endian" parameter value in SNORT's *byte_test*	21
Existance/nonexistance of *"string"* parameter in SNORT's *byte_test*	22
"operator" parameter value in SNORT's *byte_test*	26-23
Unused	31-27

Opcode value of *b_test* instruction is 10 Hex. If parameter *"relative"* in *byte_test* instruction exists, the value of bit number 20 is one otherwise it is zero. A one in bit number 21 means that the data are big endian and a zero in it, means that the data are little endian. A one/zero in bit number 22 means existence/nonexistence of parameter *"string"* in *byte_test* instruction. The second and third words in *b_test* determine parameters *"offset"* and *"value"* respectively.

The second instruction of *"SMIP"* processor is *b_jump*. *b_jump* instruction is used to introduce SNORT's *byte_jump* and all its parameters to processor. Due to parameters in *byte_jump*, *b_jump* is constructed from 2 or 3 words each has 32 bits. Opcode of *b_jump* instruction and *"bytes to convert"*, *"relative"*, *"endian"*, *"string"*, *"align"* and *"from_beginning"* parameters of SNORT's *byte_jump* are written in the first word. Table II explains how *byte_jump* parameters are placed in *b_jump*'s first word.

Opcode value of *b_jump* instruction is 20 Hex. Bit numbers 20, 21 and 22 in first word of *b_jump* are used as their corresponding bits in *b_test*. If parameter *"align"* exists in *byte_jump*, bit number 21 will be one otherwise it will be zero. A one/zero in bit number 24 means existence/nonexistence of *"from_beginning"* parameter in *byte_jump* instruction. Bit number 25 shows existence/nonexistence of parameter *"multiplier <value>"* in *byte_jump*. Being one means that *b_jump* has three words, otherwise being zero means that *b_jump* has two words. Second and third words of *b_jump* explain parameters *"offset"* and *"multiplier <value>"* in *byte_jump* instruction.

The last instruction of *"SMIP"* processor is *halt*. This instruction has one word and its opcode value is 30 Hex. *halt* instruction has no parameter. This instruction ends the processor run.

TABLE II. PLACEMENT OF PARAMETERS OF SNORT'S *BYTE_JUMP* INSTRUCTION IN FIRST WORD OF *SMIP*'S *B_JUMP* INSTRUCTION

Parameter Name	***Bit/bits Number***
Opcode of *b_jump*	15-0
"bytes to convert" parameter value in SNORT's *byte_jump*	19-16
Existance/nonexistance of *"relative"* parameter in SNORT's *byte_jump*	20
"endian" parameter value in SNORT's *byte_jump*	21
Existance/nonexistance of *"string"* parameter in SNORT's *byte_jump*	22
Existance/nonexistance of *"align"* parameter in SNORT's *byte_jump*	23
Existance/nonexistance of *"from_beginning"* parameter in SNORT's *byte_jump*	24
Existance/nonexistance of *"multiplier value"* parameter in SNORT's *byte_jump*	25
Unused	31-26

"SMIP" processor has several modules each doing special operations. A central control unit manages these modules. These modules include: *get_stream*, *data_index*, *data_extract*, *bcd_int_10*, *byte_test*, *byte_jump*, *controller* and *ram*. Fig. 4 exhibits internal view of *"SMIP"* processor.

The modules *get_stream*, *data_extract*, *byte_test*, *byte_jump* and *controller* are implemented through finite state machines (FSM). *data_index* and *bcd_int_10* are sequential modules that put data on output pins in next clock. Instructions of *"SMIP"* after being converted to binary codes are stored in *ram* module.

"SMIP" processor has general clock and reset. Reset is synchronized with clock and processor will reset with *rst* signal. In reset procedure, the modules that are implemented by FSM will have *s_idle* state.

With *cntrl_start* signal, processor starts up; *controller* and *get_stream* modules start to work simultaneously. *get_stream* module receives data in the payload section of TCP/IP packet a byte in each clock and stores them in a 16-bytes internal buffer. When buffer fills up, *get_stream* module gives these data to *data_extract* module through its output and enables *gs_ready* signal to say *controller* module that data are ready.

In parallel with *get_stream*, *controller* module fetches instructions from *ram* module, decodes and executes them. Management of fetch, decode and execute operations is done in *controller* module with the aid of a FSM. The *controller* module produces required control signals, enabling different modules to execute instructions and receives results from executing modules.

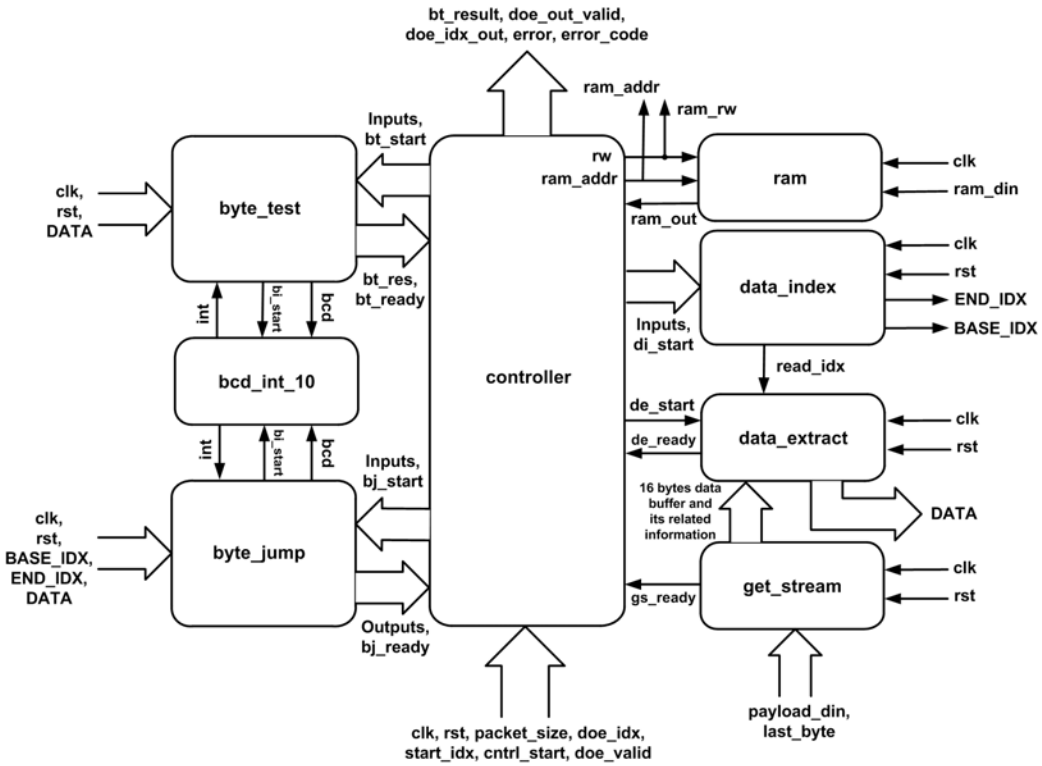

Figure 4. Internal view of *"smip"* processor

The FSM of *controller* module has two branches. The right branch executes when the decoded instruction is *b_test*. The *controller* module reads and stores the parameters of *b_test* from *ram* in *s1*, *s2* and *s3* states. The left branch executes when the decoded instruction is *b_jump*. The *controller* module reads and stores the parameters of *b_jump* from ram in *s7*, *s8* and *s9*. Fig.5 exhibits the FSM of *controller* module.

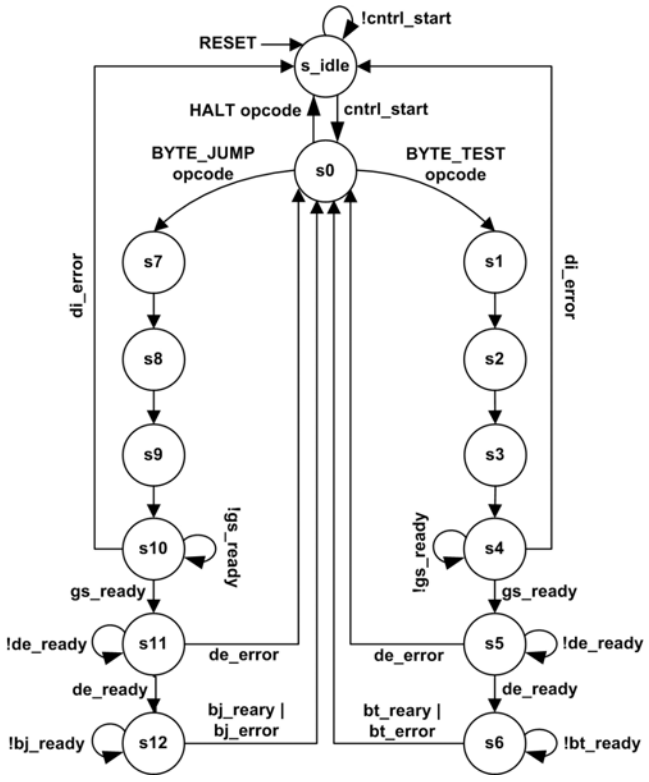

Figure 5. FSM of *controller* module

After fetch and decode steps, FSM of *controller* module enters into execute step. Depending on opcode of decoded instruction, first *data_index* module is enabled by *controller* in *s3 -> s4* or *s9 -> s10*. This module needs one clock to operate. *data_index* module computes index in TCP/IP payload where required data for *byte_jump* and *byte_test* modules start. After completion of *data_index* operation, in *s4* or *s10*, FSM of *controller* module waits for *gs_ready* signal from *get_stream* module. *gs_ready* announces *controller* module that *get_stream* output is ready to be used in other modules. Receiving *gs_ready*, FSM of *controller* enables *data_extract* module in *s4 -> s5* or *s10 -> s11* and waits in *s5* or *s11* for reception of *de_ready* or *de_error* signals from *data_extract* module.

data_extract module as its name indicates, extracts data required by *byte_test* and *byte_jump* modules from 16-bytes packet received from *get_stream*. The *controller* module enables *data_extract* module after receiving *gs_ready* signal. In data extracting process within *data_extract*, output of *data_index* modules is also exploited. After successful completing of *data_extract* operation, it gives extracted data to *byte_test* and *byte_jump* modules and enables *de_ready* signal to say *controller* module that data are ready. By error occurrence, *data_extract* enables *de_error* to inform *controller* about an error in *data_extract* module. When *controller* module receives *de_error*, puts proper values on processor outputs and goes to *s0* to start decoding of next instruction. *de_ready* signal explains the successful end of *data_extract* module operation and informs *controller* module that its output is ready. Upon receiving *de_ready*, FSM of *controller* enables *byte_test* or *byte_jump* module in *s5->s6* or *s11->s12* depending on fetched instruction opcode and waits for

bt_ready, *bj_ready*, *bt_error* or *bj_error* signals. *bt_error* or *bj_error* signals are emitted on error events in *byte_test* or *byte_jump* modules. When *controller* receives these error signals, puts proper values on processor outputs and reads next instruction in *s6 -> s0* or *s12 -> s0*. On the other hand, *bt_ready* and *bj_ready* signals tell *controller* that *byte_test* and *byte_jump* modules operations are executed successfully.

If input data to *byte_test* and *byte_jump* modules are in BCD format, they convert BCD number to integer by enabling *bcd_int_10*. *byte_jump* and *byte_test* modules give results of *byte_jump* and *byte_test* of SNORT to *controller* and enable *bj_ready* or *bt_ready* signals. When *controller* module receives *bt_ready* or *bj_ready* signals, transfers output of *byte_jump* or *byte_test* module to processor output and reads next instruction in *s6 -> s0* or *s12 -> s0*. Decode and execute of next instruction will start in *s0*.

IV. SYNTHESIS RESULTS

The proposed architecture is implemented using Verilog hardware description language and the *"SMIP"* processor is functionally tested by comparing the execution results of *b_test* and *b_jump* instructions by various parameters values and various payload data with the execution results of their counterparts in SNORT [6]. Then, the designed processor synthesized using Xilinx ISE 10.1 on Virtex5 platform. The synthesis results indicate that the proposed architecture can achieve 35.448 MHz clock rate. The post synthesized specifications are shown in Table III.

TABLE III. FPGA SYNTHESIS RESULTS FOR THE PROPOSED PROCESSOR

Longest delay path *(ns)*	28.210
Used slice registers (out of 19200)	1031
Percent of used slice registers	5
Used slice LUTs (out of 19200)	3081
Percent of used slice LUTs	16
Implementation platform	Virtex5
Circuit frequency *(MHz)*	35.448

The proposed hardware synthesized in ASIC with CMOS technology 90nm and 65nm too. The synthesis results indicate that the clock rate of proposed architecture is 166.67 MHz and 333.34 MHz in TSMC 90nm and TSMC 65nm respectively. The ASIC synthesis results are shown in Table IV.

In TSMC 90nm, the longest delay path is 6 ns. The proposed architecture processes 128 input bits in 16 clocks. We assume that clock period is 8 ns. By this click period, the proposed processor needs 128 ns for processing 128 input bits. Therefore, this hardware processes payload section of a TCP/IP stream with the rate of 1 Gb/s.

In TSMC 65nm, the longest delay path is 3 ns. In this technology we assume that the clock period is 4 ns. This means that the proposed hardware processes payload section of a TCP/IP stream with the rate of 2 Gb/s.

TABLE IV. ASIC SYNTHESIS RESULTS FOR THE PROPOSED ARCHITECTURE

Specifications	**Technology TSMC 65nm**	**Technology TSMC 90nm**
Clock frequency *(MHz)*	333.34	166.67
Combinational area	27661.320172	78161.430346
Noncombinational area	12731.760284	22459.248282
Total cell area	40393.080456	100620.678629
Global operating voltage	1.1	0.9
Cell internal power *(mW)*	5.6361	2.8697
Net switching power *(uW)*	77.1816	31.6727
Cell leakage power *(uW)*	302.4344	503.0192
Total dynamic power *(mW)*	5.7133	2.9013

V. CONCLUSION

In this paper, we introduced a new hardware implementation of basic instructions of SNORT software. This hardware solution may be used as a hardware accelerator to execute SNORT instructions. This architecture is based on modeling of operation and instructions of SNORT with the aid of related finite state machines and a central controlling block for coordinating execution of instructions. Programmability of this system may be used to increase system flexibility and making different instructions and operations in computer networks intrusion detection field.

The proposed architecture is implemented by Verilog hardware description language and functionally tested by comparing the simulation results of *b_test* and *b_jump* instructions by various parameters values and various payload data with the execution results of their counterparts in SNORT [6]. We have also synthesized the proposed architecture both in Virtex5 and ASIC (with CMOS Technology 90nm and 65nm). The ASIC synthesis results indicate that in TSMC 90nm and TSMC 65nm, the proposed hardware can process payload section of a TCP/IP stream with the rate of 1 Gb/s and 2 Gb/s respectively.

REFERENCES

[1] M. Aldwairi, "Hardware-efficient pattern matching algorithms and architecture for fast intrusion detection", Ph.D. Dissertation, North Carolina State University, Raleigh, 2006.

[2] I. Sourdis and D. Pnevmatikatos, "Pre-decoded CAMs for efficient and high-speed NIDS pattern matching," in Proc. 12th Annu. IEEE Symp. Field Programmable Custom Computing Machines, 2004, pp. 258–267.

[3] C. R. Clarck, "Design of efficient FPGA circuits for matching complex patterns in network intrusion detection systems," M.S. Dissertation, Georgia Institute of Technology, 2003.

[4] V. Rahmanzadeh, M.B. Ghaznavi-Ghoushchi, "A Multi-Gb/s Parallel String Matching Engine for Intrusion Detection Systems," Advances in Computer Science and Engineering 13th International CSI Computer Conference, CSICC 2008 Kish Island, Iran, March 9-11, 2008 Revised Selected Papers, DOI: 10.1007/978-3-540-89985-3, 2009

[5] Snort Users Manual, http://www.snort.org, viewed at 2009-5-14.

[6] Brian Caswell, "Extending snort, without knowing C for dirt", http://ftp.timisoara.roedu.net/pub/pub/packages/snort/dl/contrib/patches/snort-pcre/presentations, viewed at 2008-12-15.