Configurable String Matching Hardware for Speeding up Intrusion Detection

Monther Aldwairi*, Thomas Conte, Paul Franzon

Department of Electrical and Computer Engineering, North Carolina State University,
Box 7911, Raleigh NC, 27695
{mmaldwai, conte, paulf}@ncsu.edu

***Abstract*—Signature-based Intrusion Detection Systems (IDSs) monitor network traffic for security threats by scanning packet payloads for attack signatures. IDSs have to run at wire speed and need to be configurable to protect against emerging attacks. In this paper we consider the problem of string matching which is the most computationally intensive task in IDS. A configurable string matching accelerator is developed with the focus on increasing throughput while maintaining the configurability provided by the software IDSs. Our preliminary results suggest that the hardware accelerator offers an overall system performance of up to 14Gbps.**

***Index Terms*—Intrusion detection, Snort accelerator, string matching**

I. Introduction

Traditionally networks have been protected using firewalls that provide the basic functionality of monitoring and filtering traffic. Firewall users can then write rules that specify the combinations of packet headers that are allowed through. However, not all incoming malicious traffic can be blocked and legitimate users can still abuse their rights. Intrusion Detection Systems (IDSs) go one step further by inspecting packet payload for attack signatures. Currently, most IDSs are software based running on a general purpose processor. Snort [1] is a widely used open-source IDS in which the rules refer to the header as well as to the packet payload. A sample Snort rule that detects CGI-PHF attack is shown in Fig. 1. The rule examines the protocol, source IP address, source TCP port, destination IP address and destination TCP port. The part enclosed in parenthesis is the rule options that are executed if the packet headers match. The content option indicates that the packet payload is to be matched against the string enclosed in double quotes.

Intrusion detection can be divided into two problems; packet filtering or classification based on header fields and string matching over the packet payload. The first problem was studied extensively in the literature and many algorithms were suggested [5]. A recent study [4] showed that the string matching routines in Snort account for up to 70% of the total execution time. We also have studied the snort rules and have showed that 87% of the rules contain strings to match against. Therefore, the second problem of string matching is the most computationally intensive.

```
alert tcp any any -> 10.1.1.0/24 80 (content:
                "/cgi-bin/phf")
```

Fig. 1. Sample Snort rule

The explosion of recent attacks by Code Red and MSBlast affected the productivity of computer networks all over the world. It is also becoming

*Phone 919 513 2015; fax 919 515 2285

increasingly difficult for software based IDSs running on general purpose processors to keep up with increasing network speeds (OC192 and 10Gbps at backbone networks). This has prompted the need to accelerate intrusion detection and to maintain the configurability needed to detect new attacks. Several hardware accelerators have been proposed. For example, Deterministic Finite Automata (DFA) mapped on an FPGA has been used to accelerate string matching. However, DFA based implementations achieve low throughput and are complex to build and configure. On the other hand, discrete or parallel comparators were used to achieve higher throughput at the expense of increased area and poor scalability. CAM based solutions reduce the area used by discrete comparators and achieve similar throughput. Finally, Bloom filters and hash functions were used to compress the string set, find probable matches and reduce the total number of comparisons.

This paper focuses the most computationally intensive part of the problem that is the string matching of the packet payload against hundreds of patterns at wire-speed. We suggest a memory based accelerator that is reconfigurable and have high throughput. The IDS accelerator is composed of a software based component that runs on a general purpose processor, and a standard RAM based technique for FSM implementation. The software generates an FSM from the set of strings extracted from the Snort rule database. The FSM matches multiple strings at the same time based on the Aho-Corasick string matching algorithm. This accelerator is flexible, easy to update and can achieve high throughput. The throughput increases as the on-chip RAM bandwidth increases.

The rest of the paper is organized as follows. Section II gives a background on string matching algorithms, summarizes IDS acceleration efforts and points out some of the differences between the proposed accelerator and the other architectures. Section III describes the suggested architecture for string matching. Section IV presents the simulation results, analyzes of the performance of the accelerator and compares it with previous work. Section V summarizes the contributions of this paper and discusses directions for future work.

II. RELATED WORK

Most known IDS implementations use a general purpose string matching algorithms, such as Boyer-Moore (BM) [3]. BM is the most widely used algorithm for string matching, the algorithm compares the string to the input starting from the rightmost character of the string. To reduce the large number of comparisons, two heuristics are triggered on a mismatch. The bad character heuristic shifts the search string to align the mismatching character with the rightmost position at which the mismatching character appears in the search string. If the mismatch occurs in the middle of the search string, then there is suffix that matches. The good suffix heuristic shifts the search string to the next occurrence of the suffix in the string. Fisk and Varghese suggested a set-wise Boyer-Moore-Horspoolalgorithm specifically for intrusion detection [12]. It extends BM to match multiple strings at the same time by applying the single pattern algorithm to the input for each search pattern. Obviously this algorithm does not scale well to larger string sets.

On the other hand, Aho-Corasick (AC) [2] is a multi-string matching algorithm, meaning it matches the input against multiple strings at the same time. Multi-string matching algorithms generally preprocess the set of strings, and then search all of them together over the input text. AC is more suitable for hardware implementation because it has a deterministic execution time per packet. Tuck et al. [13] examined the worst-case performance of string matching algorithms suitable for hardware implementation. They showed that AC has higher throughput than the other multiple string matching algorithms and is able to match strings in worst-case time linear in the size of the input. They concluded that their compressed version of AC is the best choice for hardware implementation of string matching for IDS.

We use a different method to store the AC database in an SRAM that achieves a higher throughput than Tuck's implementation while having a similar memory requirement. It works by building a tree based state machine from the set of strings to be matched as follows. Starting with a default no match state as the root node, each character to be matched adds a node to the machine. Failure links that point to the longest partial match state are added. To find

matches, the input is processed one byte at a time and the state machine is traversed until a matching state is reached. Fig. 2 shows a state machine constructed from the following strings {hers, she, the, there}. The dashed lines show the failure links, however the failure links from all states to the idle state are not shown. This gives an idea of the complexity of the FSM for a simple set of strings.

There have been several attempts to accelerate IDS recently, most of the implementations used regular expressions. Regular expressions are generated for every string in the rule set and a Nondeterministic/ Deterministic Finite Automata (N/DFA) that examines the input one byte at a time is implemented. FAs are complex, hard to implement, have to be rebuilt every time a string is added and result in designs with a modest throughput. Sidhu and Prasanna mapped an NFA into an FPGA [5]. Carver et al. wrote a regular expression generator in JHDL that extracts strings from the Snort database, generates regular expressions and a netlist for a Xilinx FPGA [6].

Other architectures used discrete comparators to exploit parallelism and achieve higher throughput. The disadvantage of this approach is the large area required. Cho et al., for example, used four parallel comparators per string [10], and Sourdis et al. used pipelining as well as discrete comparators to double the throughput [7]. Several implementations [8, 11] have used CAMs and DCAMs along with comparators to reduce the area and achieve similar throughput to the discrete comparators implementations. The drawback is the high cost and the high power requirement of CAMs.

Recently, Dharmapurikar et al. [9] used bloom filters to perform string matching. The strings are compressed by calculating multiple hash function over each string. The compressed set of strings is stored into a small memory which is then queried to find out whether a given string belongs to the compressed set. If a string is found to be a member of a bloom filter, it is declared as a possible match and a hash table or regular matching algorithm is needed to verify the membership. Bloom filters use less memory, are easy to reprogram and achieve a higher throughput than DFA implementations.

Tuck et al. [13] stored the high level nodes including the pointers to the next and failure states in the RAM. Because of that a huge memory of about 53MB was needed to store the Snort rules set. They used the analogy between IP forwarding and string matching to apply bit-mapping and path compression to the AC tree, reducing its size to 2.8MB and 1.1MB, respectively. Our approach stores the state tables in the RAM and uses a minimal logic to traverse the tables and find a match. The state tables are around 3MB in size without the use of any compression techniques.

III. ACCELERATOR ARCHITECTURE

The accelerator is a part of the configurable network processor architecture shown in Fig. 3. It consists of a 2-wide multiple issue VLIW processor with hardware support for eight hyper threads. The memory system consists of multi-port RAM and a high speed DMA. A number of configurable accelerators are used to speed up specific networking tasks such as IP forwarding, quality of service and string matching for intrusion detection.

The IDS is composed of two components; a software that runs on the VLIW core and a hardware string matching accelerator. The software extracts the strings from the Snort database, creates the FSM tree and generates the state tables. The hardware is shown in Fig. 4. It implements a Mealy FSM and consists of a RAM to store the state tables, a register to hold the current state, and control logic to access the RAM and find a match.

Incoming packets need to be matched only against a subset of rules that match the packet header in the Snort database. To avoid creating one large complicated FSM for all of the strings in the database, the software performs a simple rule classification resulting in a smaller FSM or state table for every class. Rules are classified based on the header fields, mainly the protocol and port numbers, into classes such as ICMP, FTP, SMTP, Oracle, Web-CGI…etc. This makes the software faster, reduces the RAM size and exploits parallelism between packets to increase the throughput.

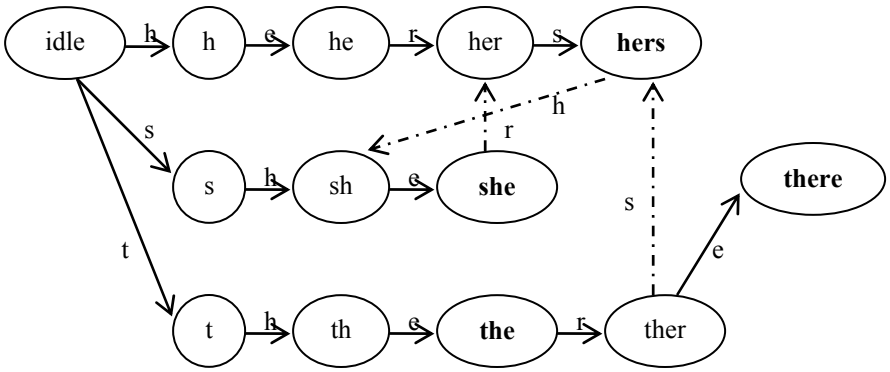

Fig. 2. Finite state machine diagram

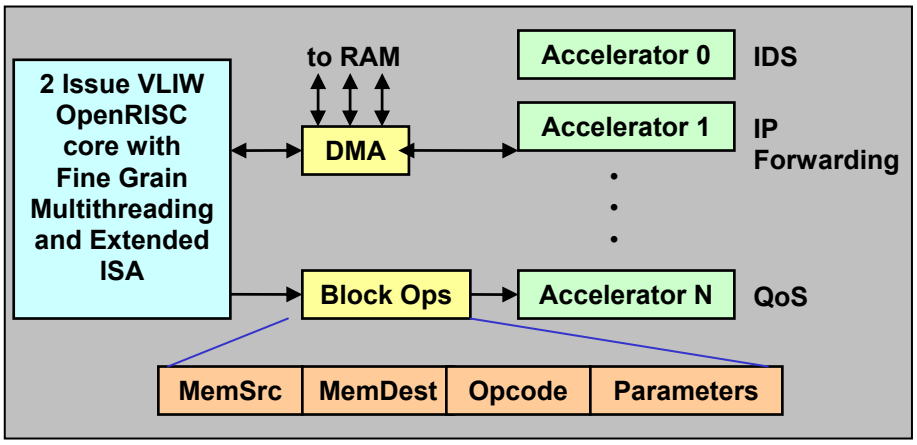

Fig. 3. Configurable network processor architecture

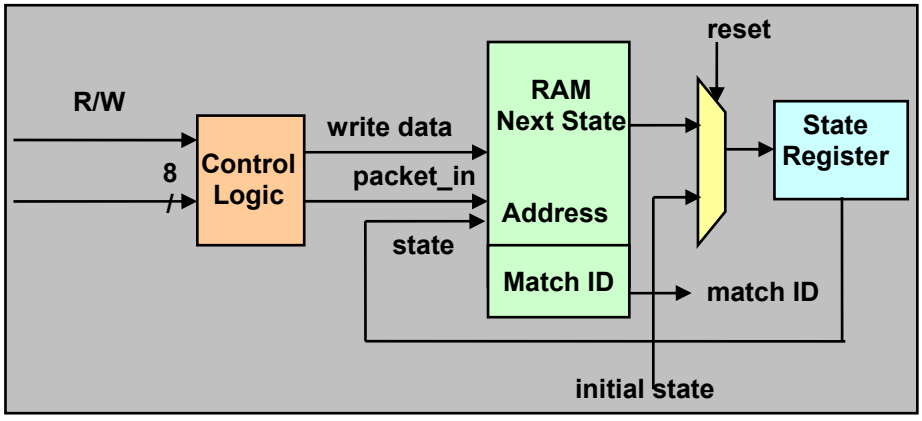

Fig. 4. String matching accelerator

Table 1. State table

	Input Character				
	e	**h**	**r**	**s**	**t**
- /(idle)	-,0	h,0	-,0	s,0	t,0
h	he,0	h,0	-,0	s,0	t,0
he	-,0	h,0	her,0	s,0	t,0
her	-,0	h,0	-,0	hers,1	t,0
hers	-,0	sh,0	-,0	s,0	t,0
s	-,0	sh,0	-,0	s,0	t,0
sh	she,2	h,0	-,0	s,0	t,0
she	-,0	h,0	her,0	s,0	t,0
t	-,0	th,0	-,0	s,0	t,0
th	the,3	h,0	-,0	s,0	t,0
the	-,0	h,0	ther,0	**s,0**	**t,0**
ther	**ther**			**he**	