FPGA-BASED ARCHITECTURE FOR PATTERN MATCHING USING CUCKOO HASHING IN NETWORK INTRUSION DETECTION SYSTEM

TRAN NGOC THINH

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENT FOR THE DEGREE OF
DOCTOR OF ENGINEERING IN ELECTRICAL ENGINEERING
FACULTY OF ENGINEERING
KING MONGKUT'S INSTITUTE OF TECHNOLOGY LADKRABANG
2009

KMITL-2009-EN-D-018-024

หัวข้อวิทยานิพนธ์	สถาปัตยกรรมตัวตรวจสอบความเหมือนของรูปประโยค โดยใช้คุกคูแฮชซิ่งสำหรับการโจมตีทางเครือข่าย
นักศึกษา	ทัน งิ้วบทิน
รหัสประจำตัว	49060028
ระดับการศึกษา	วิศวกรรมศาสตรดุษฎีบัณฑิต
สาขาวิชา	วิศวกรรมไฟฟ้า
พ.ศ.	2552
อาจารย์ที่ปรึกษาวิทยานิพนธ์	ผศ.ดร. สุรินทร์ กิตติธรกุล

บทคัดย่อ

การตรวจสอบความเหมือนของรูปประโยค การโจมตีสำหรับประยุกต์ใช้ในการตรวจจับและป้องกันการบุกรุกทางเครือข่ายจำเป็นต้องมีทรูพุทสูงอย่างยิ่ง จึงต้องมีการเพิ่มเติมรูปประโยคของการโจมตีใหม่ๆ เป็นระยะๆ ด้วยคุณลักษณะแบบขนานหรือไพพ์ไลน์ของฮาร์ดแวร์ ระบบการตรวจจับ การบุกรุกที่ใช้ฮาร์ดแวร์จึงมีความสามารถเหนือกว่าระบบที่เป็นซอฟต์แวร์ วิทยานิพนธ์ฉบับนี้จึงนำเสนอระบบการตรวจสอบความเหมือนของรูปประโยคการโจมตีโดยใช้ฮาร์ดแวร์แบบรีคอนฟิกกูเรเบิลจำนวนสองสถาปัตยกรรม โดยชนิดแรกใช้สถาปัตยกรรมอาเรย์ของตัวประมวลผล และชนิดที่สองใช้อัลกอริธึมการแฮชชื่อ “คุกคู”

วิทยานิพนธ์นี้นำเสนอการวิเคราะห์กฎการบุกรุกต่างๆ ของสนอร์ทในการสร้างตัวตรวจสอบความเหมือนชนิดแรก ด้วยสถาปัตยกรรมอาเรย์ของตัวประมวลผลจำนวนมาก โดยสามารถทำงานด้วยทรูพุทสูงสุดถึง 12.58 จิกะบิทต่อวินาที และด้วยวิธีการเข้ารหัสอย่างย่อเพื่อเป็นการประหยัดพื้นที่หน่วยความจำในการเก็บกฎต่างๆ โดยสามารถลดพื้นที่ลงได้ถึง 50% เมื่อเปรียบเทียบกับการ เข้ารหัสแบบแอสกี้

สถาปัตยกรรมที่สองใช้อัลกอริธึมการแฮชชื่อ “คุกคู” โดยมีคุณลักษณะการเพิ่มเติมรูปประโยคการโจมตีในขณะที่ยังทำงานไปพร้อมๆ กัน และตั้งชื่อว่า “พาเมล่า” โดยแบ่งขั้นตอนการพัฒนาออกเป็นสามช่วง คือ หนึ่ง การใช้การแฮชแบบคุกคูและลิงค์ลิสต์สร้างตัวตรวจสอบความเหมือนของรูปประโยคการโจมตีที่ความยาวต่างๆ สอง การเพิ่มหน่วยความจำชนิดสแตกและไฟโฟเพื่อจำกัดเวลาการเพิ่มกฎ และสาม การขยายขีดความสามารถให้ประมวลผลหลายๆ ตัวอักษรพร้อมกัน เพื่อให้ได้ทรูพุทสูงสุดถึง 8.8 จิกะบิทต่อวินาที โดยใช้ปริมาณฮาร์ดแวร์อย่างคุ้มค่ากว่าระบบอื่นๆ ที่ใช้เอฟพีจีเอของ Xilinx เช่นกัน

Thesis Title	FPGA-based Architecture for Pattern Matching using Cuckoo Hashing in Network Intrusion Detection System
Student	Mr. Tran Ngoc Thinh
Student ID	49060028
Degree	Doctor of Engineering
Program	Electrical Engineering
Year	2009
Thesis Advisor	Asst. Prof. Dr. Surin Kittitornkun

ABSTRACT

Pattern matching for network intrusion/prevention detection requires extremely high throughput with frequent updates to support new attack patterns. With naturally parallel/pipelined characteristic, current hardware implementations have outstanding performance over software implementations. In this dissertation, we propose two reconfigurable hardware engines using processor array architecture and a recently proposed hashing algorithm called Cuckoo Hashing.

In the first proposed engine, the rule set of a Network Intrusion Detection System, SNORT, is deeply analyzed. Compact encoding method is proposed to decrease the memory space for storing the payload content patterns of entire rules. This method can approximately decrease up to 50% of area cost when compared with traditional ASCII coding method. After that, a reconfigurable hardware sub-system for Snort payload matching using systolic design technique is implemented. The architecture is optimized with sharing of substrings among similar patterns and compact encoding tables. As a result, the system is a processor array architecture that can match patterns with the highest throughput up to 12.58 Gbps and area efficient manner.

The second architecture features on-the-fly pattern updates without reconfiguration, more efficient hardware utilization. The engine is named *Pattern Matching Engine with Limited-time updAte* (PAMELA). First, we implement the parallel/pipelined exact pattern matching with arbitrary length based on Cuckoo Hashing and linked-list technique.

Second, while PAMELA is being updated with new attack patterns, both stack and FIFO are incorporated to bound insertion time due to the drawback of Cuckoo Hashing and to avoid interruption of input data stream. Third, we extend the system for multi-character processing to achieve higher throughput. Our engine can accommodate the latest Snort rule-set and achieve the throughput up to 8.8 Gigabit per second while consuming the lowest amount of hardware. Compared to other approaches, PAMELA is far more efficient than any other implemented on Xilinx FPGA architectures.

Acknowledgements

First of all, I would like to deeply thank Assistant Professor Dr. Surin Kittitornkun of King Mongkut's Institute of Technology Ladkrabang, my Advisor, and Professor Dr. Shigenori Tomiyama of Tokai University, Japan, my Co-Advisor, for their helpful suggestions and constant supports during the research work of this dissertation at King Mongkut's Institute of Technology Ladkrabang and Tokai University.

I am also thankful to my dissertation committee members in the Department of Computer Engineering, Faculty of Engineering, King Mongkut's Institute of Technology Ladkrabang, for their insightful comments and helpful discussions which give me a better perspective of this dissertation.

I should also mention that my Ph.D study in King Mongkut's Institute of Technology Ladkrabang and Tokai University is entirely supported by the AUN-SeedNet Program of JICA.

Finally, I would like to acknowledge the supports of all of my beloved family and friends for all of their helps and encouragements.

Bangkok, Thailand

April, 2009

Tran Ngoc Thinh

Contents

List of Tables

Table Page

List of Figures

Chapter 1

Introduction

1.1 Motivation

Nowadays, illegal intrusions are the most serious threats to network security due to its growing frequency and complexity [88]. An intrusion is unauthorized system or network activity on one of computers or networks. According to the CERT Coordination Center (CERT/CC) [85], the number of intrusions is almost double every year. In 2003, fifty percent of companies and government agencies surveyed detected security incidents. Intrusions also cause large amounts of financial loss. It is difficult to exactly estimate the damages caused by illegal intrusion such as viruses and worms. The damages may include destruction of data, clogging of network links, and future breaches in security. In addition, the threats of intrusion have increased due to the availability of more hacking tools, which decrease the technical skills required to launch an attack while the sophistication of those attacks has risen over the same time. This trend is expected to continue. All these facts lead to a need for better network security solution.

Traditionally, networks have been protected using firewalls that provide the basic functionality of monitoring and filtering at the header level. Firewall users can then define rules of the combinations of packet headers that are allowed to pass through. Firewalls are primarily designed to deny or allow traffic to access the network, not to alert administrators of malevolent activity. Therefore, not all incoming malicious traffic can be blocked and legitimate users can still abuse their rights. A CSI/FBI security report states that most of attacks bypass firewalls [89].

Network Intrusion Detection Systems (NIDSs) go one step further by deep packet filtering for attack signatures. They watch the packets traversing the network and decide whether anything is suspicious. The NIDS differs from a firewall in that it goes beyond the header, actually searching the packet contents for various patterns that imply an attack is taking place, or that some disallowed content is being transferred across the network. In general, an NIDS searches for a match from a set of rules that have been

designed by a system administrator. These rules include information about the IP and TCP header required, and, often, a pattern that must be located in the stream. The patterns are some invariant section of the attack; this could be a decryption routine within an otherwise encrypted worm or a path to a script on a web server.

Currently, the majority of NIDSs are software applications running on a general purpose processor and standard Microsoft Windows or Linux operating systems. These platforms provide sufficient power to capture and process data packets at speeds of only a few hundreds of Mbps. Consequently, most NIDSs today are running offline, analyzing the traffic after it's allowed into the network. Alerts are sent to the security engineer identify attacks after they already happened. For real-time protection, the NIDS should inspect at the line rate of its data connection. The performance is dependent on the ability to match every incoming byte against thousands of pattern characters at line rates. So, pattern matching can be considered as one of the most computationally intensive parts of a NIDS.

To increase the throughput of pattern matching in NIDS, people tend to implement matching algorithms on hardware such as Application Specific Integrated Circuit (*ASIC*) and Field Programmable Gate Arrays (FPGA). Pattern matching can get high throughput on hardware because it can exploit parallel and pipelining capability. ASIC is so complex and expensive that it is only suitable for high volume products. FPGA is a low cost device so it is well suited for many applications. Moreover, one of the powerful characteristics of SRAM-based FPGA (Xilinx [91] or Altera [92]) in comparison with ASIC is its flexibility, i.e. the system can be easily updated or reconfigured at run time. With these advantages of FPGA, we can apply it for pattern matching in deep packet detection of NIDS.

1.2 Existing Approaches

On the hardware side, there have been a lot of ASIC and FPGA implementations of the pattern matching to accelerate the throughput.

For a line speed of gigabit network, a variety of FPGA approaches of NIDS/NIPS have been proposed, for example: shift-and-compare; state machine such as

Nondeterministic/Deterministic finite automata (NFA/DFA), Aho-Corasick algorithm; and finally hashing.

Firstly, some shift-and-compare methods are [2] and [3]. They apply parallel comparators and deep pipelining on different, partially overlapping, positions in the incoming packet. The simplicity of the parallel architecture can achieve high throughput when compared to software approaches. The drawback of these methods is the high area cost. To decrease the area cost and achieve a high clock rate, many improvements are proposed. The work [5] is extended from [2] to share common substrings. Predecoded shift-and-compare architectures ([7], [17]) convert the incoming characters to bit lines to decrease the size of comparators. A variation in tree-based optimization ([8], [9]) divides the pattern set into partitions to share similar characters resulting in excellent area performance.

The next approach exploits state machines (NFAs/DFAs) [4], [18]. The state machines can be implemented on hardware platform to work all together in parallel. By allowing multiple active states, NFA is used in [18] to convert all the Snort static patterns into a single regular expression. Moscola et al. [4] recognized that most of minimal DFAs content fewer states than NFAs, so their modules can automatically generate the DFAs to search for regular expressions. Like the shift-and-compare implementations, the predecoded method is also used in [6] to improve area performance of NFAs. The main advantage of regular expression format as compared with static pattern one is that a single regular expression can describe a set of static patterns by using meta-characters with special meaning. As a result, recently, a special format of regular expressions such as Perl Compatible Regular Expressions (PCRE) is added in Snort [1] instead of static patterns, and some new works tried to improve on PCRE matching [19], [20]. However, most of these systems suffer scalability problems, i.e. too many states consume too many hardware resources and long reconfiguration time.

Another approach of the state machine method used for static pattern matching is the Aho-Corasick algorithm [21]. By modifying this algorithm on hardware, the implementations in [22]–[24] can get high performance. Aldwairi et al. [22] partitioned the rule set into small ones according to the type of attacks in Snort database. The state machine in [23] is split into smaller FSMs which can run in parallel to improve memory

requirements. This bit-split FSM can fit over 12k characters of Snort rule set to 3.2 Mbits memory at 10 Gbps on ASIC implementation and can update new rules in the order of seconds with no interruption. Nonetheless, its FPGA implementation [24] can achieve lower throughput rate while using larger memory.

Finally, hashing approaches [10]–[14] can find a candidate pattern at constant look up time. The authors in [11], [14] use perfect hashing for pattern matching. Although their system memory usage is of high density, the systems require hardware reconfiguration for updates. Papadopoulos et al. proposed a system named HashMem [12] system using simple CRC polynomials hashing implemented with XOR gates that can use efficient area resources than before. For the improvement of memory density and logic gate count, they implemented V-HashMem [13]. Moreover, VHashMem is extended to support the packet header matching. However, these systems have some drawbacks: 1) To avoid collision, CRC hash functions must be chosen very carefully depending on specific pattern groups; 2) Since the pattern set is dependent, probability of redesigning system and reprogramming the FPGA is very high when the patterns are being updated; 3) By using glue logic gates for simplicity, the long patterns processing is also ineffective for updating.

On the other hand, Dharmapurikar et al. propose to use Bloom Filters to do the deep packet inspection [25]. Unlike other hashing approaches mentioned above, the pattern update process can be done easily without reprogramming FPGA. A Bloom Filter with multiple hash functions up to 35 probes is used to check whether or not a pattern is member of the set. Nevertheless, its main problem is due to false positive matches, which requires extra cost of hardware to confirm the match.

1.3 Statement of Problem

Pattern matching is the most computationally intensive part of high speed deep packet filtering systems (31% of the total execution time [86]) because of the following factors. First, the size of pattern set is large and requires one packet matched against thousands of attack signatures. Second, we often do not know the location of signatures in the packet payload. Hence, we need to check every byte of the packet payload.

Moreover, the normal processing speed of Internet core is over 10 Gbps. Therefore, pattern matching must be performed at Gigabit rates to build a practical in-network worm detection system. However, NIDS systems running in general purpose processor, for example SNORT, cannot support a throughput higher than a few hundreds of Mbps [87]. These rates are not sufficient to meet the needs of even medium-speed access or edge networks.

With a powerful reconfigurable architecture, current state-of-the-art FPGAs offers tremendous opportunity to implement pattern matching at high throughput. The performance of existing FPGA-based approaches can satisfy current gigabit networks. However, the drawback of hardware-based systems is the flexibility. Although reconfiguration is one of the advantages of SRAM-based FPGA; this process for adding or subtracting a few rules can take several minutes to several hours to complete. Today, such latency may not be acceptable for high speed real-time networks when the update process can be vital requirement for deployed NIDS systems. Another requirement is low area cost. The large pattern set continues growing very fast, almost doubling every two years. So the consuming hardware resource is as small as possible to fit the whole pattern set and update the system later.

To achieve these goals, it calls for effective and efficient design methodologies that can achieve high throughput, low area cost and rapid pattern set update.

1.4 Contributions

This dissertation exploits the use of reconfigurable hardware to achieve high throughput of pattern matching. We proposed two FPGA-based architectures for pattern matching in NIDS/NIPS. The first implementation using processor array can achieve extremely high throughput due to the simplicity in architecture. The system only exploits the efficient use of logic gates of FPGA. The second one applies recently developed algorithm named Cuckoo Hashing. It can both achieve high-throughput with rapid pattern set updating and balance the two area metrics: logic gates and memory blocks.

In the first architecture, in order to decrease the area cost, we analyze and preprocess an entire SNORT rule set before storing and matching it in hardware. By applying the compact encoding method, we separate the patterns to smaller groups

that can be encoded 3-5 bits instead of 8 bits as traditional ASCII code. We then use a simple processor array architecture to search on these groups [74]. The system is deeply improved with sharing of substrings among similar patterns and compact encoding tables so that the area cost can save up to 65%. With simple hardware architecture, our implementations achieve the highest throughput, up to 12.58 Gbps, as compared with any previous implementations that process with the same number of characters per clock cycle.

Based on a novel Cuckoo Hashing [15], we implement the second parallel architecture of variable-length pattern matching best suited for FPGA named *PAttern Matching Engine with Limited-time updAte* (PAMELA) [75]. Patterns can be easily added to or removed out of the Cuckoo hash tables. Unlike most previous FPGA-based systems, the proposed architecture can update the static pattern set on-the-fly without interruption of incoming data. This system reaches not only high flexibility but also best performance. In general, our contributions include:

- *Parallel Cuckoo Hashing for short and long patterns* [16, 90] : With our best knowledge, PAMELA is the first application of Cuckoo Hashing for pattern matching in NIDS/NIPS. With the parallel lookup, our improved system is more efficient in terms of performance as applied on hardware. First, we apply Cuckoo Hashing for parallel exact pattern matching up to 16 characters long. Then, we extend parallel hash engine for patterns at different lengths by using a simple but efficient linked-list technique. It enables the engine to accommodate the entire current Snort pattern set with over 68k of characters.
- *Rapid bounded time and dynamic update:* based on our theoretical analysis and simulation results, the insertion time of a new pattern is about 19-43 clock cycles in average. The deletion time of a pattern only depend on the length of patterns. To bound the insertion time and prevent the interruption of the incoming data, both small stack and FIFO are utilized as updating pattern set. We can prove that the insertion time of a new pattern is limited to 17 microseconds at 200 MHz clock frequency. As a result, a new rule set can be updated to PAMELA on line and on the fly.

- *Massively parallel processing*. The engine can simultaneously process multiple characters per clock cycle to gain higher throughput. With the power of massively parallel processing, the speedup of UPM is up to 128*X* as compared with serial Cuckoo implementation. Our engine can simply reach very high clock rate of any Xilinx FPGA architectures. This feature can result in the throughput up to 8.8 Gbps for 4-character processing.
- *The best in performance:* we optimize both kinds of FPGA resources which are logic cell and block RAM memory. PAMELA can save 30% of area compared with the best system. As a result, performance per area is far more efficient than any other FPGA systems.

1.5 Organization

This dissertation is organized in the following manner. Chapter 2 presents the background for the work presented in this dissertation. The background discusses related researches and concepts that contributed to this dissertation. The first architecture using systolic processor array for pattern matching of deep packet filtering in Network Intrusion Detection System implementation on FPGA is presented in Chapter 3. As one of the major contributions, Chapter 4 describes the second architecture of FPGA implementation, called PAMELA: PAttern Matching Engine with Limited-time UpdAte for NIDS/NIPS, using Cuckoo Hashing. Finally, Chapter 5 concludes this research work and provides some important issues for future work.

Chapter 2

Background and Related Approaches

In this chapter, the notion of Network Intrusion Detection System is first introduced. Then we discuss the existing pattern matching of NIDS software running in general purpose processors and show that none of them can operate at gigabit rates, given thousands of complex signatures. Consequently, we present some emerging hardware technologies that can boost the pattern matching to current network rates. Besides, the theory for our architecture in chapter 4, Cuckoo Hashing, is also reviewed.

2.1 Network Intrusion Detection Systems (NIDS)

In recent years, Network Intrusion Detection/ Prevention Systems (NIDSs/NIPSs) are more and more necessary for network security. Normally, traditional firewalls only examine packet headers to determine whether to block or pass the packets. Due to busy network traffic and smart attacking schemes, firewalls are not as effective as they used to be. NIDSs/NIPSs are designed to examine not only the headers but also the payload of the packets to match and identify intrusions. Intrusion detection systems can run in one of several modes: intrusion detection or inline NIDS. In intrusion detection mode, the NIDSs monitor the traffic offline and draw the attention of network administrator to suspicious activities by sending alerts. An inline intrusion detection system or Intrusion Prevention System (IPS) actively filters exploits from traffic in real-time. It can forge resets, drop packets, or modify the packets in transit to defeat an attack. IPSs have to be extremely fast and reliable to process packets in real-time and should be completely transparent, so there is no need to change the network configuration.

The NIDSs can be further segmented into one of two techniques: anomaly detection or misuse detection (signature based). Anomaly detection is based on searching for discrepancies from the models of normal behavior. These models are obtained by performing a statistical analysis on the history of system calls [66,67] or by using rule

based approaches to specify behavior patterns [68,69]. Signature based detection is based on searching packets for attack signatures. It is much faster than anomaly detection, but can detect only those attacks that already have signatures. On the other hand, anomaly detection have the advantage of being able to detect previously unknown attacks, however it suffers from a large number of false positives.

There are many signature based NIDSs that require deep packet inspections such as SNORT [1], Bro [70]. These systems are all open source systems, which allow us to perform a detailed analysis and show their abilities and constraints. Most modern NIDS/NIPSs apply a set of rules that lead to a decision regarding whether an activity is suspicious.

They have well over a thousand rules. As the number of known attacks grows, the patterns for these attacks are made into signatures (pattern set). The simple rule structure allows flexibility and convenience in configuring NIDS. However, checking thousands of patterns to see whether it matches becomes a computationally intensive task as the highest network speed increases to several gigabits per second (Gbps). Current high-performance systems can barely process that many rules on a 100 Mbps moderately loaded network [71]. To handle fully loaded gigabit networks, an NIDS must either drop some of the rules or drop some of the packets it analyzes. Neither solution is desirable since they both compromise security.

2.1.1 Snort NIDS

Snort is an open source NIDS that uses a portable library called libcap. Libcap allows the program to examine the network packet for its length, content, and header. Snort can perform traffic analysis, IP packet logging, protocol analysis, and payload content search. Furthermore, Snort can be configured to detect a variety of abnormal packet behaviors, such as buffer overflows, stealth port scans, CGI attacks, SMB probes, and OS fingerprinting attempts.

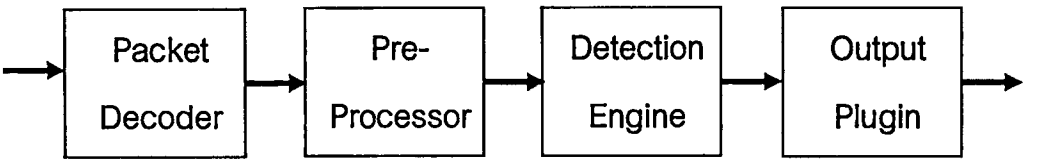


Figure 2.1: SNORT architecture

Figure 2-1 illustrates the Snort architecture. It consists of the following components. When a network packet goes into the system, it is passed to the decoder component. Here the link level information, such as the Ethernet packet header, is removed. Then, the packet enters the pre-processor block, which performs a couple of functions such as packet defragmentation and reassembles the TCP stream, manipulate or examine packets prior to forwarding them to the detection engine. Finally and most importantly, the detection engine performs tests on the packet data forwarded by the preprocessors, using the Snort rules and signatures as a baseline. If suspicious activity is identified by the detection engine, output plug-ins are called to generate administrative alerts, e.g., "drop this packet", or "log this packet".

Deep Packet Inspection Rules

SNORT contains thousands of rules, each containing attack signatures. The structure of a rule consists of two components: a rule header and a rule option. Each rule file can contain more than one rule signature with the form as shown below.

Action Protocol SrcIPAddr/Port Direction DstIPAddr/Port — Rule Header

Options — Rule Option

The rule header is a classification rule that applies to the packet header. This rule header consists of five fixed fields: protocol, source IP, source port, destination IP, and destination port.

Figure 2-2 gives an example SNORT rule that detects a MS-SQL worm probe. Here, the rule header specifies that this rule applied to User Datagram Protocol (UDP) packet from external network to a computer in the protected network with port 1434.

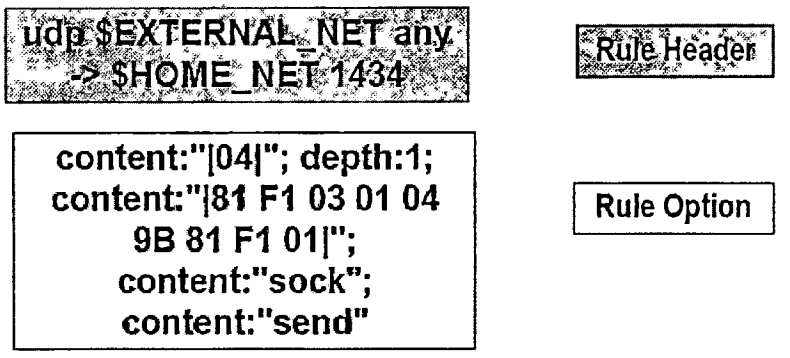


Figure 2.2: SNORT rule example

The rule option is more complicated: it specifies which intrusion patterns are to be used to scan packet payload. The most computationally intensive option is called *'content.'* This option is the key to better packet filtering in deep packet inspection firewall. There are two main types of patterns: static string patterns and full regular expression patterns. For example, the MS-SQL worm detection rule in Fig. 2-2 requires a sequential matching of a correlated pattern that is called static string patterns. Recently, SNORT also incorporates a large number of regular expression patterns. For example, the pattern for detecting Internet Message Access Protocol (IMAP) email server buffer overflow attacks is ".*AUTH\s[^\n]{100}". This signature detects the case where there are 100 non-return characters "[^\n]" after matching of keyword AUTH\s. Matching of these signatures is the core component of the SNORT system.

When the signature is loaded on to Snort, the system will 'alert' the administrator if the packet under examination has matching protocol, IP addresses, ports, and other packet characteristics describe within the parenthesis. Above rule will cause the system to specifically search for the pattern ".ida?" in all the payloads of the packets that match the header specifications in the rule signature.

2.1.2 Pattern Matching in Software NIDS Solutions

At the core of every intrusion detection system is a pattern matching algorithm. From a stream of packets, the algorithm identifies those packets that contain data matching the signatures of a known attack. The intrusion detection system then takes action that could vary from alerting the system administrator to dropping the packet in

the case of inline NIDS. The problem of pattern matching is well researched, many algorithms exist and they can be classified into either single pattern string matching or multiple pattern string matching. In single pattern string matching the packet is searched for a single pattern at a time. On the other hand, in multiple pattern string matching the algorithm searches the packet for the set of patterns all at once.

Several string pattern matching algorithms have been recently proposed in NIDS especially for SNORT's open source NIDS. First versions of SNORT used brute force pattern matching, which was very slow, making clear that using a more efficient string matching algorithm, would improve performance. The first implementations that improved SNORT used the parallel Boyer-Moore algorithm [50] for fast matching of multiple strings. This implementation improved SNORT performance 200-500% [1]. The Boyer-Moore algorithm is one of the most well-known algorithms that use two heuristics to decrease the number of comparisons. It first aligns the pattern and the incoming data (text), the comparison begins from the right-most character, and in case of mismatch the text is properly shifted. The search time for an m byte pattern in an n byte packet is $O(n+m)$. If there are k patterns, the search time is $O(k(n+m))$, which grows linearly in k. Hence, this method is slow when there are thousands of patterns. The parallel Boyer-Moore algorithm used in SNORT can potentially decrease the running time to sub-linear time in k for certain packets. However, this performance is not guaranteed, and for some packets it requires super-linear in k time to perform matching.

Fisk et al. [52] introduced Set-wise Boyer Moore-Hospool algorithm, which is an adaptation of Boyer-Moore, and is shown to be faster for matching less than 100 patterns. It extends Boyer-Moore to match multiple patterns 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 pattern sets.

On the other hand, Aho-Corasick (AC) [55] is a multiple pattern string matching algorithm, meaning it matches the input against multiple patterns at the same time. Multiple pattern string matching algorithms generally preprocess the set of patterns, and then search all of them together over the packet content. AC is more suitable for hardware implementation because it has a deterministic execution time per packet. Tuck et al. [56] examined the worst-case performance of pattern matching algorithms

suitable for hardware implementation. They showed that AC has higher throughput than the other multiple pattern matching algorithms and is able to match patterns 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 pattern matching for NIDS. Aho-Corasick works by building a tree based state machine from the set of patterns 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. Figure 2.1 shows a state machine constructed from the following patterns {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 patterns.

Recently, new pattern matching algorithms are proposed to boost the pattern matching speed of SNORT. For example, the Aho-Corasick-Boyer-Moore (AC_BM) algorithm proposed by Silicon Defense [57] combines the Boyer-Moore and Aho-Corasick algorithms. Another algorithm named Wu-Mander multi-pattern matching (MWM) algorithm [54]. The MWM algorithm improves the Boyer-Moore algorithm by performing a hash on 2-character prefix of the input data, to index into a group of patterns. The MWM algorithm is the default engine of the Snort when the search-set size exceeds 10 [53]. When the Snort uses the MWM algorithm, the matching speed becomes much faster than when using the AC and other Boyer-Moore like algorithms.

Finally, Markatos et al. proposed E^2xB algorithm, which provides quick negatives when the search pattern does not exist in the incoming data [58-60]. Compared to Fisk et al., E^2xB is faster, while for large incoming packets and less than 1k-2k rules it outperforms MWM [60].

These algorithms greatly improve SNORT's pattern matching speed to a few hundred Mbps at most, e.g., 50Mbps with the Pentium IV, 250Mbps with the SUN SDA [61]. However, it is still below the line rate needed for network deployment.

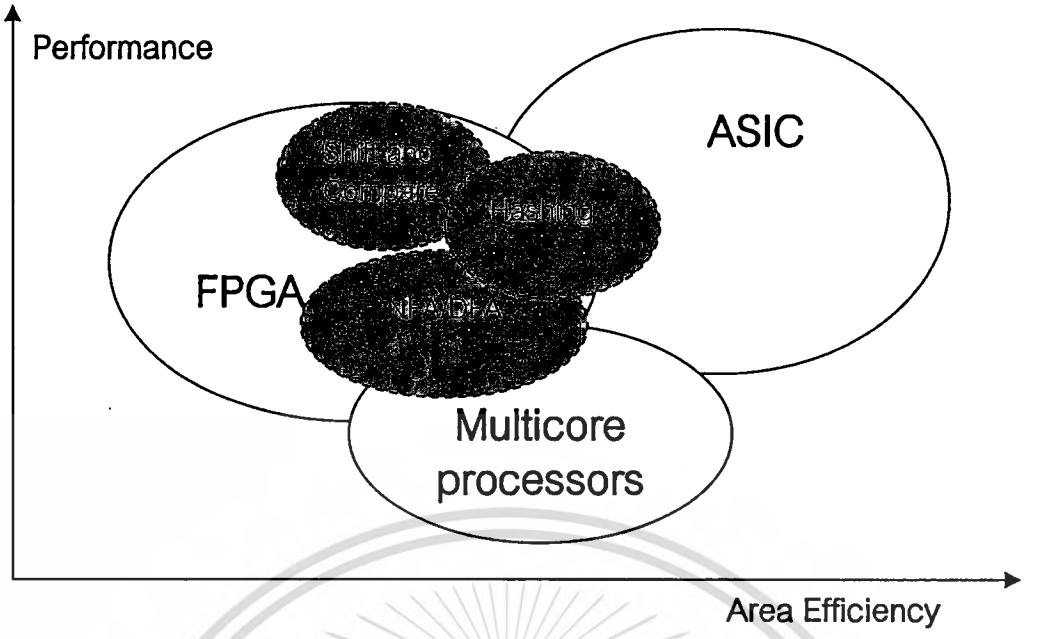


Figure 2.3: Abstract illustration of performance and area efficiency for various hardware pattern matching techniques

2.1.3 Hardware-based Pattern Matching Architectures in NIDS

Given the processing bandwidth limitations of General purpose processors (GPP), which can serve only a few hundred Mbps throughput, Hardware-based NIDS (Multi-core Processors, ASIC or FPGA) as illustrated in Fig. 2.3 is an attractive alternative solution.

ASIC Technique

Many ASIC intrusion detection systems have been commercially developed [62-65]. Such systems usually store their rules using large memory blocks, and examine incoming packets in integrated processing engines.

In academic research, there are several pattern matching solutions designed for ASIC. In order to support pattern set modifications, ASIC designs need to narrow their design alternatives down to only memory-based solutions. Hence, they often exploit the FSMs methods [23, 76-78] which base their functionality on the contents of a memory; for example, the memory may store the required state transitions to match a specific pattern. It could be expected that an ASIC pattern matching approach would be up to an order of magnitude faster than reconfigurable hardware designs, however this does

not hold true. The memory latency severely degrades ASIC pattern matching performance.

Generally, ASIC approaches for pattern matching are expensive, complicated, and although they can support higher throughput compared to GPP, they may offer higher performance than a reconfigurable one, at the cost however of limited flexibility and higher fabrication cost.

Multi-core processors Technique

Recently, multi-core processors' implementations are becoming popular for designing NIDS due to flexibility. Different from the traditional single core processors, multi-core processes combine two or more independent processors into a single package. These independent processors can run in parallel hence can provide higher computation power. Networking equipment vendors are commonly using multi-core processors. The widely used Intel Network Processor Units (NPUs) have 8-16 cores [79-80].

We can easily port software approaches such as SNORT to multi-core environments. Some multi-core processors systems are proposed for NIDS. In [81], Bruijn et al. developed a practical system, SafeCard, capable of reconstructing and scanning TCP streams at Gbps rates while preventing polymorphic buffer-overflow attacks. In [82], NNIDS prototype, a combination between IXP 1200 network processors and Xilinx Virtex FPGAs to build NIDS can keep up with traffic up to 100 Mbps. Recently, Ruler [83], a flexible high-level language for deep packet inspection, provides packet matching and rewriting based on regular expressions. Ruler is implemented on the Intel IXP2xxx NPU with processing rates at less than 1 Gbps.

However, multi-core processors also have some limitations. The limitation in number of processors requires smart algorithms to partition different tasks and patterns into the different cores. For example, Intel IXP2800 network processor [79] has 16 cores, which is much smaller than the number of patterns. The size of on-chip memory is limited. For example, the IBM cell processor has 8 synergistic processor elements, each with 128 KB local memory [80].

FPGA Technique

On the other hand, FPGAs are more suitable, because they are reconfigurable; they provide hardware speed and exploit parallelism. An FPGA-based system can be entirely changed with only the reconfiguration overhead, by just keeping the interface constant. This characteristic of reconfigurable devices allows updating or changing the rule set, adding new features, even changing systems architecture, without any hardware cost.

Next subsections present some main approaches for harware-based systems in academic researches. Most of them are implemented on FPGA platform.

2.1.3.1 CAMs & Shift-and-compare

An easy approach for pattern matching is to use Content Addressable Memories (CAMs) [3, 7-8, 31, 33-35] or shift-and-compare [2, 5, 9, 32, 36, 37, 39]. They apply parallel comparators and deep pipelining on different, partially overlapping, positions in the incoming packet. Current FPGAs give designers the opportunity to use integrated block RAMs for constructing regular CAM. Other researchers preferred to use shift-and-compare, which leads to designs that operate at higher frequency. Shift-and-compare architecture uses one or more comparators for every matching pattern. Generally, this approach uses FPGA logic cells to store each pattern. Every LUT can store a half-byte (4-bit) of a pattern, and the flip-flops that already exist in logic cells can be used to create a pipeline, without any overhead. The simplicity of the parallel architecture can achieve high throughput when compared to software approaches. The drawback of these methods is the high area cost. To decrease the area cost and achieve a high clock rate, many improvements are proposed.

Gokhale, et al. [31] used CAM to implement Snort rules NIDS on a Virtex XCV1000E. They performed both header and payload matching on CAMs, Their hardware runs at 68MHz with 32-bit data every clock cycle, giving a throughput of 2.2 Gbps, and reported an almost 9-fold improvement on a 1 GHz PowerPC G4. Another of CAM implementation [3] uses deep pipeline duplicate comparators, exploits parallelism to increase processing bandwidth, and uses a fast fan-out tree to distribute the incoming data to each comparator. The design implemented in a Virtex2-6000 device runs at 250MHz, achieving 8 Gbps throughput when processing 4 characters per clock

cycle. They require about 19-20 logic cells to match a single character, and therefore can store only a few hundreds patterns in a single FPGA.

Yu et al. proposed the use of Ternary Content Addressable Memory (TCAM) for pattern matching [33]. They break long patterns to fit them into the TCAM width and achieve 1-2 Gbps throughput. Additionally, Bu et al. improved CAM-like structures in [34, 35] achieving 2-3 Gbps and requiring 0.5-1 logic cells per matching character. Finally, a variation in tree-based optimization [8] represents multiple patterns in the form of a Binary Decision Diagram (BDD) and divides the pattern set into partitions to share similar characters resulting in excellent area performance.

Cho et al. [32, 2] designed a deep packet filtering firewall on a FPGA and automatically translated each pattern-matching component into structural VHDL. They presented a block diagram of a complete FPGA-based NIDS, and implemented the shift-and-compare pattern matching unit for more than a hundred signatures. The content match micro-architecture used 4 parallel comparators for every pattern so that the system advances 4 bytes of input packet every clock cycle and finally the results of the four parallel comparators are OR-ed. The design [32] implemented in an Altera EP20K device runs at 90MHz, achieving 2.88 Gbps throughput. They require about 10 logic cells per search pattern character.

To decrease the area cost and achieve a high clock rate, many improvements are proposed. The work [5] is extended from [2] to share common substrings. Pre-decoding is the significant improvement for these approaches. It was applied by Baker et al. [39] and Sourdis et al. [7]. The main idea of this technique is that incoming characters are pre-decoded in a centralized decoder resulting in each unique character being represented by a single wire. The incoming data are decoded, subsequently properly delayed and the shifted, decoded characters are AND-ed to produce the match signal of the pattern. Baker et al. further improved the efficiency of pre-decoding by sharing sub-patterns longer than a single character in [9, 36, 37].

The next improvement is our first pattern matching solution [74], described in Section 3. The architecture is based on systolic like array and compact encoding, showing that a processing throughput of 12 Gbps is feasible for pattern matching designs implemented in FPGA devices.

In summary, CAM and shift-and-compare can achieve high processing throughput exploiting parallelism and pipelining. Their high resource requirements can be tackled by pre-decoding, a technique which shares their character comparators among all pattern matching blocks, or compact encoding, a technique which converts characters 8-bit ASCII code to 3-5 bits. In general, a throughput of 2.5-12 Gbps can be achieved in technologies such as Xilinx Virtex2 and Virtex4.

2.1.3.2 Nondeterministic/Deterministic Finite Automata

An alternative appoach exploits state machines [4], [18], [40]. The state machines can be implemented on hardware platform to work all together in parallel. There are two main options for implementations of state machines. The first one is using Non-deterministic Finite Automata (NFAs), having multiple active states at a single cycle, while the second is Deterministic Finite automata (DFAs) which allow one active state at a time and result in a potentially larger number of states compared to NFAs. State machines produce designs with low cost, but at a modest throughput. Theoretically, DFA can be exponentially larger than NFA, but in practice often DFAs have, as compared to NFAs, a similar number of states.

Sidhu and Prassanna [40] introduced regular expressions and Nondeterministic Finite Automata (NFAs) for finding matches to a given regular expression. Their automata matched one text character per clock cycle. Hutchings et al. [18] expanding on Sidhu et al. work, used regular expressions, with more complex syntax and meta-characters such as "?" and ".", to describe patterns extracted from Snort database. Hutchings et al. were the first that mentioned the performance bottleneck that occurs in such systems due to large fan-out. Their solution was to arrange flip-flops in a fan-out tree. They managed to include up to 16,000 characters requiring 2.5-3.4 logic cells per matching character. The operating frequency of the synthesized modules was about 50 MHz on a Virtex XCV2000E.

Moscola et al. used the Field Programmable Port Extender (FPX) platform, to perform pattern matching for an Internet firewall [4]. They recognized that most of minimal DFAs content fewer states than NFAs, so their modules can automatically generate the DFAs to search for regular expressions. Each regular expression is parsed

and sent through JLex [49] to get a representation of the DFA required to match the expression. Finally, JLex representation is converted to VHDL. The authors finally described a technique to increase processing bandwidth. Incoming packets arrive in 32-bit words, and are dispatched to one of the four content scanners. This implementation can operate at 37 MHz on a Virtex XCV2000E and throughput is 1.184 Gbps.

Like the shift-and-compare implementations, the predecoded method is also used in [38, 6] to improve area performance of NFAs. Pre-decoded regular expressions have similar area cost with pre-decoded CAMs, however, they fall short in terms of performance.

Another approach of the state machine method used for static pattern matching is the Aho-Corasick algorithm [21]. By modifying this algorithm on hardware, the implementations in [22]–[24] can get high performance. Aldwairi et al. [22] partitioned the rule set into small ones according to the type of attacks in Snort database. The state machine in [23, 76, 77] is split into 8 smaller FSMs which can run in parallel to improve memory requirements. This bit-split FSM can fit over 12k characters of Snort rule set to 3.2 Mbits memory at 10 Gbps on ASIC implementation and can update new rules in the order of seconds with no interruption. Nonetheless, its FPGA implementation [24] can achieve lower throughput rate while using larger memory. Furthermore, Brodie et al. proposed a generic FSM design to support DFA matching in ASIC [78] and achieved 16 Gbps in 65 nm technology. These approaches have significant memory requirements and are rather rigid in accommodating patterns with extreme characteristics e.g. patterns that require a larger number of states than the allocated on-chip memory per FSM.

The main advantage of regular expression format as compared with static pattern one is that a single regular expression can describe a set of static patterns by using meta-characters with special meaning. As a result, recently, a special format of regular expressions such as Perl Compatible Regular Expressions (PCRE) [94] is added in Snort instead of static patterns, and some new works tried to improve on PCRE matching [19], [20].

In general, finite automata machines suffer scalability problems. They are complex and hard to implement. Too many states consume too many hardware resources. Every time a new attack is characterized and a signature is added to the database the FA have to be rebuilt again and it requires long reconfiguration time.

2.1.3.3 Hash Functions

The last pattern matching approach that we want to present is hashing. Hashing an incoming data may select only one or a few search patterns out of a set which will possibly match. In most cases the hash function provides an address to retrieve the candidate patterns from a memory, and subsequently, a comparison between the incoming data and the candidate patterns will determine the output.

The important characteristic of a hash function used for pattern matching are collision-free for matching. The guarantee of collision-free hashing makes sure that constant memory access can retrieve the possibly matching pattern and therefore offers a guaranteed throughput. In case a hash function is not collision-free, the maximum number of memory accesses to resolve possible pattern collisions is critical for the performance of the system. The complexity of the generated hash function is also significant since it may determine the overall performance and area requirements of the system. Another important characteristic is the dynamic update capacity of hashing function due to the fast growing of patterns. Finally, to more saving in hardware resources, the placement of the patterns in the memory can change using an indirection memory [11-14, 16, 75, 90].

The match unique prefixes technique of the search patterns on hardware is first proposed by Burkowski [48]. Then, Cho and Mangione-Smith utilized the same technique for intrusion detection pattern matching [5, 10]. They implemented their designs targeting FPGA devices and ASIC. Their memory requirements are similar to the size of the pattern set and the logic overhead in reconfigurable devices is about 0.26 Logic Cells/character (LC/char). The throughput is about 2 Gbps on FPGA and 7 Gbps on ASIC 0.18μm technology. The most significant drawback of the above designs, especially when implemented in ASIC where the hash functions cannot be updated, is that the prefix matching may result in collisions.

Some more efficient hashing algorithms were proposed in [11-14]. The authors in [11], [14] use perfect hashing for pattern matching. Although their system memory usage is of high density, the systems require hardware reconfiguration for updates. Papadopoulos et al. proposed a system named HashMem [12] system using simple CRC polynomials hashing implemented with XOR gates that can use efficient area resources than before. For the improvement of memory density and logic gate count, they implemented V-HashMem [13]. Moreover, VHashMem is extended to support the packet header matching. These designs support 2-3.7 Gbps, and the memory requirements are about 2.5-8x the size of the pattern set. However, these CRC hashing systems have some drawbacks: 1) To avoid collision, CRC hash functions must be chosen very carefully depending on specific pattern groups; 2) Since the pattern set is dependent, probability of redesigning system and reprogramming the FPGA is very high when the patterns are being updated; 3) By using glue logic gates for simplicity, the long patterns processing is also ineffective for updating.

Botwicz et al. proposed another hashing technique using the Karp-Rabin algorithm for the hash generation [42] and a secondary module to resolve collisions [41]. Their designs require memory of 1.5-3.2x the size of the pattern set and their performance is 2-3 Gbps in Altera Stratix2 devices.

On the other hand, the authors in [25, 44-47] propose to use Bloom Filters [43] to determine whether the incoming data can match any of the search NIDS patterns. Unlike other hashing approaches mentioned above, the pattern update process can be done easily without reprogramming FPGA. A Bloom Filter with multiple hash functions up to 35 probes is used to check whether or not a pattern is member of the bit set. In case all the hash functions agree and indicate a "hit" then incoming data may match one of the NIDS search patterns. Nevertheless, its main problem is due to false positive matches. In order to resolve false positives, the authors used a secondary hash module which possibly accesses multiple times an external memory. This decision may degrade the overall pattern matching performance. In case of successive accesses to the external memory, the overall performance is determined by the throughput of the secondary hash module.

In summary, the area cost of the hash functions used above is low requiring only a few gates for their implementation. However, in most cases the dynamic update is poor and therefore needs reconfiguration to resolve collisions or change the pattern set.

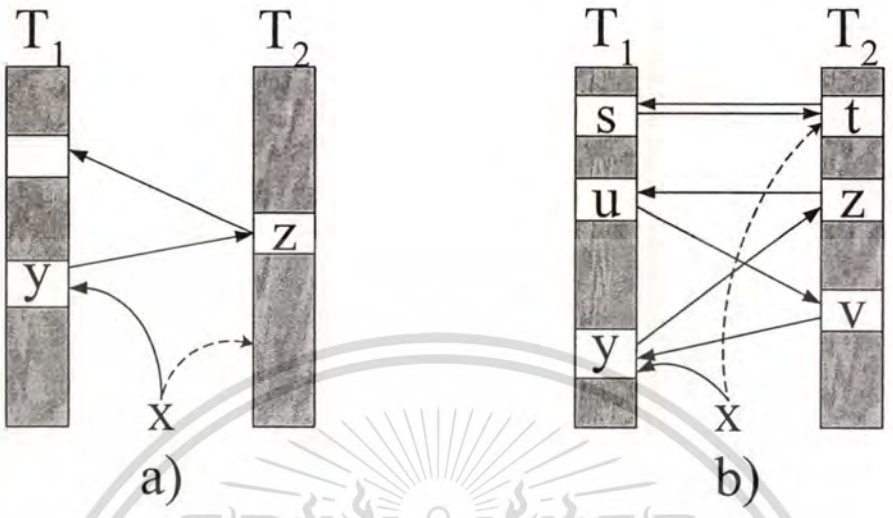


Figure 2.4: Original Cuckoo Hashing [15], a) Key *x* is successfully inserted by moving *y* and *z*, b) Key *x* cannot be accommodated and a rehash is required

2.2 Cuckoo Hashing

Cuckoo hashing is proposed by Pagh and Rodler [15] as an algorithm for maintaining a dynamic dictionary with constant lookup time in the worst case scenario. The algorithm utilizes two tables T_1 and T_2 of size $m = (1+\varepsilon)n$ for some constant $\varepsilon > 0$, where n is the number of elements (strings). Cuckoo hashing guarantees $O(n)$ space and does not need perfect hash functions that is very complicated if the set of elements stored changes dynamically under the insertion and deletion just like Snort rule set. Given two hash functions h_1 and h_2 from U to $[m]$, one maintains the invariant that a key x presently stored in the data structure occupies either cell $T_1[h_1(x)]$ or $T_2[h_2(x)]$ but not both. Given this invariant and the property that h_1 and h_2 may be evaluated in constant time, lookup and deletion procedures run in worst case constant time. In addition, the lookup procedure queries only two memory entries which are independent and can be queried in parallel.

Pagh and Rodler described a simple procedure for inserting a new key x in expected constant time. If cell $T_1[h_1(x)]$ is empty, then x is placed there and the insertion

is complete; if this cell is occupied by a key y which necessarily satisfies $h_1(x) = h_1(y)$, then x is put in cell $T_1[h_1(x)]$ anyway, and y is kicked out. Then, y is put into the cell $T_2[h_2(y)]$ of second table in the same way, which may leave another key z with $h_2(y) = h_2(z)$ nestless. In this case, z is placed in cell $T_1[h_1(z)]$, and continues until the key that is currently nestless can be placed in an empty cell as in Figure 5.1(a). The authors show that if the hash functions are chosen independently from an appropriate universal hash family, then with probability $1-O(1-n)$, the insertion procedure successfully places all n keys with at most $3/log_{1+\varepsilon}m$ evictions for the insertion of any particular key. However, it can be seen that the cuckoo process may not terminate as Figure 5.1(b). As a result, the number of iterations is bounded by a bound $M=3/log_{1+\varepsilon}m$. In this case everything is rehashed by reorganizing the hash table with two new hash functions h_1 and h_2 and newly inserting all keys currently stored in the data structure, recursively using the same insertion procedure for each key. However, rehashing is expensive on hardware platform as show in detail later in chapter 4.

For the improvement of element storage, Fotakis et al. [84] generalize Cuckoo Hashing to *d-ary Cuckoo Hashing*. Their method can decrease the space requirement to $(1+\varepsilon)n$ by allowing fixed number $d = d(\varepsilon) \geq 2$ of hash functions instead of just two. In their analysis, it was assumed that the hash functions are fully random. A depth first search tree algorithm is used in insertion process, but its worst case performance can be polynomial. Therefore, this improvement is not suitable for implementing on hardware due to the hash function easily implemented on hardware are not enough random. Moreover, the insertion process is complex so the hardware implementation and on-the-fly update for new patterns are difficult.

Chapter 3

Processor Array-Based Architectures for Pattern Matching

In this chapter, the pattern set of a Network Intrusion Detection System, SNORT [1], is deeply analyzed and a compact encoding method to decrease the memory space for storing the entire set of suspicious patterns is proposed.

The drawback of hardware-based systems is a large number of resources required to process the pattern set. Therefore, the common factor of efforts is continuous drive for lower and lower cost with the same or better of performance. In order to decrease the area cost, we deeply analyze and preprocess an entire SNORT pattern set before storing and matching it in hardware. By applying the compact encoding method [72], we separate the patterns to smaller groups that can be encoded 3-5 bits instead of 8 bits as traditional ASCII code. This method can approximately decrease up to 50% of area cost compared with traditional ASCII encoding method. After that, we implement a reconfigurable hardware sub-system for Snort payload matching using systolic design technique. Our architecture is highly scalable to process multi-character every clock cycle. With the simple and regular hardware architecture, our implementation is a processor array architecture that can achieve the highest throughput, ranging 3.14-12.58 Gbps. Our throughput per area cost is also far better than logic gate-based systems that are similar to our architecture [7-9].

This chapter is organized as follows. In section 3.1, our design methodology is elaborated. Next, the FPGA implementation and its experimental results are discussed in section 3.2. Finally, the results of system and the comparison with previous systems are discussed in the last section.

3.1 Processor Array-Based Architecture for pattern maching in NIDS

We divide the pattern set into smaller groups whose patterns are composed of similar characters. Then we apply the systolic technique for pattern matching in every group. Systolic processor array is an array of Processing Elements (PEs) which can

compute in pipelining and in parallel fashions. The system decreases the area of hardware and still keeps the high throughput. As a result, the system performance is improved significantly.

An architectural overview of our system is shown in Fig 3.1. The system consists of three parts. Match Processor Array (MPA) is the main part of system that stores compact encoding pattern set used to compare with incoming packets. Compact Encoding Table & Fan-out Tree converts incoming characters from 8 bits to 3-5 bits suitable for MPAs. The third part, Address Calculation Logic, calculates the addresses of the rules that cause matches.

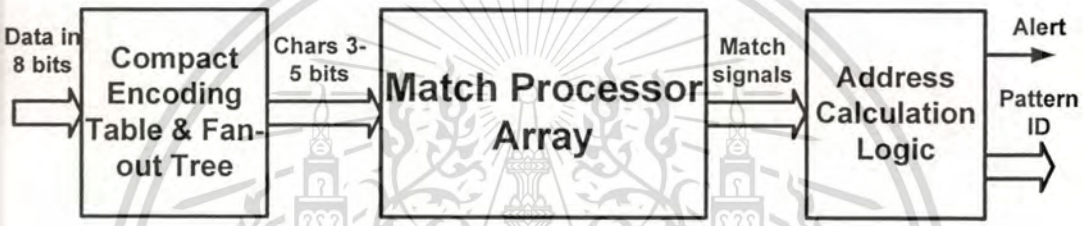


Figure 3.1: Overview of Processor Array-Based Architecture for pattern maching in NIDS

3.1.1 Compact encoding of pattern and text

Most of the previous hardware-based systems represent the pattern set and the incoming text in ASCII code with 8-bit data. Moreover, there are thousands of patterns in SNORT with over 37K of characters and the traditional storage method occupies a lot of logic gates or memory cells. Therefore, we apply a compact encoding method for pattern set of NIDS to save the area of hardware. This method is proposed by S.Kim et al. [72].

For a given pattern P and text T, we firstly count the number of distinct characters in P. Let D be the number of distinct characters in P and E be the smallest integer such that $(2^E - 1) \geq D$. Then we can encode any character in P and T with E bits by assigning distinct E bits for each character in P and assigning distinct E bits for any character that does not occur in P but occurs in T. The following example illustrates this scheme.

Consider a pattern *P* = *"encoding"* and *T* = *"Compact encoding can"*. Since we have 7 distinct characters in *P*, each character can be encoded in 3 bits ($(2^3 - 1) \geq 7$). Let's introduce a function ENCODE for encoding characters: ENCODE(e) = 001, ENCODE(n) = 010, ENCODE(c) = 011, ENCODE(o) = 100, ENCODE(d) = 101, ENCODE(i) = 110, ENCODE(g) = 111, and ENCODE(-) = 000 for any character - that does not occur in *P*. Then, *P* is encoded as 001 010 011 100 101 110 010 111 and *T* as 000 100 000 000 000 011 000 000 001 010 011 100 101 110 010 111 000 011 010.

In June 2006, there are 58,158 characters in 3,462 string patterns of Snorts rule set. However, during the analysis, we found that a lot of the rules look for the same string patterns but with different headers. Through simple preprocessing, we can eliminate duplicate patterns and decrease the number of patterns from 3,462 down to 2,378 unique patterns which contain 37,873 characters.

In the entire pattern set, there are 241 distinct characters, and the compact encoding method is not efficient for large distinct characters. Therefore, we have to divide it to groups with smaller number of distinct characters. Following parts are the analysis of how to group the pattern set.

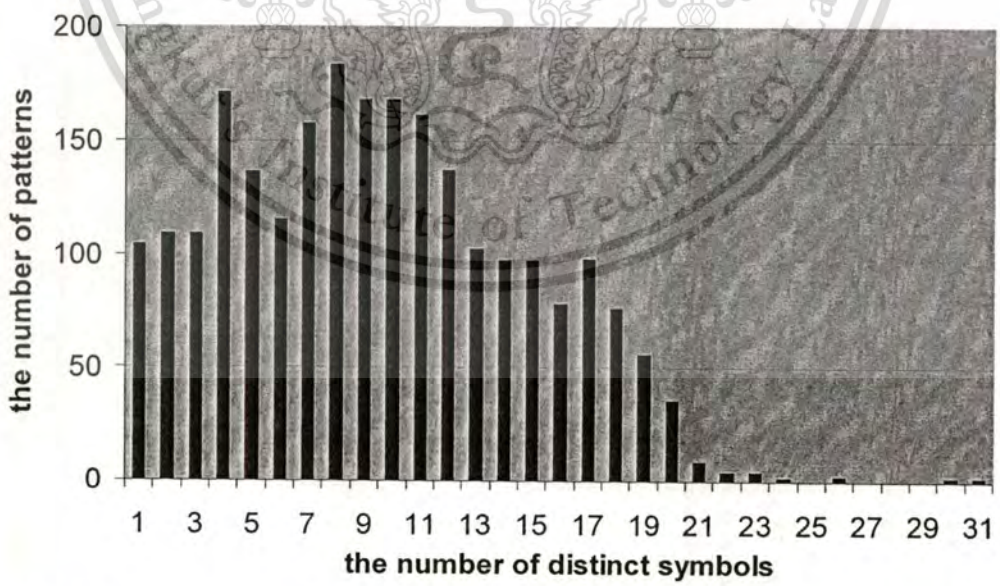


Figure 3.2: Histogram of the number of distinct characters of pattern strings

Figure 3.2 shows a histogram of the number of distinct characters of each unique pattern in the default database. In Snort pattern set, the maximum number of distinct characters in one pattern is 31 and their distribution is from 1 to 31. So we can expect that the encoding functions for each pattern should be less than or equal to 5 bits. By experimental analysis, we know that encoding functions with 3, 4 and 5 bits is the best choice. Fig. 3.3 illustrates a method to separate of patterns into 3-5 bits encoded groups. We separate the pattern set into three clusters C1, C2, and C3 that have upper bounds M1 = 7, M2 = 15, M3 = 31, i.e. C1 includes patterns that have D $\leq$ M1, C2 includes patterns that have M1 < D $\leq$ M2, and C3 includes patterns that have M2 < D $\leq$ M3. Let n_i be the number of patterns content i distinct characters and N_{C1}, N_{C2}, N_{C3} are the number of patterns in 3 clusters, respectively. With totally 2,378 unique patterns, we have the number of patterns in each cluster is

$$N_{C1} = \sum_{i=1}^{7} n_i = 901$$
$$N_{C2} = \sum_{i=8}^{15} n_i = 1113 \qquad (3.1)$$
$$N_{C1} = \sum_{i=16}^{31} n_i = 364$$

Next, we have to separate patterns in these clusters into small groups. Let σ be the alphabet of a group, and $|\sigma|$ be the number of characters in σ such that $|\sigma| \leq$ M1 in cluster C1, M1 < $|\sigma| \leq$ M2 in cluster C2 and M2 < $|\sigma| \leq$ M3 in cluster C3. In every cluster, to add into any group, a pattern P will search for any group such that union of P with group does not exceed upper bound M of its cluster. If the outcome satisfies then pattern P adds into group, otherwise it will create a new group by itself. The long patterns in every cluster will be distributed before the shorter one. This method does not guarantee the smallest number of groups but it is the simple method.

Let G_7, G_{15}, G_{31} be the numbers of groups of C1, C2, C3, respectively. We can achieve

$$G_7 = 43$$
$$G_{15} = 74 \qquad (3.2)$$
$$G_{31} = 32$$

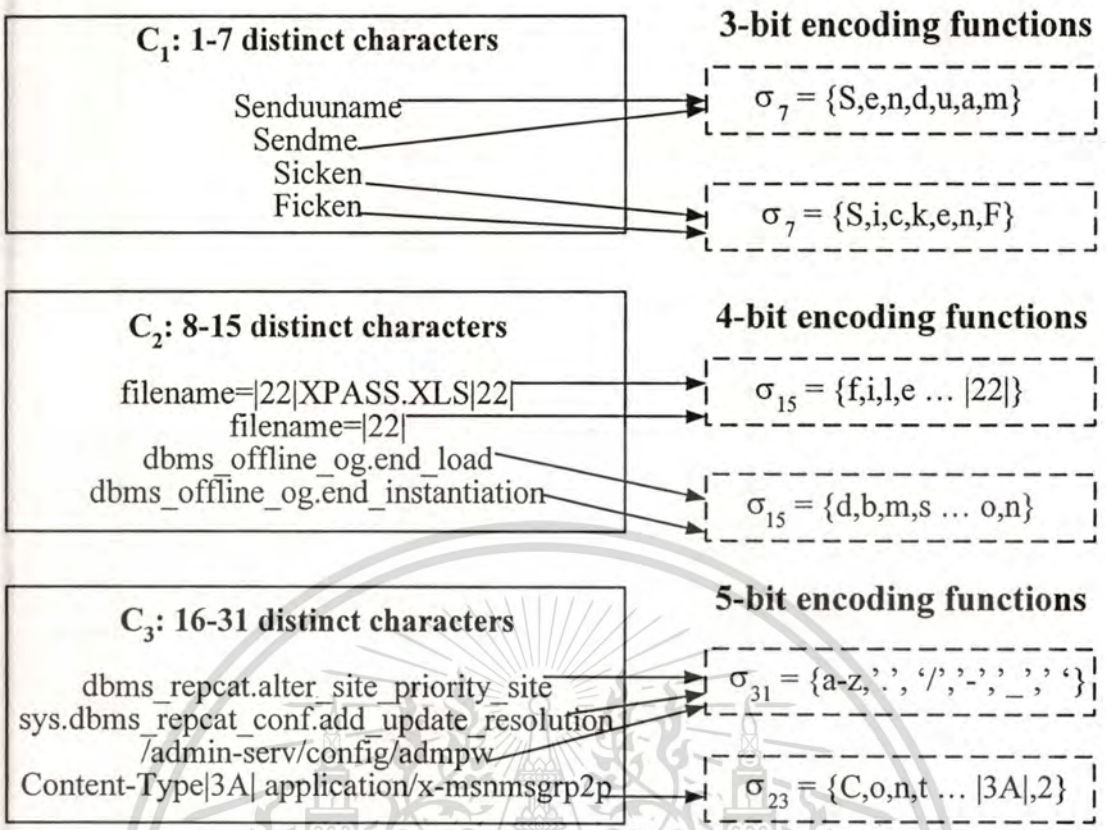


Figure 3.3: Compact Encoding Method for patterns in SNORT

These results can be achieved after merging some too small groups in smaller clusters to other groups in bigger cluster. With totally 149 groups, the number of encoded tables is correlative and the average number of patterns in one group is about 16. These outcomes are suitable for hardware design.

3.1.2 Match Processor Array

In this part, a novel systolic processor array [73] is presented. All of the patterns of one group are arranged in one 2-D array of processing elements (PEs) called Match Processor Array as Fig 3.4. Each PE represents one character in the rule set. In Fig 3.4, when one incoming character enters each group, it is encoded to compact code and then it will be compared against all PEs of MPA at one clock cycle. The match output signal is active only when both following conditions satisfy, the current incoming character matches with the stored character and match input signal is active. Then this match signal is transferred to the next PE in current pattern. When the last PE of the

current pattern has an active match signal, that mean the substring of current string (packet) coincides with the pattern.

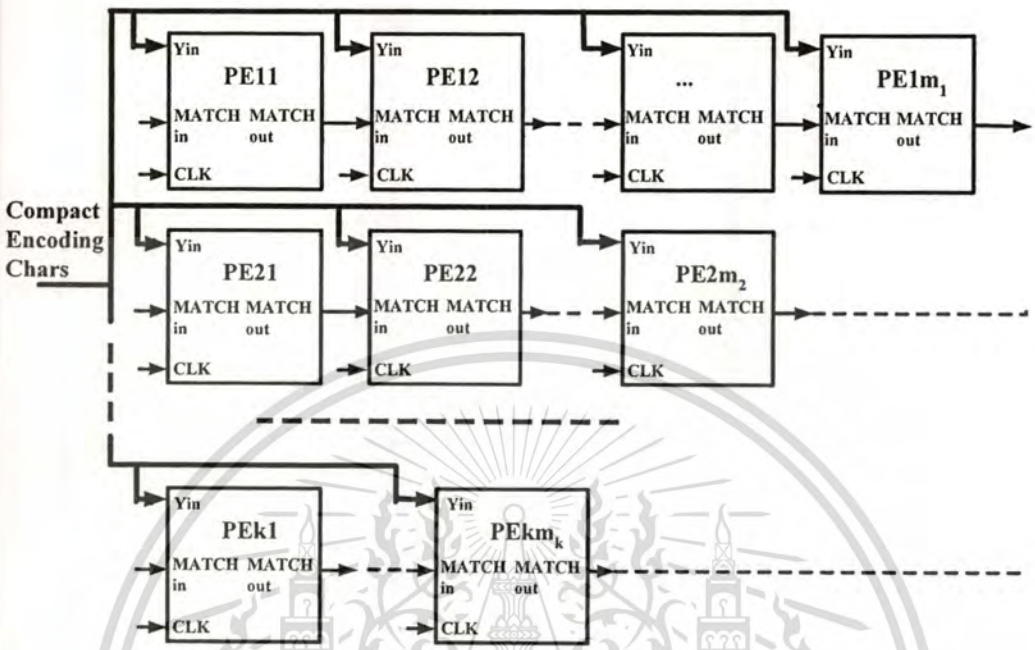


Figure 3.4: Match Processor Array

For example, the input string is AABC and the current pattern includes ABC, the match signal output in last PE of current pattern equals 1 after 4 clock cycles since the first character of input string has entered the first PE of current pattern as Fig. 3.5.

Inside each PE, look-up tables (LUTs) are used to store pattern characters and to compare with incoming characters. As the fundamental element of Xilinx FPGA, a logic cell includes a 4-bit LUT and a flip-flop. An LUT can be programmed as ROM, RAM, SRL (Shift Register LUT) or any maximum 4-input combinational function; and it can be reconfigurable at compile or run time. With this design, a PE includes 1-2 4-input LUTs depend on the number of encoding bits of incoming character as Figure 5. When the encoding character is 3 bits, a PE requires only one LUT4 that is used as a function of ROM 8x1bit and AND gate. However, when the encoding character is 4 bits or 5 bits, a PE requires two LUT4s, one LUT4 is used as a function of ROM 16x1bit and the other is the AND gate or AND gate associated with the one more bit data. With only 1-2 logic cells of Xilinx FPGA chip, we can save up to 50% area of hardware.

	A (PE1)	B (PE2)	C (PE3)
CLK1	A(1)	A(0)	A(0)
CLK2	**A(1)**	A(0)	A(0)
CLK3	B(0)	**B(1)**	B(0)
CLK4	C(0)	C(0)	**C(1)**

Figure 3.5: Example of Match Processor Array

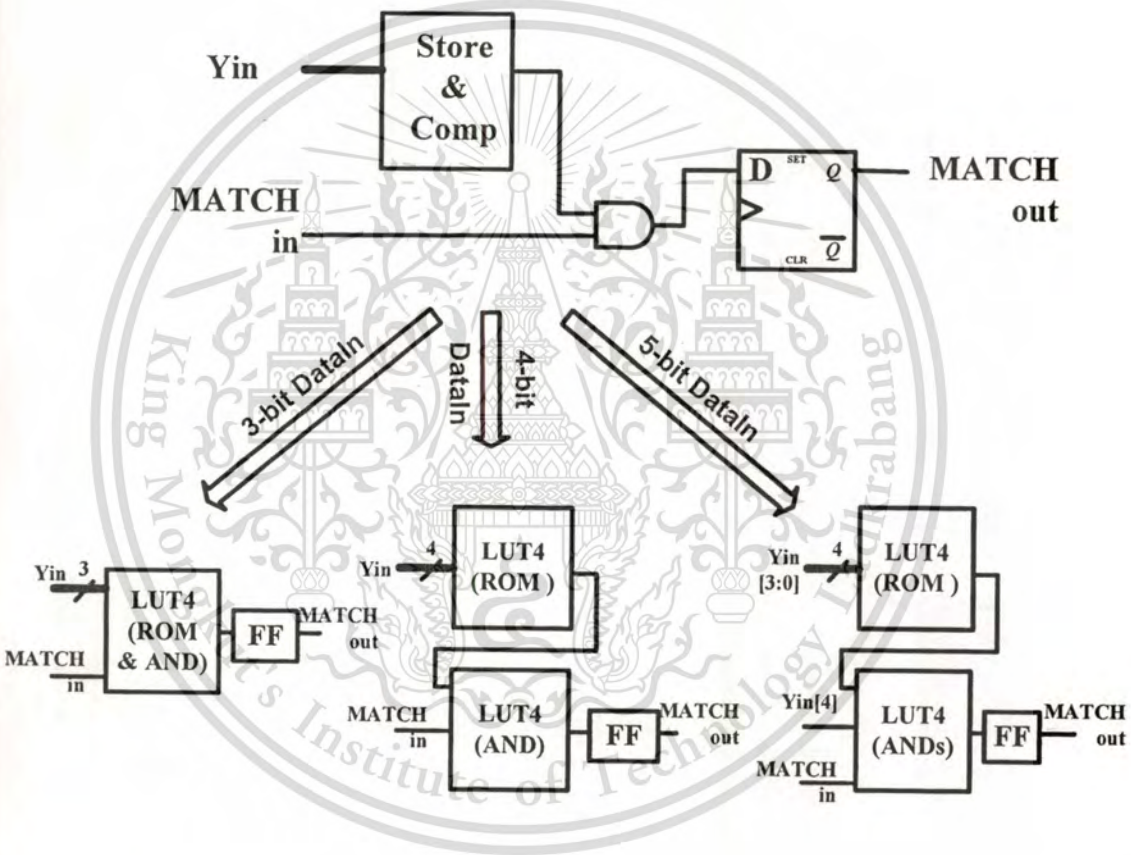


Figure 3.6: MicroArchitecture of a PE in Match Processor Array

The Match Processor Array outputs a "match" signal indicating that a match is found. However, for the system to be useful in the context of NIDS, we also need to know which signature causes the match. The Address Calculation Logic is used to find rules causing a match. In order to generate these addresses, we use the match signals generated by the Match Processor Array.

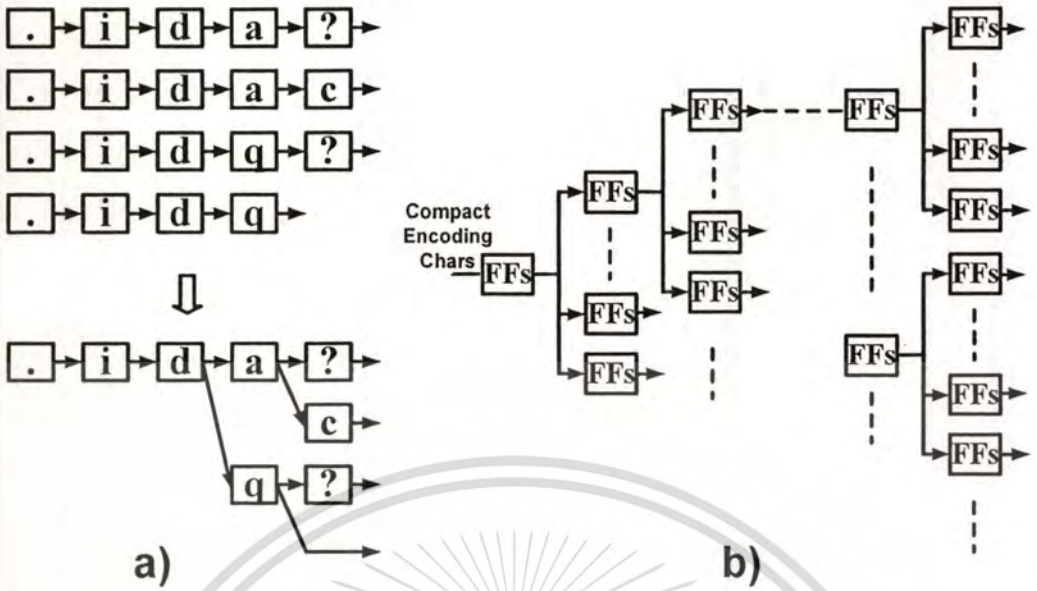


Figure 3.7: a) Example of Sharing of prefixes with 4 patterns ".ida?", ".idac",".idq?" and ".idq" b) Fan-out tree for the MPA

3.1.3 Area and Performance Improvement

The drawback of broadcast data in hardware is caused by the large fan-out from outputs of the encoding table of input strings, which are propagated to all PEs of each group. The implementation of our processor array architecture is optimized to radically decrease long propagation delays due to large fan-out, and achieve a fast speed, by constructing fan-out trees for every group of the MPA as illustrated in Figure 3.7.b. This is done without incurring extra area as we use the flip flops within the logic cells whose LUTs are used for building encoding tables and PEs of MPA.

From experimenting with the smallest to the biggest pattern group, we find the depth of fan-out tree to be 4 with the number of nodes in every level from 1 to 16. The largest fan-out will be 16 that are enough to keep high frequency, and the latency is only four clock cycles more. When we apply this optimization for MPA above, operating frequency is able to achieve approximately the full fabric speed.

One efficient way to save the area for storing the patterns is Aho and Corasicks keyword tree [55]. A keyword tree is used in many software pattern search algorithms,

including the Snort NIDS. This algorithm is already used in some reconfigurable implementations to decrease the logic area [5, 10].

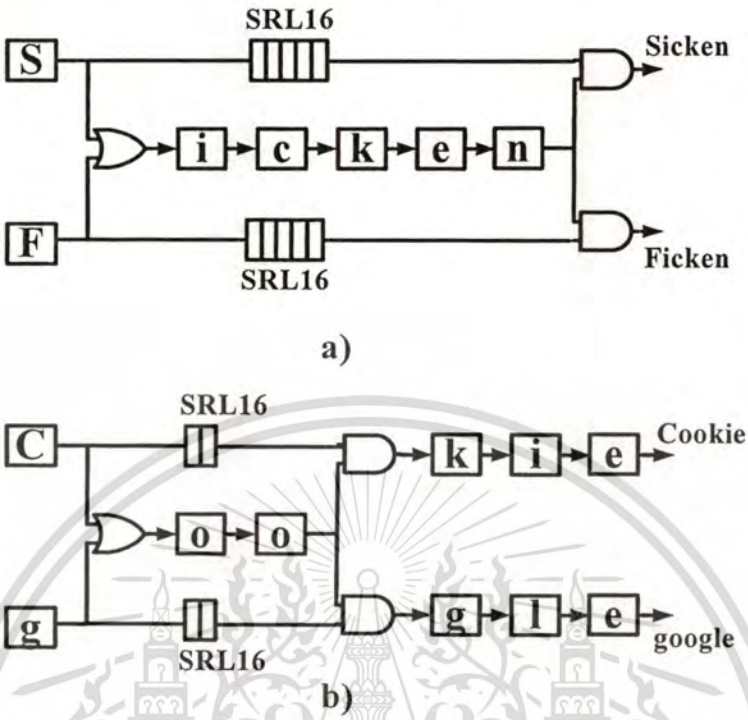


Figure 3.8: a) Example of Sharing of *suffix* of 2 patterns "Sicken" and " Ficken". The match signals of PEs that content 'S' and 'F' are delayed 5 clock cycles by SRL16. b) Example of Sharing of *infix* of 2 patterns "Cookie" and " google". The match signals of PEs that content 'C' and 'g' are delayed 2 clock cycles by SRL16.

A keyword tree in Figure 3.7.a shows how it can optimize the memory utility by reusing the keywords which are *prefixes* of patterns. The conversion not only decreases the amount of required storage, but also narrows the number of potential patterns as the pattern search algorithm traverses the tree. Since the output of the previous Processing Element is forwarded to enable the next stage, no additional logic is required for this area improvement. By applying this optimization over the entire groups, the total logic area can be saved around 35% of its initial size [74].

We can further improve for area by sharing *infix* and *suffix* due to similarity of patterns in Snort. As in Fig. 3.8.a, two patterns "Sicken, Ficken" can share the suffix "icken". The match signals of PEs that content 'S' and 'F' are delayed 5 clock cycles, afterthat "AND" with 'n', output of the last sharing PE, to determine matching.

As mentioned above, a LUT in Virtex-2 or a SLICEM of Virtex-4 can also be configured as a 16-bit shift register (SRL16) without using the flip-flops available in a slice. As a result, each LUT can delay serial data anywhere from one to 16 clock cycles. Therefore, we can use SRL16 to delay match signals. The infix sharing of is similarity with the suffix sharing. For example in Fig.3.8.b, two patterns "Cookie" and " google" share the infix "oo". The match signals of PEs that content 'C' and 'g' are delayed 2 clock cycles by SRL16. By applying the improvements over the entire groups, the total logic area can be sharply decreased to 65% of its initial size.

With the number of groups of 149, it can consume a lot of hardware for creating the compact encoding tables. In practice, we can share the encoding tables for groups that just differ from only a few characters. These same characters have the same codes; the last different characters have the private codes of the exact group that they belong to and the general code "000..." for other groups. For example, two groups "abcdefg" and "abcdefh" have the same codes for prefix "abcdef". The last character 'g' has the code "111" in the first group and the code "000" in the second group. Conversely, character 'h' has the code "111" in the second group and the code "000" in the first group.

Sharing encoding tables decreases the number of encoding tables from 149 to 67 (3-bit encoding tables reducing from 43 to 22, 4-bit encoding tables reducing from 74 to 27, and 5-bit encoding tables reducing from 32 to 18). Therefore, the area cost for encoding tables decreases up to 59%.

To increase the processing throughput of the system, we can use parallelism. We can widen the distribution paths by a factor of N providing N copies of engines. Figure 3.9 illustrates this point for N = 3. The single pattern ABC is searched for starting at offset 0, 1 or 2 within the 3-byte wide input stream. PE_{xk} of Engine *i* connected to offset *[(i-1)+(k-1)] mod N,* such as in Fig. 3.9, character 'A' at offset 0 is applied to PE_{x1}, PE_{x4} in Engine1, PE_{x3} in Engine2, and PE_{x2} in Engine3. Note that the micro-architecture of PE is slightly different with the one-character architecture, one flip-flop is used for N= 3 PEs. Therefore, we can slightly decrease the required area. This technique can be used for any value of N, not restricted to powers of three.

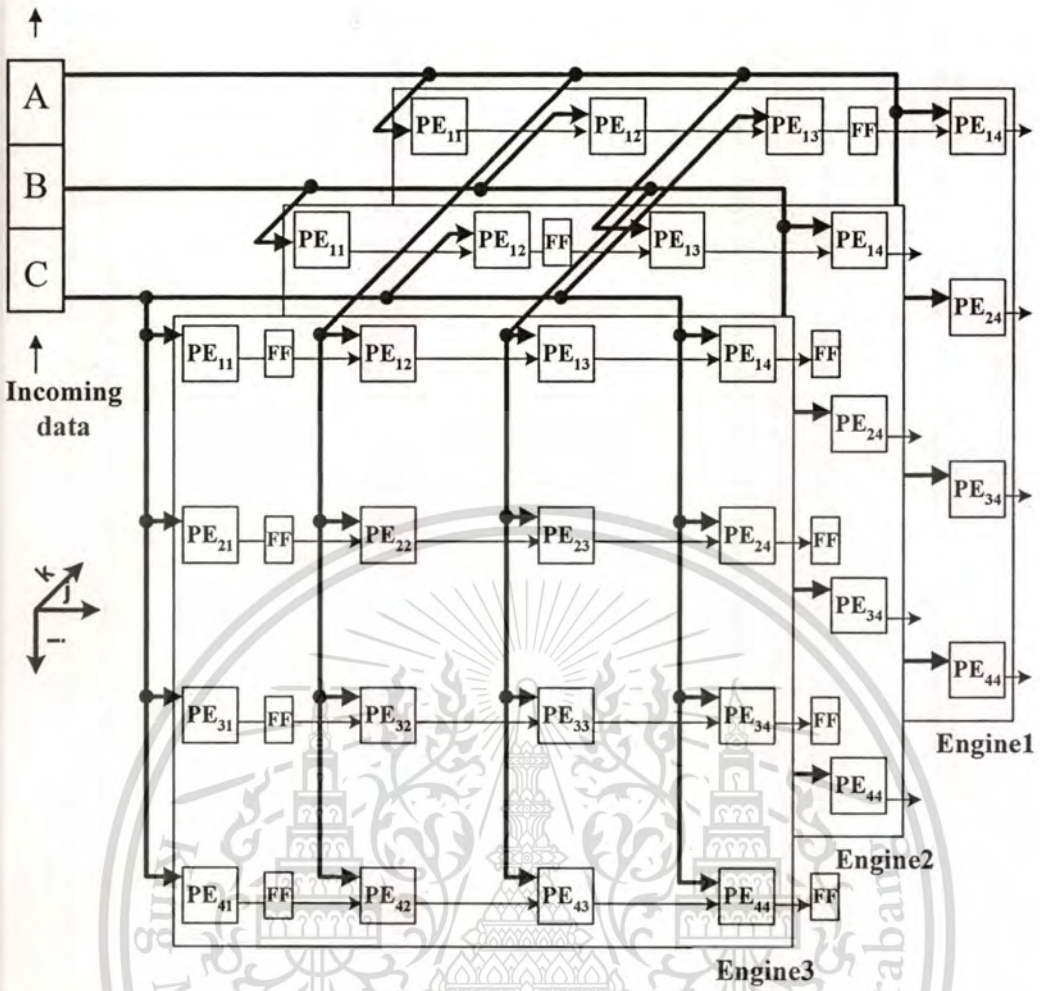


Figure 3.9: Multi-character processing by using N (N=3) engines of MPAs. Note that the micro-architecture of the PE has no Flip-flop,

3.2 FPGA Implementation of Processor-based Architecture

The design is coded by Verilog HDL and the design environment Xilinx's ISE 8.1i is used for all parts of the design flow including synthesis, mapping, and placing and routing. The target chips are Virtex4 XC4VLX100 and Virtex2 XC2V6000. Totally, we can fit 37,873 characters of entire ruleset in the FPGA chips.

We evaluate the efficiency of our processor array architecture and implementation using two metrics that are commonly used in previous works [7-12, 24]: performance in terms of operating frequency, and area cost in terms of required FPGA logic cells. The operating frequency is measured by using Xilinx Timing Analyzer. The throughput of a design is calculated by multiplying the clock frequency with the incoming data width (ranging from 8 bits to 32 bits in our evaluation). The logic cells are counted after synthetic process.

First, to evaluate the impact of resource sharing on our proposed architecture, we considered two implementations. The first one named *PA-1* shares only prefix as our previous publication [74] and the second named *PA-2* shares all substrings of patterns and compact encoding tables as improvement mentioned above. We use 3 rule set sizes ranging from 6,835 to 37,873 search pattern characters. The first set only contains cluster C1 with N_{C1}= 901 patterns of 6,835 characters. The second set is added C2 with $N_{C1}+N_{C2}$= 2,014 patterns of 26,469 characters. The last set contains entire SNORT rule set with $N_{C1}+N_{C2}+N_{C3}$= 2,378 patterns of 37,873 characters.

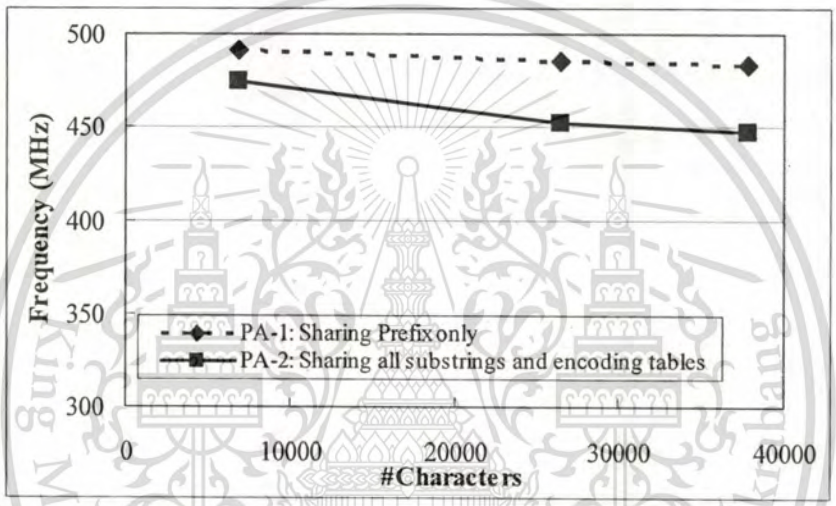


Figure 3.10: The clock frequency (MHz) of two implementations of one-character processing (N=1): *PA-1:* sharing the prefix only; and *PA-2:* sharing all substrings and compact encoding tables, on Virtex4 device.

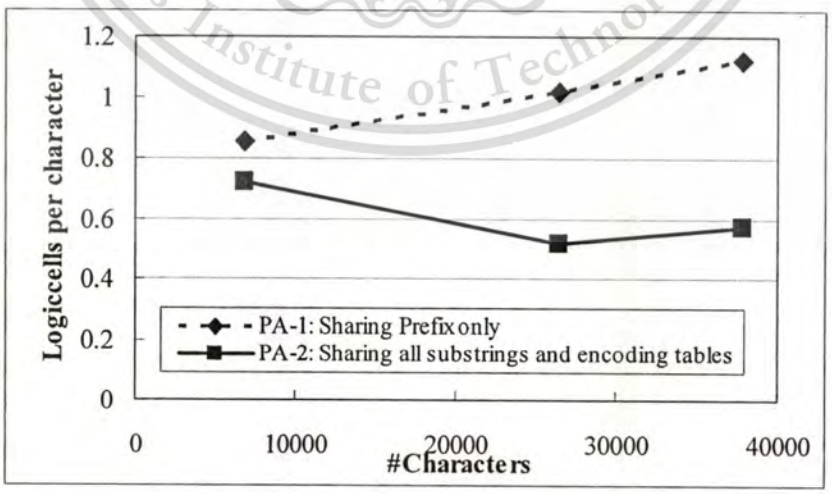


Figure 3.11: The area cost (Logiccells per character) of two implementations of one-character processing (N=1): *PA-1 and PA-2*, on Virtex4 device.

It can be seen in Fig. 3.10 that the clock frequency on Virtex4 of *PA-1* is extremely high, 483 MHz for processing the entire Snort pattern set. The reason of this result is the simplicity of architecture and most components used in design are the primitives of Xilinx FPGA chip. Moreover, the pattern set is partitioned into smaller groups that help our system to avoid the high fan-out as a text string is compared with thousands of patterns at the same time. Fig. 3.10 also shows that the frequency slightly drops down from 491 to 483 when the number of characters increases from 6,835 to 37,873. For *PA-2*, the clock frequency is lower than *PA-1* but still very high, 447 MHz on Virtex4. However, the drop of frequency is bigger, from 475 to 447 MHz. As compared with *PA-1, PA-2* has the operation frequency lower than of 7.5%. However, the area cost is significantly improved with more than 50% as in Fig. 3.11. It just occupied 0.52-0.72 LCs/char as compared with 0.85-1.12 LCs/char of *PA-1*. In summary, the throughput per area cost of *PA-2* is far better than of *PA-1*.

As described earlier, we can utilize parallelism to increase the processing throughput of the system. Our system is highly scalable that it can process *N* characters every clock cycle. In practice, we evaluate the performance and cost of our system with two and four parallel characters, i.e. systems that processes 2 and 4 input bytes per cycle. From now on, the systems under test are based on *PA-2*, the implementation of sharing all subpatterns and compact encoding tables.

Figure 3.12 plots the performance in terms of operating frequency of multi-character processing for pattern set sizes ranging from 6,835 to 37,873 search pattern characters. As expected, the operating frequency is inverse proportion to the number of characters processed per clock cycle. The performance also drops as the pattern set size increases. For two-character (16-bit) processing, the system can operate at around 424-465 MHz. While the four-character (32-bit) system operates at 393-450 MHz. These frequencies correspond to processing throughput of 6.78-7.44 and 12.58-14.40 Gbps respectively.

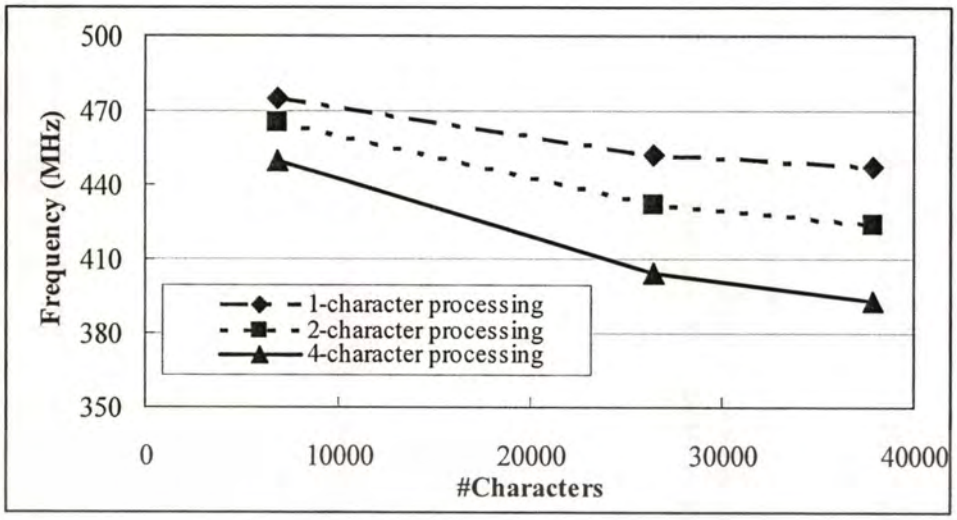


Figure 3.12: The clock frequency (MHz) of multi-character designs, on Virtex4 device.

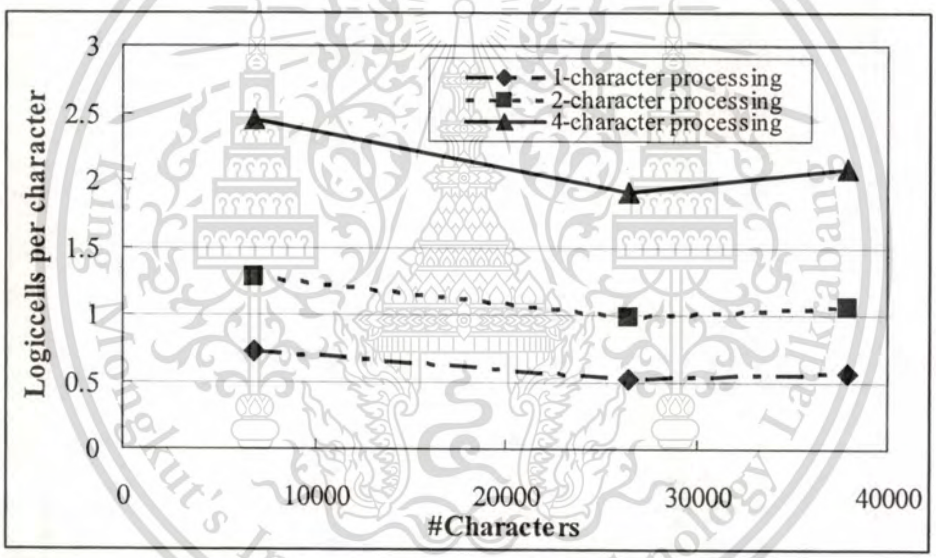


Figure 3.13: The area cost (Logiccells per character) of multi-character designs, on Virtex4 device.

The area cost per search pattern character is shown in Figure 3.13. Depending on the parallelism, the number of required logic cells per search pattern character is between 0.57 and 2.08 for the entire pattern set. Since each pattern character is searched for in two or four duplicate PEs to increase parallelism, the actual area cost of matching one character at one PE is smaller than our one-character design. In general, the higher number of character processed per clock cycle is the higher of utilization.

Table 3.1 shows the comparison of PA-2 with both kinds of existing approaches that are memory-based systems [10-12, 24] and logic gate-based systems [7-9]. For comparison purposes, a device-neutral metric called logic cells per character (LCs/char) is used. This metric is determined by dividing the total number of logic cells used in a design by the total number of characters programmed into the design. As can be seen in Table 3.1, our LCs/char is as small as logic gate-based systems. The logic gate utilization of memory-based systems [10-12] is extremely efficient because they use block RAMs of FPGA to store some parts [10] or the entire pattern rule set [11-12]. However, the throughput of them is usually lower than logic gate-based systems due to the complexity in architecture and the low speed of block RAM.

Our system has very high throughput, 3.14 to 12.58 Gbps, depend on the kinds of FPGA chips implemented and the number of bits processed per clock cycle. Our throughput is the best among logic gate-based systems and far better than memory based systems by at least *1.5x*.

To evaluate the throughput per area cost, a Performance Efficiency Metric (PEM) is used as the ratio of throughput in Gbps to the logic cell per each pattern character.

$$PEM = \frac{Throughput}{\frac{No.Logiccells + \frac{Membytes}{12}}{No.Characters}} \quad (3.3)$$

Assuming that the cost of 12 bytes block RAMs is equivalent to a logic cell [30], Eq. (3.3) takes into account both block RAMs and logic cells area metrics for fair comparison [14] between the memory-based systems and the logic gate-based systems. As PEMs in the range of 5.32-6.40, our systems are continually far better than the logic gate-based systems and acceptable as compared to memory-based systems.

Table 3.1 Comparison of Processor Array-based Architecture and previous FPGA-based pattern matching architectures.

System	Device (XC-)	Bits /cycle	Freq. (Mhz)	No. Chars	No. LCs	Mem (kbits)	LCs /char	T-put (Gbps)	PEM
Our system *(PA-2)*	2V6000	8	392		22,346		0.59	3.14	5.32
	4VLX100	8	447		21,652		0.57	3.58	6.26
	2V6000	16	368	37,873	42,150	0	1.11	5.89	5.29
	4VLX100	16	424		40,164		1.06	6.78	6.40
	2V6000	32	346		80,640		2.13	11.07	5.20
	4VLX100	32	393		78,762		2.08	12.58	6.05
PreD-CAM[7]	2V3000	8	372	18,036	19,854	0	1.100	2.98	2.70
BDDs [8]	2V8000	8	300	19,715	N/A	0	0.60	2.50	4.17
Prefix Tree [9]	2V3000	8	372	18,036	19,854	0	1.10	2.98	2.70
	2V6000	32	303		64,268		3.56	9.70	2.72
Bit-Split FSM[24]	4FX100	8	200	16,715	4,514	6,000	0.270	1.60	0.39
HashMem [12]	2V1000	8	250	18,636	2,570	630	0.140	2.00	4.01
	2V3000	16	232		5,230	1,188	0.280	3.71	3.86
PH-Mem [11]	2V1000	8	263	20,911	6,272	288	0.300	2.11	4.71
	2V1500	16	260		10,224	306	0.490	4.16	6.44
ROM+ Coproc[10]	4VLX15	8	260	32,384	8,480	276	0.260	2.08	5.90

Chapter 4

Parallel Cuckoo Hashing Architecture

In the previous chapter, systolic paradigm is applied for pattern matching in deep packet filtering of NIDS; it helps NIDS check all of incoming packets at over 10Gbps network rate. However, the drawback of our systolic processor array architecture and recently hardware-based systems is the flexibility. With emergence of new worms and viruses, the rule set must be frequently updated. For proposed FPGA-based NIDS/NIPSs [2]–[14], [74] adding or subtracting a few rules requires recompilation (synthesizing, placing and routing) and reconfiguration of some parts or the entire design. Although reconfiguration is one of the advantages of SRAM-based FPGA; this process can take several minutes to several hours to complete. Today, such latency may not be acceptable for most networks when new attacks are released frequently.

Moreover, the processor array architecture still has high hardware area cost due to not utilizing the block RAM available in FPGA chips. With the latest SNORT rule set such as of May 2007, it has over 68k characters and the processor array architecture can hardly fit entire pattern set on single FPGA chip with 4-character processing. It is necessary to build a system that can achieve high throughput per area with rapid rule set update.

In this chapter, based on a recently proposed hashing algorithm called Cuckoo Hashing [15], we propose an FPGA-based architecture of variable-length pattern matching, named *PAttern Matching Engine with Limitedtime updAte* (PAMELA). Patterns can be easily added to or removed out of the Cuckoo hash tables. Unlike most previous FPGA-based systems, PAMELA can rapidly update the static pattern set without reconfiguration thanks to Cuckoo Hashing. PAMELA reaches not only high flexibility but also best performance [75, 90, 16].

In addition, based on our analysis, the main drawback of Cuckoo Hashing is time consuming that can make the rule update time unbounded. We propose to use a stack to prevent rehashing and a FIFO to buffer the incoming data while updating the pattern

set. Our theoretical analysis and simulation results show that the insertion time of a new pattern is limited to 17 microseconds at 200 MHz system clock frequency. As a result, a new rule set can be updated to on line and on the fly. The power of massively parallel processing is discussed in this chapter. PAMELA is scalable enough to process N characters per clock cycle. With several improvements, PAMELA can save 30% of the area compared with the best system, and the throughput can achieve up to 8.8 Gbps for 4-character processing.

The rest of this chapter is organized as follows. Section 4.1 proposes the architecture of PAMELA engine. Next, performance analysis and simulation of PAMELA are discussed in Section 4.2. The experimental results and comparison with previous FPGA based systems are presented in Section 4.3.

4.1 PAMELA: Pattern Matching Engine with Limited-time Update for NIDS/NIPS

For pattern matching in NIDS/NIPS, the patterns are searched upon incoming data (packets). The matched pattern can occur anywhere as the longest substring. Normally, the pattern set is preprocessed and built in a system. The incoming data are buffered to compare against all patterns. As a result, hashing method is a good candidate for fast multi-pattern matching. FPGA-based Cuckoo Hashing is an appropriate choice for guaranteeing the lookup time with line rate of incoming data. Furthermore, dynamic update for the pattern set does not affect the performance of lookup. Fig. 4.1a shows an overview of *PAMELA: PAttern Matching Engine with Limited-time UpdAte for NIDS/NIPS* using FPGA-based Cuckoo Hashing. Each module named *Cuckoo* L_i stores patterns of the same length i characters. PAMELA also includes the update module that can update the pattern set rapidly and easily without interrupting the incoming data.

In order to process at the network speed in Gbps range, we have to construct Cuckoo Hashing module for every pattern length from $L_{min} = 1$ up to L_{max} characters. The priority circuit then selects the longest pattern if multiple matches happen. However, L_{max} can grow up to hundreds in most of NIDS/NIPS systems. The requirement of hardware resources is enormous so we cannot implement the system on a single-chip. Thus, we

build the Cuckoo Hashing modules for short patterns with the maximum length L_{max_s} according to the distribution of patterns in NIDS/NIPS. In Snort pattern set, the value of L_{max_s} is 16 characters. For longer patterns, we can break them into shorter segments so that we can insert those segments to the Cuckoo modules of short patterns. We then use simple address linked-lists to combine these segments later. Fig. 4.1b shows our optimized architecture for pattern matching.

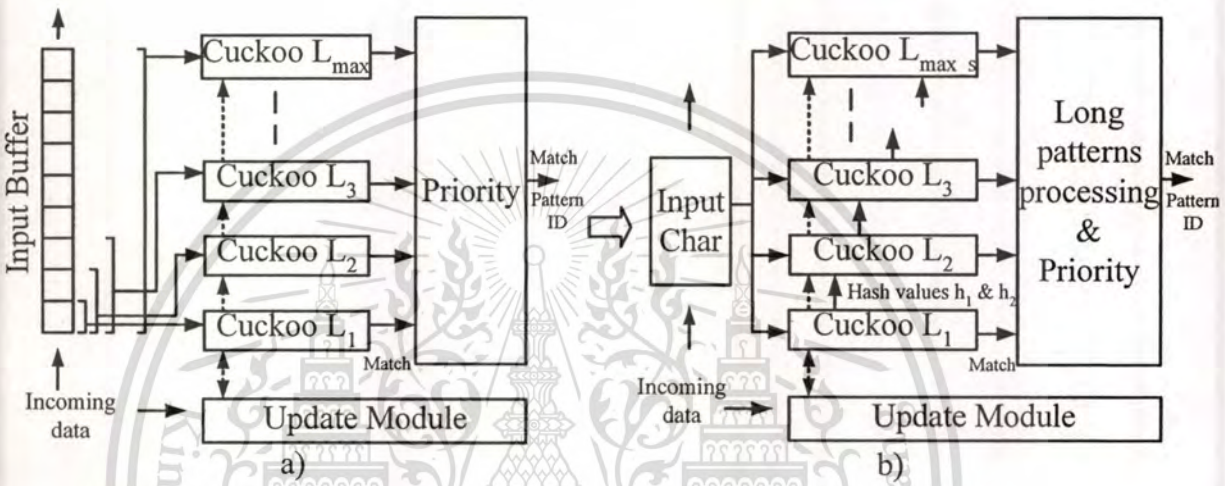


Figure 4.1: *PAMELA: PAttern Matching Engine with Limited-time UpdAte for NIDS/NIPS* using FPGA-based Cuckoo Hashing. a) General Model b) Optimized Hardware Model

4.1.1 FPGA-Based Cuckoo Hashing Module

In order to increase memory utilization, we build up a hashing module for each pattern length and use indirect storage. Small and sparse hash tables contain indices of patterns which are the addresses of a condensed pattern-stored table. Our approach is also to change the lookup to parallel processing. The insertion can be changed for better selection of the available space in both hash tables. With some improvements in architecture, ours can take full advantages of hardware.

The architecture of a FPGA-based Cuckoo Hashing module as shown in Fig. 4.2 consists of three tables. Two index tables (hash tables) T_1 and T_2 are single-port SRAMs and a pattern-stored table T_3 is a double-port SRAM for concurrent processing. Hash functions can be changed if they are required to rehash. Two registers named *key* and

index are the pattern to be searched for and the memory address of that pattern in T_3, respectively. Besides, two multiplexers are used to select addresses of T_3. The output of first multiplexer (*MUX*1) is the address of *portA* that is the read-only port. *MUX*1's inputs are the output value of T_1 (*index_*T_1) and T_2 (*index_*T_2). The output of second multiplexer (*MUX*2) is the address of *portB* that is both reading and writing port. *MUX*2 selects *index* T_2 as lookup function or *index* of key as insertion function. Finally, a comparator is used for exact matching of *key* with two candidate patterns from T_3.

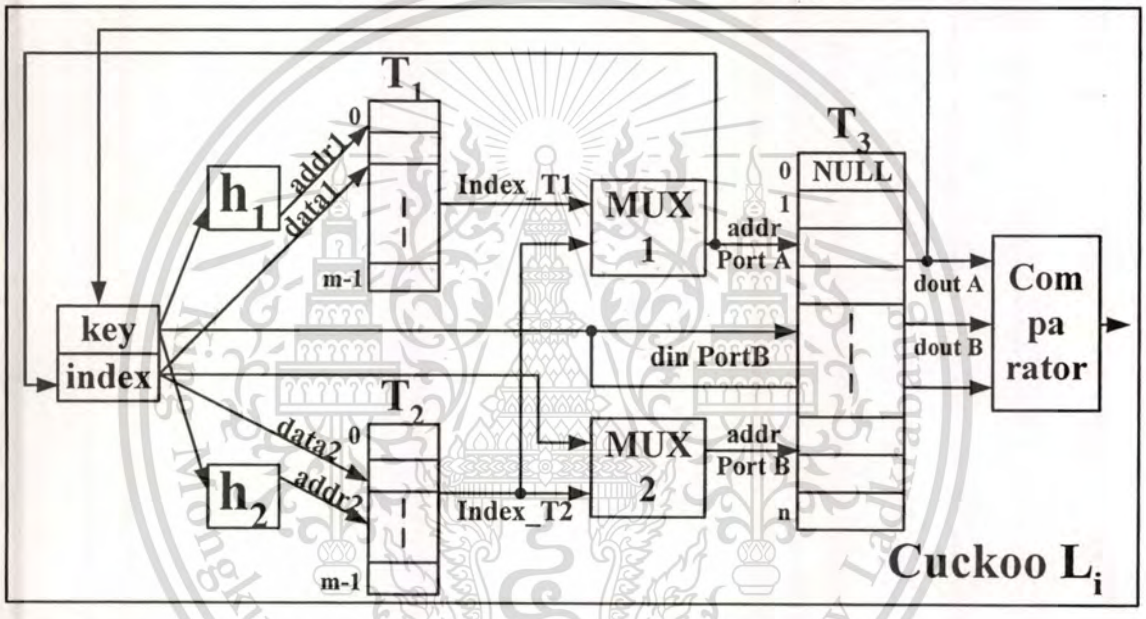


Figure 4.2: FPGA-based Cuckoo Hashing module with parallel lookup. Tables T_1, T_2 store the key indices; Table T_3 stores the keys

4.1.1.1 Parallel Lookup:

By using parallel and multi-phase pipeline architecture, PAMELA can look up patterns every clock cycle. Fig. 4.3 is the pseudo-code of parallel Cuckoo lookup function. In the first phase, a pattern *x* is hashed by two hash functions concurrently. In the second phase, the values of two hash functions are used as the addresses of two index tables. In the third phase, the outputs of two index tables are used as the addresses of T_3. In the last phase, to determine the match, the outputs of T_3 are compared with *x*.

```
function lookup(x)
  select index_T1 in MUX1  and index_T2 in MUX2;
     index_T1 = T1(h1(x));              // phase1+2
     index_T2 = T2(h2(x));
     doutA = PortA(index_T1);           // phase3
     doutB = PortB(index_T2);
     return(doutA = x or doutB = x); // phase4
end
```

Figure 4.3: Pseudo-code of Parallel Cuckoo Lookup Algorithm

```
procedure insert(x)
    if (lookup(x)) return;
    select index_T1 in MUX1  and index in MUX2;
    PortB(index) = x;
    if(index_T1 == NULL){
       T1(h1(x))= index;   return;}
    else if(index_T2 == NULL){
       T2(h2(x))= index;   return;}
    select index_T1 or index_T2 in MUX1
       //depending on balance of hash table
    loop MaxLoop times   // "Cuckoo process"
       if (select index_T1 in MUX1){
           key = PortA(index_T1);
           index = index_T1;
           select index_T2 in MUX1;}
       else{ // (select index_T2 in MUX1)
           key = PortA(index_T2);
           index = index_T2;
           select index_T1 in MUX1;}
       index_T1 = T1(h1(x));
       index_T2 = T2(h2(x));
       if (select index_T1 in MUX1){
           if(index_T1 == NULL) return;
           doutA = PortA(index_T1);}
       else{ // (select index_T2 in MUX1)
           if(index_T2 == NULL) return;
           doutA = PortA(index_T2);}
    end loop
    rehash(); insert(x);
end
```

Figure 4.4: Pseudo-code of Parallel Cuckoo Insertion Algorithm

4.1.1.2 Dynamic Insertion and Deletion

When a key insertion occurs, we consider both key and its index. The key is used as an input for two hash functions and is stored in table T_3. Its index is used as the address for lookup the space of key in T_3 and is also stored in T_1 or T_2.

The insertion of a new pattern x can be described in Fig. 4.4. First of all, x is inserted into T_3 with address of next available space of T_3. We have some improvements as compared with the original Cuckoo Hashing. We consider both tables to decrease the insertion time. If one of the outputs of two index tables is empty (NULL), the index of x is inserted into T_1 or T_2 and the insertion is complete. If the outputs of both T_1 and T_1 are not NULL, we insert the *index* into the table with less number of patterns. At the same time, the "kicked-out" *index_*T_1 (*index_*T_2) and its data from T_3 will be written into the *index* and *key* registers to start the hashing process. Then, M, $\lceil 3/log_{1+\varepsilon}m \rceil$ [15], is decreased and the key value is hashed by hash function h_2 (h_1). The output data will be checked for whether the value is NULL. If it is NULL, the process ends with successful insertion. On the other hand, the process is continued by taking in turns hashing from h_2 (h_1) to h_1 (h_2). The worst case happens when M decreases to zero. Hence, a rehash is required. Two new hash functions h_1 and h_2 are issued by a pseudo-random number generator. As rehashing cost can be expensive, the choice of a suitable hash function has to be discussed in the next subsection.

```
procedure delete(x)
    if (!lookup(x)) return;
    key = NULL;
    index = index_T1 or index_T2;
            //depending on the lookup result
    PortB(index) = key;
    T1(h1(x))= NULL or T2(h2(x))= NULL;
    return;
end
```

Figure 4.5: Pseudo-code of Parallel Cuckoo Deletion Algorithm

For deletion, the algorithm in Fig. 4.5 is as simple as the lookup process. If the lookup succeeds, *MUX*1 will select exactly one of the outputs of two index tables to write into the index register. We then write NULL into table T_3 at the address pointed by the index register. After that, we reset the index register to NULL and write it into appropriate T_1 or T_2. The deletion process results in some "holes" in T_3. When the number of "holes" is greater than a threshold, the rehash for rearranging of T_3 can be implemented. This rehashing does not require new hash functions.

4.1.1.3 Recommended Hash Function

The choice of hash functions greatly affects the performance of the system. Moreover, the probability for rehashing in Cuckoo Hashing is also based on the randomized property of hash functions. In Cuckoo Hashing, the authors use the Siegel's universal hashing [95] that has a constant evaluation time. However, this constant time is not small and complex in practice.In this subsection, we discuss some simple and fast hash functions for string such as CRC, tabulation table and shift-add-xor; and choose the suitable one that is easily implemented on hardware.

CRC (cyclic redundancy check) is a polynomials function that is significantly cheap implementation on hardware [29], but its randomness characteristic is poor. The universal class of hash functions [26] perfom well which can support various input keys by randomly selecting hash functions from the family. An example of such construct is modular hash functions. However, it is not suitable for hardware because of the complexity of the prime modulo operation. A fast way of generating a class of universal hash function without the modular operation and hardware-friend, tabulation based hashing method [26], is defined as follows:

$$H_t(x) = a_t[0][x_0] \oplus a_t[1][x_1] \oplus \ldots \oplus a_t[n-1][x_{n-1}] \qquad (4.1)$$

A table contains a 2-D array of random numbers in the hashing space. A key is string n characters $x_0x_1..x_{n-1}$ and the hash value is calculated by bit-wise exclusive-or ($\oplus$) a sequence of values $a_t[i][x_i]$, which is indexed by each byte value of x_i and position of i in the string. The drawback of this method is that the size of random table is very large and depends on the key length.

Another class of simple hash function for hashing character strings named *shift-add-xor* (SAX) [27] utilizes only the simple and fast operations of shift, exclusive-or and addition.

$$H_i = H_{i-1} \oplus (S_L(H_{i-1}) + S_R(H_{i-1}) + c_i) \quad (4.2)$$

Two operators S_L and S_R denote the shift left and right, respectively. The symbol c_i is the character i^{th} of string and H_i is an intermediate hash value after examination of i characters. The initial value H_0 can be generated randomly. The authors have shown that the class is likely to be universal and good performance can be achieved in practice by randomly choosing functions from this class. The main advantages of SAX as compared with random-table are very few required hardware resources and high achievable clock frequency due to the architectural simplicity. To generate the new SAX hash function in case of rehashing, we only need to change the value of H_0 by simple pseudo-random circuit LFSR [28]. Therefore, the SAX is the best choice for our system and the practical performance will be shown in the next section.

4.1.1.4 Hardware Optimization for Cuckoo Module

We can significantly decrease large amount of hardware by exploiting accumulative characteristic of SAX hash function. From Eq. (4.2), to calculate hash value of an incoming pattern with length i characters in the hash module i^{th}, the requisite inputs are the hash values of i-1 characters calculated beforehand in the hash module $(i-1)^{th}$ and the character i^{th}. Therefore, the values of previous hash module can be reused for the next hash module. As shown in Fig. 4.1b, two hash values h_1 and h_2 of *Cuckoo* L_{i-1} are fed into *Cuckoo* L_i ($2 < i \leq L_{max\_s}$). In addition, the buffer for incoming data is also replaced by only a one-character shift register.

For instance, a pattern "*abc*" is stored in *Cuckoo* L_3 and an incoming data "*. . . 123abcde. . .* " is passed through the system. In clock cycle t, character '*a*' of data comes from the shift register; it is broadcasted to every Cuckoo module. At *Cuckoo* L_1, its hash values are passed to *Cuckoo* L_2. In next clock cycle, $t + 1$, *Cuckoo* L_2 uses the hash values from *Cuckoo* L_1 and the new incoming character '*b*' to calculate its hash values for string *"ab"*. Similarly, in clock cycle t+2, *Cuckoo* L_3 uses the hash values from

Cuckoo L_2 and the new incoming character '*c*' to calculate its hash values for string *"abc"*. A match signal is generated here. If many matches happen at the same time then a priority circuit selects the longest pattern and does not take interested in the other results.

In comparison with previous implementations [10-14], the module i^{th} requires all i characters of the input string for every calculation and no reused of previous hash values at all. We share some ideas with those of [25], but they use numerous hash functions up to 35. Their weak points lead to the consumption of the remarkable number of logic gates for implementing hash functions.

Nevertheless, our hardware optimization can increase the probability of rehash on the system. For example, when a rehash by collision happens at Cuckoo module i, new functions h_1 and h_2 of module i make the inputs of module i+1 changed and the hash values of module i + 1 will be incorrect. The process is going on recursively up to module L_{max_s}. As a result, the rehash can be forced from Cuckoo module L_{i+1} to module L_{max_s}. This thing, however, only affects the insertion process. This trade off is good enough because the first priority is fast lookup with smaller hardware.

4.1.2 Matching Long Patterns

This subsection will present the matching of long patterns with arbitrary lengths. We break the longer (> 16-Character) patterns into variable-length segments of 1 to $L_{max\,s}$ = 16 characters. The above Cuckoo Hashing modules can then be used for matching these individual segments. After that, these segments of a long pattern are combined to a chain that can be implemented by simple linked-list technique. The data structure for storing linked-lists is a table named T_4 whose depth is the depth of T_3 multiplied by the number of Cuckoo modules. Each address represents one segment and its content is an address of next segment in the same long pattern.

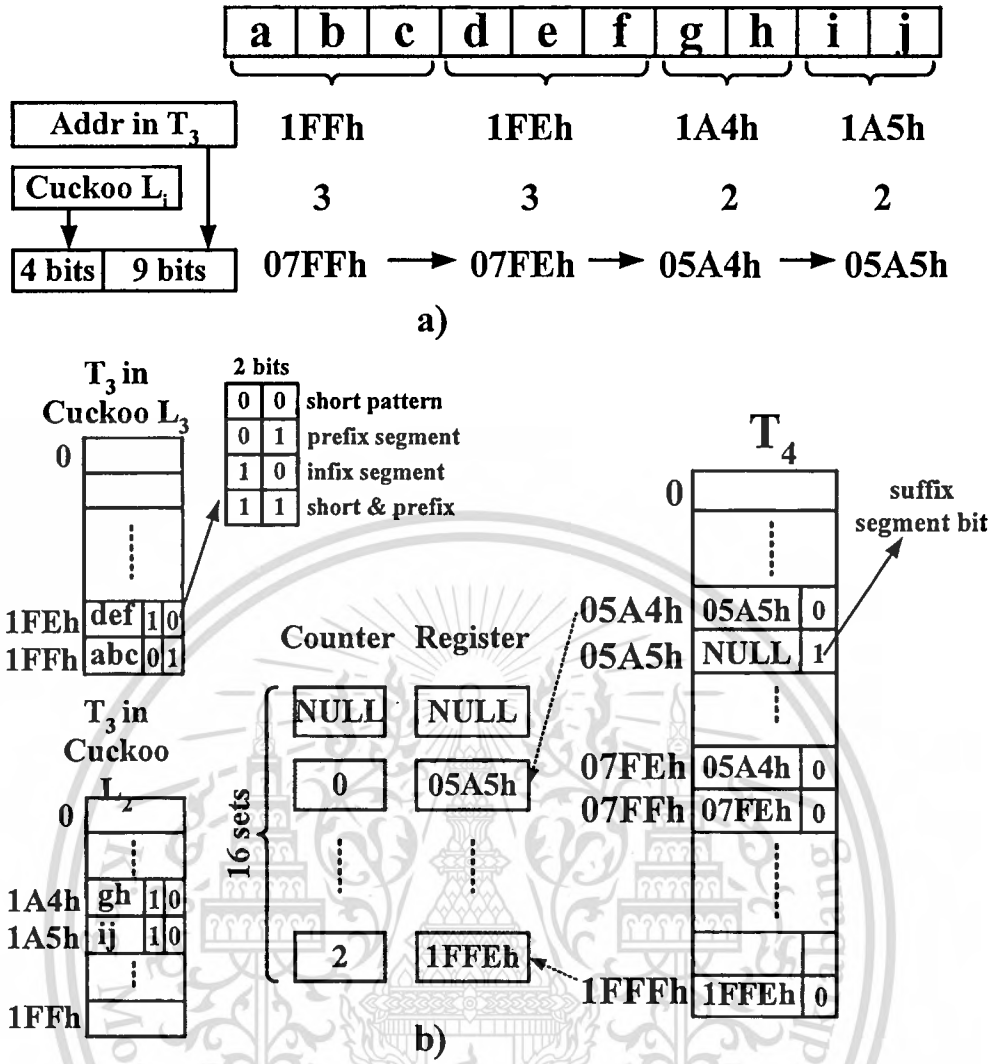


Figure 4.6: Matching long patterns. a) Example of breaking a long pattern "abcdefghij". b) How to store a long pattern in table T_4 as a linked-list

For more details, we describe the technique using a simple example. We assume that string *"abcdefghij"* is a long pattern that is broken into smaller segments with proportion 3 : 3 : 2 : 2 as in Fig. 4.6a. Segments 1 and 2 are hashed and stored in T_3 of *Cuckoo* L_3 at address 1*FFh* and 1*FEh* while segments 3 and 4 are hashed and stored in T_3 of *Cuckoo L2* at address 1*A4h* and 1*A5h*. We combine the addresses in T_3*s* together with the number of Cuckoo modules to link these individual segments. If the depth of T_3 in every module and the number of Cuckoo modules are 512 and 16 then the bit number is 13 for representing a position of individual segment: 9 least significant bits for representing the address of T_3 and 4 most significant bits for representing the number of Cuckoo modules. The segment addresses of *"abcdefghij"* in T_4 are 7*FFh*, 7*FEh*, 5*A4h*,

and 5*A*5*h* as shown in Fig. 4.6b. The matching process is described as the following. When we get a match for segment *"abc"*, we have address 7*FFh* whose content in T_4 is 7*FEh*. After 3 clock cycles, if the mach segment is *"def"*, the process is continued by jumping from 7*FFh* to 7*FEh*. Otherwise, the detective process finishes without match and resets for other patterns. Similarly, from 7*FEh*, if the next match segment after 2 clock cycles is *"gh"*, address 5*A*4*h* is considered. Finally, if the last segment *"ij"* is detected 2 next clock cycles later then the content of address 5*A*5*h* is read and the match of this pattern is reported.

Two more bits are added in every entry of T_3 as in Fig. 4.6b. They are used to describe a string which can be the short pattern, the first segment of long pattern called *prefix segment*, the body of long pattern called *infix segment*, or even the short pattern which is also the prefix segment. In case of the end of long pattern called *suffix segment*, we can determine it by checking the content of its address in T_4 whether it is NULL. However, if a long pattern is a prefix substring of another one then it cannot be detected. For example, a long pattern *"abcdef"* can be the prefix substring of *"abcdefghij"*. To improve this case, we add one more bit in table T_4. When this bit is active, a suffix segment can also be an infix segment of another pattern. In other words, the content of a suffix segment in T_4 can be a pointer to another address.

The above paragraphs describe the method for holding one long pattern per time. Each time, if the system detects one segment with length L, it has to wait for the next segment in L clock cycles. During this time, if other matches happen then the system cannot detect them. For example, the incoming data is *"...abcdefghijk..."* and we have 2 long patterns *"abcdmnpq"* and *"bcdefghi"* in the pattern set. We assume that these patterns are broken into segments *"abcd"*, *"mnpq"* and *"bcde"*, *"fghi"*, respectively. In clock cycle t, if the matched string is *"abcd"* then the system waits for the next match in 4 clock cycles. However, the next match for the first pattern does not happen in this case. Meanwhile, in clock cycle $t + 1$, the system cannot detect the segment *"bcde"* that is the prefix for the second pattern.

To detect all segments of long patterns while matching in every clock cycle, we use $L_{max\_s} = 16$ down counters and registers for storing the match values read from T_4; the length of segment and its content in T_4 are stored in an available counter and register,

respectively. When a segment match happens and it is a prefix segment, its position in T_4 is considered. If it is an infix or a suffix of current long pattern candidates, it must be simultaneously compared to all registers whose counters are currently zero values to determine the unique address of T_4. Because the length of segments can reach at L_{max_s}, the maximum number of current long pattern candidates is L_{max_s}.

```
procedure insert_long(x)
    sh: short pattern in table T3;
    prefix(k,x): prefix with length k of x;
    suffix(k,x): suffix with length k of x;
    //  lookup the longest prefix of x
    if(lookup(x)) return;
    if(lookup(sh=prefix(k,x)) and sh without "011"){
        //short & prefix segment
        change the end of sh to "011";
        call part(suffix(k(x-sh),x),"010","110");
    }
    else
        call part(x,"001","110");
end

procedure part(x,pre_id,suf_id)
    L: length of x;
    #T3(L): #patterns in T3 of length L;
    Th_L:threshold of length L;
    if(L > L_max_s or #T3(L) > Th_L or lookup(x)){
        part(prefix(L-L/2,x),pred_id,"010");
        part(suffix(L/2,x),"010",suf_id);}
    else{
        if(pre_id=="001")       //prefix segment
            insert(x) with the end "001";
        else if(suf_id=="110") //suffix segment
            insert(x) with the end "110";
        else                    //infix segment
            insert(x) with the end "010";
    }
end
```

Figure 4.7: Pseudo-code of Long Pattern Insertion Algorithm

We use an automatic generator to partition the long patterns in pattern set. Fig. 4.7 is the pseudocode of long pattern insertion. First, we consider long patterns as the incoming strings pass through the engine for lookup. If a short pattern in the engine is the longest prefix of a candidate long pattern then we break the long pattern such that its first segment is the same length as the short pattern. We mark the short pattern which is also the prefix segment to avoid checking again. After that, we break the rest of the long pattern into halves if the length of the rest is greater than L_{max_s} or the number of patterns in the same table T_3 is greater than a certain threshold *Th*, somewhat arbitrarily set less than the size of hash table, to avoid the high probability of rehash. Otherwise, the rest can be a suffix segment. The partitioning can continue if any segment matches with a string in the engine.

4.1.3 Massively Parallel Processing

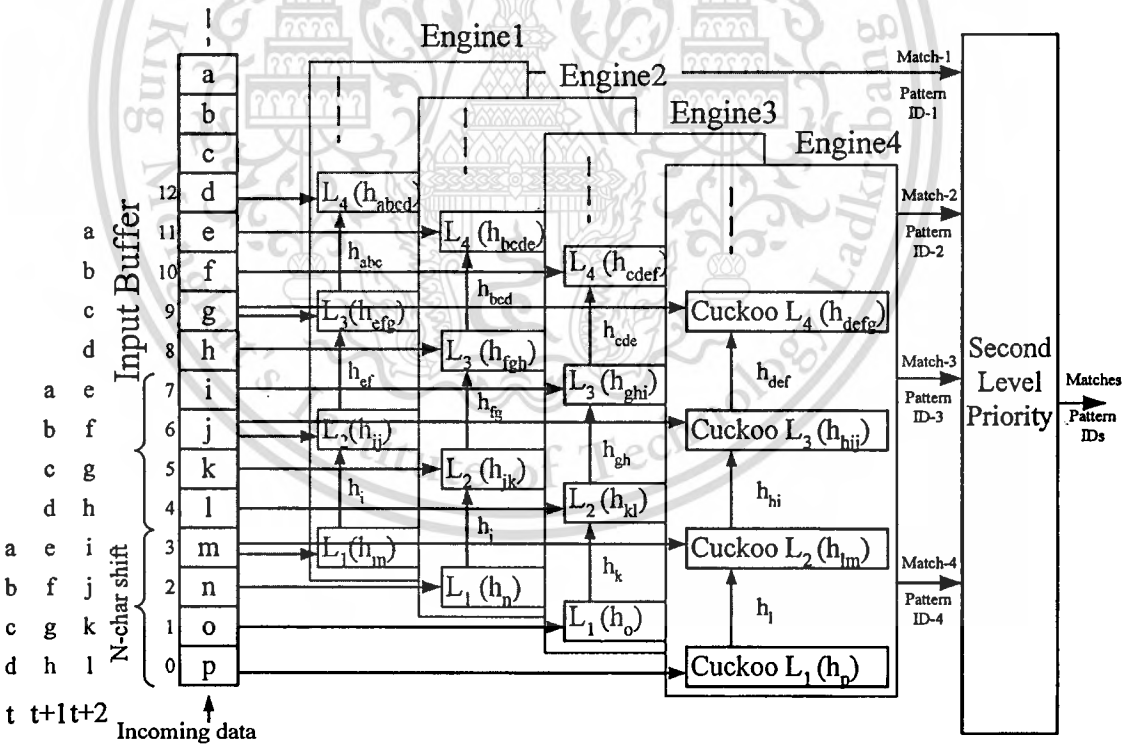


Figure 4.8: PAMELA for parallel processing of N-character (N = 4). Cuckoo modules are connected to the input buffer at the pre-determined addresses. The input data is string *"...abcdefghijklmnop..."* and PAMELAs are being the state of time t + 3. h_xs represent for hash values h_1 and h_2.

To increase the throughput, PAMELA engine can be easily extended for multi-character processing. Instead of reading one character per clock cycle, the system shifts N characters into an input buffer every clock cycle. The system has N engines corresponding to N characters needed to process. Inside the engine, each Cuckoo module receives one character from the pre-determined address of input buffer and the hash value from the previous module. Fig. 4.8 is an example of our system to process 4 characters per clock cycle. Due to higher addressing complexity, we need to determine the address of input buffer that each Cuckoo Module is connected to. We assume that the incoming data is the string "$x_0x_1...x_k$, x_{k+1}..." and we shift N characters each clock cycle. In each engine j ($1 \leq j \leq N$), if the address connected to the previous module is $A_{i-1,j}$ ($2 \leq i \leq L_{max\_s}$) and contents character x_k then the address connected to the next module, $A_{i,j}$, has to shift N-1 from $A_{i-1,j}$ and the next module processes the character x_{k+1}. For illustration, the incoming data is *"...abcdefghijklmnop..."* and PAMELAs consume N = 4 characters every clock cycle. In clock cycle t, we shift string *"abcd"* into the buffer, *Cuckoo* L_1 of every engine will consume one character for hashing corresponding to one cell of the buffer; *Cuckoo* L_1 of *Engine1* consumes the character *'a'* at address $A_{1,1}$ = 3. In next clock cycle, t + 1, in *Engine1*, *Cuckoo* L_2 of *Engine1* consumes the character '*b*' at address $A_{2,1}$ = 6 and the hash values from *Cuckoo* L_1 to calculate hash values for string *"ab"*. Similarly for other engines, the previous modules consume the character '*b*', '*c*' and '*d*' at address 2, 1 and 0 then the next module consumes the character '*c*', '*d*' and '*e*' at address 5, 4 and 3 one clock cycle later. As can be seen in Fig. 4.8 at clock cycle t + 3, PAMELAs have consumed 16 characters of incoming text with massively pipeline/parallel computation in 4 clock cycles. In general, the address in the buffer connected to Cuckoo *Li* of engine *j* is:

$$\begin{aligned} A_{i,j} &= A_{i-1,j} + (N-1) = A_{i-2,j} + 2(N-1) \\ &= ... = A_{1,j} + (i-1)(N-1) \end{aligned} \qquad (4.3)$$

From Eq. (4.3), we can see that when N = 1, the address of any module coincides with the first address. So it is correct with our one-character design. We can also

calculate the size of input buffer from Eq. (4.3). The last address is A_{Lmax_s},1 when i = L_{max_s}. If the lowest address is zero then $A_{1,1}$ is N-1. The size of the buffer can be calculated as follows.

$$\begin{aligned} S_{Buffer} &= A_{L_{max\_s},j} + 1 = A_{1,1} + (L_{\max\_s} - 1)(N-1) + 1 \\ &= (L_{\max\_s} - 1)(N-1) + 1 \end{aligned} \tag{4.4}$$

The resources for storing patterns in multi-character processing scheme can be significantly decreased by sharing SRAMs together. If the number of ports of SRAMs is two times of the number of processed characters, T_3 can be shared for Cuckoo modules that process the same pattern length. Moreover, the processing of long pattern can be also carried out only one table T_4 for the whole system.

4.2 Performance Analysis

4.2.1 Theoretical Analysis

In this subsection, we will analyze the time to insert a pattern. Based on the worst case scenario, we propose a method for bounding insertion time of a new pattern. In addition, some metrics such as the lookup time, latency, speedup and hardware utilization are also explored to show the efficiency of PAMELA engine. To facilitate the theoretical analysis, some main notations are listed in Table 4.1.

4.2.1.1 Insertion time

The time to process a pattern of length L (characters or bytes) is a function of the number of cycles needed to calculate the hash values T_{hash}, and the number of cycles needed to access memory including hash tables $T_{mem1,2}$ and storage-pattern table T_{mem3}. We assume that the time needed to access every memory table is constant and $T_{mem1,2}$ = T_{mem3} = T_{mem} and our engine takes one character per clock cycle, so T_{hash} = L. The equation for the processing time T_{procL} is expressed as

$$T_{proc_L} = T_{hash} + 2 \times T_{mem} = L + 2 \times T_{mem} \tag{4.5}$$

Table 4.1 Summary of main notations used in the performance analysis

Symbol	Units	Description
L	bytes	The length of a pattern
T_{hash}	cycles	The time to calculate the hash values
T_{mem}	cycles	The time to access any memory table
T_{procL}	cycles	The time to process a pattern of length L
T_{mtch}	cycles	The time of the comparison
T_{luL}	cycles	The lookup time of a pattern
T_{inL}	cycles	The insertion time of a pattern
T_{in_wL}	cycles	The worst case insertion time
T_{lu_long}	cycles	The time for lookup of a long pattern
T_{in_long}	cycles	The time for insertion of a long pattern
S_S	bytes	The size of the stack
T_{in_uL}	cycles	The unsuccessful insertion time
T_{partL}	cycles	The online partitioning time
S_F	bytes	The size of the FIFO
$T_{latency}$	cycles	The latency time
$S_{nomatch}$		The no-match speedup of each engine
S_{match}		The match speedup of each engine
S_{avg}		The average speedup of N engines
α		Load factor
U_{mem}		The average memory utilization
R_{LC}		The ratio logic gates of the general architecture and the optimized architecture

The lookup time is determined by the processing time T_{procL} as calculated in Eq. (4.5) adding one more stage, T_{mtch} - the time for comparing the candidate pattern after reading out of table T_3 with the part of incoming data. The equation for the lookup time is calculated as follows

$$T_{lu_L} = T_{proc_L} + T_{mtch} = L + 2 \times T_{mem} + T_{mtch} \quad (4.6)$$

Since the execution order of the stages is independent, all stages can execute as a pipeline. Inside of every stage, we can divide it into small pipelined phases. Therefore, the total lookup time for pattern set varies depending on the number of pipeline stages, and the number of Cuckoo modules executed in parallel. Because of the collision-free lookup, the system throughput only depends on the number of bits which can be processed in every clock cycle.

The insertion time T_{inL} of a pattern x with length L can include some shuffles of other patterns in hash table. Let h denote the number of shuffles then the average time of successful insertion is expressed as: $T_{in\_avgL} = (1+h) \times T_{procL}$. At the best case, successful insertion of a pattern requires no shuffle of other patterns in hash tables, $T_{in\_bestL} = T_{procL}$.

The worst case can happen as h reaches M and the rehash occurs. According to [15], probability of this case is $O(1/n^2)$ when M is $3log_{\varepsilon+1}m$. Before the rehashing process begins, the hash tables have to clear every occupied space. The reset time is T_{reset}. If N_L is the number of insertions for all patterns with length L until the successful insertion of pattern x then the time of rehash is $N_L \times T_{procL}$. In addition, according to the accumulative characteristic of SAX hash function, the rehashes are also required for the patterns of lengths from L+1 to L_{max_s}. Finally, the worst-case insertion time of a pattern can be calculated as

$$T_{in\_w_L} = M \times T_{proc_L} + \sum_{i=L}^{L_{max\_s}} (N_i \times T_{proc_i} + T_{reset}) \quad (4.7)$$

Where T_{proci} denotes the processing time of pattern with length i *and* N_i denotes the number of insertion time of all patterns with length i

For long patterns, we consider three more parameters: the time to access table T_4, $T_{mem4} = T_{mem}$; the time to compare and connect segments, $T_{connect}$; and the number of

segments of a pattern, *s*. We express the time for lookup and insertion of a long pattern as follows.

$$T_{lu\_long} = (T_{lu_L} + T_{mem} + T_{connect}) \times s \quad (4.8)$$

$$T_{in\_long} = (T_{in_L} + T_{mem} + T_{connect}) \times s \quad (4.9)$$

4.2.1.2 Limited-time Update

As the analyzed worst case of insertion in Eq. (4.7), updating time of a pattern can be lengthy. For real-time protection, some practical systems can be vulnerable. To avoid this weakness, the basic solution is that we build the duplicate modules or separate modules for updating only [10], [23]. However, this solution consumes a lot of hardware resources and requires re-compiling some parts of the system. We propose a simple and fast method that needs minimum hardware based on the capability of breaking a pattern into segments.

The details of our solution can be described as follows. If the worst case happens as a new pattern is inserted in the system then we report unsuccessful insertion instead of rehashing the hash tables. However, at that time, some positions of the hash tables are modified. Thus, we add a *stack* to trace the insertion process. Every step of insertion process is stored in the stack. If *M* is reached then we copy the traces from the stack back to the hash tables to restore the system. For illustration, we use a simple example as in Fig. 4.9. We insert a new element *"4"* into hash tables that stored *"0,2"* at addresses *3, 0* of T_1 and *"1, 3"* at addresses *4, 0* of T_2. At every step of collisions, we store the old information of "kick-out" elements in the stack including of the address in hash table $Addr_{hash}$, the content at this address $Content_{hash}$, and the order number of hash table Id_{hash}. As in Fig. 4.9, insertion process is infinite. Therefore, we restore steps one by one from the stack to the hash tables until the stack is empty with the last element of unavailable space being *"4"*. We can calculate the size of the stack as follows

$$S_S = M \times \lceil Addr_{hash} + Content_{hash} + Id_{hashtable} \rceil_{byte} \quad (4.10)$$

In Eq. (4.10), the symbol $\lceil\ \rceil_{byte}$ denotes rounding to byte of the bit sum of fields in the stack.

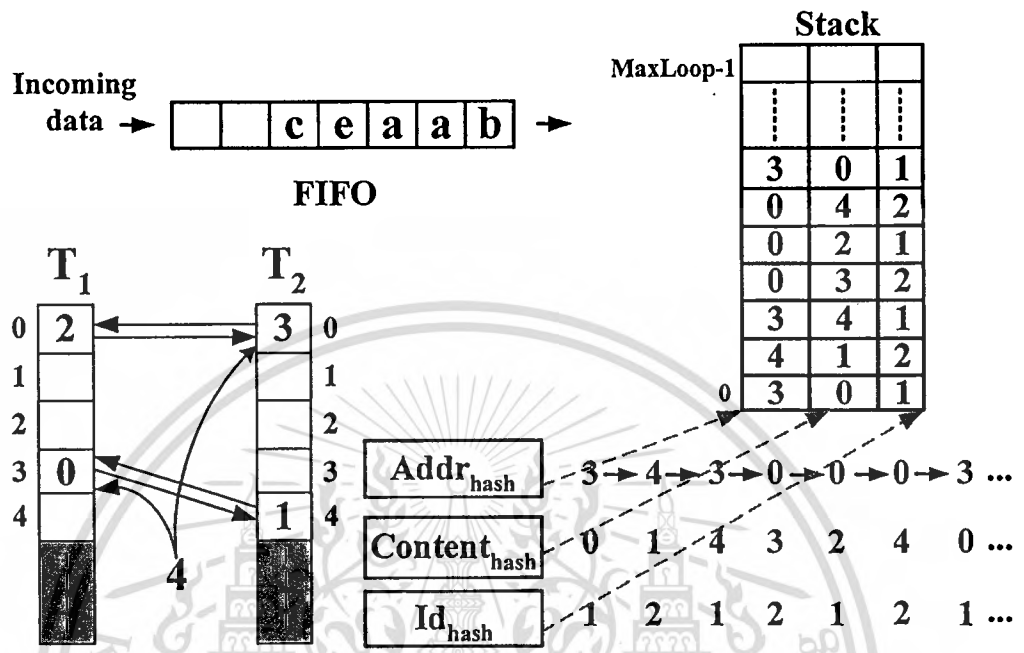


Figure 4.9: Example of Limited-time pattern update. A *stack* traces the insertion process, the old information of "kick-out" elements, including the address in hash table $Addr_{hash}$, the content of this address $Content_{hash}$, and the ID number of hash table Id_{hash}. A *FIFO* buffers the incoming data while updating patterns.

Next, we consider the new pattern as a long pattern and break it into segments to re-insert to the system. This process is recursive until the successful insertion occurs. We can prove that the update process of a string (pattern or segment) is limited by following paragraphs.

After an unsuccessful insertion of a string less than or equal L_{max_s} characters, the string must be broken into halves. Note that the partitioning of long patterns greater than L_{max_s} is preprocessed off-line before updating. The online partitioning is slightly different from off-line method as mentioned in section 4.1.2. We do not consider whether the short pattern in the engine can be the prefix segment to save time. So the unsuccessful insertion time T_{in_uL} and the partitioning time T_{partL} can be calculated as follows.

$$T_{in\_u_L} = M \times T_{proc_L} + M = M(T_{proc_L} + 1) \quad (4.11)$$

$$T_{part_L} = 2T_{lu_{L/2}} + 2T_{in_{L/2}} \quad (4.12)$$

The unsuccessful insertion time equals to the sum of insertion time and back tracking time. The partitioning time includes the lookup and insertion time of two halves. The worst case insertion time can be expressed by the sum of the unsuccessful time and the partitioning time.

$$\begin{aligned} T_{in\_w_L} &= T_{in\_u_L} + T_{part_L} \quad (4.13) \\ &= T_{in\_u_L} + 2T_{lu_{L/2}} + 2T_{in_{L/2}} \end{aligned}$$

Let us assume that $L_{max\_s} = 2^K$ and the shortest segment is one-character which is always successful insertion, so the insertion time $T_{im} = T_{in\_w1}$.

Lemma 1. *The worst case insertion time of a new string of length* $L_{max\_s} = 2^K$ ($K \geq 1$) *is limited by the following.*

$$T_{in\_w_{2^K}} \leq \sum_{i=0}^{K-1} 2^i (T_{in\_u_{2^{K-i}}} + 2T_{lu_{2^{K-i-1}}}) + 2^K T_{in\_w_1}$$

Proof. The proof is mathematical induction using Eq.(4.13).

- *Basis*: for $K = 1$, according to Eq.(4.13), it is true.

$$\begin{aligned} T_{in\_w_2} &= T_{in\_u_2} + 2T_{lu_1} + 2T_{in_1} \\ &\leq T_{in\_u_2} + 2T_{lu_1} + 2T_{in\_w_1} \end{aligned}$$

- *Inductive step*: we assume that the result holds for some unspecified value of K.

  We must be shown that the result holds for $K+1$, that is:

$$
\begin{aligned}
T_{in\_w_{2^K+1}} &= T_{in\_u_{2^K+1}} + 2T_{lu_{2^K}} + 2T_{in_{2^K}} \\
&\le T_{in\_u_{2^K+1}} + 2T_{lu_{2^K}} \\
&\quad + 2\left[\sum_{i=0}^{K-1} 2^i (T_{in\_u_{2^{K-i}}} + 2T_{lu_{2^{K-i-1}}}) + 2^K T_{in_{2^K/2^K}}\right] \\
&\le \sum_{i=0}^{K} 2^i (T_{in\_u_{2^{K+1-i}}} + 2T_{lu_{2^{K-i}}}) + 2^{K+1} T_{in\_w_1}
\end{aligned}
$$

Hence, the proof.

From Eq.(4.11), (4.5) and (4.6):

$$
\begin{aligned}
T_{in\_w_{2^K}} \le \sum_{i=0}^{K-1} 2^i \big[& M(2^{K-i} + 2T_{mem} + 1) \\
& + 2(2^{K-i-1} + 2T_{mem} + T_{mtch})\big] + 2^K T_{in\_w_1}
\end{aligned}
\tag{4.14}
$$

Moreover, for on-the-fly update without interrupting the data stream, we can use a *FIFO* to buffer data stream. While a pattern is being updated, the data stream enters the FIFO. After a successful update of a pattern, PAMELA will get data from the FIFO. The next update can continue as soon as the FIFO is empty. Because the network load is not 100% at all time, this on-the-fly update process is practical. If the line rate network is equivalent to one byte per clock cycle which can be in order of Gbps, the size (bytes) of the FIFO equals to the upper bound of a string insertion.

We assume that all timing parameters in Eq.(4.14) consuming one clock cycle, K = 4, and M = 30 so we have the maximum successful insertion time of a pattern less than 3,440 clock cycles as follows.

$$
\begin{aligned}
T_{in\_w_{2^K}} &\le \sum_{i=0}^{K-1} \big[2^K (M+1) + 2^i (3M+6) + 2^K \\
&\le 109M + 170 = 3440 = S_F
\end{aligned}
\tag{4.15}
$$

where S_F is the size of the FIFO.

For massively parallel processing, we can use only one stack and one FIFO for all engines, the size of the stack can keep as one-chacracter processing. Meanwhile, the size of the FIFO can multiply 3,440 to the number of engines, N.

4.2.1.3 Latency and Speedup

Based on the architecture of PAMELA, we can determine some parameters of on-line performance such as the latency and the system speedup.

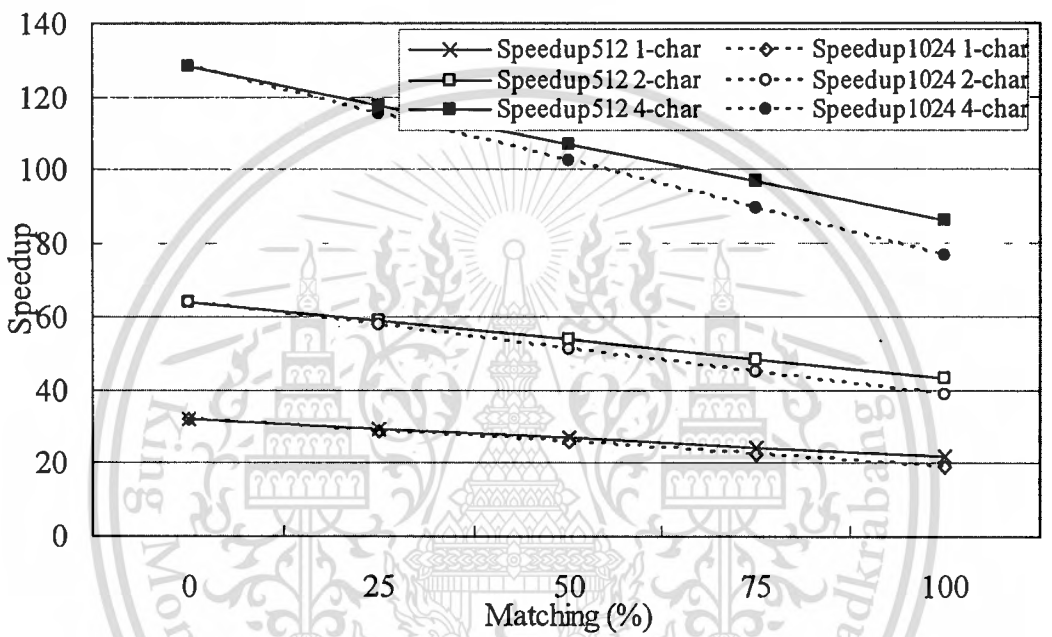


Figure 4.10: Speedup of PAMELAs with multiple characters per clock cycle processing compared with baseline serial Cuckoo Hashing (one-character processing). *Matching* (%) is the percentage of suspicious patterns that require pattern matching.

The latency time is the time that the system can answer when the match happens after reading the input data. It can be determined by the lookup time as in Eq. (4.6) and the time to determine the actual match, $T_{priority}$, when multi-match happens at the same time.

$$\begin{aligned} T_{latency} &= T_{lu_L} + T_{priority} \\ &= L + 2T_{mem} + T_{mtch} + T_{priority} \end{aligned} \quad (4.16)$$

For speedup calculation, we assume that the serial Cuckoo model can process the input data every clock cycle and the length of patterns is less than 16-character. We

consider two cases that are no-match speedup and match speedup. When the system has no match, the serial model has to lookup on both hash tables. With 2-lookup in every module and 16 Cuckoo modules running in parallel, the no-match speedup of each PAMELA engine is calculated as follows.

$$S_{nomatch} = 16 \times \frac{T_{lu\_T_1} + T_{lu\_T_2}}{T_{lu\_T_1}} = 32 \quad (4.17)$$

When the match happens, the serial model firstly searches in T_1. If there is no match then the second time of lookup is in T_2. The probability of access to one or two tables depends on the distribution of patterns in T_1 and T_2. Normally, this distribution in T_1 is around 60%-75% depending on the sizes of hash tables. The match speedup of each PAMELA engine is expressed as follows.

$$S_{match} = 16 \times \frac{\%T_{lu\_T_1} + \%T_{lu\_T_2}}{T_{lu\_T_1}} \quad (4.18)$$

Finally, we can determine the average speedup of our massively parallel processing based on the baseline of serial Cuckoo model.

$$S_{avg} = N \times (\%S_{match} + \%S_{nomatch}) \quad (4.19)$$

From Eq. (4.19), the average speedup changes depending on the matching percentage. We demonstrate clearly in Fig. 4.10. It shows the speedup of PAMELAs (multiple characters processing) compared with baseline serial Cuckoo Hashing (one-character processing). With 4-character processing, the speedup of PAMELAs can be up to *128X* as compared to the baseline of Cuckoo model. The speedup decreases as the match percentage increases. That is due to the speedup of match case is less than the speedup of the no-match case as in Eq. (4.18) and Eq. (4.19). The size of hash tables also affects to the speedup. The speedup increases when the size of hash tables increases. In summary, the speedup increases when the system can process several characters per clock cycle.

4.2.1.4 Hardware Utilization

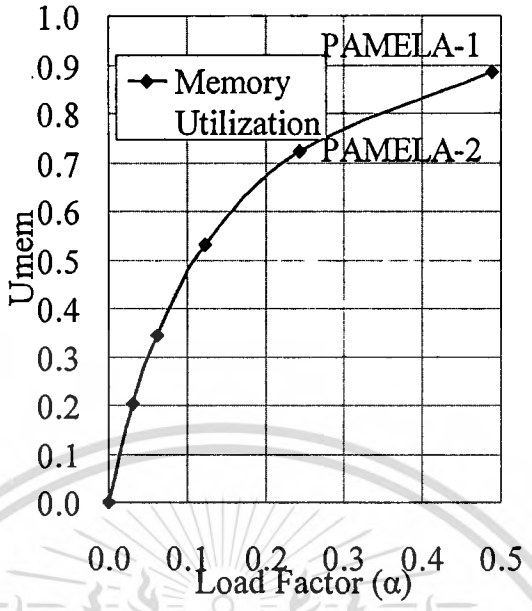


Figure 4.11: Memory Utilization vs. Load Factor of hash tables, T_1, T_2. *PAMELA-1* has Memory Utilization U_{mem} of 0.88, *PAMELA-2* has U_{mem} of 0.72

For memory utilization analysis, an important metric in hashing method is *load factor*, the number of elements per hash table size. In the direct storage method, load factor presents the efficiency of memory utilization. The maximum of load factor is one, that is, the hash method is minimum perfect hashing (MPH). Nevertheless, MPH is complex and only suitable for static case. In Cuckoo Hashing, load factor is less than 0.5 to guarantee successful insertions. If we define n_{avg} as the average number of patterns and m_{avg} as the average size of hash table in each Cuckoo module, then average load factor α is given by the formula

$$\alpha = \frac{n_{avg}}{2 \times m_{avg}} \tag{4.20}$$

In our indirect storage method, both hash tables only store the indices of patterns. Therefore, the memory utilization metric of our system has to take into account the condensed table T_3. If we assume that the size in bits of T_3 is equal to the number of bits of pattern storage required, then it is the result of n_{avg} in a module multiplied by the average number of bits ($8 \times L_{avg}$) per pattern of this module. The number of bits to

encode the address of storage table T_3 is $\lceil log_2 n_{avg} \rceil$, where $\lceil\ \rceil$ denotes rounding up. Finally, the average memory utilization U_{mem} is calculated as

$$U_{mem} = \frac{n_{avg} \times \lceil \log_2 n_{avg} \rceil + n_{avg} \times 8 \times L_{avg}}{2 \times m_{avg} \times \lceil \log_2 n_{avg} \rceil + n_{avg} \times 8 \times L_{avg}} \quad (4.21)$$

In Eq.(4.21), U_{mem} is the ratio of the patterns and their indices with the total sizes of T_1, T_2 and T_3. Fig. 4.11 shows the effect of U_{mem} based on load factor α. With the maximum value of α of 0.5, U_{mem} is approximately 0.88 when m_{avg} = 512, $n_{avg} \approx 500$ and $L_{avg} \approx 8$; we name it *PAMELA-1*. As α decreases by half to 0.25, U_{mem} slightly decreases and equals to 0.72 when m_{avg} = 1024; we name it *PAMELA-2*. To consider lower value of load factor, it is not interesting due to too many memory resources. Therefore, we only select these two systems for further testing. If the balance between the high flexibility of update and the area is the first priority then *PAMELA-2* is selected. Otherwise, hardware-efficient *PAMELA-1* is the choice. Next section will show the practical comparison of two systems.

Besides the memory utilization, the logic gate (logic cell) utilization is also of our interest. We assume that if a character needs p_{LC} logic gates for implementing hash function as in Fig. 4.1a then each pattern of length L characters needs $L \times p_{LC}$ logic gates. With L_{max_s} = 16 modules, the number of logic gates is very large for the general architecture. However, as in Fig. 4.1b, our optimized architecture only uses p_{LC} logic gates for each module. The notation R_{LC}, the ratio between logic gates of the general architecture and the optimized architecture, is

$$R_{LC} = \frac{\sum_{L=1}^{L_{max\_s}} (L \times p_{LC})}{L_{max\_s} \times p_{LC}} = \frac{L_{max\_s}(L_{max\_s}+1) \times p_{LC}}{2 \times L_{max\_s} \times p_{LC}} = \frac{L_{max\_s}+1}{2} \quad (4.22)$$

As L_{max_s} is 16 characters, the saving ratio R_{LC} is over 8 times for implementing the hash functions in our optimized architecture.

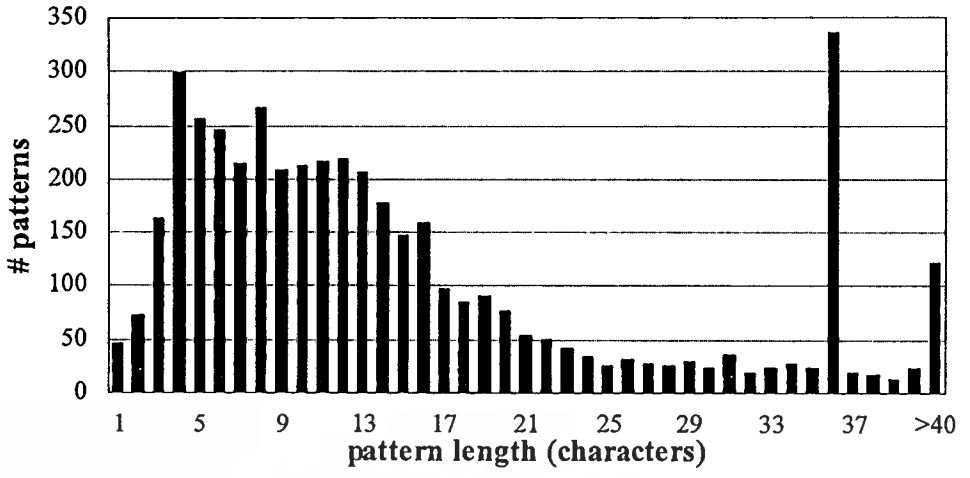


Figure 4.12: Pattern length distribution of pattern set of SNORT in Dec 2006.

4.2.2 Performance Simulations

As discussed in the previous subsection, we already showed the theoretical analysis of PAMELA. This subsection will present some simulation results of offline insertion of short and long patterns in design stage, on-line update of new patterns. The verification of hardware utilization and lookup throughput will be presented in section 4.3.

4.2.2.1 Off-line Insertion of Short Patterns

On Dec 15, 2006, there were 4,748 unique patterns equivalent to 64,873 characters in Snorts rule set. Fig. 4.12 shows the distribution of the pattern lengths in Snort database from 1 up to 109 characters. Fortunately, 65% of total numbers of patterns are up to 16 characters. Therefore, we build the Cuckoo Hashing modules for short patterns which are less than or equal to L_{max_s} = 16 characters according to this fact.

For practical comparison, we implement hash functions with patterns of the lengths from 2 to 16 characters. For pattern length of one character, we directly match the patterns to save the hardware. In all experiments from now on, the number of trials is 1,000. We define a parameter named %*Rehash* as the following equation to determine the possibility of rehash happening in every trial in every pattern length.

$$\%\mathrm{Re}hash = \frac{\text{the number of rehashes}}{\text{the number of trials}} \times 100\% \qquad (4.23)$$

Table 4.2 presents the number of insertions of Cuckoo Hashing with three hash functions: SAX (*shift-add-xor*), random table and CRC (cyclic redundancy check). Although CRC is a polynomials function, it is a suitable candidate for testing due to its significantly cheap implementation on hardware [29]. Three names, *SAX hard*, *Tab hard* and *CRC hard*, are the FPGA-based systems whose architectures are changed as in Section 4.1 of this paper with patterns inserted in balancing. In this first test, the size of index table is 256 for saving hardware resources. Remember that the theoretical Cuckoo Hashing requires the size of one table greater than the number of patterns. However, in this test, the maximum numbers of patterns in some pattern lengths are greater than 256. The practical insertions are still successful because the total size of two tables is 512 and greater than the maximum number of patterns. This experience also shows that the system can be updated over its capacity in the future. We can observe that %*Rehash* is high when the number of pattern is approximately equal to or greater than the table size.

Table 4.2 Comparison of the number of insertions of various hash functions. index table size is 256. The number of trials is 1000. CRC_hard, Tab_hard and SAX_hard are the FPGA-based systems

The length	No. patterns	SAX hashing		Tab hashing		CRC hashing		*SAX_hard*		*Tab_hard*		*CRC_hard*	
		#Ins	%Rehash	#Ins	%Rehash	#Ins	%Rehash	#Ins	%Rehash	#Ins	%Rehash	#Ins	%Rehash
2	72	83	0.0	79	0.0	83	0.0	75	0.0	73	0.0	80	0.0
3	162	187	0.0	193	0.0	223	3.5	167	0.0	164	0.0	169	0.0
4	299	757	236.4	671	210.5	1,598	236.8	678	109.5	550	155.3	751	181.3
5	255	530	84.7	414	72.3	838	75.6	420	59.7	359	27.8	407	97.0
6	245	404	17.2	389	5.3	407	41.1	295	16.8	289	2.0	298	20.2
7	214	321	54.3	345	54.6	382	42.0	259	2.0	232	5.1	263	2.5
8	266	468	75.0	441	73.0	667	82.5	361	36.0	415	14.4	447	75.2
9	208	305	5.7	280	10.8	301	0.0	235	0.0	248	11.1	283	2.7
10	213	305	51.9	299	24.6	361	19.4	242	7.0	249	18.0	264	8.6
11	217	333	0.0	331	13.1	470	27.9	238	0.0	252	7.0	259	0.0
12	218	300	7.2	302	0.0	339	6.7	232	0.0	242	0.0	252	1.3
13	206	289	47.1	295	17.9	287	1.6	228	0.0	224	0.0	235	3.0
14	178	232	0.0	232	0.0	238	2.6	182	0.0	183	1.0	193	2.5
15	147	174	0.0	173	0.0	157	2.0	159	0.0	151	0.0	166	0.0
16	159	200	0.0	199	0.0	201	0.0	170	0.0	173	0.0	185	0.0

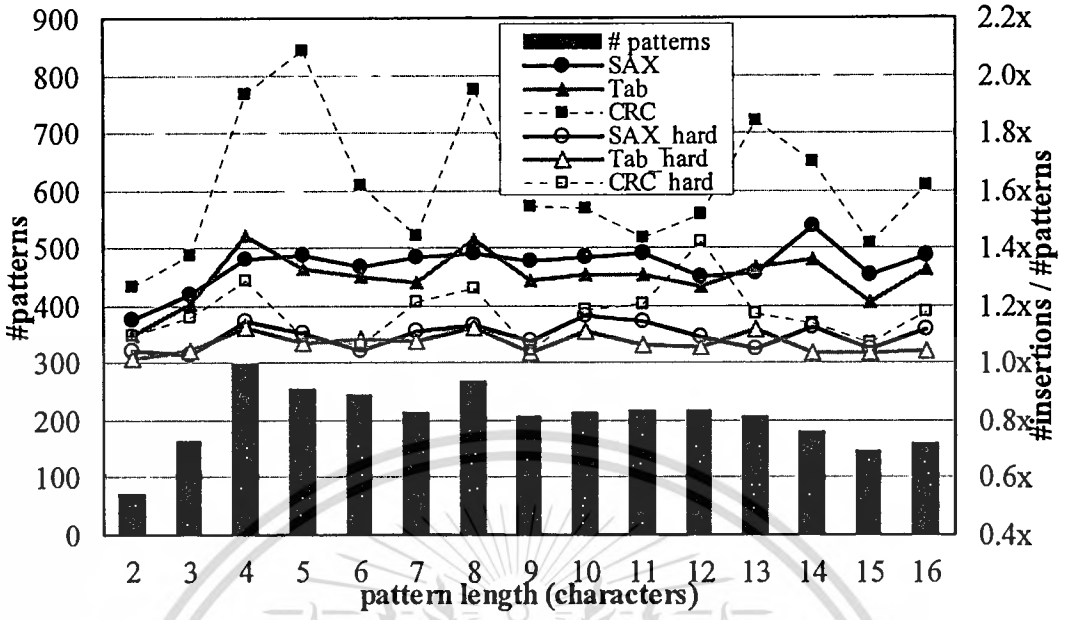


Figure 4.13: The number of insertions of various hash functions vs. pattern length (characters). Bar graphs are the numbers of patterns. Line graphs are the ratio of numbers of insertions and the numbers of patterns. Index (hash) table size is 512. The number of trials is 1,000

To decrease the number of insertions and the probability of rehashing, the size of index table is increased to double. The results in Fig. 4.13 show that the number of insertions of SAX and random table functions significantly decrease to very small %*Rehash* of less than 5%. On the other hand, the number of insertions of CRC hashing slightly decreases but it is still high as compared with the others. The results in Table 4.2 and Fig. 4.13 also show that the FPGA-based systems have the number of insertions less than the original systems by 20% and the performance of SAX hash function is close to that of random table. With the index table size of 512, the average load factor of index table is about 1/4. The remaining space can be used for fitting the segments of patterns with length of over 16 characters.

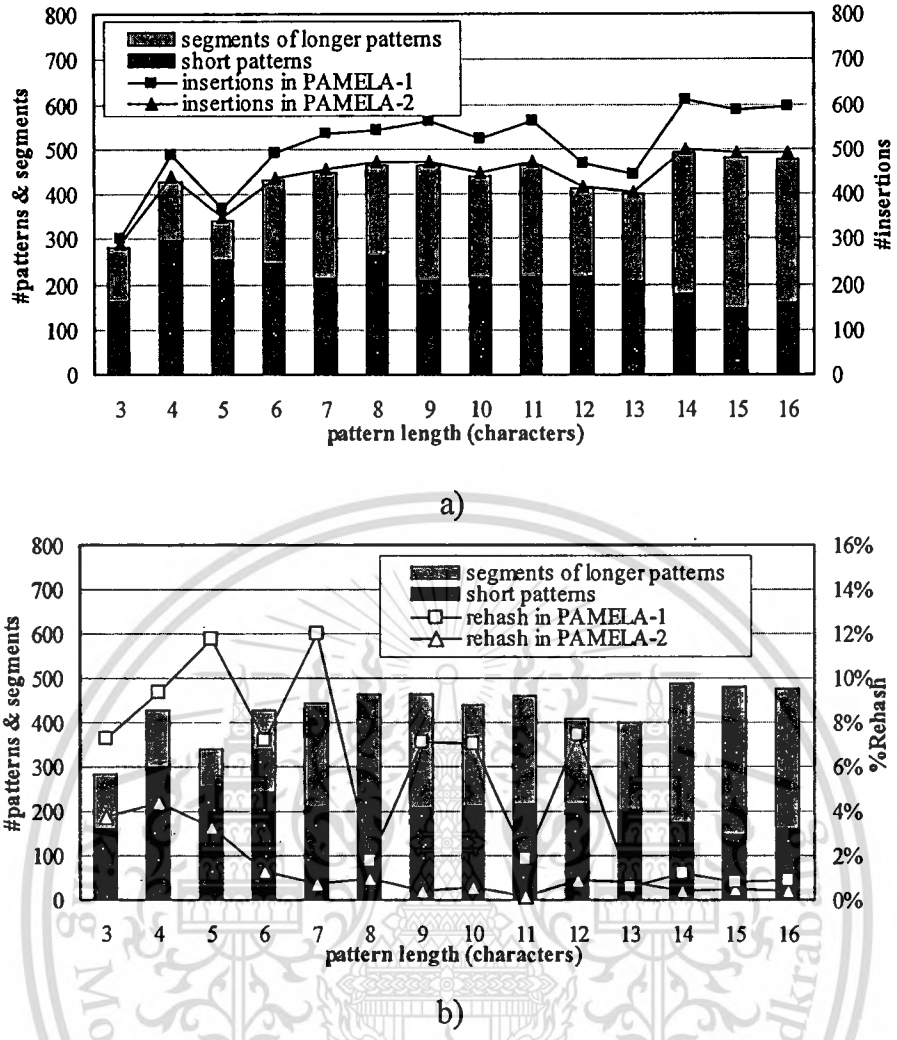


Figure 4.14: a) The number of insertions after addition of longer patterns vs. pattern lengths. b) %Rehash after addition of longer patterns vs. pattern lengths (*L*). *PAMELA-1* and *PAMELA-2* have the index table sizes of 512 and 1,024, respectively. Both systems are based on SAX hash function and our improved architecture. The number of trials is 1,000.

4.2.2.2 Off-line Insertion of Long Patterns

We break 1,643 long patterns of Snort rule set into over 3,500 segments of lengths from 3-16 characters as the described technique in Section 4.1.2. By sharing the prefixes with short patterns of the engine, the number of unique segments decreases by about 12%. Let a *string* denote either a short pattern or a segment of long pattern. Totally, there are 6,136 strings of pattern set. Note that the distribution of segments, *Th*, is also considered to make the string number less than 512 in every length. This

condition helps decrease the number of block RAMs of FPGA for implementing T_3 as well as T_1 and T_2.

The design can be parameterized with different table depths of 512 and 1,024 entries as *PAMELA-1* and *PAMELA-2*, respectively as defined in the previous subsection. These systems are used to evaluate the trade off between hardware utilization and performance of insertion. Both systems are based on SAX hash function and the FPGA-based Cuckoo architecture. Fig. 4.14a shows the number of insertions as well as the number of strings in every length after adding segments of long patterns. In every length of string, the number of insertions in *PAMELA-2* is just slightly greater than the number of strings. Unfortunately for *PAMELA-1*, the number of insertions is about 16% greater than the number of strings. %*Rehash*es of both systems are showed in Fig. 4.14b to explain why the number of insertions increases in *PAMELA-1*. Due to high memory utilization, the collision (%*Rehash*) increases as the number of patterns increases, approximately 12% as compared with 4% of *PAMELA-2*.

4.2.2.3 Dynamic Update for New Patterns

Snort rule database must be updated to handle the new attacks. Fig. 4.15 shows the number of pattern set from Jun, 04 to May, 07 [1]. It can be seen that the number of patterns and characters are doubled every 24 months. Normally, the web site Snort (www.snort.org) generates new rules every one or two weeks. As a result, PAMELA can also be rapidly and easily updated to avoid vulnerability.

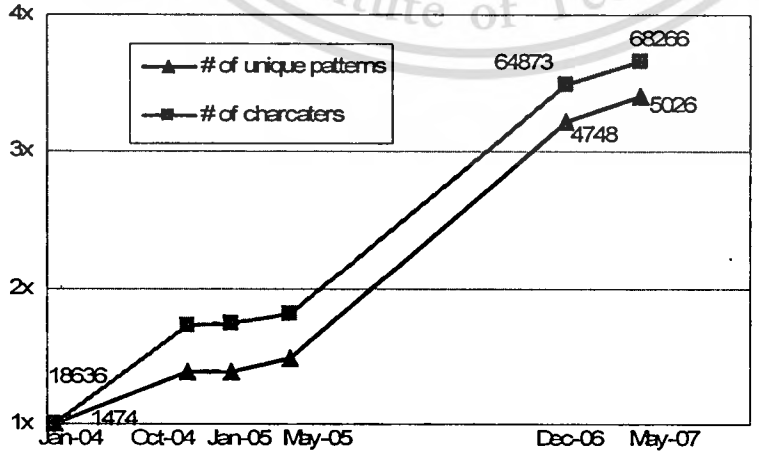


Figure 4.15: Growth of the SNORT rule set over the last two years.

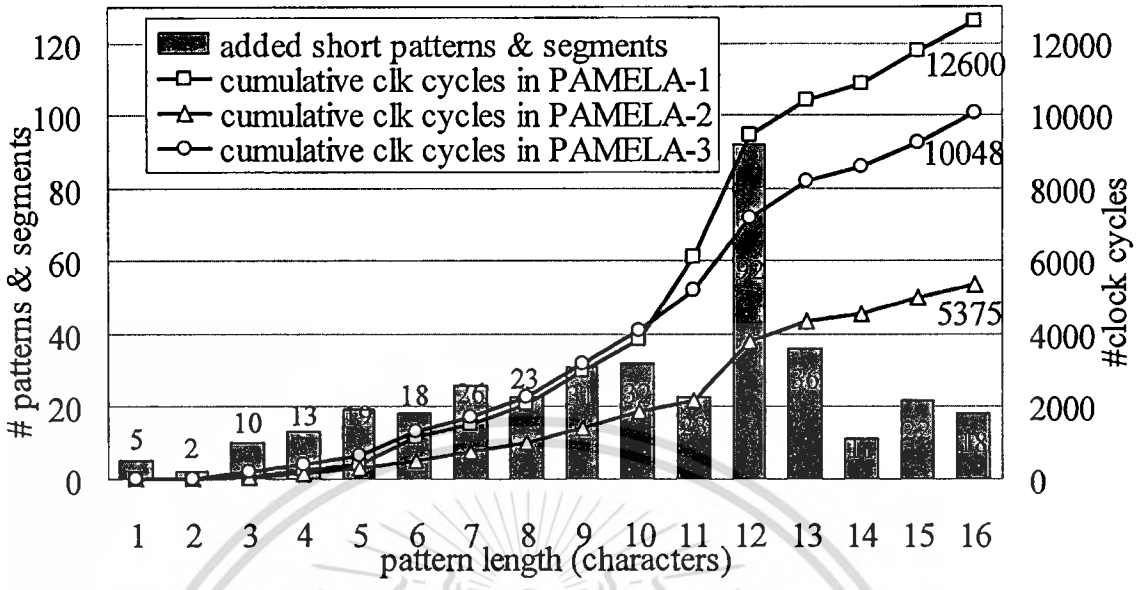


Figure 4.16: The average insertion time (clock cycles) for inserting 381 new strings (patterns & segments). *PAMELA-3* is extended from *PAMELA-1* by adding a stack and a FIFO for limited-time and uninterruptible update. The number of trials is 1,000.

To demonstrate the dynamic update ability of PAMELA, we accumulated the latest Snort pattern set on May 14, 2007 to 5,026 unique patterns and 68,266 characters. Based on the pattern set on Dec 15, 2006, there were 293 unique and new patterns consisting 3,476 characters and 15 unique patterns consisting 83 characters deleted. Note that all new patterns are searched in the system before inserting. Within these 293 new patterns, 65 patterns are longer than 16 characters. These long patterns are broken off-line into segments as described in Section 4.1.2. Finally, only 381 unique short patterns and segments are inserted.

The average insertion time in number of clock cycles of each PAMELA system is compared in Fig. 4.16. To determine M, we can assign α = 1, m_{avg} = 1,024 for all systems; thus the result of M is 30. Due to bigger table size of 1,024 entries, *PAMELA-2* takes 5,375 clock cycles to insert without rehashing. In *PAMELA-1*, the numbers of clock cycles at pattern lengths of 6, 11, 12 and 15 characters rapidly increase due to the high penalty of rehashing. With %*Rehash* from 0.6%- 2.9%, it takes 12,600 clock cycles to

insert that is about 2.5 times as compared with *PAMELA-2*. *PAMELA-3* is extended from *PAMELA-1* by adding a stack and a FIFO for limited-time and uninterruptible update.

According to Eq. (4.10) and (4.15), the sizes of the stack and FIFO are 90 and 3440 bytes, respectively. Some new patterns with lengths of 6, 11, 12 and 15 characters are broken into segments with lengths of 3, 5-8 characters as they are unsuccessfully inserted into the index tables. Then, the segments are inserted into the Cuckoo modules with corresponding lengths. Some segments with length of 6 characters are continually broken again due to high collision. Although the number of strings needed for insertion increases up to 390, the number of clock cycles including partitioning time in *PAMELA-3* decreases to 10,048. For 15 deleted patterns, the process is fast with around 100 clock cycles for each system.

We assume that *PAMELA-1* to *PAMELA-3* are clocked at 200 MHz which is less than our synthesized results shown later and can be achieved on many common Virtex FPGA boards. The update time ranging 5,475 to 12,700 clock cycles is about 28 to 64 microseconds (μs). Table 4.3 shows the comparison of the dynamic update time of some systems. Reference [23] shows that the system can update non-interrupting data stream by using a temporary module for updating only and the duration is less than one second for compiling and updating. Reference [10] uses a co-processor to update 30 new patterns with estimated time of 10 milliseconds (ms). Thus, the approximate time per pattern is 1/3 ms. We use no additional hardware in *PAMELA-1* and *PAMELA-2* of simulation systems. As a result, to add 293 new patterns, the average insertion time for a pattern is 19-43 clock cycles; and with added the delete time, the average time for updating one pattern is only 18-42 clock cycles, about 0.09-0.21 μs on 200 MHz FPGA systems. If the limited-time and uninterruptible update system is required then *PAMELA-3* is used; the insertion time of a new pattern is limited in 3440 clock cycles (17μs). These results show that PAMELAs are effective for updating pattern set and do not need lengthy FPGA reconfiguration time.

Table 4.3 Dynamic Update Comparison for A Pattern

System	Clock Freq.(Mhz)	Update time (μs)	Additional hardware requirement
PAMELA-1 (index table size:512)	200 (assumed)	0.22	No
PAMELA-2 (index table size:1024)		0.09	No
PAMELA-3 (extended from PAMELA-1)		0.17	A stack & FIFO for uninterruptible update
Bit-split AC [23]	N/A	~10^6	A temporary module
Rom+CoProc [10]	260	~1/3 x 10^3	A co-processor

4.3 FPGA Implementation Results of PAMELA

Our design is developed in Verilog hardware description language and by Xilinx's ISE 8.1i for hardware synthesis, mapping, and placing and routing. The target chips are some major Xilinx FPGA chips such as Virtex2, Virtex-Pro and Virtex-4. To decrease the number of memory blocks in FPGA, we can implement two index tables in the same block RAM; T_1 is in a low addresses part and T_2 is in a high addresses part. The block RAM of Xilinx FPGA can be configured as dual port mode that can be accessed concurrently. Therefore, the performance of the system still remains the same.

Based on our parallel pattern matching engine described earlier, we can measure the cost based on the numbers of block RAMs and logic cells in Table 4.4. With the maximum capacity of 18 kbits, block RAMs can be programmed as 18K x 1 bit to 512 x 36 bits, in various depth and width configurations. Hence, we can configure both tables T_1 and T_2 with size 1,024 x 9 bits for each as *PAMELA-2* mentioned above. Since the number of patterns in each Cuckoo module is less than 500, we can set the depth of T_3 as 512. The width of T_3 can be determined by the pattern length added 2 more bits for controlling long patterns. The width of T_4 is equal to the number of its address bits plus 1 bit for suffix segment of long pattern. For this reason, the size of T_4 is $2^{13} \times 14$. For the short patterns of one character, we can directly compare to save hardware resources.

The other parts of the system use logic cell of FPGA. While SAX functions use few resources of logic cells, the most consuming parts are the comparators. Totally, we use only 62 block RAMs and 3,220 logic cells to fit 68,266 characters of the entire rule set on the XCV4LX100 FPGA chip. To update without interrupting the incoming data, three more block RAMs are used to implement the stack and FIFO.

For multi-character processing, it can be seen in Table 4.4, the number of block RAMs for tables T_1, T_2 and T_3 increases corresponding to the number of characters processed per clock cycle. Since current Xilinx FPGA chips only have 2-port block RAM, we have to store duplicate the pattern in T_3. Table T_4 now can be used to process 2 lookups at the same time to save the resource of block RAMs. Some components such as the controls, counters, registers, LFSR, etc. are also shared inside the system. Therefore, the numbers of logic cells are 6,120 and 11,980 for 2-character scheme and 4-character scheme, respectively.

Table 4.4. Logic and Memory Cost of PAMELA on Xilinx Virtex-4

Component	1-character			2-character			4-character			Note for 1-character
	Quantity	Block RAMs	Logic cells	Quantity	Block RAMs	Logic cells	Quantity	Block RAMs	Logic cells	
Table T_1 & T_2	15x2	15	0	60	30	0	120	60	0	2 tables in a BRAM
Table T_3	16	39	0	31	77	0	62	154	0	The BRAM numbers for lengths 1-4 : 1; 5-8 : 2 9-13: 3; 14-16:4
Table T_4	1	8	0	1	8	0	2	16	0	
SAX hashes	15x2	0	840	60	0	1,680	120	0	3,360	
Multiplexer	15x2	0	300	60	0	600	120	0	1,200	
Comparator	32	0	1,320	64	0	2,640	128	0	5,280	16 for long patterns
Buffer, Priority, LFSR, etc.		0	760		0	1,200		0	2,140	
Total		**62**	**3,220**		**115**	**6,120**		**230**	**11,980**	

Table 4.5 shows the comparison of our synthesized systems with other recent FPGA systems. Two metrics, logic cells per character (LCs/char) and SRAM bits per character (bits/char), are used to evaluate the efficiency of FPGA utilization. For state machine approach, we just show some interesting implementations that are commonly used for static pattern matching only. We can see that the hashing systems [10]–[13] are almost better than state machine [6], [24] and compare-and-shift ones [7], [9] in term of hardware utilization.

As compared to V-HashMem [13], PAMELAs are 30% fewer in LCs/char and bits/char. Note that V-HashMem supports header matching with less hardware. On the contrary, with 6,500 strings inserted into the system, the storage capacity of synthesized PAMELA can support 1500 more strings without additional hardware resource. As compared to [10], [11], the Block RAM usage of our architecture is 1.22-2.07 times greater than theirs, but our logic cell usage is significantly smaller than theirs by 4.3-5.2 times. In summary, PAMELAs are the most efficient ones in using logic cells of FPGA at a cost of 0.047-0.175 LCs/char. In addition, the memory usage of our architecture is of high density (16.74-62.10 bits/char) and is acceptable as compared to other systems.

For throughput comparison, our throughput can vary from 1.78-8.8 Gbps depending on the models of FPGA chips and the number of characters being processed per clock cycle. Some works can also process multi-character so they have very high throughput up to 10 Gbps, especially shift-and-compare architectures. However, they have high area cost. Therefore, a Performance Efficiency Metric (PEM) was used in chapter 3 as the ratio of throughput in Gbps to the logic cell per each pattern character for performance evaluation.

$$PEM = \frac{Throughput}{\frac{No.Logiccells + \frac{Membytes}{12}}{No.Characters}} \qquad (4.24)$$

As PEMs in the range of 8.03-10.92, PAMELAs are the best of the FPGA-based hashing systems, far better than the shift-and-compare systems by at least two times, and better than the state machine systems over three times.

Table 4.5. Performance Comparison of FPGA-based Systems for NIDS/NIPS

System	Dev.-XC (Xilinx)	Bits/ cycle	Freq. (MHz)	No. chars	No. LCs	Mem (kbits)	LCs/ char	Mem/ char (bits)	Through-Put (Gbps)	PEM
PAMELA-2	4VLX100	8	285	68,266	3,220	1,116	0.047	16.74	2.28	10.29
	4VLX100	16	282		6,120	2,070	0.090	31.05	4.51	10.92
	4VLX100	32	275		11,980	4,140	0.175	62.10	8.80	10.70
	2VP20	8	272		3,266	1,116	0.048	16.74	2.18	9.79
	2V6000	8	223		3,266	1,116	0.048	16.74	1.78	8.03
	2V6000	16	218		6,212	2,070	0.091	31.05	3.49	8.42
PAMELA-2 (with a FIFO and a stack)	4VLX100	8	285	68,266	3,233	1,170	0.047	17.55	2.28	9.91
V-HashMem [13]	2VP30	8	306	33,613	2,084	702	0.060	21.39	2.49	8.60
HashMem [12]	2V1000	8	250	18,636	2,570	630	0.140	34.62	2.00	4.01
	2V3000	16	232		5,230	1,188	0.280	65.28	3.71	3.86
PH-Mem [11]	2V1000	8	263	20,911	6,272	288	0.300	14.10	2.11	4.71
	2V1500	16	260		10,224	306	0.490	14.98	4.16	6.44
ROM+Coproc[10]	4VLX15	8	260	32,384	8,480	276	0.260	8.73	2.08	5.90
Bit-Split FSM[24]	4FX100	8	200	16,715	4,514	6,000	0.270	184	1.60	0.39
Prefix Tree [9]	2VP100	8	191	39,278	12,176	0	0.310	0	1.53	4.93
PreD-CAM[7]	2V3000	8	372	18,036	19,854	0	1.100	0	2.98	2.70

Chapter 5

Conclusions and Future Works

5.1 Conclusions

This dissertation presents the designs and implementations of two efficient pattern matching engines developed on FPGA chips for Network Intrusion Detection System.

In the first engine, a novel linear systolic processor array architecture implementation on FPGA is proposed. The main contribution of this work is the deeper parallel/pipeline in processor array and compact encoding that can achieve high clock frequency and low cost area. According to our implementation result, the throughput is over 12Gbps with 4-byte processing. Compared to other logic gate based systems, our performance is the best.

The second architecture is a pattern matching engine based on Cuckoo hashing for NIDS/NIPS named PAMELA is proposed. To the best of our knowledge, our system is the first successful application of Cuckoo Hashing on hardware. PAMELA engine can update the new patterns rapidly in the orders of microseconds. PAMELA can also guarantee that the data stream cannot be interrupted during pattern updates by adding both small stack and FIFO. According to the implementation results, the performance of PAMELA is the best when compared with other previous systems and the achievable throughput can be up to 8.8 Gbits/s. The current scheme of PAMELA is also scalable enough to process N characters every clock cycle.

5.2 Future Works

The network intrusion detection/prevention is broad and never ending. Several issues of this dissertation should be improved. As future works for pattern matching using FPGA-based Cuckoo Hashing in NIDS, we can further optimize inside the system by removing some redundant elements to increase memory utilization. The utilization of memory in Cuckoo Hashing technique is still lower than 50% due to the limitation of the number of hash functions and the ports of memory. If the number of ports is bigger than

two, we can store several close string lengths in the same memory block. We can also increase the number of hash functions or the number of cells at each address. According to some improvements of Cuckoo Hashing [84, 93], the utilization of memory can be over 90%.

Another improvement direction can be explored that is combining two techniques: Cuckoo Hashing and compact encoding to save the pattern storage. Before the patterns are hashed and stored in memory, the compact encoding technique can be applied to decrease the bit number of each character. The incoming data is compacted as well before it enters the system. This combination requires the deep analysis in pattern set. Especially, the partitioning into the groups with different lengths and similar characters is more suitable for large pattern set such as CLAM-AV anti-virus [51] with over 80K patterns.

Due to the rapid increase of the number of regular expressions in Snort rule set, explore the possibility of mapping PERL regular expression string matching [94] is one of our interesting future works. This would become a better solution for deep packet inspection problem. In addition, IP header packet classification and matching can be combined to complete NIDS system. For example, hashing algorithms have been used in the past for packet classification. It would be interesting to investigate the efficiency of a modified Cuckoo Hashing algorithm for such a task.

This dissertation showed that reconfigurable hardware is suitable for pattern matching in NIDS. However, since PAMELA requires no reconfiguration at all, this engine can be applied on ASIC at much better than FPGA. ASIC is lower power consumption and higher throughput than FPGA.

Besides NIDS, our systems can be also applied for other applications in network security such as anti-virus CLAM-AV or other fields that use pattern matching as main task, for example image detection, handwriting recognition, text matching, computational biology, etc.

Bibliography

[1] http://www.snort.org, "SNORT official website,"

[2] Y.H. Cho, S. Navab, and W.H. Mangione-Smith, "Specialized hardware for deep network packet filtering," *Proceedings of the 12th International Conference on Field-Programmable Logic and Applications*, pp.452–461, 2002.

[3] I. Sourdis and D.N. Pnevmatikatos, "Fast, large-scale string match for a 10gbps FPGA-based network intrusion detection system," *Proceedings of the 12th International Conference on Field-Programmable Logic and Applications*, pp.880–889, 2003.

[4] J. Moscola, J. Lockwood, R.P. Loui, and M. Pachos, "Implementation of a content-scanning module for an internet firewall," *Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pp.31–38, 2003.

[5] Y.H. Cho and W.H. Mangione-Smith, "Deep packet filter with dedicated logic and read only memories," *Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pp.125–134, 2004.

[6] C.R. Clark and D.E. Schimmel, "Scalable pattern matching for high speed networks," *Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pp.249–257, 2004.

[7] I. Sourdis and D. Pnevmatikatos, "Pre-decoded CAMs for efficient and high-speed NIDS pattern matching," *Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pp.258–267, 2004.

[8] S. Yusuf and W. Luk, "Bitwise optimised CAM for network intrusion detection systems.," *Proceedings of 15th International Conference on Field-Programmable Logic and Applications*, pp.444–449, 2005.

[9] Z.K. Baker and V.K. Prasanna, "High-throughput linked-pattern matching for intrusion detection systems," *Proceedings of the 2005 symposium on Architecture for networking and communications systems*, pp.193–202, 2005.

[10] Y.H. Cho and W.H. M-Smith, "Fast reconfiguring deep packet filter for 1+ gigabit network.," *Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pp.215–224, 2005.

[11] I. Sourdis, D.N. Pnevmatikatos, S. Wong, and S. Vassiliadis, "A reconfigurable perfect-hashing scheme for packet inspection," *Proceedings of 15th International Conference on Field-Programmable Logic and Applications*, pp.644–647, 2005.

[12] G. Papadopoulos and D.N. Pnevmatikatos, "Hashing + memory = low cost, exact pattern matching," *Proceedings of 15th International Conference on Field-Programmable Logic and Applications*, pp.39–44, 2005.

[13] D. Pnevmatikatos and A. Arelakis, "Variable-length hashing for exact pattern matching," *Proceedings of 16th International Conference on Field-Programmable Logic and Applications*, pp.1–6, 2006.

[14] I. Sourdis, D.N. Pnevmatikatos, and S. Vassiliadis, "Scalable multigigabit pattern matching for packet inspection," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol.16, no.2, pp.156–166, 2008.

[15] R. Pagh and F.F. Rodler, "Cuckoo hashing," *Journal of Algorithms*, vol.51, no.2, pp.122–144, 2004.

[16] T.N. Thinh, S. Kittitornkun, and S. Tomiyama, "Applying Cuckoo hashing for FPGA-based pattern matching in NIDS/NIPS," *Proceedings of IEEE International Conference on Field-Programmable Technology (ICFPT)*, pp.121–128, 2007.

[17] Z.K. Baker and V.K. Prasanna, "Time and area efficient pattern matching on FPGAs," *Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, pp.223–232, 2004.

[18] B.L. Hutchings, R. Franklin, and D. Carver, "Assisting network intrusion detection with reconfigurable hardware," *Proceedings of the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pp.111–120, 2002.

[19] J.C. Bispo, I. Sourdis, J.M. Cardoso, and S. Vassiliadis, "Regular expression matching for reconfigurable packet inspection," *Proceedings of International Conference on Field-Programmable Technology*, pp.119–126, 2006.

[20] A. Mitra, W. Najjar, and L. Bhuyan, "Compiling PCRE to FPGA for accelerating SNORT IDS," *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*, pp.127–136, 2007.

[21] A.V. Aho and M.J. Corasick, "Efficient string matching: an aid to bibliographic search," *Communications of the ACM*, vol.18, no.6, pp.333–340, 1975.

[22] M. Aldwairi, T. Conte, and P. Franzon, "Configurable string matching hardware for speeding up intrusion detection," *SIGARCH Comput. Archit. News*, vol.33, no.1, pp.99–107, 2005.

[23] L. Tan and T. Sherwood, "A high throughput string matching architecture for intrusion detection and prevention," *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pp.112–122, 2005.

[24] H.J. Jung, Z.K. Baker, and V.K. Prasanna, "Performance of FPGA implementation of bit-split architecture for intrusion detection systems," *Proceedings of the Reconfigurable Architectures Workshop*, 2006.

[25] S. Dharmapurikar, P. Krishnamurthy, T.S. Sproull, and J.W. Lockwood, "Deep packet inspection using parallel bloom filters," *Symposium on High Performance Interconnects*, pp.44–51, 2003.

[26] L. Carter and M.N. Wegman, "Universal classes of hash functions.," *Journal of Computer and System Sciences*, vol.18, no.2, pp.143–154, 1979.

[27] M.V. Ramakrishna and J. Zobel, "Performance in practice of string hashing functions," *Proceedings of the Fifth International Conference on Database Systems for Advanced Applications*, pp.215–224, 1997.

[28] http://www.xilinx.com/bvdocs/appnotes/xapp211.pdf, "Xilinx application note,"

[29] J. Stone, M. Greenwald, C. Partridge, and J. Hughes, "Performance of checksums and CRC's over real data," *IEEE/ACM Transactions on Networking*, vol.6, no.5, pp.529–543, 1998.

[30] T. Sproull, G. Brebner, and C. Neely, "Mutable codesign for embedded protocol processing," *Proceedings of 15th International Conference on Field-Programmable Logic and Applications*, pp.52–56, 2005.

[31] M. Gokhale, D. Dubois, A. Dubois, M. Boorman, S. Poole, and V. Hogsett, "Granidt: Towards gigabit rate network intrusion detection technology," *In 12th Conference on Field Programmable Logic and Applications*, pp. 404–413, September 2002. Springer-Verlag.

[32] Y. H. Cho, S. Navab, and W. H. Mangione-Smith, "Deep network packet filter design for reconfigurable devices," *In 12th Conference on Field Programmable Logic and Applications*, pp. 452–461, September 2002. Springer-Verlag.

[33] F. Yu, R. H. Katz, and T. V. Lakshman, "Gigabit rate packet patternmatching using tcam," in *ICNP '04: Proceedings of the Network Protocols, 12th IEEE International Conference on (ICNP'04)*, (Washington, DC, USA), pp. 174–183, IEEE Computer Society, 2004.

[34] L. Bu and J. A. Chandy, "FPGA based network intrusion detection using content addressable memories," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 316–317, April 2004.Napa, CA, USA.

[35] J. Singaraju, L. Bu, and J. A. Chandy, "A signature match processor architecture for network intrusion detection.," in *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 235–242, 2005.

[36] Z. K. Baker and V. K. Prasanna, "Automatic Synthesis of Efficien Intrusion Detection Systems on FPGAs," in *14th International Conference on Field Programmable Logic and Applications*, pp. 311–321, 2004.

[37] Z. K. Baker and V. K. Prasanna, "Automatic synthesis of efficient intrusion detection systems on FPGAs.," *IEEE Trans. Dependable Sec. Comput.*, vol. 3, no. 4, pp. 289–300, 2006.

[38] C. R. Clark and D. E. Schimmel, "Efficient reconfigurable logic circuit for matching complex network intrusion detection patterns," in *13th International Conference on Field Programmable Logic and Applications*, pp. 956–959, 2003.

[39] Z. K. Baker and V. K. Prasanna, "A Methodology for Synthesis of Efficient Intrusion Detection systems on FPGAs," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 135–144, 2004.

[40] R. Sidhu and V. K. Prasanna, "Fast regular expression matching using fpgas," *In Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 227–238, April 2001.

[41] J. Botwicz, P. Buciak, and P. Sapiecha, "Building dependable intrusion prevention systems," in *DepCoS-RELCOMEX*, pp. 135–142, 2006.

[42] R. M. Karp and M. O. Rabin, "Efficient randomized pattern-matching algorithms," *IBM Journal of Ressearch and Development*, vol. 31, no. 2, pp. 249–260, 1987.

[43] B. H. Bloom, "Space/time trade-offs in hashing coding with allowable errors," in *Communications of the ACM,* Vol 13, no. 7, pp. 422–426, July 1970.

[44] S. Dharmapurikar and J. Lockwood, "Fast and scalable pattern matching for content filtering," in *ANCS '05: Proceedings of the 2005 symposium on Architecture for networking and communications systems*, (New York, NY, USA), pp. 183–192, ACM Press, 2005.

[45] S. Dharmapurikar and J. W. Lockwood, "Fast and Scalable Pattern Matching for Network Intrusion Detection Systems," *IEEE Journal on Selected Areas in Communications*, vol. 24, no. 10, pp. 1781–1792, 2006.

[46] H. Song and J. W. Lockwood, "Multi-pattern signature matching for hardware network intrusion detection systems," in *IEEE Globecom 2005*, (St. Louis, MO), pp. 1686–1690, Nov. 2005.

[47] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood, "Fast hash table lookup using extended bloom filter: An aid to network processing," in *ACM SIGCOMM*, (Philadelphia, PA), pp. 181–192, Aug. 2005.

[48] F. J. Burkowski, "A Hardware Hashing Scheme in the Design of a Multiterm String Comparator," *IEEE Transactions on Computers*, vol. 31, no. 9, pp. 825–834, 1982.

[49] E. Berk and C. Ananian, "Jlex: A lexical analyzer generator for java".

[50] R. Boyer and J. Moore, "A fast string match algorithm," *In Commun. ACM, vol. 20, no. 10*, pp.762–772, October 1977.

[51] Clam Antivirus. http://www.clamav.net.

[52] M. Fisk and G. Varghese, "Analysis of fast string matching applied to content based forwarding and intrusion detection," *In Techical Report CS2001-0670 (updated version)*, University of California- San Diego, 2002.

[53] Sourcefire. Snort 2.0 - detection revised. http://www.snort.org/docs/Snort 20-v4.pdf

[54] S. Wu and U. Mander, "A fast algorithm for multi-pattern searching," *In Techical Report TR-94-17*, University of Arizona, 1994.

[55] A. V. Aho and M. J. Corasick, "Efficient String Matching: An Aid to Bibliographic Search," *Communications of ACM*, vol. 18, pp.333-343, 1975.

[56] N. Tuck, T. Sherwood, B. Calder and G. Varghese, "Deterministic Memory-Efficient String Matching Algorithms for Intrusion Detection". In *Proceedings of the IEEE Infocom conference*, March 2004.

[57] C. J. Coit, S. Staniford, and J. McAlerney, "Towards Faster String Matching for Intrusion Detection or Exceeding the Speed of SNORT," *DARPA Information Survivability Conference and Exposi-tion (DISCEX II'01)*, 2001.

[58] E. P. Markatos, S. Antonatos, M. Polychronakis, and K. G. Anagnostakis, "Exb: exclusion-based signature matching for intrusion detection," *In Proceedings of CCN'02*, November 2002.

[59] K. G. Anagnostakis, E. P. Markatos, S. Antonatos, and M. Polychronakis, "E^2xb: a domain-specific string matching algorithm for intrusion detection," *In Proceedings of the 18th IFIP International Information Security Conference*, May 2003.

[60] S. Antonatos, K. G. Anagnostakis, E. P. Markatos, and M. Polychronakis, "Performance analysis of content matching intrusion detection systems," *In Proceedings of the International Symposium on Applications and the Internet*, January 2004.

[61] "Sun Managed Security Services: Multidimensional Defense-in-Depth," A Sun Microsystems White Paper. http://www.sun.com/service/managedservices/Man_Sec_WP11.15.pdf.

[62] Safenet inc. safexcel-4850 2004. http://www.safenet-inc.com/Library/3/SafeXcel 4850ProductBrief.pdf, "

[63] Cisco Inc., "Cisco pix 500 series firewalls," 2004.

[64] NetScreen Technologies Inc., "Netscreen-idp 10/100/500/1000 specifications," 2003.

[65] PMC Siera Inc., "Pm2329 classipi network classification processor datasheet," 2001.

[66] P. Helman and G. Liepins, "Statistical Foundations of Audit Trail Analysis for the Detection of Computer Misuse", *In IEEE Transactions on Software Engineering*, vol.9, , pp. 886–901, 1993.

[67] C. Warrender, S. Forrest, and B.A. Pearlmutter, "Detecting intrusions using system calls: Alternative data models", *In IEEE Symposium on Security and Privacy*, pp. 133–145, 1999.

[68] D.Wagner and D. Dean, "Intrusion Detection via Static Analysis", In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, IEEE Press, 2001.

[69] C. Ko, M. Ruschitzka, and K. Levitt, "Execution Monitoring of Security-Critical Programs in Distributed Systems: A Specification-based Approach", *In Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pp. 175–187, 1997.

[70] "Bro Intrusion Detection System." http://bro-ids.org/Overview.html.

[71] L. Schaelicke, T. Slabach, B. Moore, and C. Freeland, "Characterizing the performance of network intrusion detection sensors," *In Proceedings of the International Symposium on Recent Advances in Intrusion Detection*, 2003.

[72] S. Kim and Y. Kim, "A Fast Multiple String-Pattern Matching Algorithm", *Proceedings of 17th AoM/IAoM Conference on Computer Science*, 1999.

[73] S.Y. Kung, "VLSI Array Processors", (Englewood Cliffs, New Jersey:Prentice Hall,1988).

[74] T.N. Thinh and S. Kittitornkun, "Systolic Array For String Matching in NIDS" *Proceedings of 4th IASTED Asian Conference Communication Systems and Networks*, 2007.

[75] T.N. Thinh, S. Kittitornkun, and S. Tomiyama, "PAMELA: Pattern Matching Engine with Limited-time Update for NIDS/NIPS," *to appear in IEICE Transactions of Information and Systems*, Vol. E92-D, No.5, May 2009

[76] L. Tan and T. Sherwood, "A High Throughput String Matching Architecture for Intrusion Detection and Prevention," *Proc. 32nd Ann. Int'l Symp. Computer Architecture* (ISCA 05), IEEE CS Press, pp. 112-122, 2005.

[77] L. Tan and T. Sherwood, "Architectures for bit-split string scanning in intrusion detection," *IEEE Micro*, vol. 26, no. 1, pp. 110–117, 2006.

[78] B. C. Brodie, D. E. Taylor, and R. K. Cytron, "A Scalable Architecture for High-Throughput Regular-Expression Pattern Matching," in *33rd International Symposium on Computer Architecture (ISCA'06)*, pp. 191–202, 2006.

[79] "Intel® IXP2800 network processor." http://www.intel.com/design/network/products/npfamily/ixp2800.htm.

[80] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introduction to the Cell multiprocessor," *IBM J. RES. & DEV.*, vol. 49, July/September 2005.

[81] Willem de Bruijn, Asia Slowinska, Kees van Reeuwijk, Tomas Hruby, Li Xu, and Herbert Bos, "Safecard: A gigabit IPS on the network card", In *RAID*, pages 311–330, 2006.

[82] Herbert Bos and Kaiming Huang, "Towards software-based signature detection for intrusion prevention on the network card", In *RAID*, pages 102–123, 2005.

[83] Tomas Hruby, Kees van Reeuwijk and Herbert Bos, "Ruler: high-speed packet matching and rewriting on NPUs", *In Proceedings of the 2007 ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, ANCS 2007, pages 1–10, 2007.

[84] D. Fotakis, R. Pagh, P. Sanders and P. G. Spirakis, "Space Efficient Hash Tables with Worst Case Constant Access Time," *Theory of Computing Systems*, vol 38, no. 2, pp. 229-248, 2005.

[85] http://www.cert.org/stats/cert_stats.html

[86] Mike Fisk and George Varghese, "Applying Fast String Matching to Intrusion Detection" *Techical Report CS2001-0670* , University of California - San Diego, 2004.

[87] S. Antonatos, K. G. Anagnostakis, E. P. Markatos, and M. Polychronakis, "Performance Analysis of Content Matching Intrusion Detection Systems," *in Proceedings of the International Symposium on Applications and the Internet*, (Los Alamitos, CA, USA), pp. 208–218, 2004.

[88] Paulson, L.D.; "Stopping intruders outside the gates", *Computer*, Volume: 35 Issue: 11, pp. 20 -22, Nov. 2002.

[89] CSI/FBI Computer Crime and Security Survey SI Computer Security Issues & Trends 2002

[90] T.N. Thinh and S. Kittitornkun, "FPGA-based Cuckoo Hashing for Pattern Matching in NIDS/NIPS" *Proceedings of 10th Asia-Pacific Network Operations and Management Symposium, (APNOMS2007),* pp. 334-343, LNCS 4773, Springer-Verlag, 2007.

[91] Xilinx, Inc. http://www.xilinx.com.

[92] Altera, Inc. http://www.altera.com.

[93] Martin Dietzfelbinger and Christoph Weidling "Balanced allocation and dictionaries with tightly packed constant size bins" *Theoretical Computer Science*, Volume 380, Issues 1-2, pp. 47-68, June 2007.

[94] PCRE -Perl Compatible Regular Expressions, http://www.pcre.org/.

[95] Alan Siegel, "On universal classes of fast high performance hash functions, their timespace tradeoff, and their applications". *In Proceedings of the 30th Annual Symposium on Foundations of Computer Science (FOCS '89)*, pages 20-25. IEEE Comput. Soc. Press, Los Alamitos, CA, 1989.

Appendix A

Publication List

[1] T.N. Thinh, S. Kittitornkun, and S. Tomiyama, "PAMELA: Pattern Matching Engine with Limited-time Update for NIDS/NIPS," *to appear in IEICE Transactions of Information and Systems*, Vol. E92-D, No.5, May 2009.

[2] T.N. Thinh, S. Kittitornkun, and S. Tomiyama, "Applying Cuckoo hashing for FPGA-based pattern matching in NIDS/NIPS," *Proceedings of IEEE International Conference on Field-Programmable Technology (ICFPT)*, pp.121–128, 2007.

[3] T.N. Thinh and S. Kittitornkun, "FPGA-based Cuckoo Hashing for Pattern Matching in NIDS/NIPS" *Proceedings of 10th Asia-Pacific Network Operations and Management Symposium, (APNOMS2007)*, pp. 334-343, LNCS 4773, Springer-Verlag, 2007.

[4] T.N. Thinh and S. Kittitornkun, "Systolic Array For String Matching in NIDS" *Proceedings of 4th IASTED Asian Conference Communication Systems and Networks*, 2007.

IEICE TRANS. INF. & SYST., VOL.E92–D, NO.5 MAY 2009

PAPER

PAMELA: Pattern Matching Engine with Limited-time Update for NIDS/NIPS*

Tran NGOC THINH†, Surin KITTITORNKUN†, *Nonmembers*, *and* Shigenori TOMIYAMA††, *Member*

SUMMARY Several hardware-based pattern matching engines for network intrusion/prevention detection systems (NIDS /NIPSs) can achieve high throughput with less hardware resources. However, their flexibility to update new patterns is limited and still challenging. This paper describes a *PAttern Matching Engine with Limited-time updAte* (PAMELA) engine using a recently proposed hashing algorithm called *Cuckoo Hashing*. PAMELA features on-the-fly pattern updates without reconfiguration, more efficient hardware utilization, and higher performance compared with other works. First, we implement the improved parallel exact pattern matching with arbitrary length based on Cuckoo Hashing and linked-list technique. Second, while PAMELA is being updated with new attack patterns, both stack and FIFO are utilized to bound insertion time due to the drawback of Cuckoo Hashing and to avoid interruption of input data stream. Third, we extend the system for multi-character processing to achieve higher throughput. Our engine can accommodate the latest Snort rule-set, an open source NIDS/NIPS, and achieve the throughput up to 8.8 Gigabit per second while consuming the lowest amount of hardware. Compared to other approaches, ours is far more efficient than any other implemented on Xilinx FPGA architectures.

key words: Cuckoo Hashing, Dynamic Update, Pattern Matching, FPGA, NIDS/NIPS.

1. Introduction

Nowadays, illegal intrusion is one of the most serious threats to network security. Network Intrusion Detection/Prevention Systems (NIDS/NIPSs) are designed to examine not only the headers but also the payload of the packets to match and identify intrusions. Most modern NIDS/NIPSs apply a set of rules that lead to a decision regarding whether an activity is suspicious. For example, Snort [1] is an open source network intrusion detection and prevention system utilizing a rule-driven language. As the number of known attacks grows, the patterns for these attacks are made into Snort signatures (pattern set). The simple rule structure allows flexibility and convenience in configuring Snort. However, checking thousands of patterns to see whether it matches becomes a computationally intensive task as the highest network speed increases to several gigabits per second (Gbps).

To improve the performance of pattern matching in Snort, various implementations of field programmable gate array (FPGA) systems have been proposed. These systems can simultaneously process thousands of patterns relying on native parallelism of hardware so that their throughput can satisfy current gigabit networks. The drawback of hardware-based systems, however, is the flexibility. With emergence of new worms and viruses, the rule set must be frequently updated. For recently proposed FPGA-based NIDS/NIPSs [2]–[14], adding or subtracting a few rules requires recompilation (synthesizing, placing and routing) and reconfiguration of some parts or the entire design. Although reconfiguration is one of the advantages of SRAM-based FPGA; this process can take several minutes to several hours to complete. Today, such latency may not be acceptable for most networks when new attacks are released frequently. Moreover, the measure of throughput per area should be used in all hardware implementations. It is necessary to build a system that can achieve high-throughput per area with rapid rule set update.

Based on a novel Cuckoo Hashing [15], we implemented an FPGA-based architecture of variable-length pattern matching in our previous paper [16]. Unlike most previous FPGA-based systems, ours [16] can update the static pattern set without reconfiguration thanks to Cuckoo Hashing. In this paper, we propose a system named *PAttern Matching Engine with Limited-time updAte* (PAMELA) that extends and improves some disadvantages of [16]. Based on our analysis, the main drawback of Cuckoo Hashing is time consuming that can make the rule update time unlimited. We propose to use a stack to prevent rehashing and a FIFO to buffer the incoming data while updating the pattern set. Based on our theoretical analysis and simulation results, the insertion time of a new pattern is limited to 17 microseconds at 200 MHz clock frequency. As a result, a new rule set can be updated to PAMELA on line and on the fly. The power of high throughput process-

Manuscript received September 09, 2008.
Manuscript revised January 16, 2009.

†Tran Ngoc Thinh and Surin Kittitornkun are with the Department of Computer Engineering, Faculty of Engineering, King Mongkut's Institute of Technology Ladkrabang, Bangkok, 10520 Thailand (emails: tnthinh@cse.hcmut.edu.vn, kksurin@kmitl.ac.th)

††Shigenori Tomiyama is with the Department of Embedded Technology, School of Information and Telecommunication Engineering, Tokai University, Minato-ku Tokyo, 108-8619 Japan (email: tomiyama@dt.u-tokai.ac.jp)

*The previous version of this paper was presented at International Conference on Field-Programmable Technology 2007.

ing is addressed in this paper. With the extension for multi-character processing, the current system can sustain higher throughput than the previous one. Moreover, the linked-list method is improved to detect the continuous matches of long patterns; performance analysis, more experimental verification and comparison are carried out. With several improvements, PAMELA can save 30% of the area compared with the best system, and the throughput can achieve up to 8.8 Gbps for 4-character processing.

The rest of our paper is organized as follows. In Section 2, some previous hardware implementations of static pattern matching and Cuckoo Hashing are presented. Section 3 proposes the architecture of PAMELA engine. Next, performance analysis of PAMELA is discussed in Section 4. The extension for multi-character processing is mentioned in Section 5 before the experimental results on FPGA are presented in Section 6. Finally, future works are suggested in the conclusion.

2. Background and Related Works

2.1 Pattern Matching on NIDS/NIPS

For a line speed of gigabit network, a variety of FPGA approaches of NIDS/NIPS have been proposed, for example: shift-and-compare; state machine such as Nondeterministic/Deterministic finite automata (NFA/DFA), Aho-Corasick algorithm; and finally hashing. Firstly, some of the shift-and-compare method are [2], [3]. They apply parallel comparators and deep pipelining on different, partially overlapping, positions in the incoming packet. The simplicity of the parallel architecture can achieve high throughput when compared to software approaches. The drawback of these methods is the high area cost. To reduce the area cost and achieve a high clock rate, many improvements are proposed. The work [5] is extended from [2] to share common substrings. Predecoded shift-and-compare architectures ([7], [17]) convert the incoming characters to bit lines to reduce the size of comparators. A variation in tree-based optimization ([8], [9]) divides the pattern set into partitions to share similar characters resulting in excellent area performance.

The next approach exploits state machines (NFAs/DFAs) [4], [18]. The state machines can be implemented on hardware platform to work all together in parallel. By allowing multiple active states, NFA is used in [18] to convert all the Snort static patterns into a single regular expression. Moscola et al. [4] recognized that most of minimal DFAs content fewer states than NFAs, so their modules can automatically generate the DFAs to search for regular expressions. Like the shift-and-compare implementations, the predecoded method is also used in [6] to improve area performance of NFAs. The main advantage of regular expression format as compared with static pattern one is that a single regular expression can describe a set of static patterns by using meta-characters with special meaning. As a result, recently, a special format of regular expressions such as Perl Compatible Regular Expressions (PCRE) is added in Snort [1] instead of static patterns, and some new works tried to improve on PCRE matching [19], [20]. However, most of these systems suffer scalability problems, i.e. too many states consume too many hardware resources and long reconfiguration time.

Another approach of the state machine method used for static pattern matching is the Aho-Corasick algorithm [21]. By modifying this algorithm on hardware, the implementations in [22]–[24] can get high performance. Aldwairi et al. [22] partitioned the rule set into small ones according to the type of attacks in Snort database. The state machine in [23] is split into smaller FSMs which can run in parallel to improve memory requirements. This bit-split FSM can fit over 12k characters of Snort rule set to 3.2 Mbits memory at 10 Gbps on ASIC implementation and can update new rules in the order of seconds with no interruption. Nonetheless, its FPGA implementation [24] can achieve lower throughput rate while using larger memory.

Finally, hashing approaches [10]–[14] can find a candidate pattern at constant look up time. The authors in [11], [14] use perfect hashing for pattern matching. Although their system memory usage is of high density, the systems require hardware reconfiguration for updates. Papadopoulos et al. proposed a system named HashMem [12] system using simple CRC polynomials hashing implemented with XOR gates that can use efficient area resources than before. For the improvement of memory density and logic gate count, they implemented V-HashMem [13]. Moreover, V-HashMem is extended to support the packet header matching. However, these systems have some drawbacks: 1) To avoid collision, CRC hash functions must be chosen very carefully depending on specific pattern groups; 2) Since the pattern set is dependent, probability of redesigning system and reprogramming the FPGA is very high when the patterns are being updated; 3) By using glue logic gates for simplicity, the long patterns processing is also ineffective for updating.

On the other hand, Dharmapurikar et al. propose to use Bloom Filters to do the deep packet inspection [25]. Unlike other hashing approaches mentioned above, the pattern update process can be done easily without reprogramming FPGA. A Bloom Filter with multiple hash functions up to 35 probes is used to check whether or not a pattern is member of the set. Nevertheless, its main problem is due to false positive matches, which requires extra cost of hardware to confirm the match.

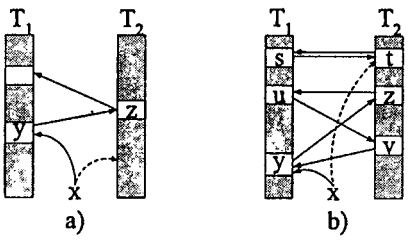


Fig. 1 Original Cuckoo Hashing [15] a) A pattern x is successfully inserted by moving y and z. b) A pattern x cannot be accommodated and a rehash is required.

2.2 Cuckoo Hashing

Cuckoo Hashing is proposed by Pagh and Rodler [15] as an algorithm for maintaining a dynamic dictionary with constant lookup time in the worst case scenario. The algorithm utilizes two tables, T_1 and T_2, of $m = (1+\epsilon)n$ cells each, where ϵ is some constant ($\epsilon > 0$), n is the number of elements (strings). Cuckoo Hashing does not need perfect hash functions that is very complicated if the set of stored elements dynamically changes under the insertion and deletion. Given two hash functions, h_1 and h_2, mapping from universe U to $[m]$, a pattern x can be exactly stored in either cell $T_1[h_1(x)]$ or $T_2[h_2(x)]$ but not both. So, lookup of x requires at most two positions. Deletion procedure can also run in worst case constant time.

Pagh and Rodler described a procedure to insert a new pattern x in expected constant time. For the first time, x is always placed into cell $T_1[h_1(x)]$. If this cell is empty, the insertion is complete; if it is occupied by a pattern y which necessarily satisfies $h_1(x) = h_1(y)$, then x is put in cell $T_1[h_1(x)]$ anyway, and y is kicked out. Then, y is put into the cell $T_2[h_2(y)]$ of the second table similarly, which is also possibly occupied by another pattern z with $h_2(y) = h_2(z)$. In this case, z is placed in cell $T_1[h_1(z)]$, and the process repeats until the pattern can be placed in an empty cell as in Fig. 1a. However, it can be seen that the "cuckoo process" may not terminate as Fig. 1b. As a result, the number of iterations is limited by a bound M chosen beforehand. In this case every entry must be rehashed with two new hash functions.

3. FPGA-based Pattern Matching Engine in NIDS/NIPSs using Cuckoo Hashing

For pattern matching in NIDS/NIPS, the patterns are searched on the incoming data (packets). The matched pattern can occur anywhere as the longest substring. This problem is called multi-pattern matching with a set of patterns $P = \{p_1, p_2, \ldots, p_n\}$ and the incoming data T. Normally, the pattern set is preprocessed and built in a system. The incoming data is entered into a FIFO to compare with all of patterns. As a result,

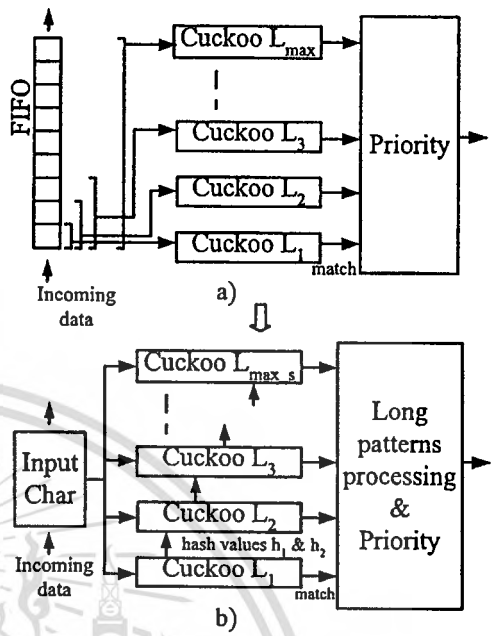


Fig. 2 FPGA-based Pattern Matching Engine in NIDS/NIPSs using Cuckoo Hashing. a) General Model b) Optimized Hardware Model

Cuckoo Hashing is a good candidate for multi-pattern matching with constant lookup time. Furthermore, dynamic update for the pattern set does not affect the performance of lookup. Fig. 2a shows an overview of multi-pattern matching engine using FPGA-based Cuckoo Hashing. Each module named *Cuckoo* L_i can process patterns of i characters long.

In order to process at the network speed in Gbps, we have to construct Cuckoo Hashing module for every pattern length from $L_{min} = 1$ up to L_{max} characters. The priority circuit then selects the longest pattern if multi matches happen. However, L_{max} can grow up to hundreds in most of NIDS/NIPS systems. Thus, we build the Cuckoo Hashing modules for short patterns with the maximum length L_{max_s} according to the most distribution of patterns in NIDS/NIPS. In Snort pattern set, L_{max_s} gets the best value of 16 characters. For longer patterns, we can break them into shorter segments so that we can insert those segments to the Cuckoo modules of short patterns. We then use simple address linked-lists to combine these segments later. Fig. 2b shows our optimized architecture for pattern matching.

3.1 FPGA-Based Cuckoo Hashing Module

In order to increase memory utilization, we build up a hashing module for each pattern length and use indirect storage. Small and sparse hash tables contain indices of patterns which are the addresses of a condensed pattern-stored table. Our approach is also to

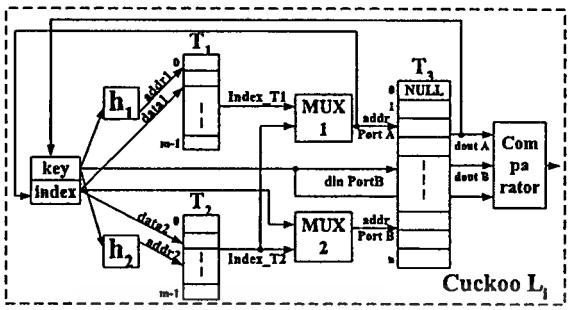


Fig. 3 FPGA-based Cuckoo Hashing module with parallel lookup. Tables T_1 and T_2 store the key indices; Table T_3 stores the keys.

```
function lookup(x)
  select index_T1 in MUX1  and index_T2 in MUX2;
     index_T1 = T1(h1(x));                    // phase1+2
     index_T2 = T2(h2(x));
     doutA = PortA(index_T1);                 // phase3
     doutB = PortB(index_T2);
     return(doutA = x or doutB = x); // phase4
end
```

Fig. 4 Pseudo-code of Parallel Cuckoo Lookup Algorithm

```
procedure insert(x)
    if (lookup(x)) return;
    select index_T1 in MUX1  and index in MUX2;
    PortB(index) = x;
    if(index_T1 == NULL){
        T1(h1(x))= index;    return;}
    else if(index_T2 == NULL){
        T2(h2(x))= index;    return;}
    select index_T1 or index_T2 in MUX1
        //depending on balance of hash table
    loop MaxLoop times  // "Cuckoo process"
        if (select index_T1 in MUX1){
            key = PortA(index_T1);
            index = index_T1;
            select index_T2 in MUX1;}
        else{ // (select index_T2 in MUX1)
            key = PortA(index_T2);
            index = index_T2;
            select index_T1 in MUX1;}
        index_T1 = T1(h1(x));
        index_T2 = T2(h2(x));
        if (select index_T1 in MUX1){
            if(index_T1 == NULL) return;
            doutA = PortA(index_T1);}
        else{ // (select index_T2 in MUX1)
            if(index_T2 == NULL) return;
            doutA = PortA(index_T2);}
    end loop
    rehash(); insert(x);
end
```

Fig. 5 Pseudo-code of Parallel Cuckoo Insertion Algorithm

change the lookup to parallel processing. The insertion can be changed for better selection of the available space in both hash tables. With some improvements in architecture, ours can take full advantages of hardware.

The architecture of a FPGA-based Cuckoo Hashing module as shown in Fig. 3 consists of three tables. Two index tables (hash tables) T_1 and T_2 are single-port SRAMs and a pattern-stored table T_3 is a double-port SRAM for concurrent processing. Hash functions can be changed if they are required to rehash. Two registers named *key* and *index* are the pattern to be searched for and the memory address of that pattern in T_3, respectively. Besides, two multiplexers are used to select addresses of T_3. Finally, a comparator is used for exact matching of *key* with two candidate patterns from T_3.

3.1.1 Parallel Lookup of An Incoming Pattern

By using parallel and multi-phase pipeline architecture, PAMELA can look up patterns every clock cycle. Fig. 4 is the pseudo-code of parallel Cuckoo lookup function. A pattern x is hashed by two hash functions concurrently. Then, the values of two hash functions are used as the addresses of two index tables. After that, the outputs of two index tables are used as the addresses of T_3. Finally, to determine the match, the outputs of T_3 are compared with x.

3.1.2 Dynamic Pattern Insertion and Deletion

The insertion of a new pattern x can be described in Fig. 5. We have some improvements as compared with the original Cuckoo Hashing. We consider both tables to reduce the insertion time. If one of the outputs of two index tables is empty (NULL), the index of x is inserted into T_1 or T_2 and the insertion is complete. If the outputs of both T_1 and T_2 are not NULL, we insert the *index* into the table with less number of patterns. At the same time, the "kicked-out" *index*_T_1 (*index*_T_2) and its data from T_3 will be written into the *index* and *key* registers to start the hashing process. Then, M, $\lceil 3log_{1+\epsilon}m \rceil$ [15], is decreased and the key value is hashed by hash function h_2 (h_1). The output data will be checked for whether the value is NULL. If it is NULL, the process ends with successful insertion. On the other hand, the process is continued by taking in turns hashing from h_2 (h_1) to h_1 (h_2). The worst case happens when M decreases to zero. Hence, a rehash is required. Two new hash functions h_1 and h_2 are issued by a pseudo-random number generator.

For deletion, the algorithm in Fig. 6 is as simple as the lookup process. If the lookup succeeds, $MUX1$ will select exactly one of the outputs of two index tables to write into the index register. We then write NULL into table T_3 at the address pointed by the index register. After that, we reset the index register to NULL and write it into appropriate T_1 or T_2. The deletion process results in some "holes" in T_3. When the number of "holes" is greater than a threshold, the rehash for rearranging of T_3 can be implemented.

3.1.3 Recommended Hash Function

The hash function of choice greatly affects the perfor-

```
procedure delete(x)
    if (!lookup(x)) return;
    key = NULL;
    index = index_T1 or index_T2;
          //depending on the lookup result
    PortB(index) = key;
    T1(h1(x))= NULL or T2(h2(x))= NULL;
    return;
end
```

Fig. 6 Pseudo-code of Parallel Cuckoo Deletion Algorithm

mance of the system. A fast way of generating a class of universal hash function that is hardware-friendly and tabulation based method [26], is defined as follows:

$$H_t(x) = a_t[0][x_0] \oplus a_t[1][x_1] \oplus \ldots \oplus a_t[n-1][x_{n-1}] \quad (1)$$

A table contains a 2-D array of random numbers in the hashing space. A key x is a string of n characters, $x_0 x_1 .. x_{n-1}$, and the hash value is calculated by bit-wise exclusive-or ($\oplus$) a sequence of values $a_t[i][x_i]$, which is indexed by each byte value of x_i and position of i in the string. The drawback is that the size of random table is very large and depends on the key length.

Another class of simple hash function for hashing character strings named *shift-add-xor* (SAX) [27] utilizes only the simple and fast operations of shift, exclusive-or and addition.

$$H_i = H_{i-1} \oplus (S_L(H_{i-1}) + S_R(H_{i-1}) + c_i) \quad (2)$$

Two operators S_L and S_R denote the shift left and right, respectively. The symbol c_i is the character i^{th} of string and H_i is an intermediate hash value after examination of i characters. The initial value H_0 can be generated randomly. The main advantage of SAX as compared with random-table is very few hardware resources required. To generate the new SAX hash function in case of rehashing, we only need to change the value of H_0 by a simple pseudo-random circuit LFSR [28]. Therefore, SAX is a suitable choice for PAMELA.

3.1.4 Hardware Optimization for Cuckoo Module

We can significantly reduce large amount of hardware by exploiting accumulative characteristic of SAX hash function. From Eq. (2), to calculate hash value of an incoming pattern with length i characters in the hash module i^{th}, the requisite inputs are the hash values of $i-1$ characters calculated beforehand in the hash module $(i-1)^{th}$ and the i^{th} character. Therefore, the values of previous hash module can be reused for the next hash module. As shown in Fig. 2b, two hash values h_1 and h_2 of *Cuckoo* L_{i-1} are fed into *Cuckoo* L_i $(2 < i \leq L_{max\_s})$. In addition, the FIFO for the incoming data is also replaced by only a one-character shift register.

Nevertheless, our hardware optimization can increase the probability of rehash on the system. For example, when a rehash by collision happens at Cuckoo module i, new functions h_1 and h_2 of module i make the inputs of module $i+1$ changed and the hash values of module $i+1$ will be incorrect. The process is going on recursively up to module L_{max_s}. As a result, the rehash can be forced from Cuckoo module $i+1$ to module L_{max_s}. This thing, however, only affects the insertion process. This trade off is good enough because the first priority is fast lookup with smaller hardware.

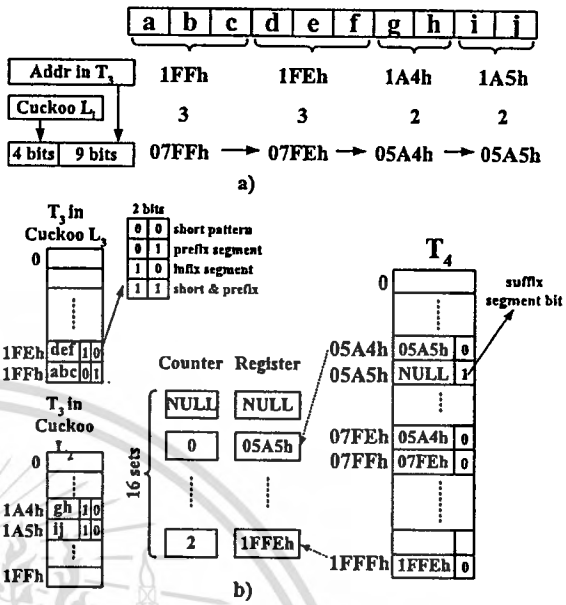


Fig. 7 Matching long patterns. a) Example of breaking a long pattern "abcdefghij". b) How to store a long pattern in table T_4 as a linked-list

3.2 Matching Long (>16-Character) Patterns

We break the longer (> 16-Character) patterns into variable-length segments of 1 to $L_{max\_s} = 16$ characters. The above Cuckoo Hashing modules can then be used for matching these individual segments. After that, these segments of a long pattern are combined to a chain that can be implemented by simple linked-list technique. The data structure for storing linked-lists is a table named T_4 whose depth is the depth of T_3 multiplied by the number of Cuckoo modules. Each address represents one segment and its content is an address of next segment in the same long pattern.

For more details, we describe the technique using a simple example. We assume that string *"abcdefghij"* is a long pattern that is broken into smaller segments with proportion 3 : 3 : 2 : 2 as in Fig. 7a. Segments 1 and 2 are hashed and stored in T_3 of *Cuckoo* L_3 at address $1FFh$ and $1FEh$ while segments 3 and 4 are hashed and stored in T_3 of *Cuckoo* L_2 at address $1A4h$ and $1A5h$. We combine the addresses in $T_3 s$ together with the number of Cuckoo modules to link these individual segments. If the depth of T_3 in every module

```
procedure insert_long(x)
    sh: short pattern in table T3;
    prefix(k,x): prefix with length k of x;
    suffix(k,x): suffix with length k of x;
    // lookup the longest prefix of x
    if(lookup(x)) return;
    if(lookup(sh=prefix(k,x)) and sh without "011"){
        //short & prefix segment
        change the end of sh to "011";
        call part(suffix(k(x-sh),x),"010","110");}
    else
        call part(x,"001","110");
end

procedure part(x,pre_id,suf_id)
    L: length of x;
    #T3(L): #patterns in T3 of length L;
    Th_L:threshold of length L;
    if(L > L_max_s or #T3(L) > Th_L or lookup(x)){
        part(prefix(L-L/2,x),pred_id,"010");
        part(suffix(L/2,x),"010",suf_id);}
    else{
        if(pre_id=="001")       //prefix segment
            insert(x) with the end "001";
        else if(suf_id=="110") //suffix segment
            insert(x) with the end "110";
        else                    //infix segment
            insert(x) with the end "010";}
end
```

Fig. 8 Pseudo-code of Long Pattern Insertion Algorithm

and the number of Cuckoo modules are 512 and 16 then the bit number is 13 for representing a position of individual segment: 9 least significant bits for representing the address of T_3 and 4 most significant bits for representing the number of Cuckoo modules. The segment addresses of *"abcdefghij"* in T_4 are $7FFh$, $7FEh$, $5A4h$, and $5A5h$ as shown in Fig. 7b. The matching process is described as the following. When we get a match for segment *"abc"*, we have address $7FFh$ whose content in T_4 is $7FEh$. After 3 clock cycles, if the mach segment is *"def"*, the process is continued by jumping from $7FFh$ to $7FEh$. Otherwise, the detective process finishes without match and resets for other patterns. Similarly, from $7FEh$, if the next match segment after 2 clock cycles is *"gh"*, address $5A4h$ is considered. Finally, if the last segment *"ij"* is detected 2 next clock cycles later then the content of address $5A5h$ is read and the match of this pattern is reported.

Two more bits are added in every entry of T_3 as in Fig. 7b. They are used to describe a string which can be the short pattern, the first segment of long pattern called *prefix segment*, the body of long pattern called *infix segment*, or even the short pattern which is also the prefix segment. In case of the end of long pattern called *suffix segment*, we can determine it by checking the content of its address in T_4 whether it is NULL. However, if a long pattern is a prefix substring of another one then it cannot be detected. For example, a long pattern *"abcdef"* can be the prefix substring of *"abcdefghij"*. To improve this case, we add one more bit in table T_4. When this bit is active, a suffix segment can also be an infix segment of another pattern. In other words, the content of a suffix segment in T_4 can be a pointer to another address.

The above paragraphs describe the method for holding one long pattern per time. Each time, if the system detects one segment with length L, it has to wait for the next segment in L clock cycles. During this time, if other matches happen then the system cannot detect them. For example, the incoming data is *"...abcdefghijk..."* and we have 2 long patterns *"abcdmnpq"* and *"bcdefghi"* in the pattern set. We assume that these patterns are broken into segments *"abcd"*, *"mnpq"* and *"bcde"*, *"fghi"*, respectively. In clock cycle t, if the matched string is *"abcd"* then the system waits for the next match in 4 clock cycles. However, the next match for the first pattern does not happen in this case. Meanwhile, in clock cycle $t+1$, the system cannot detect the segment *"bcde"* that is the prefix for the second pattern.

To detect all segments of long patterns while matching in every clock cycle, we use $L_{max\_s} = 16$ down counters and registers for storing the match values read from T_4; the length of segment and its content in T_4 are stored in an available counter and register, respectively. When a segment match happens and it is a prefix segment, its position in T_4 is considered. If it is an infix or a suffix of current long pattern candidates, it must be simultaneously compared to all registers whose counters are currently zero values to determine the unique address of T_4. Because the length of segments can reach at L_{max_s}, the maximum number of current long pattern candidates is L_{max_s}.

We use an automatic generator to partition the long patterns in pattern set. Fig. 8 is the pseudo-code of long pattern insertion. First, we consider long patterns as the incoming strings pass through the engine for lookup. If a short pattern in the engine is the longest prefix of a candidate long pattern then we break the long pattern such that its first segment is the same length as the short pattern. We mark the short pattern which is also the prefix segment to avoid checking again. After that, we break the rest of the long pattern into halves if the length of the rest is greater than L_{max_s} or the number of patterns in the same table T_3 is greater than a certain threshold Th, somewhat arbitrarily set less than the size of hash table, to avoid the high probability of rehash. Otherwise, the rest can be a suffix segment. The partitioning can continue if any segment matches with a string in the engine.

4. Performance Evaluation: Theoretical Analysis and Simulation

4.1 Theoretical Analysis

In this subsection, we will analyze the time to lookup and insert a pattern. Based on the worst case scenario,

Table 1 Summary of main notations used in the performance analysis

Symbol	Units	Description
L	bytes	The length of a pattern
T_{hash}	cycles	The time to calculate the hash values
T_{mem}	cycles	The time to access any memory table
T_{proc_L}	cycles	The time to process a pattern of length L
T_{mtch}	cycles	The time of the comparison
T_{lu_L}	cycles	The lookup time of a pattern
T_{in_L}	cycles	The insertion time of a pattern
$T_{in\_w_L}$	cycles	The worst case insertion time
T_{lu_long}	cycles	The time for lookup of a long pattern
T_{in_long}	cycles	The time for insertion of a long pattern
S_S	bytes	The size of the stack
$T_{in\_u_L}$	cycles	The unsuccessful insertion time
T_{part_L}	cycles	The online partitioning time
S_F	bytes	The size of the FIFO
U_{mem}		The average memory utilization
R_{LC}		The ratio logic gates of the general architecture and the optimized architecture

we propose a method for bounding insertion time of a new pattern. In addition, two common area metrics of SRAM-based FPGA, memory and logic gate utilization are also explored to show the efficiency of PAMELA engine. To facilitate the theoretical analysis, some main notations are listed in Table 1.

4.1.1 Lookup time and insertion time

The time to process a pattern of length L (characters or bytes) is a function of the number of cycles needed to calculate the hash values T_{hash}, and the number of cycles needed to access memory including hash tables $T_{mem_{1,2}}$ and storage-pattern table T_{mem_3}. We assume that the time needed to access every memory table is constant and $T_{mem_{1,2}} = T_{mem_3} = T_{mem}$. Our engine takes one character per clock cycle, so $T_{hash} = L$. The equation for the processing time T_{proc_L} is expressed as

$$T_{proc_L} = T_{hash} + 2T_{mem} = L + 2T_{mem} \quad (3)$$

For pattern lookup, we need one more stage to compare the candidate pattern after reading out of table T_3 with the part of the incoming data. With the time of the comparison T_{mtch}, the equation for the lookup time is calculated as follows.

$$T_{lu_L} = T_{proc_L} + T_{mtch} = L + 2T_{mem} + T_{mtch} \quad (4)$$

The insertion time T_{in_L} of a pattern x with length L can include some shuffles of other patterns in the hash table. Let h denote the number of shuffles then the average time of successful insertion is expressed as: $T_{in\_avg_L} = (1+h) \times T_{proc_L}$. At the best case, successful insertion of a pattern requires no shuffle of other patterns in hash tables, $T_{in\_best_L} = T_{proc_L}$.

The worst case can happen as h reaches M and the rehash occurs. According to [15], probability of this case is $O(1/n^2)$ when M is $3log_{\epsilon+1}m$. Before the rehashing process begins, the hash tables have to clear every occupied space. The reset time is T_{reset}. If N_L is the number of insertions for all patterns of length L until the successful insertion for pattern x then the time of rehash is $N_L \times T_{proc_L}$. In addition, according to the accumulative characteristic of SAX hash function, the rehashes are also required for patterns of lengths from $L+1$ to L_{max_s}. Finally, the worst-case insertion time of a pattern can be calculated as

$$T_{in\_w_L} = M \times T_{proc_L} + \sum_{i=L}^{L_{max\_s}} (N_i \times T_{proc_i} + T_{reset}) \quad (5)$$

where T_{proc_i} denotes the processing time of pattern with length i and N_i denotes the number of insertion time of all patterns with length i.

For long patterns, we consider three more parameters: the time to access table T_4, $T_{mem_4} = T_{mem}$; the time to compare and connect segments, $T_{connect}$; and the number of segments of a pattern, s. We express the time for lookup and insertion of a long pattern as follows.

$$T_{lu\_long} = (T_{lu_L} + T_{mem} + T_{connect}) \times s \quad (6)$$

$$T_{in\_long} = (T_{in_L} + T_{mem} + T_{connect}) \times s \quad (7)$$

4.1.2 Limited-time Update

As the analyzed worst case of insertion in Eq. (5), updating time of a pattern can be lengthy. For real-time protection, some practical systems can be vulnerable. To avoid this weakness, the basic solution is that we build the duplicate modules or separate modules for updating only [10], [23]. However, this solution consumes a lot of hardware resources and require re-compiling some parts of the system. We propose a simple and fast method that needs minimum hardware based on the capability of breaking a pattern into segments.

The details of our solution can be described as follows. If the worst case happens as a new pattern is inserted in the system then we report unsuccessful insertion instead of rehashing the hash tables. However, at that time, some positions of the hash tables are modified. Thus, we add a *stack* to trace the insertion process. Every step of insertion process is stored in the stack. If M is reached then we copy the traces from the stack back to the hash tables to restore the system. For illustration, we use a simple example as in Fig. 9. We insert a new element *"4"* into hash tables that stored *"0,2"* at addresses *3, 0* of T_1 and *"1, 3"* at addresses *4, 0* of T_2. At every step of collisions, we store the old information of "kick-out" elements in the stack including of the address in hash table $Addr_{hash}$, the content at this address $Content_{hash}$, and the order number of hash table Id_{hash}. As in Fig. 9, insertion process is infinite. Therefore, we restore steps one by one from the stack to the hash tables until the stack is

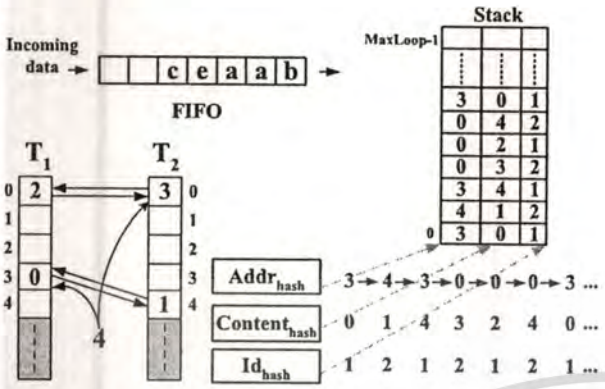


Fig. 9 Example of Limited-time pattern update. A *stack* traces the insertion process, the old information of "kick-out" elements, including of the address in hash table $Addr_{hash}$, the content at this address $Content_{hash}$, and the order number of hash table Id_{hash}. A *FIFO* buffers the incoming data as updating patterns

empty with the last element of unavailable space being "4". We can calculate the size of the stack as follows

$$S_S = M \times \lceil Addr_{hash} + Content_{hash} + Id_{hash} \rceil_{byte} \quad (8)$$

In Eq. (8), the symbol $\lceil ... \rceil_{byte}$ denotes rounding to byte of the bit sum of fields in the stack.

Next, we consider the new pattern as a long pattern and break it into segments to re-insert to the system. This process is recursive until the successful insertion occurs. We can prove that the update process of a string (pattern or segment) is limited by following paragraphs.

After un unsuccessful insertion of string less than or equal L_{max_s}, the string must be broken into halves. Note that the partitioning of long patterns greater than L_{max_s} is preprocessed off-line before updating. The online partitioning is slightly different from off-line method as mentioned in section 3.2. We do not consider whether the short pattern in the engine can be the prefix segment to save time. So the unsuccessful insertion time and the partitioning time can be calculated as follows.

$$T_{in\_u_L} = M \times T_{proc_L} + M = M(T_{proc_L} + 1) \quad (9)$$

$$T_{part_L} = 2T_{lu_{L/2}} + 2T_{in_{L/2}} \quad (10)$$

The unsuccessful insertion time equals to the sum of insertion time and back tracking time. The partitioning time includes the lookup and insertion time of two halves. The worst case insertion time can be expressed by the sum of the unsuccessful time and the partitioning time.

$$\begin{aligned} T_{in\_w_L} &= T_{in\_u_L} + T_{part_L} \quad (11) \\ &= T_{in\_u_L} + 2T_{lu_{L/2}} + 2T_{in_{L/2}} \end{aligned}$$

Let us assume that $L_{max\_s} = 2^K$ and the shortest segment is one-character which is always successful, so the insertion time $T_{in_1} = T_{in\_w_1}$.

Lemma 1. *The worst case insertion time of a new string of length $L_{max\_s} = 2^K (K \geq 1)$ is limited by the following.*

$$T_{in\_w_{2^K}} \leq \sum_{i=0}^{K-1} 2^i (T_{in\_u_{2^{K-i}}} + 2T_{lu_{2^{K-i-1}}}) + 2^K T_{in\_w_1}$$

Proof. The proof is mathematical induction using Eq.(11).

- *Basis*: for $K = 1$, according to Eq.(11), it is true.

$$\begin{aligned} T_{in\_w_2} &= T_{in\_u_2} + 2T_{lu_2} + 2T_{in_1} \\ &\leq T_{in\_u_2} + 2T_{lu_2} + 2T_{in\_w_1} \end{aligned}$$

- *Inductive step*: we assume that the result holds for some unspecified value of K.
We must be shown that the result holds for $K+1$, that is:

$$\begin{aligned} T_{in\_w_{2^{K+1}}} &= T_{in\_u_{2^{K+1}}} + 2T_{lu_{2^K}} + 2T_{in_{2^K}} \\ &\leq T_{in\_u_{2^{K+1}}} + 2T_{lu_{2^K}} \\ &\quad +2[\sum_{i=0}^{K-1} 2^i (T_{in\_u_{2^{K-i}}} + 2T_{lu_{2^{K-i-1}}}) \\ &\quad +2^K T_{in_{2^K/2^K}}] \\ &\leq \sum_{i=0}^{K} 2^i (T_{in\_u_{2^{K+1-i}}} + 2T_{lu_{2^{K-i}}}) \\ &\quad +2^{K+1} T_{in\_w_1} \end{aligned}$$

Hence, the proof. □

From Eq.(9), (3) and (4):

$$\begin{aligned} T_{in\_w_{2^K}} \leq \sum_{i=0}^{K-1} 2^i [M(2^{K-i} + 2T_{mem} + 1) \quad (12) \\ +2(2^{K-i-1} + 2T_{mem} + T_{mtch})] + 2^K T_{in\_w_1} \end{aligned}$$

Moreover, for on-the-fly update without interrupting the data stream, we can use a *FIFO* to buffer data stream. While a pattern is being updated, the data stream enters the FIFO. After the successful or unsuccessful update of a pattern, PAMELA will get data from the FIFO. The next update can continue as soon as the FIFO is empty. Because the network load is not 100% at all time, this on-the-fly update process is practical. If the line rate network is one byte per clock cycle, the size in byte of FIFO equals to the upper bound of a string insertion.

We assume that all time parameters in Eq.(12) consuming one clock cycle, $K = 4$, and $M = 30$ so we have the maximum successful insertion time of a pattern less than 3440 clock cycles as follows.

$$T_{in\_w_{2K}} \leq \sum_{i=0}^{K-1}[2^K(M+1)+2^i(3M+6)]+2^K \quad (13)$$
$$\leq 109M + 170 = 3440 = S_F$$

where S_F is the size of the FIFO.

4.1.3 Hardware Utilization

In the direct storage method, the number of elements per hash table size named *load factor* presents the efficiency of memory utilization. In Cuckoo Hashing, load factor is less than 0.5 to guarantee success of pattern insertion. If we define n_{avg} as the average number of patterns and m_{avg} as the average size of hash table in each Cuckoo module, then the average load factor α is given by the formula $\alpha = n_{avg}/(2 \times m_{avg})$

In our indirect storage method, the hash tables only store the indices of patterns using a few hardware resources, and most of resources are for pattern storage. Therefore, the memory utilization metric of our system has to take into account the condensed table T_3. If we assume the size in bit of T_3 is equal to the number of bits of pattern storage required, then it is the result of n_{avg} in a module multiplied by the average number of bits $(8 \times L_{avg})$ per pattern of this module. The number of bits to encode the address of storage table T_3 is $\lceil log_2 n_{avg} \rceil$, where $\lceil \rceil$ denotes rounding up. Finally, the average memory utilization U_{mem} is calculated as

$$U_{mem} = \frac{n_{avg} \times \lceil log_2 n_{avg} \rceil + n_{avg} \times 8 \times L_{avg}}{2 \times m_{avg} \times \lceil log_2 n_{avg} \rceil + n_{avg} \times 8 \times L_{avg}} \quad (14)$$

In Eq.(14), U_{mem} is the ratio of the patterns and their indices with the total sizes of T_1, T_2 and T_3. Fig. 10 shows the effect of U_{mem} based on load factor α. With the maximum value of α of 0.5, U_{mem} is approximately 0.88 when $m_{avg} = 512$, $n_{avg} \simeq 500$ and $L_{avg} \simeq 8$; we name it *PAMELA-1*. As α reduces by half to 0.25, U_{mem} slightly reduces and equals to 0.72 when $m_{avg} = 1,024$; we name it *PAMELA-2*. To consider lower value of load factor, it is not interesting in due to wasting too many memory resources. Therefore, we only select these two systems for further testing. If the balance between the high flexibility of update and the area is the first priority then *PAMELA-2* is selected. Otherwise, hardware-efficient *PAMELA-1* is the choice. Next section will show the practical comparison of two systems.

Besides the memory utilization, the logic gate (logic cell) utilization is also our interest. We assume that if a character needs p_{LC} logic gates for implementing hash function as in Fig. 2a then each pattern with length L needs $L \times p_{LC}$ logic gates. With $L_{max\_s} = 16$ modules, the number of logic gates is very large for the general architecture. However, as in Fig. 2b, our optimized architecture only uses p_{LC} logic gates for each

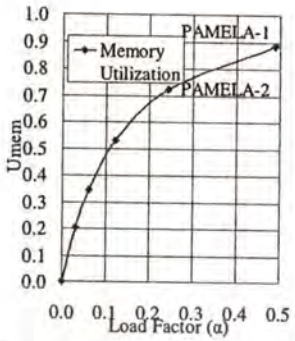


Fig. 10 Memory Utilization vs. Load Factor. *PAMELA-1* has Memory Utilization U_{mem} of 0.88, *PAMELA-2* has U_{mem} of 0.72

module. The notation R_{LC} to represent the ratio between logic gates of the general architecture and the optimized architecture is

$$R_{LC} = \frac{\sum_{L=1}^{L_{max\_s}}(L \times p_{LC})}{L_{max\_s} \times p_{LC}} \quad (15)$$
$$= \frac{L_{max\_s}(L_{max\_s}+1) \times p_{LC}}{2 \times L_{max\_s} \times p_{LC}} = \frac{(L_{max\_s}+1)}{2}$$

If L_{max_s} is 16 then the saving ratio is over 8 times for implementing the hash functions.

4.2 Performance Simulations

4.2.1 Off-line Insertion of Short Patterns

On Dec 15, 2006, there were 4,748 unique patterns with 64,873 characters in Snorts rule set. The distribution of the pattern lengths in Snort database is from 1 up to 109 characters. Fortunately, 65% of total numbers of patterns are up to 16 characters. Therefore, we build the Cuckoo Hashing modules for short patterns which are less than or equal to $L_{max\_s} = 16$ characters according to this fact.

For practical comparison, we implement hash functions with patterns of the lengths from 2 to 16 characters. For pattern length of one character, we directly match the patterns to save the hardware. In all experiments from now on, the number of trials is 1,000. We define a parameter named *%Rehash* as the following equation to determine the possibility of rehash happening in every trial in every pattern length.

$$\%Rehash = \frac{the\ number\ of\ rehashes}{the\ number\ of\ trials} \times 100\% \quad (16)$$

Figure 11 presents the number of insertions of Cuckoo Hashing with three hash functions: SAX, random table and CRC in which the size of index table is 512. Although CRC is a polynomials function, it is a suitable candidate for testing due to its significantly cheap implementation on hardware [29]. Three names, *SAX_hard*, *Tab_hard* and *CRC_hard*, are the FPGA-based systems whose architectures are changed as in

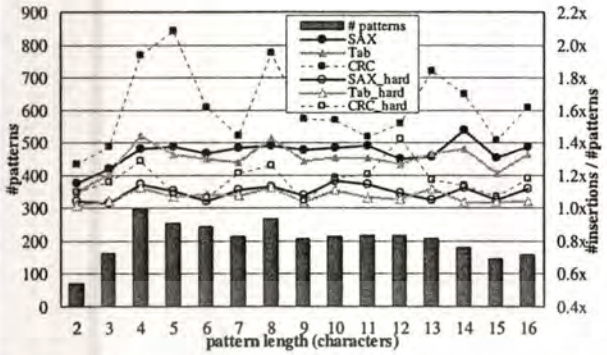


Fig. 11 The number of insertions of various hash functions vs. pattern length (characters). Bar graphs are the numbers of patterns. Line graphs are the ratio of numbers of insertions over numbers of patterns. Index (hash) table size is 512. The number of trials is 1,000

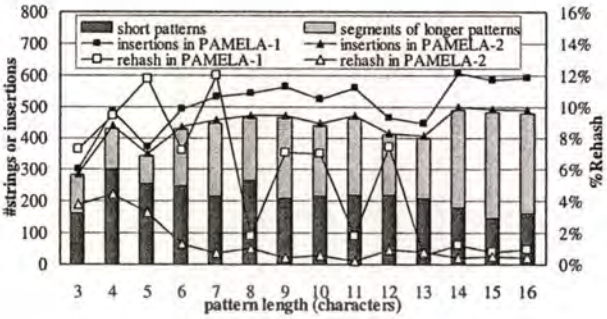


Fig. 12 The number of insertions and %Rehash after addition of longer patterns vs. pattern lengths (L). *PAMELA-1* and *PAMELA-2* have the index table sizes of 512 and 1,024, respectively. Both systems are based on SAX hash function and our improved architecture. The number of trials is 1,000.

Section 3.1 of this paper with patterns inserted in balancing.

The results in Fig. 11 show that the FPGA-based systems have the number of insertions less than the original systems by 20% and the performance of SAX hash function is close to that of random table. The results in Fig. 11 also show that the SAX and random table functions are significantly more efficient than CRC function with very small *%Rehash* of less than 5%. With the index table size of 512, the average load factor of index table is about 1/4. The remaining space can be used for fitting the segments of patterns with lengths of over 16 characters.

4.2.2 Off-line Insertion of Long Patterns

We break 1,643 long patterns of Snort rule set into over 3,500 segments of lengths from 3-16 characters as the described technique in Section 3.2. By sharing the prefixes with short patterns in the engine, the number of unique segments reduces about 12%. Let *string* denote the short pattern and the segment of long pattern. Totally, there are 6,136 strings of pattern set. Note that the distribution of segments, Th, is also considered to make the string number less than 512 in every length. This condition helps reduce the number of block RAMs of FPGA for implementing T_3 as well as T_1 and T_2.

The design can be parameterized with different table depths of 512 and 1,024 entries as *PAMELA-1* and *PAMELA-2*, respectively as defined in the previous subsection. These systems are used to evaluate the trade off between hardware utilization and performance of insertion. Both systems are based on SAX hash function and the FPGA-based Cuckoo architecture. Fig. 12 shows the number of insertions as well as number of strings in every length after adding segments of long patterns. In every length of string, the number of insertions in *PAMELA-2* is just greater than the number of strings slightly. Unfortunately for *PAMELA-1*, the number of insertions is about 16% greater than the number of strings. *%Rehash*es of both systems are showed in Fig. 12 to explain why the number of insertions increases in *PAMELA-1*. Due to high memory utilization, the collision (*%Rehash*) increases as the number of patterns increases, approximately 12% as compared with 4% of *PAMELA-2*.

4.2.3 Dynamic Update for New Patterns

Snort rule database must be updated to handle the new attacks. Normally, the web site of Snort generates new rules every one or two weeks. As a result, PAMELA can also be rapidly and easily updated to avoid vulnerability.

To demonstrate the dynamic update ability of PAMELA, we accumulated the latest Snort pattern set on May 14, 2007 to 5,026 unique patterns and 68,266 characters. Based on the pattern set on Dec 15, 2006, there were 293 unique and new patterns consisting 3,476 characters and 15 unique patterns consisting 83 characters deleted. Note that all new patterns are searched in the system before inserting. Within these 293 new patterns, 65 patterns are longer than 16 characters. These long patterns are broken off-line into segments as described in Section 3.2. Finally, only 381 unique short patterns and segments are inserted.

The average insertion time in number of clock cycles of each PAMELA system is compared in Fig. 13. To determine M, we can assign $\epsilon = 1$, $m_{avg} = 1{,}024$ for all systems; thus the result of M is 30. Due to bigger table size of 1,024 entries, *PAMELA-2* takes 5,375 clock cycles to insert without rehashing. In *PAMELA-1*, the numbers of clock cycles at pattern lengths of 6, 11, 12 and 15 characters rapidly increase due to the high penalty of rehashing. With *%Rehash* from 0.6%-2.9%, it takes 12,600 clock cycles to insert that is about 2.5 times as compared with *PAMELA-2*. *PAMELA-3* is extended from *PAMELA-1* by adding a stack and a FIFO for limited-time and uninterruptible update. Ac-

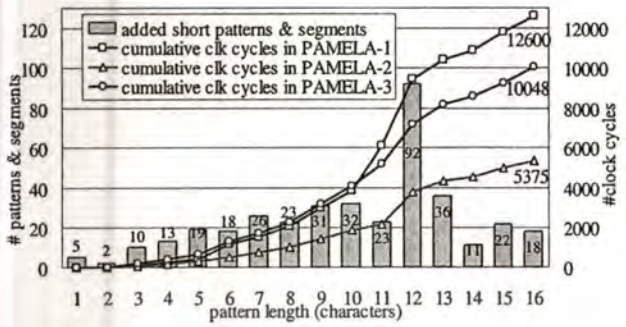


Fig. 13 The average insertion time (clock cycles) for inserting 381 new strings (patterns & segments). *PAMELA-3* is extended from *PAMELA-1* by adding a stack and a FIFO for limited-time and uninterruptible update. The number of trials is 1,000.

Table 2 Dynamic Update Comparison for A Pattern

System	Clock Freq.(Mhz)	Update time(μs)	Additional hardware requirement
PAMELA-1 (index table size:512)	200 (assumed)	0.21	No
PAMELA-2 (index table size:1,024)		0.09	No
PAMELA-3 (extended from *PAMELA-1*)		0.17	A stack & FIFO for uninterruptible update
Bit-split AC [23]	N/A	$\sim 10^6$	A temporary module
Rom+CoProc [10]	260	$\sim 1/3 \times 10^3$	A co-processor

cording to Eq. (8) and (13), the sizes of the stack and FIFO are 90 and 3440 bytes, respectively. Some new patterns with lengths of 6, 11, 12 and 15 characters are broken into segments with lengths of 3, 5-8 characters as they are unsuccessfully inserted into the index tables. Then, the segments are inserted into the modules with corresponding lengths. Some segments with length of 6 are continually broken again due to high collision but the total of partitioning times is only 2. Although the number of strings needed for insertion increases up to 390, the number of clock cycles including partitioning time in *PAMELA-3* reduces to 10,048. For 15 deleted patterns, the process is fast with around 100 clock cycles for each system.

We assume that *PAMELA-1* to *PAMELA-3* are clocked at 200 MHz which is less than our synthesized results shown later and can be achieved on many common Virtex FPGA boards. With 5,475 to 12,700 clock cycles, the update time is about 28 to 64 microseconds (μs). Table 2 shows the comparison of the dynamic update time of some systems. [23] shows that the system can update non-interrupting data stream by using a temporary module for updating only and the duration is less than one second for compiling and updating. [10] uses a co-processor to update with estimated time of 10 milliseconds (ms) for 30 new patterns. Thus we can show the approximate time for one pattern is 1/3 ms. We use no additional hardware in *PAMELA-1* and *PAMELA-2* of simulation systems. As a result, to add 293 new patterns, the average insertion time for a pattern is 19-43 clock cycles; and with added the delete time, the average time for updating one pattern is only 18-42 clock cycles, about 0.09-0.21 μs on 200 MHz FPGA systems. If the limited-time and uninterruptible update system is required then *PAMELA-3* is used; the insertion time of a new pattern is limited in 3440 clock cycles ($\sim 17\mu s$). These results show that PAMELAs are very efficient for updating pattern set without lengthy FPGA reconfiguration time.

5. Extension for multi-character processing

To increase the throughput, PAMELA can be easily extended for multi-character processing. Instead of reading one character per clock cycle, the system shifts N characters into an input buffer every clock cycle The system has N blocks corresponding to N characters needed to process. Inside the block, each Cuckoo module receives one character from the pre-determined address of input buffer and the hash value from the previous module. Fig. 14 is an example of our system to process 4-character per clock cycle. Due to higher addressing complexity, we need to determine the address of input buffer that each Cuckoo Module is connected to. In each block j, if the address connected to the previous module is $A_{i-1,j}$ $(2 \le i \le L_{max\_s})$ and contents character x_k then the address connected to the next module, $A_{i,j}$, has to shift $N-1$ from $A_{i-1,j}$ and the next module processes the next character. For example, if the string is "...*abcd*...", each time we shift $N = 4$ characters, the previous module consumes the character 'a' at address $A_{i-1,j}$ then the next module consumes the character 'b' at address $A_{i-1,j} + (N-1)$ one clock cycle later. In general, the address in the buffer connected to Cuckoo i of block j is:

$$A_{i,j} = A_{i-1,j} + (N-1) = A_{i-2,j} + 2(N-1) \quad (17)$$
$$= \ldots = A_{1,j} + (i-1)(N-1)$$

From Eq.17, we can see that when $N = 1$, the address of any module coincides with the first address. So it is correct with our one-character design. We can also calculate the size of input buffer from Eq.17. The last address is $A_{L_{max\_s},1}$ when $i = L_{max\_s}$. If the lowest address is zero then $A_{1,1}$ is $N-1$. The size of the buffer can be calculated as follows.

$$\begin{aligned} S_{Buffer} &= A_{L_{max\_s},1} + 1 \\ &= A_{1,1} + (L_{max\_s} - 1)(N-1) + 1 \\ &= L_{max\_s}(N-1) + 1 \end{aligned} \quad (18)$$

The resources for storing patterns in multi-character processing scheme can be significantly reduced by sharing SRAMs together. If the number of ports of SRAMs

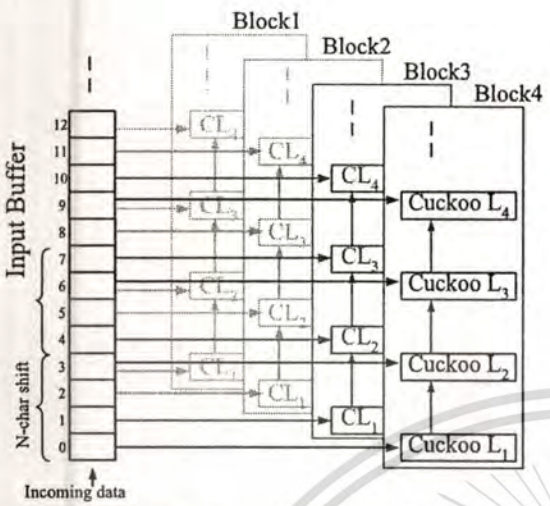


Fig. 14 PAMELA for N-character processing ($N = 4$). Cuckoo Modules are connected to the determined addresses of input buffer.

are two times of the number of processed characters, T_3 can be shared for Cuckoo modules that process the same pattern length. Moreover, the processing of long pattern can be also carried out only one table T_4 for the whole system.

6. FPGA Implementation Results

Our design is developed in Verilog hardware description language and by Xilinx's ISE 8.1i for hardware synthesis, mapping, and placing and routing. The target chips are some major Xilinx FPGA chips such as Virtex2, Virtex-Pro and Virtex-4. To reduce the number of memory blocks in FPGA, we can implement two index tables in the same block RAM; T_1 is in a low addresses part and T_2 is in a high addresses part. The block RAM of Xilinx FPGA can be configured as dual-port mode that can be accessed concurrently.

Based on our parallel pattern matching engine described earlier, we can measure the cost based on the numbers of block RAMs and logic cells in Table 3. With the maximum capacity of 18 kbits, block RAMs can be programmed as 18K x 1 bit to 512 x 36 bits, in various depth and width configurations. Hence, we can configure both tables T_1 and T_2 with size 1,024 x 9 bits for each as *PAMELA-2* mentioned above. Since the number of patterns in each Cuckoo module is less than 500, we can set the depth of T_3 as 512. The width of T_3 can be determined by the pattern length added 2 more bits for controlling long patterns. For table T_4, the width of T_4 is equal to the number of its address bits plus 1 bit for suffix segment of long pattern. For this reason, the size of T_4 is $2^{13} \times 14$. For the short patterns of one character, we can directly compare to save hardware resources. The other parts of the system use logic cell of FPGA. While SAX functions use few resources of logic cells, the most consuming parts are the comparators. Totally, we use only 62 block RAMs and 3,220 logic cells to fit 68,266 characters of the entire rule set on the XCV4LX100 FPGA chip. To update without interrupting the incoming data, three more block RAMs are used to implement the stack and FIFO. For multi-character processing, since current Xilinx FPGA chips only have 2-port block RAM, we have to duplicate the pattern in T_3s. In addition, T_4s can only process 2 lookups at the same clock cycle. Some components such as the controls, counters, registers, LFSR, etc. are also shared inside the system.

Table 3 Logic and Memory Cost of Components in Virtex-4

Component	Quantity	Block RAMs	Logic cells	Note
Tables T_1 & T_2	15x2	15	0	2 tables in a BRAM
Table T_3	16	39	0	The BRAM numbers for lengths 1-4 : 1; 5-8: 2 9-13: 3; 14-16: 4
Table T_4	1	8	0	
SAX hashes	15x2	0	840	
Multiplexer	15x2	0	300	
Comparator	32	0	1,320	16 for long patterns
Counter, Priority, Reg, LFSR, etc.		0	760	
Total		62	3,220	

Table 4 shows the comparison of our synthesized systems with other recent FPGA systems. Two metrics, logic cells per character (LCs/char) and SRAM bits per character (bits/char), are used to evaluate the efficiency of FPGA utilization. For state machine approach, we just show some interesting implementations that are commonly used for static pattern matching only. We can see that the hashing systems [10]–[13] are almost better than state machine [6], [24] and compare-and-shift ones [7], [9] in term of hardware utilization. As compared to V-HashMem [13], PAMELAs are 30% fewer in LCs/char and bits/char. Note that V-HashMem supports header matching with less hardware. On the contrary, with 6,500 strings inserted into the system, the storage capacity of synthesized PAMELA can support 1500 more strings without additional hardware resource. As compared to [10], [11], the Block RAM usage of our architecture is 1.22-2.07 times greater than theirs, but our logic cell usage is significantly smaller than theirs by 4.3-5.2 times. In summary, PAMELAs are the most efficient ones in using logic cells of FPGA at a cost of 0.047-0.175 LCs/char. In addition, the memory usage of our architecture is of high density (16.74-62.10 bits/char) and is acceptable as compared to other systems.

For throughput comparison, our throughput (T-put) can vary from 1.78-8.8 Gbps depending on the kinds of FPGA chips and the number of characters processed per clock cycle. Some works can also process multi-character at very high throughput up to 10

Table 4 Performance Comparison of FPGA-based Systems for NIDS/NIPS

System	Device (Xilinx)	bits/ cycle	Freq. (MHz)	No. chars	No. LCs	Mem (kbits)	LCs/ char	Mem per char (bits)	T-put (Gbps)	PEM
PAMELA-2	**XC4VLX100**	**8**	**285**	**68,266**	**3,220**	**1,116**	**0.047**	**16.74**	**2.28**	**10.29**
	XC4VLX100	**16**	**282**		**6,120**	**2,070**	**0.090**	**31.05**	**4.51**	**10.92**
	XC4VLX100	**32**	**275**		**11,980**	**4,140**	**0.175**	**62.10**	**8.80**	**10.70**
	XC2VP20	**8**	**272**		**3,266**	**1,116**	**0.048**	**16.74**	**2.18**	**9.79**
	XC2V6000	**8**	**223**		**3,266**	**1,116**	**0.048**	**16.74**	**1.78**	**8.03**
	XC2V6000	**16**	**218**		**6,212**	**2,070**	**0.091**	**31.05**	**3.49**	**8.42**
PAMELA-2 (with a FIFO and a stack)	**XC4VLX100**	**8**	**285**	**68,266**	**3,233**	**1,170**	**0.047**	**17.55**	**2.28**	**9.91**
V-HashMem [13]	XC2VP30	8	306	33,613	2,084	702	0.060	21.39	2.49	8.60
HashMem [12]	XC2V1000	8	250	18,636	2,570	630	0.140	34.62	2.00	4.01
	XC2V3000	16	232		5,230	1,188	0.280	65.28	3.71	3.86
PH-Mem [11]	XC2V1000	8	263	20,911	6,272	288	0.300	14.10	2.11	4.71
	XC2V1500	16	260		10,224	306	0.490	14.98	4.16	6.44
ROM+Coproc [10]	XC4VLX15	8	260	32,384	8,480	276	0.260	8.73	2.08	5.90
Prefix Tree [9]	XC2VP100	8	191	39,278	12,176	0	0.310	0	1.53	4.93
PreD-CAM [7]	XC2V3000	8	372	18,036	19,854	0	1.100	0	2.98	2.70
	XC2V6000	32	303		64,268	0	3.560	0	9.70	2.72
FPGA-based Bit-Split [24]	XC4FX100	8	200	16,715	4,514	6,000	0.270	184	1.60	0.39
PreD-NFA [6]	XC2V8000	32	219	17,537	54,890	0	3.130	0	7.00	2.24

Gbps, especially shift-and-compare architectures. However, the area cost is high. Therefore, a Performance Efficiency Metric (PEM) is used as the ratio of throughput in Gbps to the logic cell per each pattern character for performance evaluation.

$$PEM = \frac{Throughput}{\frac{No.Logiccells + \frac{Membytes}{12}}{No.Characters}} \quad (19)$$

Assuming that the cost of 12 bytes block RAMs is equivalent to a logic cell [30], Eq. (19) takes into account both block RAMs and logic cells area metrics for fair comparison [14] between the memory-based systems and the logic gate-based systems. As PEMs in the range of 8.03-10.92, PAMELAs are the best of the FPGA-based hashing systems, far better than the shift-and-compare systems by at least two times, and better than the state machine systems over three times.

7. Conclusion

A pattern matching engine based on Cuckoo hashing for NIDS/NIPS named PAMELA is proposed. PAMELA engine can update the new patterns rapidly in the orders of microseconds. PAMELA can also guarantee that the data stream cannot be interrupted during pattern updates by adding both small stack and FIFO. According to the implementation results, the performance of PAMELA is the best when compared with other previous systems and the achievable throughput can be up to 8.8 Gbits/s. The current scheme of PAMELA is also scalable enough to process N characters every clock cycle. Since PAMELA requires no reconfiguration at all, this engine can be applied on ASIC at much higher performance than FPGA.

Acknowledgments

This work was supported by the AUN-SeedNet Program of JICA.

References

[1] http://www.snort.org, "SNORT official website,"
[2] Y.H. Cho, S. Navab, and W.H. Mangione-Smith, "Specialized hardware for deep network packet filtering," Proc. Int. Conf. on Field-Programmable Logic and Applications, pp.452–461, 2002.
[3] I. Sourdis and D.N. Pnevmatikatos, "Fast, large-scale string match for a 10gbps FPGA-based network intrusion detection system," Proc. Int. Conf. on Field-Programmable Logic and Applications, pp.880–889, 2003.
[4] J. Moscola, J. Lockwood, R.P. Loui, and M. Pachos, "Implementation of a content-scanning module for an internet firewall," Proc. IEEE Symp. on Field-Programmable Custom Computing Machines, pp.31–38, 2003.
[5] Y.H. Cho and W.H. Mangione-Smith, "Deep packet filter with dedicated logic and read only memories," Proc. IEEE Symp. on Field-Programmable Custom Computing Machines, pp.125–134, 2004.
[6] 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, 2004.
[7] 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, 2004.
[8] S. Yusuf and W. Luk, "Bitwise optimised CAM for network intrusion detection systems.," Proc. Int. Conf. on Field-Programmable Logic and Applications, pp.444–449, 2005.
[9] Z.K. Baker and V.K. Prasanna, "High-throughput linked-pattern matching for intrusion detection systems," Proc. ACM/IEEE Symp. on Architecture for networking and communications systems, pp.193–202, 2005.
[10] Y.H. Cho and W.H. M-Smith, "Fast reconfiguring deep packet filter for 1+ gigabit network.," Proc. IEEE Symp.

on Field-Programmable Custom Computing Machines, pp.215–224, 2005.
[11] I. Sourdis, D.N. Pnevmatikatos, S. Wong, and S. Vassiliadis, "A reconfigurable perfect-hashing scheme for packet inspection.," Proc. Int. Conf. on Field-Programmable Logic and Applications, pp.644–647, 2005.
[12] G. Papadopoulos and D.N. Pnevmatikatos, "Hashing + memory = low cost, exact pattern matching.," Proc. Int. Conf. on Field-Programmable Logic and Applications, pp.39–44, 2005.
[13] D. Pnevmatikatos and A. Arelakis, "Variable-length hashing for exact pattern matching.," Proc. Int. Conf. on Field-Programmable Logic and Applications, pp.1–6, 2006.
[14] I. Sourdis, D.N. Pnevmatikatos, and S. Vassiliadis, "Scalable multigigabit pattern matching for packet inspection.," IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol.16, no.2, pp.156–166, 2008.
[15] R. Pagh and F.F. Rodler, "Cuckoo hashing," Journal of Algorithms, vol.51, no.2, pp.122–144, 2004.
[16] T.N. Thinh, S. Kittitornkun, and S. Tomiyama, "Applying Cuckoo hashing for FPGA-based pattern matching in NIDS/NIPS," Proc. Int. Conf. on Field-Programmable Technology, pp.121–128, 2007.
[17] Z.K. Baker and V.K. Prasanna, "Time and area efficient pattern matching on FPGAs," Proc. ACM/SIGDA Symp. on FPGAs, pp.223–232, 2004.
[18] B.L. Hutchings, R. Franklin, and D. Carver, "Assisting network intrusion detection with reconfigurable hardware," Proc. IEEE Symp. on Field-Programmable Custom Computing Machines, pp.111–120, 2002.
[19] J.C. Bispo, I. Sourdis, J.M. Cardoso, and S. Vassiliadis, "Regular expression matching for reconfigurable packet inspection," Proc. Int. Conf. on Field-Programmable Technology, pp.119–126, 2006.
[20] A. Mitra, W. Najjar, and L. Bhuyan, "Compiling PCRE to FPGA for accelerating SNORT IDS," Proc. ACM/IEEE Symp. on Architecture for networking and communications systems, pp.127–136, 2007.
[21] A.V. Aho and M.J. Corasick, "Efficient string matching: an aid to bibliographic search," Communications of the ACM, vol.18, no.6, pp.333–340, 1975.
[22] M. Aldwairi, T. Conte, and P. Franzon, "Configurable string matching hardware for speeding up intrusion detection," SIGARCH Comput. Archit. News, vol.33, no.1, pp.99–107, 2005.
[23] L. Tan and T. Sherwood, "A high throughput string matching architecture for intrusion detection and prevention," Proc. IEEE Symp. on Computer Architecture, pp.112–122, 2005.
[24] H.J. Jung, Z.K. Baker, and V.K. Prasanna, "Performance of FPGA implementation of bit-split architecture for intrusion detection systems," Proc. Reconfigurable Architectures Workshop, 2006.
[25] S. Dharmapurikar, P. Krishnamurthy, T.S. Sproull, and J.W. Lockwood, "Deep packet inspection using parallel bloom filters," Symposium on High Performance Interconnects, pp.44–51, 2003.
[26] L. Carter and M.N. Wegman, "Universal classes of hash functions.," Journal of Computer and System Sciences, vol.18, no.2, pp.143–154, 1979.
[27] M.V. Ramakrishna and J. Zobel, "Performance in practice of string hashing functions," Proc. Int. Conf. on Database Systems for Advanced Applications, pp.215–224, 1997.
[28] http://www.xilinx.com/bvdocs/appnotes/xapp211.pdf, "Xilinx application note,"
[29] J. Stone, M. Greenwald, C. Partridge, and J. Hughes, "Performance of checksums and CRC's over real data," IEEE/ACM Transactions on Networking, vol.6, no.5, pp.529–543, 1998.
[30] T. Sproull, G. Brebner, and C. Neely, "Mutable codesign for embedded protocol processing," Proc. Int. Conf. on Field Program Logic and Applications, pp.52–56, 2005.

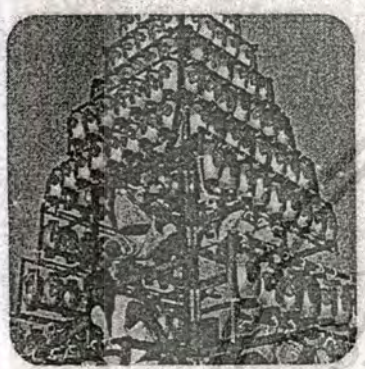
Tran Ngoc Thinh received the B.E. degree from the Ho Chi Minh City University of Technology, Vietnam, in 1999, and the M.E. from the King Mongkut's Institute of Technology Ladkrabang, Thailand, in 2006, all in computer engineering. He is currently pursuing the Ph.D. degree in computer engineering from the King Mongkut's Institute of Technology Ladkrabang, Thailand. His research interests include Network security and bioinformatics on reconfigurable devices.

Surin Kittitornkun received his Ph.D. and MS.EE. from University of Wisconsin-Madison, USA in 2002 and 1997, respectively. His research interest includes high performance/parallel computing in FPGA. He was awarded Gerald Holdridge Course/Lab Development Teaching Assistant Award in 2002. Prior to that he was an intern at Motorola Incorporation, Schaumburg, Illinois. He is now an Assistant Professor at the Department of Computer Engineering, Faculty of Engineering, King Mongkut's Institute of Technology Ladkrabang, Bangkok, Thailand.

Shigenori Tomiyama received the B.S. and M.S. degrees from Tokai University, Kanagawa, Japan, in 1966 and 1968 respectively. He received the Ph.D. degree in Engineering from Tokai university in 1988. In 1968, he joined Department of Communications Engineering, Tokai University, as an assistant professor. He has been a Professor since 1988. He is currently a professor with the Department of Embedded Technology, Tokai University. His research interests include system characteristics approximation, digital filter design and FPGA. He is a member of IEEE.

December 12 – 14, 2007, Kitakyushu, JAPAN



ICFPT 2007

International Conference on Field-Programmable Technology


Editors:
Hideharu Amano
Andy Ye
Takeshi Ikenaga



Co-sponsors:

IEICE-ISS, Technical Committee on RECONF
City of Kitakyushu

CITY OF KITAKYUSHU

Technical co-sponsors:

IEEE

IEEE Electron Devices Society

Applying Cuckoo Hashing for FPGA-based Pattern Matching in NIDS/NIPS

Tran Ngoc Thinh*[1], Surin Kittitornkun*[2] and Shigenori Tomiyama**[3]
** Department of Computer Engineering, Faculty of Engineering,*
King Mongkut's Institute of Technology Ladkrabang, Bangkok, 10520 Thailand
***Department of Embedded Technology,*
School of Information Technology and Electronics, Tokai University, Japan
[1]tnthinh@cse.hcmut.edu.vn, [2]kksurin@kmitl.ac.th, [3]tomiyama@dt.u-tokai.ac.jp

Abstract

Pattern matching for network intrusion/prevention detection requires extremely high throughput with frequent updates to support new attack patterns. Most of current hardware implementations have outstanding performance over software implementations. However, the requirement for dynamic update pattern set is still challenging for hardware researchers. This paper describes a novel FPGA-based pattern matching architecture using a recent hashing algorithm called Cuckoo Hashing. The proposed architecture features on-the-fly pattern updates without reconfiguration, more efficient hardware utilization, and higher performance. Through various algorithmic changes of Cuckoo Hashing, we can implement parallel pattern matching on SRAM-based FPGA. Our system can accommodate the latest Snort rule-set, an open source Network Intrusion Detection/Prevention System, and achieve the highest utilization in terms of SRAM per character and Logic Cells per character at 17 bits/character and 0.043 Logic Cells/character, respectively on major Xilinx Virtex architectures. Compared to others, ours is much more efficient than any other Xilinx FPGA architectures.

1. Introduction

In recent years, Network Intrusion Detection/Prevention Systems (NIDSs/NIPSs) are more and more necessary for network security. Normally, traditional firewalls only examine packet headers to determine whether to block or pass the packets. Due to busy network traffic and smart attacking schemes, firewalls are not as effective as they used to be. NIDSs/NIPSs are designed to examine not only the headers but also the payload of packets to match and identify intrusions.

To define suspicious activities, most modern NIDSs/NIPSs rely on a set of rules which are applied to matching packets. At the heart of almost every NIDS is pattern matching algorithms. For example, Snort [1] is an open source network intrusion detection and prevention system utilizing a rule-driven language, which combines the benefits of signature, protocol and anomaly based inspection methods. Snort relies on a set of rules to filter the incoming packets. As the number of known attacks grows, the patterns for these attacks are made into Snort signatures (patterns or strings). The simple rule structure allows flexibility and convenience in configuring Snort. However, checking thousands of patterns to see if it matches with incoming packets becomes a computationally intensive task as the highest network speed increases to several gigabits per second (Gbps).

To improve the performance of Snort, various implementations of FPGA-based systems have been proposed. These systems can simultaneously process thousands of rules relying on native parallelism of hardware so their throughput can satisfy current gigabit networks. However, the drawback of hardware-based systems is the flexibility. With emergence of new worms and viruses, the rule set must be frequently updated. Although SRAM-based field programmable gate array (FPGA) can be reconfigured; the process of recompiling the updated FPGA design can be lengthy. For some recently proposed FPGA-based NIDSs/NIPSs, adding or subtracting any number of rules requires recompilation of some parts or the entire design. The compilation process takes several minutes to several hours to complete. Today, such latency in compilation may be not accepted for most networks when new attacks are released frequently.

Based on Cuckoo Hashing [2], we implement a novel architecture of variable-length pattern matching best suited for FPGA. New patterns can be added to or removed out of the Cuckoo hash tables. Unlike most previous FPGA-based systems, the proposed architecture can update the static pattern set on-the-fly without reconfiguration thanks to Cuckoo Hashing. Our contributions also include parallel Cuckoo

1-4244-1472-5/07/$25.00 © 2007 IEEE

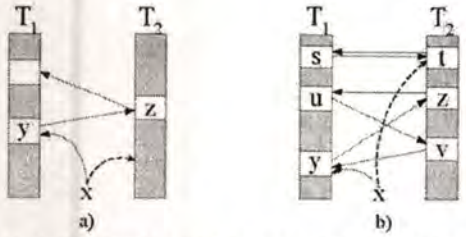


Fig. 1. Original Cuckoo Hashing [2], a) Key *x* is successfully inserted by moving *y* and *z*, b) Key *x* cannot be accommodated and a rehash is necessary

Hashing, better hardware utilization and high matching throughput reaching multiple Gbps.

The paper is organized as follows. In section 2, some previous FPGA implementations of static pattern matching and Cuckoo Hashing are presented. Section 3 proposes the architecture of FPGA-based Cuckoo Hashing. Next, the FPGA implementation of Cuckoo Hashing for multiple pattern matching and its experimental results are discussed in Section 4 and 5, respectively. Finally, future works are suggested in the conclusion.

2. Background and Related Works

2.1. Pattern matching on NIDS/NIPS

For a line speed of gigabit network, many previous FPGA approaches of NIDS/NIPS are proposed. Some of them as [3, 4] implement regular expression matching (NFAs/DFAs) and [5] uses content addressable memory (CAM). They suffer scalability problems such as too many states consume too many hardware resources and FPGA device has to re-programmed every time patterns changed. Furthermore, to match incoming characters in parallel with all matchers, these systems require the use of extensive pipelined trees to achieve a high clock rate.

Another hardware approach implements hash functions [6, 7, 8, 9, 10] to find a candidate of pattern match. Dionisios et al. proposed a system named HashMem [6] system using simple CRC polynomials hashing implemented with XOR gates that can use efficient area resource of FPGA than before. For the improvement of memory density and logic gate count, they implemented V-HashMem [7]. However, these systems have some drawbacks: 1) To reduce the sparse of memory and avoiding collision, CRC hash functions have to be chosen carefully depending on specific pattern groups, 2) since pattern set is dependent, probability of redesigning the system and the reprogramming the FPGA is very high every time patterns are changed. On the other hand, Dharmapurikar proposed to use Bloom Filters to do the deep packet inspection [10]. This method does not require reprogramming of FPGA for patterns added. Nevertheless, the blooming filter method could generate a false positive match, which requires extra cost of hardware for the match rechecked.

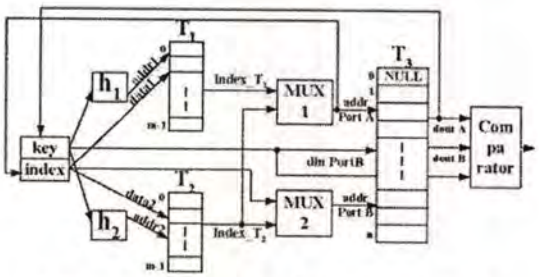


Fig. 2. FPGA-based Cuckoo Hashing

2.2. Cuckoo Hashing

Cuckoo hashing is proposed by Pagh and Rodler [2] as an algorithm for maintaining a dynamic dictionary with constant lookup time in the worst case scenario. The algorithm utilizes two tables T_1 and T_2 of size $m = (1+\varepsilon)n$ for each, where some constant $\varepsilon > 0$, n is the number of elements (strings). Cuckoo hashing does not need perfect hash functions that is very complicated if the set of elements stored changes dynamically under the insertion and deletion. Given two hash functions h_1 and h_2 from U to $[m]$, a key x can be exactly stored in either cell $T_1[h_1(x)]$ or $T_2[h_2(x)]$ but not both. So, lookup of x requires at most two positions. Deletion procedure can also run in worst case constant time.

Pagh and Rodler described a procedure for inserting a new key x in expected constant time. For the first time, x is always placed into cell $T_1[h_1(x)]$. If this cell is empty, the insertion is complete; if it is occupied by a key y, then y is kicked out. Then, y is put into the cell $T_2[h_2(y)]$ of second table in the same way, which is also possibly occupied by another key z. In this case, z is placed in cell $T_1[h_1(z)]$, and Cuckoo process repeats until the key that is currently nestless can be placed in an empty cell as in Fig. 1.a However, it can be seen that the Cuckoo process may not terminate as Fig. 1.b As a result, the number of iterations is bounded by a bound *MaxLoop* chosen beforehand. In that case everything is rehashed by reorganizing the hash table with new hash functions h_1 and h_2 and newly inserting all keys currently stored in the data structure.

3. FPGA-based Cuckoo Hashing

To apply FPGA-based Cuckoo Hashing for variable pattern lengths, the memory efficiency for storing patterns in hash tables is required because of the limited numbers of hardware resources. In direct

```
function lookup(x)
  select index_T1 in MUX1  and index_T2 in MUX2;
     index_T1 = T1(h1(x));              // phase1+2
     index_T2 = T2(h2(x));
     dataA = PortA(index_T1);           // phase3
     dataB = PortB(index_T2);
     return(dataA = x or dataB = x); // phase4
end
```

Fig. 3. Pseudo-code of FPGA-based Cuckoo Lookup Algorithm

storage method, the width of hash table must be equal to the longest pattern in the rule set. Thus, the remaining short patterns will waste a lot of memory space. In order to increase memory utilization, we build up a hash module for each length of pattern and use indirect storage. Small and sparse hash tables contain indices of keys which are the addresses of a condensed pattern-stored table.

The architecture of a FPGA-based Cuckoo Hashing module as shown in Fig. 2 includes three tables: two index tables (hash tables) T_1 and T_2 are the single-port SRAMs and a pattern-stored table T_3 is the double-port SRAM for concurrent processing. Hash functions are any universal functions that can change if they are required to rehash. Two registers named *key* and *index* contain the pattern to be searched for and the memory address of that pattern in T_3, respectively. Two multiplexers are used to select addresses for two ports of T_3. The output of first multiplexer (*MUX1*) is the address of *port A* that is the read-only port. *MUX1*'s inputs are the output value of T_1 (*index_T1*) and T_2 (*index_T2*). The output of second multiplexer (*MUX2*) is the address of *port B* that is both reading and writing port. *MUX2* selects *index_T2* as lookup function or *index* of key as insertion function. A comparator is used for exact matching with two candidate patterns from T_3.

3.1. Parallel Lookup

A lookup function of element (key) x can be divided into four phases. In each phase, instructions can be done in parallel. In the first phase, x is hashed by two hash functions concurrently. In the second phase, the values of two hash functions are used as the address for reading data of two index tables. In the third phase, to read data from T_3, two ports of T_3 use the outputs of two index tables as their addresses. In the last phase, the data outputs of T_3 are compared with the incoming character to determine the match. Fig.3 is the pseudo-code of parallel Cuckoo lookup function. By using multi-phase pipeline architecture, the FPGA-based Cuckoo Hashing can look up the keys in streaming with each key in clock cycle.

3.2. Dynamic Insertion & Deletion

```
procedure insert(x)
    if (lookup(x)) return;
    select index_T1 in MUX1  and index in MUX2;
    PortB(index) = x;
    if(index_T1 == NULL){
        T1(h1(x)) = index; return;}
    else if(index_T2 == NULL){
        T2(h2(x)) = index; return;}
    loop MaxLoop times
        if (select index_T1 in MUX1){
            key = PortA(index_T1);
            index = index_T1;
            select index_T2 in MUX1;}
        else{ // (select index_T2 in MUX1)
            key = PortA(index_T2);
            index = index_T2;
            select index_T2 in MUX1;}
        index_T1 = T1(h1(x));
        index_T2 = T2(h2(x));
        if (select index_T1 in MUX1){
            if(index_T1 == NULL) return;
            dataA = PortA(index_T1);}
        else{ // (select index_T2 in MUX1)
            if(index_T2 == NULL) return;
            dataA = PortA(index_T2);}
    end loop
    rehash(); insert(x);
end
```

Fig. 4. Pseudo-code of FPGA-based Cuckoo Insertion Algorithm

When a key insertion occurs, we consider both key and its index. The key is used as an input for two hash functions and stores in table T_3. Its index is used for storing in table T_1 or T_2 and also as the address for lookup the space of key in T_3.

The insertion of element x as description of C-like pseudo-code in Fig.4 is only started after the lookup process failed. First of all, x is inserted into T_3 with address of next available space of T_3. If one of outputs of two index tables is empty (NULL), index of x is inserted into T_1 or T_2 and the insertion complete. As compared to Cuckoo Hashing algorithm described above, the *index* is always inserted into T_1 without referring to the value of T_2. Our improvement system considers both tables to reduce the insertion time. If both of the outputs of T_1 and T_2 are not NULL, we insert the key index into T_1. At the same time, the "kick-off" *index_T1* and its data from T_3 will be written into the index storage and the key storage for starting of Cuckoo process. Then, the *MaxLoop*, $[3\log_{1+\epsilon}m]$, is decreased and the key value is hashed by hash function h_2. The output data will be checked for whether the value is NULL. If it is NULL, the process ends with successful insertion. Conversely, the process is continued by taking in turns hashing from h_2 to h_1.

The worst case happens when the *MaxLoop* decreases to zero. Hence, rehashing is required. Two new hash functions h_1 and h_2 are issued by a pseudo-random number generator.

For deletion, it is as simple as the lookup process. If the lookup succeeds, *MUX1* will select exactly one of the outputs of two index tables to write into the *index* register. We then write NULL into table T_3 at the

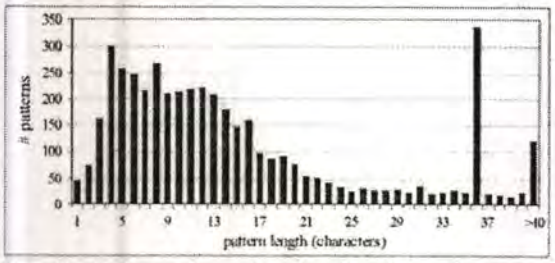


Fig. 5. Pattern length distribution of pattern set of SNORT in Dec 2006

address pointed by the *index* register. After that, we reset the *index* register to NULL and write into appropriate T_1 or T_2. The deletion process results some "holes" in T_3, but rehashing for rearranging of T_3 can be implemented as table T_3 is too sparse. This rehashing does not require new hash functions.

3.3. Selection of Hash Functions

The choice of hash functions greatly affects the performance of the system. Moreover, the probability for rehashing in Cuckoo Hashing is also based on the randomized property of hash functions. In this section, we discuss some simple and fast hash functions for string and choose the good one that is easily implemented on hardware.

The universal class of hash functions [11] has the good performance which can be guaranteed independent of the input keys. An example of such construct is modular hash functions. However, it is not suitable for hardware because of the complexity of the prime modulo operation. A fast way of generating a class of universal hash function and hardware-friendly, tabulation based hashing method [11], is defined as follows:

$$H_i(x) = a_i[0][x_0] \oplus a_i[1][x_1] \oplus \ldots \oplus a_i[n\text{-}1][x_{n-1}] \quad (1)$$

A table contains a 2-D array of random numbers. A key is string n characters $x_0x_1..x_{n-1}$ and the hash process is calculated by bit-wise exclusive-or ($\oplus$) a sequence of values $a_i[i][x_i]$, which is indexed by each byte value of x_i and position of i in the string. The drawback of this method is that the size of random table is very large and depends on the length of key.

Another class of simple hash function for hashing character strings named *shift-add-xor* (SAX) [12] utilizes only the simple and fast operations of shift, exclusive-or and add.

$$H_i = H_{i-1} \oplus (S_L(H_{i-1}) + S_R(H_{i-1}) + c_i) \quad (2)$$

Two operators S_L and S_R denote the shift left and right, respectively. The symbol c_i is the character i^{th} of string and H_i is an intermediate hash value after examination of i characters. The authors have shown that the class is likely to be universal. Good performance can be achieved in practice. Moreover, the main advantages of SAX as compared with random-table are the using small space of hardware and achieving high clock frequency by the simplicity of architecture. To generate the new SAX hash function in case of rehashing, we only need to change the value of H_0 by simple pseudo-random circuit LFSR [13]. Therefore, the SAX is a good choice for our system. The practical performance will be shown in the next section.

4. Implementation of FPGA-based Cuckoo Hashing for Multi-pattern matching in NIDS/NIPS

4.1. Short Patterns (less than or equal 16 characters)

For pattern matching in NIDS/NIPS, the patterns are searched on incoming data (packets). The matched pattern can occur anywhere as the longest substring. Normally, the pattern set is preprocessed and built in a system. The streaming data is entered into a FIFO to compare with all of patterns. FPGA-based Cuckoo Hashing guarantees the lookup time with line rate of streaming data.

On Dec 15, 2006, there were 4,748 unique patterns which contain 64,873 characters in Snorts rule set. Fig. 5 shows the distribution of the pattern lengths in Snort database of from 1 up to 109 characters. We can see that 65% of total numbers of patterns are up to 16 characters. Therefore, we build the Cuckoo Hashing modules for short patterns which are less than or equal 16 characters according to this fact. For longer patterns, we can break them into shorter segments so that we can insert those segments to the Cuckoo modules of short patterns. We then use simple address linked-lists to combine these segments later.

Fig. 6 shows the architecture of our pattern matching. Each module named *Cuckoo* L_i is used to process patterns that have same length i characters. We can reduce significantly large amount of hardware area resource by accumulative characteristic of SAX hash function. From (2), to calculate hash value of pattern of length i characters in the hash module i^{th}, the requisite inputs are hash value of i-1 characters calculated beforehand in the hash module $(i\text{-}1)^{th}$ and the i^{th} character. Therefore, the value of previous hash module can be reused for the next hash module. As in Fig 6, two hash values h_1 and h_2 of *Cuckoo* L_{i-1} are fed into *Cuckoo* L_i $(2<i\leq 16)$.

For instance, a pattern *"abc"* is stored in *Cuckoo* L_3 and an incoming text *"...123abcde..."* is passed through the system. In clock cycle t, the character *'a'*

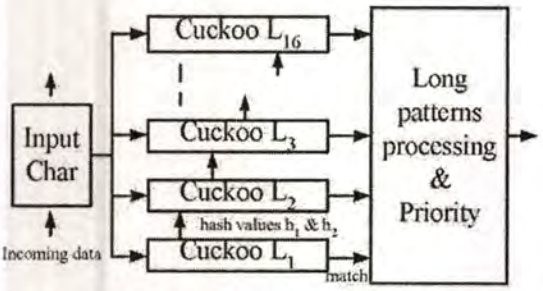


Fig. 6. FPGA-based Cuckoo Hashing for pattern matching in NIDSs/NIPSs

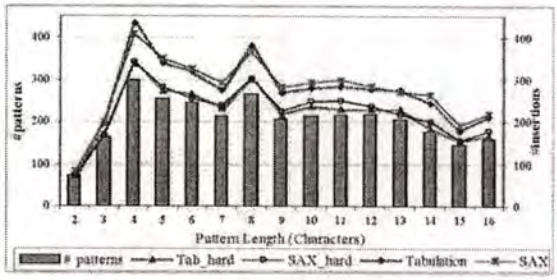


Fig. 7. The number of insertions of hash functions vs. pattern lengths. Bar graphs are the numbers of patterns. The lines are the numbers of insertions. Index tablesize is 512.

of text comes from the shift register; it is broadcasted to every Cuckoo module. At *Cuckoo* L_1, its hash values are passed to *Cuckoo* L_2. In next clock cycle, $t+1$, *Cuckoo* L_2 uses the hash values from *Cuckoo* L_1 and the new incoming character '*b*' to calculate its hash values for string *"ab"*. Similarly, in clock cycle $t+2$, *Cuckoo* L_3 uses the hash values from *Cuckoo* L_2 and the new incoming character '*c*' to calculate its hash values for string *"abc"*. A match signal is generated here. If many matches happen at the same time then a priority circuit selects the longest pattern and does not take interested in the other results.

Our architecture has a weak point here. The influence of rehash in the insertion process is remarkable. For example, when the rehash by collision happens at Cuckoo module i, new functions h_1 and h_2 of module i make the inputs of module $i+1$ changed and the hash values of module $i+1$ will be incorrect. The process is going on recursively up to module 16. As a result, the rehash can be forced from Cuckoo module $i+1$ to module *16*. This thing, however, only affects the insertion process. The trade off is good enough because the first priority is fast lookup with smaller hardware.

In comparison with previous implementations [6-9], the module i^{th} requires all i characters of the input string for every calculation and no reused of previous hash values at all. Their weak points can lead to the consumption of a remarkable the number of logic gates for implementation of hash functions.

Now, we show that SAX hash function also has good performance in insertion task. For practical comparison, we implement SAX and random table functions with patterns of the length from 2 to 16 characters. In all experiments from now on, the number of trials is 1,000. We define a parameter named *%Rehash* as following equation to determine the possibility of rehash happening in every trial of pattern length.

$$\%Rehash = \frac{the\ number\ of\ rehashes}{the\ number\ of\ trials} \times 100\%$$

Fig.7 shows the number of insertions of Cuckoo Hashing with two methods: random table and SAX function in which the size of each index table is 512. Two parameters *SAX_hard* and *Tab_hard* are the SAX and random table hash functions whose Cuckoo architectures are changed as in section 3 of this paper with patterns inserted initially in parallel. The results in Fig. 7 show that the parallel systems have the insertion time less than the original systems by about 20% and the SAX's hash function performance is close to that of random table. *%Rehash* is very small of less than 5% for all implementations of hash functions. With the index table size of 512, the average load factor of index table, the number of elements per table size, is about ¼. The remaining space can be used for fitting the segments of patterns with size of over 16.

4.2. Long Patterns (greater than 16 characters)

This subsection will present the method of matching long patterns of arbitrary length. We break the longer patterns into variable-length segments of 1÷16 characters. The above Cuckoo Hashing modules can then be used for matching these individual segments. After that, these segments of a pattern are combined to a chain that can be implemented by simple linked-list technology.

For more details, we describe the algorithm by a simple example. We assume that string *"abcdefghij"* is a long pattern that needs to break into smaller segments. The length of segments can be variable and the segments are unique. For example, with proportion 3:3:2:2 as in Fig. 8.a, segments 1 and 2 are hashed and stored in T_3 of *Cuckoo* L_3 at address 1FFh and 1FEh while segments 3 and 4 are hashed and stored in T_3 of *Cuckoo* L_2 at address 1A4h and 1A5h. We combine the addresses in tables T_3 together with the number of Cuckoo modules to link these individual segments. For instance, the depth of T_3 in every module is 512 and there are 16 Cuckoo modules. Therefore, the bit number is 13 for representing a position of individual segment: 9 least significant bits for the address of T_3 and 4 most significant bits for the number of Cuckoo

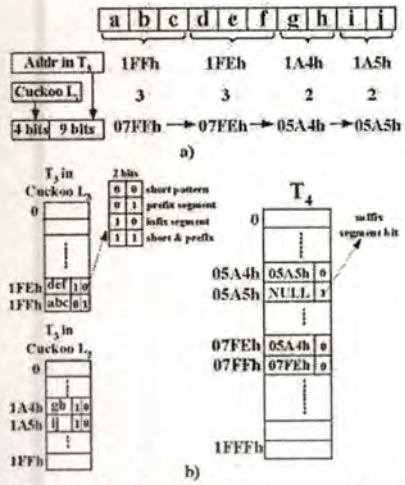


Fig. 8. Matching long patterns. a) Example of breaking a long pattern *"abcdefghij"* b) Linked-list storing in table T_4

modules. The segment addresses of *"abcdefghij"* are 7FFh, 7FEh, 5A4h, and 5A5h.

The data structure for storing linked-lists is a table named T_4 whose depth is the depth of T_3 multiplying the number of Cuckoo modules. Each address represents one segment and its content is an address of next segment in the same long pattern.

Returning to our example, the segment addresses of long pattern *"abcdefghij"* are stored in T_4 as Fig. 8.b. The match process is described as following. When we get a match for segment *"abc"*, we have address 7FFh whose content in T_4 is 7FEh. After 3 clock cycles, if the mach segment is *"def"*, the process is continued by jumping from 7FEh to 5A4h. Otherwise, the detective process finishes without match and resets for other patterns. Similarly, from 5A4h, if the next match segment after 2 clock cycles is *"gh"*, address 5A5h is considered. Finally, the match of pattern is reported after 2 next clock cycles if the last segment *"ij"* is detected.

Some additional parameters in Fig. 8.b need to be explained details about their functions. When a match happens in a Cuckoo module, we do not know a matched string is a short pattern or a segment of long pattern. Moreover, we have to determine the beginning of long pattern. So 2 bits are added in every entry of T_3 as in Fig. 8.b. They are used to describe a string is the short pattern, the first segment of long pattern called *prefix segment*, the body of long pattern called *infix segment*, or even the short pattern which is also the prefix segment. In case of the end of long pattern called *suffix segment*, we can determine it by checking the content of its address in T_4 whether it is NULL. However, if a long pattern is a prefix substring of another one then it can not be detected. For example, long pattern *"abcdef"* can be a prefix substring of

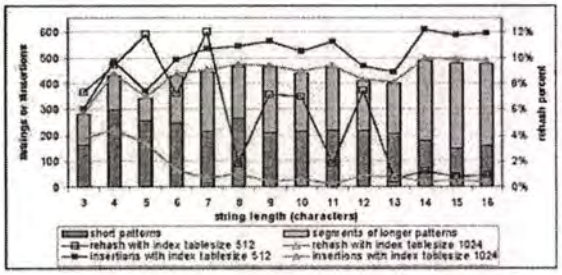


Fig. 9. The number of insertions and rehash percent *(%Rehash)* after addition of longer patterns vs. pattern lengths

"abcdefghij". To improve this case, we add 1 bit in table T_4. This bit allows a segment to be both a *suffix segment* and an *infix segment*. The content of a suffix segment in T_4 can be a pointer to another address so the number of segments can be decreased.

Besides, there are some cases that long patterns share the same prefix, infix or suffix. If long patterns have the same segments of these kinds then the number of segments can be reduced sharply. To process these cases, however, the algorithm will be more complicated and the insertion and deletion of new patterns will be difficult. As a result, we do not discuss these cases in this paper.

After processing 1,643 long patterns in Snort rule set, we obtain over 3,000 segments with length from 3-16 characters. We define *string* represents short pattern and segment of long pattern. The distribution of segments is also considered to make the balance in every length of string satisfying less than 512. This condition helps to reduce the number of block RAMs of FPGA for implementing the depth of T_3 as well as the width of T_1 and T_2. Totally, there are 6,136 *strings of pattern set*.

The design is implemented on two systems named *system-1* and *system-2* with index table depths of 512 and 1024, respectively. These systems are used to evaluate the trade off between hardware utilization and performance of insertion. Both of systems are based on SAX hash function and parallel Cuckoo hashing. Fig. 9 shows the number of insertions as well as number of string in every length after adding segments of long patterns. In every length of string, the number of insertions in *system-2* is just greater than the number of strings slightly. Unfortunately for *system-1*, the number of insertions is greater than the number of strings about 16%. In Fig. 9, *%Rehash* of *system-1* and *system-2* are also shown for clearer understanding why the number of insertions increases in *system-1*.

5. Experimental Results

Our design is developed in Verilog hardware description language and Xilinx's ISE 8.1i for

Table 1: Logic and memory cost of components in Xilinx Virtex-4

Component	Quantity	No. Block RAM	Logic cell	Note
T_1 & T_2	15x2	15	0	2 tables in a BRAM
T_3	16	39	0	The BRAM numbers for lengths 1-4 : 1; 5-8 : 2 9-13 : 3; 14-16: 4
T_4	1	8	0	
SAXs h_1 & h_2	15x2	0	840	
Control Block	16	0	175	
Multiplexer	15x2	0	300	
Comparator	18	0	1192	2 for long patterns
Register, buffer, LFSR...		0	475	
Total		62	2982	

hardware synthesis, mapping, and placing and routing. The target chip is a Virtex4 XCV4LX25 which has 24,192 logic cells and 72 RAM memory blocks. To reduce the number block RAMs, we can implement two index tables in the same block RAM: T_1 is in a low addresses part and T_2 is in a high addresses part. The block RAM of Xilinx FPGA can be configured as dual-port mode that can be accessed concurrently.

Based on our parallel pattern matching system described earlier, we can measure the cost from the required block RAMs and the number logic cells in Table 1. With 18 kbits, block RAM can be programmed from 16K x 1 to 512 x 36, in various depth and width configurations. Hence, we can configure tables T_1 & T_2 with size 1024 x 9 for each one. The size of table T_3 depends on the length of patterns. With the number of patterns in every Cuckoo module less than 500, we can set the depth of T_3 as 512. The width of T_3 can be determined by the length of patterns, adding 2 bits more for controlling long patterns. For the last table, the width of T_4 equals the number of its address bits adding 1 bit for suffix segment of long pattern. For this reason, the size of T_4 is 2^{13} x 14. For the narrow patterns of one character, we can compare directly to save hardware resource. Totally, we use only 62 RAM blocks and 2,982 logic cells to fit 64,873 characters of the entire rule set in the XCV4LX25 FPGA chip. The throughput of a design can be calculated by multiplying the clock frequency with the data width (8-bit) of incoming characters. For a design running at 285 MHz clock frequency, the throughput is 2.28 Gbps.

5.1. Dynamic Update for SNORT Pattern Set

Snort rule database must be updated to handle the new attacks. Normally, the web site of SNORT generates new rules every one or two weeks. To demonstrate dynamic update ability of our system, we collect the newest SNORT rule set on May 14, 2007 with 5,026 unique patterns and 68,266 characters. When comparing to Dec 15, 2006, there are 293 added unique patterns consisting 3,476 characters and 15 deleted unique patterns consisting 83 characters. In 293 added patterns, there are 65 patterns with the length more than 16 characters. We break them into segments as the above algorithm for long pattern. Totally, we have 381 short patterns and segments. These strings are added into the Cuckoo Hashing modules.

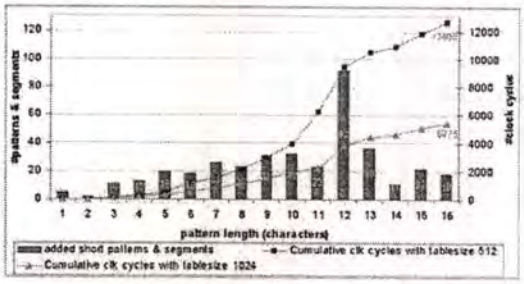


Fig. 10. The average insertion time for adding 381 new strings (patterns & segments)

Two simulation systems, *system-1* and *system-2*, with table size of 512 and 1,024 entries, respectively are compared the update time in Fig. 10. The figure shows the average insertion time in number of clock cycles for adding 381 new strings measured. Due to bigger table size of 1,024 entries, *system-2* takes 5,275 clock cycles to insert without rehashing. In *system-1*, rehashing happens at pattern lengths of 6, 11, 12 and 15 characters with small *%Rehash* from 0.6%-2.9%. However, the numbers of clock cycles at these lengths increase rapidly due to the high penalty of rehashing. Consequently, it causes 12,600 clock cycles of insertion time that is about 2.5 times comparing with *system-2*. For 15 deleted patterns, the process is fast with around 100 clock cycles for both systems. We assume that both systems are only running with frequency 200 MHz which are less than our synthesized results and can be achieved on many common Virtex FPGA boards. With 5,375 and 12,700 clock cycles, the update time is about 27 and 64 microseconds, respectively. These results show that both systems are very efficient for update new patterns without lengthy FPGA reconfiguration time.

5.2. Comparison with Other Systems

Table 2 shows the comparison of our synthesized systems with recent hashing systems built using FPGA. For ease of comparison, we also implement the system on other FPGA chips as Virtex2 XC2V3000 and VirtexPro XC2VPro20. Two metrics, Logic Cells per character (LCs/char) and SRAM bits per character (bits/char), are used to evaluate the efficiency of FPGA utilization. LCs/char is determined by dividing the total

Table 2: Comparison of FPGA-Based Systems for NIDS using Hash Method

System	Device (Xilinx)	Freq. (MHz)	No. chars	No. LCs	Mem (kbits)	LCs/ char	Mem per char (bits/char)	Throughput (Gbps)	PEM
FPGA-based Cuckoo Hashing (with pattern set in May 2007)	**XC4VLX25**	**285**	**68,266**	**2,982**	**1,116**	**0.043**	**16.74**	**2.28**	**53.02**
	XC2VP20	**272**		**3,028**		**0.044**		**2.18**	**49.55**
	XC2V3000	**223**		**3,028**		**0.044**		**1.78**	**40.45**
FPGA-based Cuckoo Hashing (with pattern set in Dec 2006)	**XC4VLX25**	**285**	**64,873**	**2,982**	**1,116**	**0.046**	**17.62**	**2.28**	**49.57**
	XC2VP20	**272**		**3,028**		**0.047**		**2.18**	**46.38**
	XC2V3000	**223**		**3,028**		**0.047**		**1.78**	**37.87**
V-HashMem [7]	XC2VP30	306	33,613	2,084	702	0.060	21.39	2.45	40.83
HashMem[6]	XC2V1000	250	18,636	2,570	630	0.140	34.62	2.00	14.50
	XC2VP70	338		2,570				2.70	19.60
PH-Mem[9]	XC2V1000	263	20,911	6,272	288	0.300	14.10	2.11	7.03
ROM+Coproc[8]	XC4VLX15	260	32,384	8,480	276	0.260	8.73	2.08	8.00

number of logic cells used in a design by the total number of characters programmed into the design. SRAM Bits/char is the ratio of memory blocks in bits per total number of characters. Both metrics are as small as possible.

With only 0.043-0.047 LCs/char, our systems are the most efficient one in using logic cells of FPGA. In addition, with 16.74-17.62 bits/char, the memory usage of our architecture is of very high density and is acceptable in comparing to other systems.

For throughput comparison, all systems in the Table 2 process one character (8-bit) per clock cycle. Our throughput is a bit smaller than some of other implementations. That is because we trade off speed with hardware area. However, the Performance Efficiency Metric (PEM) is used as the ratio of throughput in Gbps to the Logic Cells per pattern character instead. Although it does not take account the memory usage, it is commonly used in comparison of previous systems. With PEMs of 37.87-53.02, our system is far better than other FPGA hashing systems up to 1.22 - 6.63 folds.

6. Conclusion and Future Works

A novel FPGA-based pattern matching based on Cuckoo Hashing for NIDSs/NIPSs is proposed. According to the implementation results, the utilization of our system is the best when compared with other previous systems and the achievable throughput can be up to 2.28 Gbits/s. One of remarkable features of our system is dynamic pattern insertions and deletions with no FPGA reconfiguration. For future work, our system will be extended to pipeline several characters instead of one character per clock cycle. It is expected that the throughput will be multiplied proportionally.

Acknowledgment

We would like to acknowledge AUN-SeedNet Program of JICA for the scholarship and Xilinx, Inc. for donating the software tools.

References

[1] SNORT: The Open Source Network Intrusion Detection System. http://www.snort.org

[2] R. Pagh, F. F. Rodler, "Cuckoo hashing," *Journal of Algorithms*, vol 51, 2004, pp. 122-144.

[3] J. Moscola, J. Lockwood, R. P. Loui and M. Pachos, "Implementation of a content-scanning module for an internet firewall," *the 11th IEEE Symposium on Field-Programmable Custom Computing Machines*, 2003, pp.31–38.

[4] C. R. Clark and D. E. Schimmel, "Scalable pattern matching for high speed networks," *the 12th IEEE Symposium on Field-Programmable Custom Computing Machines*, 2004, pp. 249–257.

[5] I. Sourdis and D. Pnevmatikatos, "Pre-decoded cams for efficient and high-speed NIDS pattern matching," *the 12th IEEE Symposium on Field-Programmable Custom Computing Machines*, 2004, pp.258–267.

[6] G. Papadopoulos and D. Pnevmatikatos, "Hashing + memory = low cost, exact pattern matching," *the 15th International Conference on Field Programmable Logic and Applications (FPL)*, 2005, pp. 39-44.

[7] D. Pnevmatikatos and A. Arelakis, "Variable-length hashing for exact pattern matching," *the 16th International Conference on FPL*, 2006, pp. 1-6.

[8] Y. H. Cho and W. H. M-Smith, "Fast reconfiguring deep packet filter for 1+ gigabit network," *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2005, pp. 215–224.

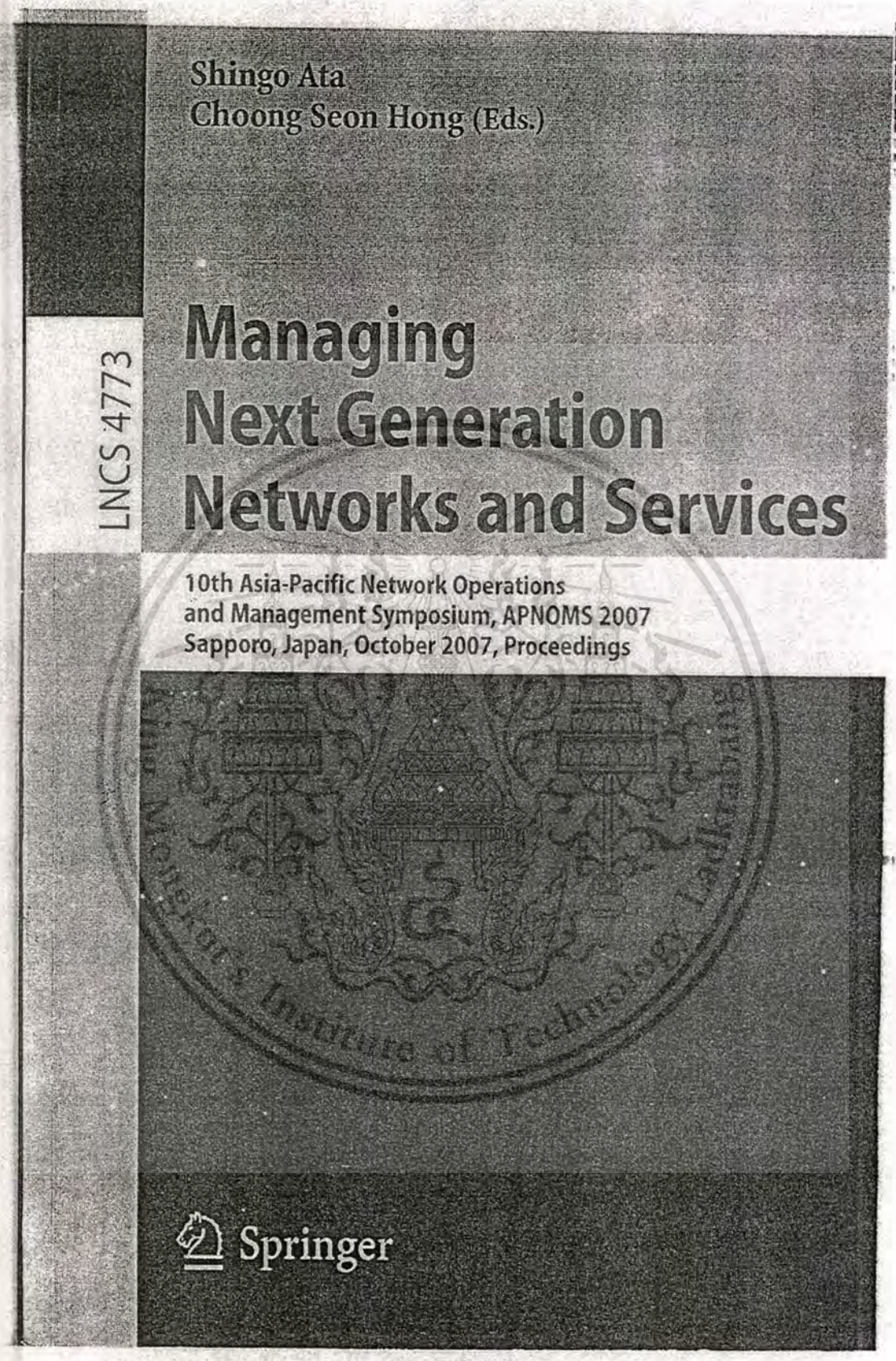
[9] I. Sourdis, D. Pnevmatikatos, S. Wong and S. Vassiliadis, "A reconfigurable perfect-hashing scheme for packet inspection," *the 15th International Conference on FPL*, 2005, pp. 644-647.

[10] S. Dharmapurikar, P. Krishnamurthy, T. Spoull and J. Lockwood, "Deep Packet Inspection using Bloom Filters," *IEEE Hot Interconnects*, 2003, pp. 44-51.

[11] J. L. Carter and M. N. Wegman, "Universal classes of hash functions," *Journal of Computer System Sci.*, vol. 18, 1979, pp. 143–154.

[12] M. V. Ramakrishna and J., Zobel, "Performance Practice of String Hashing Functions," *the Fifth International Conference on Database Systems for Advanced Applications*, vol. 6, 1997, pp.215-224.

[13] Xilinx App. Note, http://www.xilinx.com/bvdocs/appnotes/xapp211.pdf



FPGA-Based Cuckoo Hashing for Pattern Matching in NIDS/NIPS

Thinh Ngoc Tran and Surin Kittitornkun

Dept.of Computer Engineering, Faculty of Engineering,
King Mongkut's Institute of Technology Ladkrabang, Bangkok, 10520 Thailand
tnthinh@dit.hcmut.edu.vn,kksurin@kmitl.ac.th

Abstract. Pattern matching for network intrusion/prevention detection demands exceptionally high throughput with recent updates to support new attack patterns. This paper describes a novel FPGA-based pattern matching architecture using a recent hashing algorithm called *Cuckoo Hashing*. The proposed architecture features on-the-fly pattern updates without reconfiguration, more efficient hardware utilization, and higher throughput. Through various algorithmic changes of Cuckoo Hashing, we can implement parallel pattern matching on SRAM-based FPGA. Our system can accommodate the newest Snort rule-set, an open source Network Intrusion Detection/Prevention System, and achieve the highest utilization in terms of SRAM per character and Logic Cells per character at 15.63 bits/character and 0.033 Logic Cells/character, respectively on major Xilinx Virtex FPGA architectures. Compared to others, ours is more efficient than any other Xilinx FPGA architectures.

Keywords: NIDS, NIPS, Cuckoo Hashing, FPGA, Pattern Matching.

1 Introduction

In recent years, Network Intrusion Detection/Prevention Systems (NIDSs/NIPSs) are more and more necessary for network security. Normally, traditional firewalls only examine packet headers to determine whether to block or pass the packets. Due to busy network traffic and smart attacking schemes, firewalls are not as effective as they used to be. NIDSs/NIPSs are designed to examine not only the headers but also the payload of packets to match and identify intrusions.

To define suspicious activities, most modern NIDSs/NIPSs rely on a set of rules which are applied to matching packets. At the heart of almost every NIDS is pattern matching algorithms. For example, Snort [1] is an open source network intrusion prevention and detection system utilizing a rule-driven language, which combines the benefits of signature, protocol and anomaly based inspection methods. Snort uses a set of rules to filter the incoming packets. As the number of known attacks grows, the patterns for these attacks are made into Snort signatures (patterns or strings). The simple rule structure allows flexibility and convenience in configuring Snort. However, unfortunately, checking every byte of every packet to see if it matches one

S. Ata and C.S. Hong (Eds.): APNOMS 2007, LNCS 4773, pp. 334–343, 2007.

of a set of thousand strings becomes a computationally intensive task as the highest network speed increases to several gigabits/second.

To improve the performance of Snort, various implementations of FPGA-based systems have been proposed. These systems can simultaneously process thousands of rules relying on native parallelism of hardware so their throughput can satisfy current gigabit networks. However, the drawback of hardware-based systems is the flexibility. With emergence of new worms and viruses, the rule set must be frequently updated. Although SRAM-based field programmable gate array (FPGA) can be reconfigured; the process of recompiling the updated FPGA design can be lengthy. For recently proposed FPGA-based NIDSs/NIPSs, adding or subtracting any number of rules requires recompilation of some parts or the entire design. The compilation process takes several minutes to several hours to complete. Today, such latency in compilation may be not accepted for most networks when new attacks are released at a high frequency. It is necessary to update pattern database faster to reduce down time.

Based on Cuckoo Hashing [2], we implement a novel architecture of variable-length pattern matching best suited for FPGA. New patterns can be added to or removed out of the Cuckoo hash tables. Unlike most previous FPGA-based systems, the proposed architecture can update the rule set on-the-fly without reconfiguration thanks to Cuckoo Hashing. Our contributions also include parallel Cuckoo hashing, better hardware utilization and high matching throughput reaching multiple gigabits per second.

The paper is organized as follows. In section 2, some previous FPGA implementations of pattern matching and Cuckoo Hashing are presented. Section 3 proposes the architecture of FPGA-based Cuckoo Hashing. Next, the FPGA implementation of Cuckoo Hashing for multiple pattern matching and its experimental results are discussed in section 4 and 5, respectively. Finally, future works are suggested in the conclusion.

2 Background and Related Works

2.1 FPGA Implementations of NIDS

For a line speed of gigabit network, many previous FPGA approaches of NIDS are proposed. Some of them as [3, 4] implement regular expression matching (NFAs/DFAs) on FPGA. Other approach [5] uses content addressable memory (CAM). While the processing speed is fast, they suffer the two scalability problems such as too many states consume too many hardware resources and FPGA device has to re-program every time patterns be changed. Furthermore, incoming characters are broadcasted to all character matchers. This requires the use of extensive pipelined trees to achieve a high clock rate. The clock frequency of these architectures tends to drop gradually as the number of patterns increases.

Another hardware approach implements hash functions [6-10] to find a candidate of pattern match. Dharmapurikar proposed to use Bloom Filters to do the deep packet inspection [6]. Unlike other hardware approaches mentioned above, this method does not require reprogramming of FPGA for patterns added. Nevertheless, the blooming

filter method could generate a false positive match, which requires extra cost of hardware for the match rechecked.

Dionisios et al. proposed CRC hashing named HashMem [8] system using simple CRC polynomials hashing implemented with XOR gates that can use efficient area resource of FPGA than before. For the improvement of memory density and logic gate count, they implemented the V-HashMem [9]. However, these systems have some drawbacks: 1) To reduce the sparse of memory and avoiding collision, CRC hash functions have to be chosen carefully depending on specific pattern groups, 2) since pattern set is dependent, probability of redesigning the system and the reprogramming the FPGA is very high every time patterns are changed.

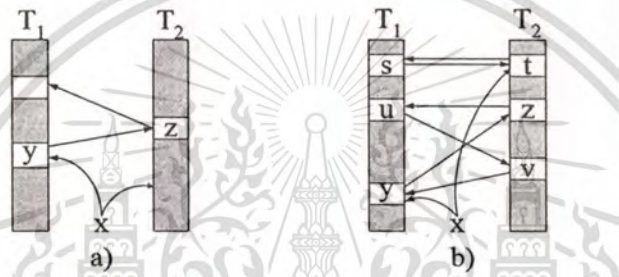


Fig. 1. Original Cuckoo Hashing [2], a) Key x is successfully inserted by moving y and z, b) Key x cannot be accommodated and a rehash is necessary

2.2 Cuckoo Hashing

Cuckoo hashing is proposed by Pagh and Rodler [2] as an algorithm for maintaining a dynamic dictionary with constant lookup time in the worst case scenario. The algorithm utilizes two tables T_1 and T_2 of size $m = (1+\varepsilon)n$ for some constant $\varepsilon > 0$, where n is the number of elements (strings). Cuckoo hashing guarantees $O(n)$ space and does not need perfect hash functions that is very complicated if the set of elements stored changes dynamically under the insertion and deletion. Given two hash functions h_1 and h_2 from universe U to $[m]$, one maintains the invariant that a key x presently stored in the data structure occupies either cell $T_1[h_1(x)]$ or $T_2[h_2(x)]$ but not both. Given this invariant and the property that h_1 and h_2 may be evaluated in constant time, lookup and deletion procedures run in worst case constant time.

Pagh and Rodler described a simple procedure for inserting a new key x in expected constant time. If cell $T_1[h_1(x)]$ is empty, then x is placed there and the insertion is complete; if this cell is occupied by a key y which necessarily satisfies $h_1(x) = h_1(y)$, then x is put in cell $T_1[h_1(x)]$ anyway, and y is kicked out. Then, y is put into the cell $T_2[h_2(y)]$ of second table in the same way, which may leave another key z with $h_2(y) = h_2(z)$ nestless. In this case, z is placed in cell $T_1[h_1(z)]$, and continues until the key that is currently nestless can be placed in an empty cell as in Figure 1(a). However, it can be seen that the cuckoo process may not terminate as Figure 1(b). As a result, the number of iterations is bounded by a bound *MaxLoop* chosen beforehand. In that case everything is rehashed by reorganizing the hash table with new hash

functions h_1 and h_2 and newly inserting all keys currently stored in the data structure, recursively using the same insertion procedure for each key.

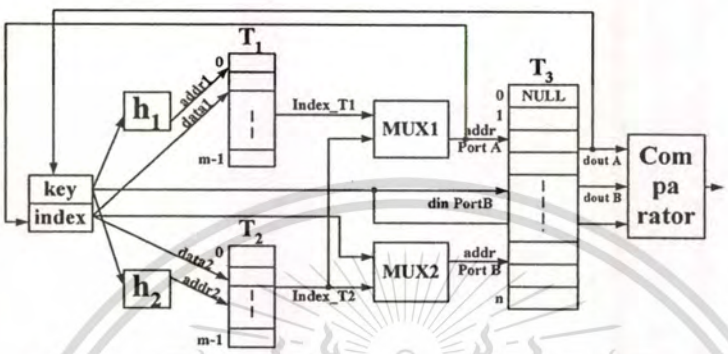


Fig. 2. FPGA-based Cuckoo Hashing. Tables T_1 and T_2 store the key indices; Table T_3 stores the keys.

3 FPGA-Based Cuckoo Hashing

To apply FPGA-based Cuckoo Hashing for variable pattern lengths, the memory efficient for storing patterns in hash tables is required because of the limited numbers of hardware resources. In the original Cuckoo Hashing, the width of table must be equal to the longest pattern in the rule set. Thus, the remaining short patterns will waste so much memory. In order to increase memory utilization, we build up the hashing module for each length of pattern and use indirect storage. Small and sparse hash tables contain indices of keys which are the addresses of a condensed pattern-stored table.

The architecture of a FPGA-based Cuckoo Hashing module as shown in Figure 2 includes three tables: two index tables (hash tables) T_1 and T_2 are single-port SRAMs and a pattern-stored table T_3 is the double-port SRAM for concurrent processing. Hash functions are any universal hashes that can change if they are required to rehash. Two multiplexers are used to select addresses for two ports of T_3. The output of first multiplexer (*MUX1*) is the address of *port A* that is the read-only port. *MUX1*'s inputs are the output value of T_1 (*index_*T_1) and T_2 (*index_*T_2). The output of second multiplexer (*MUX2*) is the address of *port B* that is both reading and writing port. *MUX2* selects *index_*T_2 as lookup function or *index* of key as insertion function.

3.1 Parallel Lookup

A lookup function of element (key) x can be divided into three phases. In each phase, instructions can be processed simultaneously. In the first phase, x is hashed by two hash functions in parallel. The values of two hash functions are used as the address for reading data of two index tables. In the second phase, output data of two index tables are used as the address for two ports of T_3 for reading. In the third phase, the

data outputs of T_3 are compared with the incoming character to determine the match. Following is the pseudo-code of parallel Cuckoo lookup function.

```
function lookup(x)
  select index_T1 in MUX1  and index_T2 in MUX2;
         index_T1 = T1(h1(x));                  // phase1
         index_T2 = T2(h2(x));
         dataA = PortA(index_T1);               // phase2
         dataB = PortB(index_T2);
         return(dataA = x or dataB = x);        // phase3
end
```

By processing simultaneously two hash functions and pipelining every step of whole process of system, the FPGA-based Cuckoo Hashing can look up the keys in streaming with each key in clock cycle.

3.2 Online Insertion and Deletion

When a key insertion occurs, we consider both key and its index. The key is used as an input for two hash functions and stores in table T_3. Its index is used for storing in table T_1 or T_2 and also as the address for lookup the space of key in T_3.

The insertion of element x as description in C-like pseudo-code below is only started after the lookup process failed. If one of outputs of two index tables is empty (NULL), *x's* index is inserted into T_1 or T_2. This is an improvement as compared with the original Cuckoo Hashing. The original one always inserts the index into T_1 without referring to the value of T_2. Otherwise, we consider both tables to reduce the insertion time. If both of the outputs of T_1 and T_2 are not NULL, we insert the key index into T_1. At the same time, the data from T_3 and its address, *index_T1*, will be written into the key storage and the index storage for starting of cuckoo process. Then, the *MaxLoop* is decreased and the key value is hashed by hash function h_2. The output data will be checked for whether the value is NULL. If it is NULL, the process ends with successful insertion. Conversely, the process is continued by taking in turns hashing from h_2 to h_1.

The worst case happens when the *MaxLoop* decreases to zero. Hence, rehashing is required. Two new hash functions h_1 and h_2 are issued by a pseudo-random number generator. As rehashing cost can be expensive, the choice of good hash function has to be discussed in the next subsection.

For deletion, it is as simple as the lookup process. If the lookup succeeds, the deletion resets the key value to become NULL and the key index to become index of either table T_1 or T_2. We then write key value to table T_3. After that, we reset the key index to NULL and write to the appropriate index table T_1 or T_2. Rehashing for deletion can be required when table T_3 is too sparse. However, this rehashing does not require new hash functions.

```
procedure insert(x)
    if (lookup(x)) return;
    select index_T1 in MUX1  and index in MUX2;
    PortB(index) = x;
    if(index_T1 == NULL){
       T1(h1(x)) = index;
```

```
    return;}
  else if(index_T2 == NULL){
    T2(h2(x))= index;
    return;}
  loop MaxLoop times
    if (select index_T1 in MUX1){              //phase1
      key = PortA(index_T1);
      index = index_T1;
      select index_T2 in MUX1;}
    else{ // (select index_T2 in MUX1)
      key = PortA(index_T2);
      index = index_T2;
      select index_T1 in MUX1;}
    index_T1 = T1(h1(x));                      //phase2
    index_T2 = T2(h2(x));
    if (select index_T1 in MUX1){              //phase3
      if(index_T1 == NULL) return;
      dataA = PortA(index_T1);}
    else{ // (select index_T2 in MUX1)
      if(index_T2 == NULL) return;
      dataA = PortA(index_T2);}
  end loop
  rehash(); insert(x);
end
```

3.3 Selection of Hash Functions

The choice of hash functions greatly affects the performance of the system. Moreover, the probability for rehashing in Cuckoo Hashing is also based on the randomized property of hash functions. In Cuckoo Hashing, the authors use the Siegel's universal hashing [11] that has a constant evaluation time. However, this constant time is not small and complex in practice. In this section, we discuss some other simple and fast hash functions for string and chose the best one that easily be implemented on hardware.

The universal class of hash functions [12] has the good performance which can be guaranteed independent of the input keys by randomly selecting hash functions from the family. An example of such construct is modular hash functions. However, it is not suitable for hardware because of the complexity of the prime modulo operation. A fast way of generating a class of universal hash function without the modular operation and hardware-friend, tabulation based hashing method [12], is defined as follows:

$$H_t(x) = a_t[0][x_0] \oplus a_t[1][x_1] \oplus \ldots \oplus a_t[n-1][x_{n-1}] \quad (1)$$

A randomized table contains a 2-D array of random numbers in the hashing space. A key is string n characters $x_0x_1..x_{n-1}$ and the hash process is calculated by bit-wise exclusive-or ($\oplus$) a sequence of values $a_t[i][x_i]$, which is indexed by each byte value of x_i and position of i in the string. The drawback of this method is that the size of random table is very large and depends on the length of key.

Another class of simple hash function for hashing character strings named *shift-add-xor* (SAX) [13] is proposed by Ramakrishna et al. The function utilizes only the simple and fast operations of shift, exclusive-or and add.

$$H_i = H_{i-1} \oplus (S_L(H_{i-1}) + S_R(H_{i-1}) + c_i) \tag{2}$$

Two operators S_L and S_R denote the shift left and right, respectively. The symbol c_i is the character i^{th} of string and H_i is an intermediate hash value after examination of i characters. The initial value H_0 can be generated randomly. The authors have shown that the class is likely to be universal. Good performance can be achieved in practice by randomly choosing functions from this class. Moreover, the main advantages of SAX over random-table are a very small space of hardware, and the simple architecture achieving high clock frequency hence the system is faster. To generate the new SAX hash function in case of rehashing, we only need to change the value of H_0 by simple pseudo-random circuit LFSR [14]. Therefore, the SAX is the best choice for our system. The practical performance will be shown in the next section.

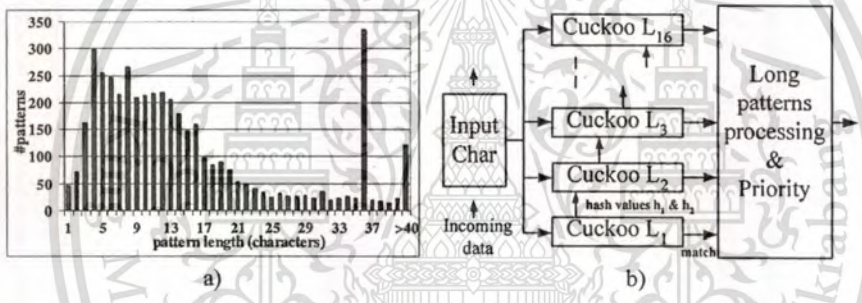


Fig. 3. a) Pattern length distribution of pattern set of SNORT in Dec 2006. b) Overview of FPGA-based Cuckoo Hashing in NIDS.

4 Implementation of FPGA-Based Cuckoo Hashing for Variable-Length Pattern Matching in NIDSs/NIPSs

In Dec 2006, there are 4,748 unique patterns which contain 64,873 characters in Snorts rule set. Figure 3(a) shows the distribution of the pattern lengths in the Snort database. For the pattern matching in the NIDS, the patterns are searched on incoming packets. The matched pattern can occur anywhere as the longest substring. In order to process at the network speed in Gigabits per second, we have to construct Cuckoo Hashing module for every length of pattern from 1 up to 109 characters. Fortunately, in Figure 3(a), we can see that 65% of total numbers of patterns are up to 16 characters long. Therefore, we build the Cuckoo Hashing modules for patterns which are less than or equal 16 characters according to this fact. For longer patterns, we can break them to shorter segments so that we can insert those segments to the Cuckoo modules of short patterns. We then use simple address link-lists to combine these segments later. Figure 3(b) shows the overview of our NIDS system.

As discussed in the above section, SAX hash function is the best choice of our system. Now, we implement SAX and random table hash with patterns of the length from 2 to 16 characters for practical comparison. Figure 4 shows the number of insertion times of Cuckoo Hashing with two methods: random tables and SAX function in which the size of each index table is 512. The lines *SAX_par* and *Tabulation_par* are the hash functions whose architectures are changed as in section 2 of this paper with keys inserted in parallel. The results show that the parallel systems have the insertion time less than the original systems by 20% and the performance of SAX hash function is close to that of random table.

Moreover, we can reduce significantly large amount of hardware area resource by accumulative characteristic of SAX hash function. From (2), to calculate hash value of pattern of length i characters in the hash module i^{th}, the requisite inputs are hash value of i-1 characters calculated beforehand in the hash module $(i\text{-}1)^{th}$ and the i^{th} character. Therefore, the value of previous hash module can be reused for the next hash module. This property of the hash functions results in a regular and less resource-consuming hash function. In comparison with previous implementations [6-10], the module i^{th} requires all i characters of the input string for every calculation and no reused of previous hash values at all. Their weak points lead to the consumption of a remarkable the number of logic gates for implementation of hash functions.

To reduce the number of memory blocks in FPGA, we can implement two index tables in the same block RAM; the T_1 is in a low addresses part and the T_2 is in a high addresses part. The block RAM of Xilinx FPGA can be configured as dual-port mode that can be accessed concurrently. Therefore, the performance of system still remains the same.

With improvements in the architecture, we can reduce a large amount of hardware resources. In next section, we will illustrate the experimental results on FPGA.

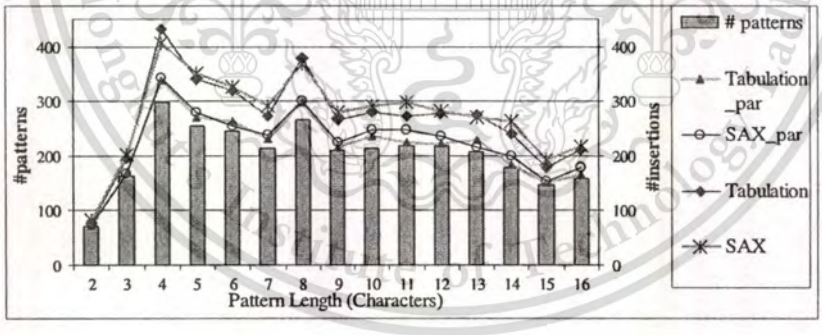


Fig. 4. The number of insertions of various hash functions vs. pattern lengths. Bar graphs are the numbers of patterns. The lines are the numbers of insertions. Index table size is 512. The number of trials is 1,000.

5 Experimental Results

Our design is developed in Verilog hardware description language and Xilinx's ISE 8.1i for hardware synthesis, mapping, and placing and routing. The target chip is a

Virtex4 XC4VLX25 which has 24,192 logic cells and 72 RAM memory blocks. Based on our parallel Cuckoo Hashing pattern matching system described earlier, the numbers of required memory blocks are 39 for pattern storage and 15 for index memories at 1k x 18bit for each block. Besides, we need 1 more memory block to match with the narrow patterns of one character. Totally, we use only 2,142 logic cells and 55 RAM blocks to fit 64,873 characters of entire rule set in the XC4VLX25 FPGA chip. The throughput of a design is calculated by multiplying the clock frequency with the data width (8-bit) of incoming characters. For a design running at 285 MHz clock frequency, the throughput is 2.28 Gigabits per second (Gbps).

Table 1 shows the comparison of our system with other hashing systems. For ease of comparison, we also implement the system on other FPGA chips as Virtex2 XC2V2000 and VirtexPro XC2VPro20. Two metrics, Logic Cells per character (LCs/char) and SRAM bits per character (bits/char), are used to compare hardware NIDS designs. LCs/char is determined by dividing the total number of logic cells used in a design by the total number of characters programmed into the design. SRAM Bits/char is the ratio of memory blocks in bits per total number of characters. With only about 0.033-0.035, our LCs/char is twice smaller than the best one, V-HashMem [9], of previous systems. With 15.63 bits/char, the memory usage of our architecture is of very high density and is acceptable in comparing to other systems.

Another metric used to compare hardware NIDS designs is the Performance Efficiency Metric (PEM) that is the ratio of throughput in Gbps to the Logic Cells per pattern character. PEM of our system is at 62.29 for Virtex2Pro and 69.09 for Virtex4 devices, the best one among all of FPGA hashing systems.

Table 1. Comparison of FPGA-Based Systems for NIDS using hash functions

System	Dev.-XC (Xilinx)	Freq. (MHz)	No. chars	No. LCs	Mem (kbits)	LCs/ char	Mem per char (bits/char)	Through-Put (Gbps)	PEM
Our System	**4VLX25**	**285**	**64,873**	**2,142**	**990**	**0.033**	**15.63**	**2.28**	**69.09**
	2VP20	**272**		**2,328**		**0.035**		**2.18**	**62.29**
	2V2000	**223**		**2,328**		**0.035**		**1.78**	**50.86**
V-HashMem [9]	2VP30	306	33,613	2,084	702	0.060	21.39	2.45	40.83
HashMem [8]	2V1000	250	18,636	2,570	630	0.140	34.62	2.00	14.50
	2VP70	338		2,570				2.70	19.60
PH-Mem [10]	2V1000	263	20,911	6,272	288	0.300	14.10	2.11	7.03
ROM+Coproc[7]	4VLX15	260	32,384	8,480	276	0.260	8.73	2.08	8.00

6 Conclusion and Future Works

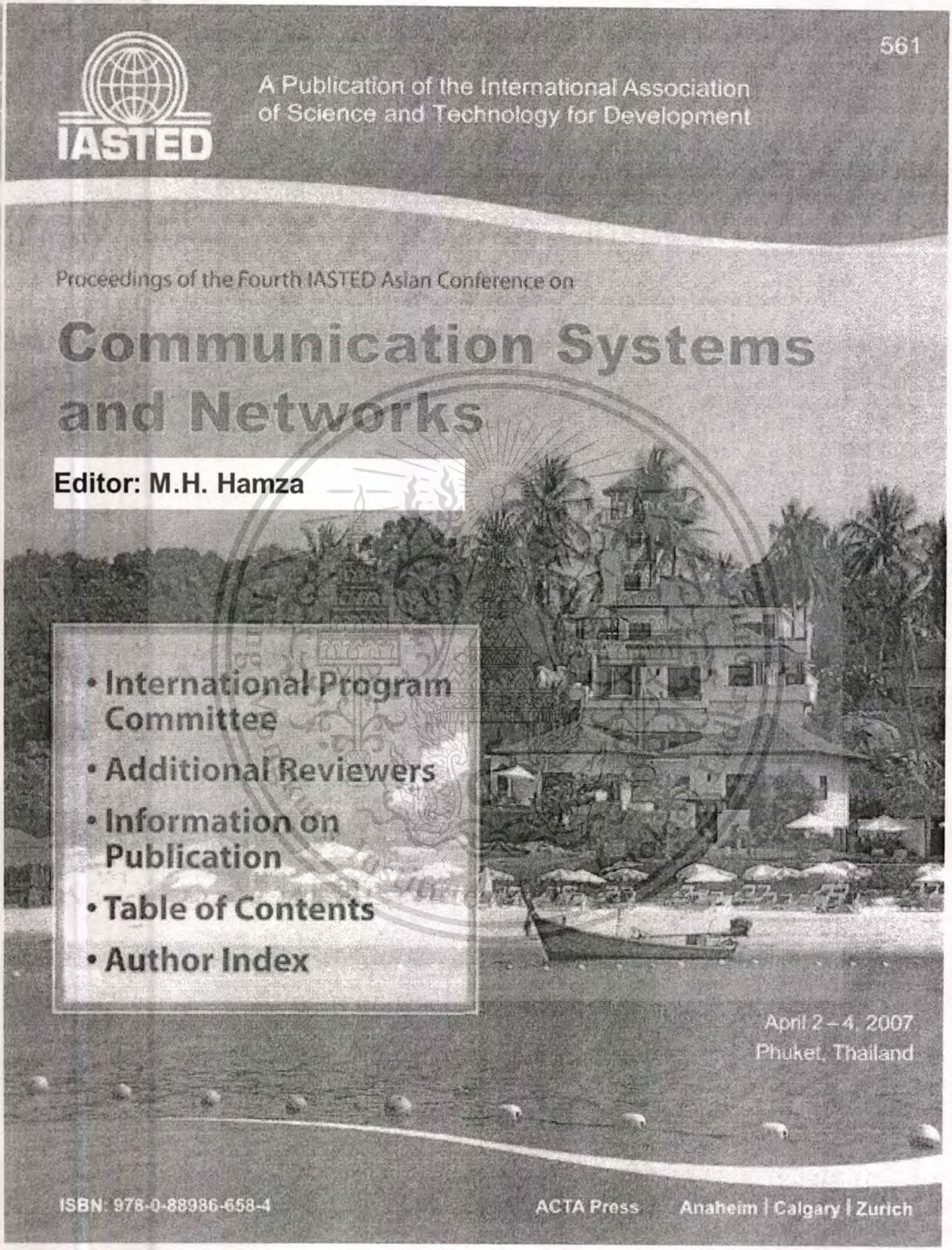
A novel FPGA-based pattern matching based on Cuckoo Hashing for NIDSs/NIPSs is proposed. In addition, selections of practical hash functions are also discussed. As a

result, the most suitable one is shift-add-xor (SAX). According to the implementation results, the utilization of our system is the best when compared with other previous systems and the achievable throughput can be up to 2.28 Gbits/s. One of remarkable features of our system is dynamic pattern insertions and deletions with no FPGA reconfiguration. For future work, we will improve our system by reducing the size of hash tables. In addition, IP header matching will be combined to complete the system.

Acknowledgments. We would like to acknowledge AUN-SeedNet Program of JICA for the scholarship and Xilinx, Inc. for donating the software tools.

References

1. SNORT: The Open Source Network Intrusion Detection System. http://www.snort.org
2. Pagh, R., Rodler, F.F.: Cuckoo hashing. Journal of Algorithms. 51, 122–144 (2004)
3. Moscola, J., Lockwood, J., Loui, R.P., Pachos, M.: Implementation of a content-scanning module for an internet firewall. In: Proceedings of the 11th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), pp. 31–38. IEEE Computer Society Press, Los Alamitos (2003)
4. Clark, C.R., Schimmel, D.E.: Scalable pattern matching for high speed networks. In: Proceedings of the 12th IEEE Symposium on FCCM, pp. 249–257. IEEE Computer Society Press, Los Alamitos (2004)
5. Sourdis, I., Pnevmatikatos, D.: Pre-decoded cams for efficient and high-speed NIDS pattern matching. In: Proceedings of the 12th IEEE Symposium on FCCM, pp. 258–267. IEEE Computer Society Press, Los Alamitos (2004)
6. Dharmapurikar, S., Krishnamurthy, P., Spoull, T., Lockwood, J.: Deep Packet Inspection using Bloom Filters. In: Hot Interconnects, pp. 44–51 (2003)
7. Cho, Y.H., M-Smith, W.H.: Fast reconfiguring deep packet filter for 1+ gigabit network. In: Proceedings of the 13th IEEE Symposium on FCCM, pp. 215–224. IEEE Computer Society Press, Los Alamitos (2005)
8. Papadopoulos, G., Pnevmatikatos, D.: Hashing + memory = low cost, exact pattern matching. In: Proceedings of the 15th International Conference on Field Programmable Logic and Applications, pp. 39–44 (2005)
9. Pnevmatikatos, D., Arelakis, A.: Variable-length hashing for exact pattern matching. In: Proceedings of the 16th International Conference on Field Programmable Logic and Applications, pp. 1–6 (2006)
10. Sourdis, I., Pnevmatikatos, D., Wong, S., Vassiliadis, S.: A reconfigurable perfect-hashing scheme for packet inspection. In: Proceedings of the 15th International Conference on Field Programmable Logic and Applications, pp. 644–647 (2005)
11. Siegel, A.: On universal classes of fast high performance hash functions, their time–space tradeoff, and their applications. In: Proceedings of the 30th Annual Symposium on Foundations of Computer Science, pp. 20–25. IEEE Computer Society Press, Los Alamitos (1989)
12. Carter, J.L., Wegman, M.N.: Universal classes of hash functions. Journal of Computer System Sci. 18, 143–154 (1979)
13. Ramakrishna, M.V., Zobel, J.: Performance in Practice of String Hashing Functions. In: Proceedings of the Fifth International Conference on Database Systems for Advanced Applications, vol. 6, pp. 215–224 (1997)
14. Xilinx Application Note. http://www.xilinx.com/bvdocs/appnotes/xapp211.pdf



Proceedings of the Fourth IASTED Asian Conference
COMMUNICATION SYSTEMS AND NETWORKS
April 2–4, 2007, Phuket, Thailand
ISBN CD: 978-0-88986-658-4

SYSTOLIC ARRAY FOR STRING MATCHING IN NIDS

Tran Ngoc Thinh
Dept.of Computer Engineering, Faculty of Engineering
King Mongkut's Institute of Technology Ladkrabang
Bangkok, 10520 Thailand
tnthinh@dit.hcmut.edu.vn

Surin Kittitornkun
Dept.of Computer Engineering, Faculty of Engineering
King Mongkut's Institute of Technology Ladkrabang
Bangkok, 10520 Thailand
kksurin@kmitl.ac.th

ABSTRACT
In this paper, the rule set of a Network Intrusion Detection System, SNORT [1], is deeply analyzed and a compact encoding method to reduce the memory space for storing the payload content strings of entire rules is proposed. This method can approximately reduce up to 50% of area cost when compared with traditional ASCII coding method. After that, we implement a reconfigurable hardware sub-system for Snort payload matching using systolic design technique. Our system is a processor array architecture that can match strings with throughput up to 3.86 Gbps and area efficient manner.

KEY WORDS
Compact Encoding, Systolic Array, String Matching, NIDS, FPGA

1 Introduction

To protect the network from the rapid increasing of malicious attacks, Network Intrusion Detection Systems (NIDSs) have been developed to integrate with general purpose firewalls. Traditional firewalls usually examine packet headers to determine whether to block or allow packets. Due to busy network traffic and smart attacking schemes, firewalls are not as effective as they used to be. NIDSs use patterns of well-known attacks to match and identify intrusions. They monitor incoming network traffic for predefined suspicious activities or data patterns and notify to system administrators when malicious traffic is detected so that appropriate action may be taken. Therefore, the NIDSs often rely on exact string matching of packet payloads to detect hostile packets and string matching is the most computationally expensive step of the detection process.

Snort is an open source network intrusion prevention and detection system utilizing a rule-driven language, which combines the benefits of signature, protocol and anomaly based inspection methods. Snort uses a set of rules to filter the incoming packets. As the number of known attacks grows, the patterns for these attacks are made into Snort signatures. The simple rule structure allows flexibility and convenience in configuring Snort. However, there is a performance disadvantage of having a long list of rules.

To improve the performance of SNORT, various implementations of FPGA-based hardware systems have been proposed. These systems can simultaneously process thousands of rules relying on native parallelism of hardware so their throughput can satisfy current gigabit networks.

The most common approach is the regular expressions matching, implemented using Finite Automata (NFAs or DFAs) [2, 3, 4]. This approach generates regular expressions for every pattern or group of patterns, represents the regular expression as a finite automata graph and then translates them directly to FPGA circuitry. By adding predecoded wide parallel inputs to standard implementations, area and throughput performance are improved significantly. For more improvement of efficient area, L.Tan et al. [5] proposed tiny Aho-Corasick state machines and Jung et al. [6] verified successful this system on FPGA.

Another more straightforward approach for FPGA-based string matching is the use of regular CAM [7, 8] and discrete comparators [9, 10]. Current FPGAs give designers the opportunity to use integrated block RAMs for constructing regular CAM. This is a simple procedure that achieves modest performance, in most cases better than simple N/DFAs architectures. Cho et al. created a content-based firewall using discrete logic filters [9]. They created automated techniques to generate highly parallel comparator structures that can be quickly configured. This work was expanded to include logic re-use and read only memory [10].

The drawback of hardware-based system is requirement of a large amount of resources for storing and processing entire SNORT rule set. Therefore, the common factor of efforts is continuous drive for lower and lower cost with the same or better of performance.

In order to reduce the area cost, we analyze and preprocess entire SNORT rule set before storing and matching it in hardware. By applying the compact encoding method [11], we separate entire rule to smaller groups that can be encoded 3-5 bits instead of 8 bits as traditional ASCII code. We then use a simple systolic technique to search on these groups. With the simplicity in architecture, our implementations achieve the highest throughput, 3.31-3.86 Gbps, when compared with previous implementations that process one character per time.

The paper is organized as follows. In section 2, our design methodology is elaborated. Next, the FPGA implementation and its experimental results are discussed in

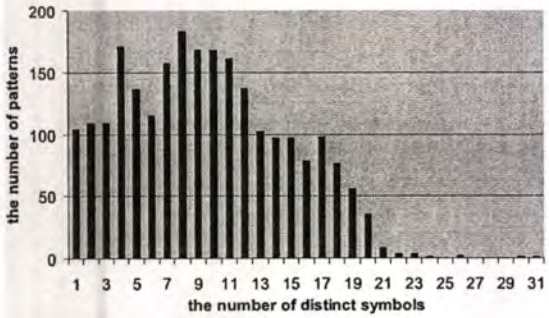


Figure 1. Histogram of the number of distinct characters of pattern strings

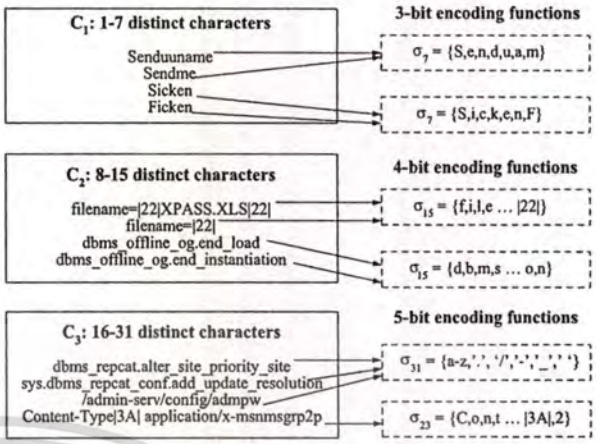


Figure 2. Compact Encoding Method for patterns in SNORT

section 3. Finally, future works are suggested in our conclusion.

2 Our Design Methodology

2.1 Compact encoding of pattern and text

In above section, most of the mentioned systems represent the pattern set and the incoming text in ASCII code with 8-bit data. Moreover, there are thousands of rules in SNORT with over 37K of characters and the traditional storing method occupies a lot of logic gates or memory of hardware. Therefore, we apply a compact encoding method for rule set of NIDS to save the area of hardware. This method is proposed by S.Kim et al. [11]. One advantage of using a compact encoding scheme is that multiple characters can be compared simultaneously.

For a given pattern P and text T, we firstly count the number of distinct characters in P. Let D be the number of distinct characters in P and E be the smallest integer such that $(2^E - 1) \geq D$. Then we can encode any character in P and T with E bits by assigning a distinct E bit for each character in P and assigning a distinct E bit for any character that does not occur in P but occurs in T. The following example illustrates this scheme.

Example 1: Consider a pattern P = "encoding" and T = "Compact encoding can". Since we have 7 distinct characters in P, each character can be encoded in 3 bits ($(2^3 - 1) \geq 7$). Let's introduce a function ENCODE for encoding characters: ENCODE(e) = 001, ENCODE(n) = 010, ENCODE(c) = 011, ENCODE(o) = 100, ENCODE(d) = 101, ENCODE(i) = 110, ENCODE(g) = 111, and ENCODE(-) = 000 for any character - that does not occur in P. Then, P is encoded as 001 010 011 100 101 110 010 111 and T as 000 100 000 000 000 011 000 000 001 010 011 100 101 110 010 111 000 011 010.

2.2 Analysis of ruleset

In June 2006, there are 58,158 characters in 3,462 string patterns of Snorts rule set. However, during the analysis, we found that a lot of the rules look for the same string patterns but with different headers. Through simple preprocessing, we can eliminate duplicate patterns and reduce the number of patterns from 3,462 down to 2,378 unique patterns which contain 37,873 characters.

In the entire pattern set, there are 241 distinct characters, and the compact encoding method is not efficient for large distinct characters. Therefore, we have to divide it to groups with smaller number of distinct characters. Following parts are the analysis of how to group the pattern set.

Figure 1 shows a histogram of the number of distinct characters of each unique pattern in the default database. In rule set, the maximum number of distinct characters in one pattern is 31 and their distribution is from 1 to 31. So we can expect that the encoding functions for each pattern should be less than or equal to 5 bits. By experimental analysis, we know that encoding functions with 3, 4 and 5 bits is the best choice. Figure 2 illustrates method for separation of patterns into 3-5 bits encoded groups. We separate the pattern set into three clusters C1, C2, and C3 that have upper bounds $M1 = 7, M2 = 15, M3 = 31$, i.e. C1 includes patterns that have $D \leq M1$, C2 includes patterns that have $M1 < D \leq M2$, and C3 includes patterns that have $M2 < D \leq M3$. Let n_i be the number of patterns content i distinct characters and N_{C1}, N_{C2}, N_{C3} are the number of patterns in 3 clusters, respectively. With totally 2,378 unique patterns, we have the number of patterns in each cluster is

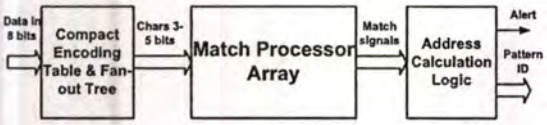


Figure 3. Overview of Deep Packet Filtering in NIDS system

$$N_{C1} = \sum_{i=1}^{7} n_i = 901$$

$$N_{C2} = \sum_{i=8}^{15} n_i = 1113$$

$$N_{C3} = \sum_{i=16}^{31} n_i = 364$$

Next, we have to separate patterns in these clusters into small groups. Let σ be the alphabet of a group, and $|\sigma|$ be the number of characters in σ such that $|\sigma| \leq M1$ in cluster C1, $M1 < |\sigma| \leq M2$ in cluster C2 and $M2 < |\sigma| \leq M3$ in cluster C3. In every cluster, to add into any group, a pattern P will search for any group such that union of P with group does not exceed upper bound M of its cluster. If the outcome satisfies then pattern P adds into group, otherwise it will create a new group by itself. The long patterns in every cluster will be distributed before the shorter one. This method does not guarantee the number of group is the smallest but it is the simple method.

Let $G1, G2, G3$ be the numbers of groups of $C1, C2, C3$, respectively.

$$G_7 = 43,$$
$$G_{15} = 74,$$
$$G_{31} = 32$$

These results are achieved after adjustment some too small groups in smaller clusters which only content few patterns to other groups in bigger cluster that can enclose them. With totally 149 groups, the number of encoded tables is correlative and the average number of patterns in one group is about 16. These outcomes are suitable for hardware design.

2.3 Design Methodology for Deep Packet Filtering in NIDS

We divided the pattern set into small groups with similar number of distinct characters in the same group and now we apply the systolic technology for string matching in every group. Systolic method has an array of Processing Elements (PEs) which can compute parallel. Therefore, the system reduces the area of hardware and still keeps the high throughput. The performance of whole system is improved significantly.

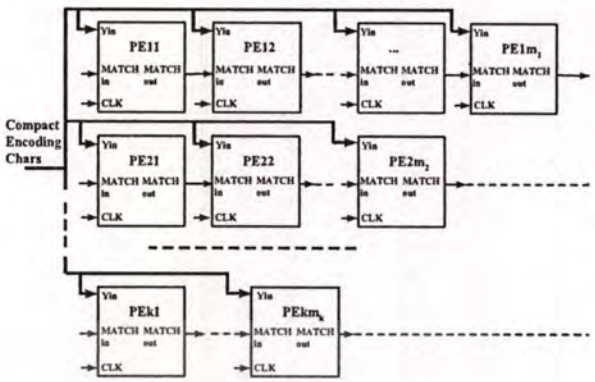


Figure 4. Match Processor Array (MPA)

An architectural overview of our system is shown in Figure.3. The system consists of three parts. Match Processor Array (MPA) is the main part of system that stores compact encoding pattern set used to compare with incoming packet. Compact Encoding Table & Fan-out Tree is the part that converts incoming characters from 8 bits to 3-5 bits suitable for MPAs. The third part Address Calculation Logic calculates the address of the rules that cause a match.

2.3.1 Match Processor Array

In this part, a novel systolic processor array [12] that uses an array of processing elements to match characters going through is presented. All of the patterns of one group are arranged in one 2-D array of processing elements (PEs) called Match Processor Array as the figure 4. Each PE represents one of the characters in the rule set. In the Figure 4, when one incoming character comes in each group, it is encoded to compact code and then it will be compared with all of PEs of MPA at one clock cycle. The match output signal is active only when both following conditions satisfy, the current incoming character matches with the inside stored character and match input signal is active. Then this match signal is transferred to the next PE in current pattern. When the last PE of current pattern has an active match signal, that mean the substring of current string (packet) coincides with the pattern.

For example, if the input string is AABC and the current pattern includes ABC, the match signal output in last PE of current pattern equals 1 after 4 clock cycles since the first character of input string has come in to the first PE of current pattern as the Figure 5.

Inside each PE, look-up tables (LUTs) are used to store pattern character and to compare with incoming character. As the fundamental element of Xilinx FPGA [13], a logic cell includes a 4-bit LUT and a flip-flop. An LUT can be programmed as ROM, RAM, SRL or any maximum 4-

	A (PE1)	B (PE2)	C (PE3)
CLK1	A(1)	A(0)	A(0)
CLK2	A(1)	A(0)	A(0)
CLK3	B(0)	B(1)	B(0)
CLK4	C(0)	C(0)	C(1)

Figure 5. Example of MPA

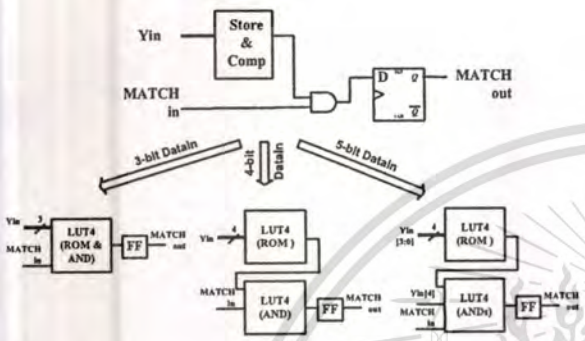


Figure 6. MicroArchitecture of a PE in MPA

input combinational function; and it can be reconfigurable at compile or run time. With this design, a PE includes 1-2 4-input LUTs depend on the number of encoding bits of incoming character as Figure 6. When the encoding character is 3 bits, a PE requires only one LUT4 that is used as a function of ROM 8x1bit and AND gate. However, when the encoding character is 4 bits or 5 bits, a PE requires two LUT4s, one LUT4 is used as a function of ROM 16x1bit and the other is the AND gate or AND gate associated with the storing one more bit. With only 1-2 logic cells of Xilinx FPGA chip, we can save up to 50% area of hardware.

2.3.2 Area and Performance Improvement

The drawback of broadcast data of systolic array in hardware is caused by the large fan-out from outputs of the encoding table of input strings, which are propagated to all PEs of each group. The implementation of our systolic structure is optimized to radically reduce long propagation delays due to fan-out, and achieve a fast speed, by constructing fan-out trees for every group of the MPA as illustrated in Figure 7.b. This is done without incurring extra area resources as we use the flip flops within the logic cells whose LUTs are used for building encoding tables and PEs of MPA.

From experimenting with the smallest to the biggest pattern group, we find the depth of fan-out tree to be 4 with the number of nodes in every level from 1 to 16. The largest fan-out will be 16 that are enough to keep high frequency, and the latency is only four clock cycles more. When we apply this optimization for MPA above, operating frequency is able to achieve approximately the full fabric speed.

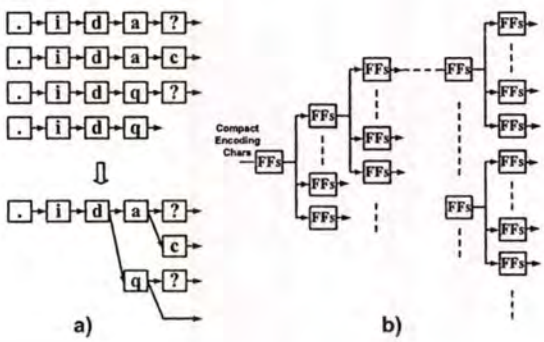


Figure 7. a) Example of Sharing of substrings with 4 strings ".ida?", ".idac",".idq?" and ".idq" b) Fan-out tree for the MPA

One efficient way to save the area for storing the patterns is Aho and Corasicks keyword tree [14]. A keyword tree is used in many software pattern search algorithms, including the Snort IDS. This algorithm is already used in some reconfigurable implementations to reduce the logic area [10].

A keyword tree in Figure 7.a shows how it can optimize the memory utility by reusing the keywords which are prefix of patterns. The conversion not only reduces the amount of required storage, but also narrows the number of potential patterns as the pattern search algorithm traverses the tree. Since the output of the previous Processing Element is forwarded to enable the next stage, no additional logic is required for this area improvement. By applying this optimization over the entire groups, the total logic area can be saved around 35% of its initial size.

3 Experimental Results and Comparison

Our design is coded by Verilog HDL and the design environment Xilinx's ISE 8.1i is used for all parts of the design flow including synthesis, mapping, and place and route. The target chip is a Virtex4 XC4VLX60 (53,248 logic cells). Totally, we can fit 37,873 characters of entire ruleset in the XC4VLX60 FPGA chip. And the maximum frequency reported by Xilinx Timing Analyzer is 483 MHz. The throughput of a design is calculated by multiplying the clock frequency with the data width (8-bit). For a design running at 483 MHz, the throughput is 3.86 Gigabits per second (Gbps).

Table.1 shows the comparison of our work with other recent related works that process one incoming character per clock cycle. To be easier in comparison, we also implement the system on other FPGA chips as Virtex2

Table 1. Comparison of FPGA-based systems for NIDS

System	Dev.-XC (Xilinx)	Freq. (MHz)	No. chars	LCs/ char	Mem (kb)	T-put (Gbps)
Our System	**2V6000 4VLX60**	**414 483**	**37,873 37,873**	**1.12 1.15**	**0 0**	**3.31 3.86**
[8]	2V8000	300	19,715	0.60	0	2.50
[10]	4VLX15	250	32,384	0.26	162	2.08
[7]	2V3000	372	19,854	1.10	0	2.98
[6]	4FX100	200	16,715	0.27	0	1.60

XC2V6000. For comparison purposes, a device-neutral metric called logic cells per character (LCs/char) is used. This metric is determined by dividing the total number of logic cells used in a design by the total number of characters programmed into the design.

As can be seen in Table.1, our system has very high frequency, 414-483 MHz, depend on the kinds of FPGA chips implemented. The reason of this result is the simplicity of our architecture. Moreover, the pattern set is partitioned into smaller groups that help our system to avoid the high fan-out as a text string is compared with thousands of patterns at the same time. From 3.31 to 3.86 Gbps, our throughput is the best among systems which compare one character per clock cycles. The area of [10] as using the ROM is the smallest since the authors use block RAM of FPGA as ROM to store some parts of pattern rule set and the LCs/char metric does not take account the ROM capacity.

4 Conclusion and Future Works

A new SRAM-base FPGA implementation of Network Intrusion Detection is proposed. We have used compact encoding method and systolic processor technique to design. According to our implementation result, the achievable throughput can be up to 3.86 Gbits/s. This is sufficient to handle intrusion detection on current gigabit networks.

In current works, we are performing to save more area by sharing not only prefix of patterns but also all positions of substring that can be happen in patterns. We are also implementing the system on the Avnet board ADS-XLX-V4-LX-EVL60 with Virtex-4 FPGA chip to verify the actual performance of our system.

Acknowledgment

We would like to acknowledge AUN-SeedNet Program of JICA for the scholarship and Xilinx, Inc. for donating the software tools.

References

[1] http://www.snort.org, SNORT: The Open Source Network Intrusion Detection System.

[2] B. L. Hutchings, R. Franklin and D. Carver, Assisting Network Intrusion Detection with Reconfigurable Hardware, *Proceedings of the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 2002, 111-120.

[3] J. Moscola, J. Lockwood, R. P. Loui and M. Pachos, Implementation of a Content-Scanning Module for an Internet Firewall, *Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 2003, 31-38.

[4] C. R. Clark and D. E. Schimmel, Scalable Pattern Matching for High Speed Networks, *Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 2004, 249-257.

[5] L. Tan and T. Sherwood, A High Throughput String Matching Architecture for Intrusion Detection and Prevention, *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, 2005, 112-122.

[6] H.J. Jung,Z. K. Baker and V. K. Prasanna, Performance of FPGA Implementation of Bit-split Architecture for Intrusion Detection Systems, *Proceedings of the Reconfigurable Architectures Workshop at IPDPS*, 2006.

[7] I. Sourdis and D. Pnevmatikatos, Pre-Decoded CAMs for Efficient and High-Speed NIDS Pattern Matching, *Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 2004, 258-267.

[8] S. Yusuf and Wayne Luk, Bitwise Optimised CAM for Network Intrusion Detection Systems, *Proceedings of 15th International Conference on Field-Programmable Logic and Applications*, 2005, 444-449.

[9] Y. H. Cho,S. Navab and W. H. Mangione-Smith, Specialized Hardware for Deep Network Packet Filtering, *Proceedings of 12th International Conference on Field-Programmable Logic and Applications*, 2002, 452-461.

[10] Y. H. Cho and W. H. Mangione-Smith, Fast Reconfiguring Deep Packet Filter for 1+ Gigabit Network, *Proceedings of the 13th IEEE Symposium on Field-Programmable Custom Computing Machines*, 2005, 215-224.

[11] S. Kim and Y. Kim, A Fast Multiple String-Pattern Matching Algorithm, *Proceedings of 17th AoM/IAoM Conference on Computer Science*, 1999.

[12] S.Y. Kung, *VLSI Array Processors*, (Englewood Cliffs, New Jersey:Prentice Hall,1988).

[13] http://www.xilinx.com, Xilinx Inc.

[14] A. V. Aho and M. J. Corasick, Efficient string matching: an aid to bibliographic search, *Communications of the ACM*, 18(6), 1975, 333-340.