國 立 交 通 大 學

電信工程學系

碩 士 論 文

應用於場效可程式邏輯陣列之高效能入侵偵測系統

High Performance FPGA-Based Intrusion Detection System

研究生：葉易霖

指導教授：李程輝

中 華 民 國 九十四 年 六 月

應用於場效可程式邏輯陣列之
高效能入侵偵測系統

High Performance FPGA-Based
Intrusion Detection System

研 究 生： 葉易霖　　Student: Yilin Yeh

指導教授： 李程輝 教授　　Advisor: Prof. Tsern-Huei Lee

國 立 交 通 大 學

電 信 工 程 學 系 碩 士 班

碩 士 論 文

A Thesis

Submitted to Institute of Communication Engineering

College of Electrical Engineering and Computer Science

National Chiao Tung University

in Partial Fulfillment of the Requirements

for the Degree of

Master of Science

in

Communication Engineering

June 2005

Hsinchu, Taiwan, Republic of China.

中 華 民 國 九 十 四 年 六 月

應用於場效可程式邏輯陣列之
高效能入侵偵測系統

研究生： 葉易霖　　　　　　　　指導教授： 李程輝 教授

國立交通大學

電信工程學系碩士班

中文摘要

近幾年來，由於科技的進步，網路的傳輸速率也不斷地提升，大家都可以享受到網路的便利性。但是在此同時，網路的安全問題卻也逐漸浮上了檯面，越來越多的病毒、木馬程式等，讓人深怕下一個受害者就是自己，許多的網路安全軟體就因此產生了。但是網路的速度越來越快，單單只靠著軟體的防護，似乎已經不太足夠了。如果能讓軟體中工作量最大的部份－字串比對－由硬體來負責的話，相信一定能夠大幅地提升工作效率。

在許多的研究中[7, 8, 9, 11, 13, 14, 15]，都使用了硬體來加速字串比對的過程。不過他們幾乎都是將規則(rule)建立在硬體上，此舉雖然提昇速度，但是卻有著相當大的缺點，那就是無法建立太多的規則。原因在於硬體的邏輯閘數目是有限，而根據不同的軟體，規則的數目有幾百幾千，甚至有上萬條的規則。

因此本篇論文的目的，在於提出一種不同的硬體實現方式，不將規則存放在硬體，而是存放在記憶體中。這種方法能夠讓硬體處理更多的規則，同時又有著不錯的傳輸速率，以期能夠配合目前網路速度的需求。

High Performance FPGA-Based Intrusion Detection System

Student: Yilin Yeh Advisor: Prof. Tsern-Huei Lee

Institute of Communication Engineering

National Chiao-Tung University

Abstract

Recent years, because of advancement of technology, network speed is increasing continuously. Everyone can enjoy the convenience of network. But in the same time, the security of network becomes a serious problem. More and more virus and Trojans make people dread that they are next victims. So, they begin to use some network security software. Nevertheless, under the increasing network speed, software may be not safe anymore. If we can let hardware do string matching that is heavy work in software, we believe that it must improve efficiency enormously.

In some research [7, 8, 9, 11, 13, 14, 15], they almost build rules on the hardware to increase throughput. But there is a big problem in this way. There are not enough logic gates to build many rules. Especially, there are hundreds, thousands, even more than ten thousands of rules in software.

Therefore, we propose a different hardware implement in this thesis. We do not build rule in hardware but store in memory. This approach not only processes more rules but also has good performance to match up the need of present network.

Acknowledgements

Deeply thank to my advisor, Professor Tsern-Huei Lee, for his continuous encouragement, support, and enthusiastic discussion throughout this research. His valuable suggestions help me to the success of this work.

Also I would like to thank the members of network technologies laboratory in the Institute of Communication Engineering, National Chiao Tung University, for their stimulating discussion and help.

This thesis is dedicated to my parents for their patience, love, encouragement and long expectation. At last, thank all my friends who support and concern me so far.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As network technologies become more advanced, the security issues related to these technologies become more complex. The purpose of a Network Intrusion Detection System (NIDS) is to help protect computer network users from malicious attacks. To protect public and private networks, software-based intrusion detection systems have become the norm, with the goal of protecting against compromise to the network's integrity, the machines that form the network, and the information contained within it. Network Intrusion Detection Systems (NIDS) are receiving considerable attention as a mechanism of last resort for shielding computer system and networks against attackers.

1.1 Background

Network Intrusion Detection Systems (NIDS) attempt to detect such attempts by monitoring incoming traffic for suspicious contents. The typical function of a NIDS is based on a set of signatures (or rules), each describing one known intrusion threat, and use them to identify possible security threats, much like virus detection software, and report offending packets to the administrators for further actions. NIDS scan packet's payload looking for pattern that would indicate security threats. String matching is generally expensive: finding a single pattern in an input string imposes computation which is at least linear to the size of the input string, and NIDS rule-sets often contain hundreds of such strings. Most known NIDS implementations use general-purpose string matching algorithms that are known to perform well. The computational burden of string matching using those algorithms is significant: measurements on SNORT show that 31% of total processing is due to string matching; the percentage goes up to 80% in the case of Web-intensive traffic [1]. So, string matching can be considered as one of the most computationally intensive parts of a NIDS and we focus on payload matching. However, intrusion detection systems running in general purpose processors can only serve up to a few hundred Mbps throughput. Therefore, seeking for hardware-based solutions is possibly the only way to increase performance for speeds higher than a few hundred Mbps.

1.2 Motivation

Most Network Intrusion Detection Systems use a general purpose processor running rule-based packet filtering software. However, with widespread availability of 1 Gbps Local Area Networks and 10 Gbps links, it has become clear that current software-based intrusion detection systems cannot process packets at those line rates, resulting in inadequate monitoring of network traffic and increasing the probability of an undetected attack.

Due to exhaustive string matching algorithm, software system running on single general purpose processor may not be able to inspect all network traffic [2][3]. ASIC commercial products can support high throughput, but constitute a relatively expensive solution. So, we choose FPGA as our design platform.

FPGA-based systems provide higher flexibility and comparable to ASICs performance. Because network threats are constantly changing, it is very important to be able to change the rule set. FPGA-based platforms can exploit the fact that the NIDS rules change relatively frequently, and use reconfiguration to reduce implementation cost. FPGAs are flexible, reconfigurable, provide hardware speed, and therefore, are suitable for implementing such systems. But large designs are complex and therefore hard to operate at high frequency. Additionally, matching a large number of patterns has high area cost, so sharing logic is critical, since it could save a significant amount of resources, and make designs smaller and faster. These are some problems that we must be care and try to solve.

1.3 Organization of the Thesis

The rest of this thesis is organized as follows. **Chapter 2** introduces the string matching algorithms. In **Chapter 3**, we will brief some software-based Network Intrusion Detection Systems, like Snort, ClamAV, etc. In **Chapter 4**, we will review hardware-based Network Intrusion Detection Systems (NIDS), previous FPGA-based pattern matching architectures. Then, we will describe our proposed approach in **Chapter 5**. It also includes results and comparison of the performance between previous approach and ours. Finally, there is the conclusion of this thesis in **Chapter 6**.

Chapter 2

String Matching Algorithm

This chapter includes the briefs of some famous string matching algorithms. We will introduce how to build keyword sets and how to work according algorithms. Finally, we are going to compare these String Matching Algorithms to know each one's superiority and shortcoming.

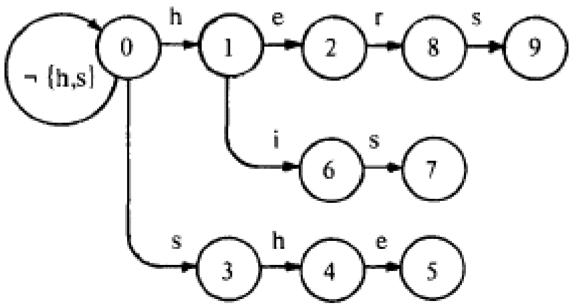
2.1 Introduction

A number of algorithms have been proposed for pattern matching in a Network Intrusion Detection Systems. The performance of each algorithm may vary according to the case in which it is applied. Some are fast for a few rules, but does not perform well when used for a large set. And some behave well on large sets, but their performances start to degrade when short patterns appear in rules. Besides, in worst case a few string matching algorithms perform well, but others are good for average case.

2.2 The Aho-Corasick Algorithm

One of the earliest algorithms in precise multi-pattern string matching was proposed by Aho and Corasick [4], which is able to match patterns in worst-case time linear in the size of the input string. Aho-Corasick works by constructing a state machine from the patterns to be matched. The behavior of the pattern matching state machine is dictated by three functions: a goto function, a failure function, and an output function.

Figure 2.1 shows the functions used by a pattern matching state machine for the set of keywords {he, she, his, hers}.



(a) Goto function.

i	1	2	3	4	5	6	7	8	9
$f(i)$	0	0	0	1	2	0	3	0	3

(b) Failure function.

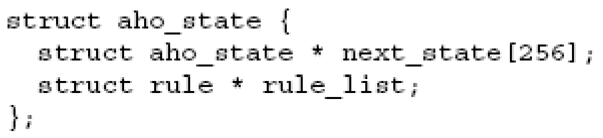
i	*output* (i)
2	{he}
5	{she, he}
7	{his}
9	{hers}

(c) Output function.

Figure 2.1 Pattern matching state machine

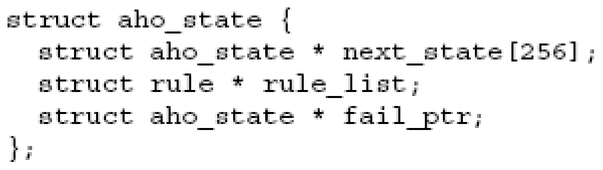
The goto function maps a pair consisting of a state and an input symbol into a state or the message *fail.* The directed graph in Figure 2.1.(a) represents the goto function. The state machine starts with an empty root node which is the default non-matching state, i.e. state 0 in Figure 2.1.(a).

The failure function maps a state into a state. The failure function is consulted whenever the goto function reports *fail*. If *fail* happens, the failure pointers are added from each state to the longest prefix of that state which also leads to a valid state in the trie. In Figure 2.2 we can see that it also cans eliminate all failure transitions by precomputing the next state for every character from every state in the machine. The Figure 2.3 is the original version.



```
struct aho_state {
  struct aho_state * next_state[256];
  struct rule * rule_list;
};
```

Figure 2.2 Base Optimized Aho-Corasick Data



```
struct aho_state {
  struct aho_state * next_state[256];
  struct rule * rule_list;
  struct aho_state * fail_ptr;
};
```

Figure 2.3 Base Un-Optimized Aho-Corasick Data

Certain states are designated as output states which indicate that a set of keywords has been found. The output function formalizes this concept by associating a set of keywords (possibly empty) with every state. If the current state's output function is a non-empty keyword, it will emit the keyword that contains by the current state.

In processing an input of length *n* makes exactly *n* goto transitions. And the total number of failure transitions must be at least one less than the total number of goto transitions. Therefore, the total number of state transitions is less than 2n. As we mentioned before that its performance is linear with the input string length in the worst-case. So, Aho-Corasick is a linear-time algorithm that is optimal in the worst-case.

2.3 The Boyer-Moore algorithm

The most well-known algorithm for matching a single pattern against an input string was proposed by Boyer and Moore [5]. The Boyer-Moore algorithm compares the search pattern with the input string, starting from the rightmost character of the search pattern. This allows the use of two heuristics, Bad Character Heuristics and Good Suffixes Heuristics, that may reduce the number of comparisons needed for pattern matching to increase matching efficiency. Both of them are triggered on a mismatch.

Bad Character: Given a single pattern of length *n* to match, one can look ahead in the input string by *n* characters. If the character at this position (the rightmost of the pattern) is not a character from our pattern, we can immediately move the search pointer ahead by *n* character without examining the other characters in between. If the character of the input string we look-ahead to does appear in the pattern, but is not the last character in the search pattern, we can skip ahead by the as large amount as possible that ensures that we have not missed a case of our pattern.

Good Suffixes: If a mismatch occurs in the middle of the pattern, there is a non-empty suffix that matches parts of pattern. Then we shift the pattern by the least amount that guarantees not to skip any occurrence of the non-empty suffix already matched.

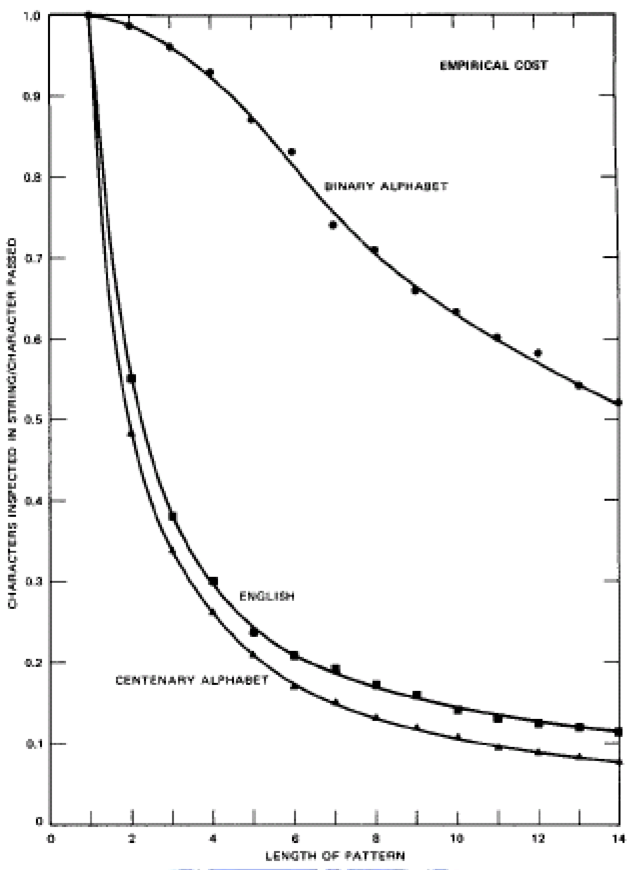


Figure 2.4 Different Data type against Boyer-Moore

Figure 2.4 shows that the average number of references to string/character in string passed is against the pattern length for each of three source strings. From Figure 2.4, we can know that the pattern is more like English text, it cans pass more character. So, it is possible to actually skip a large portion of the text while searching, leading to faster than linear algorithm in the average case. But it is somewhat worse for small alphabets, and its worst case behavior was quadratic.

2.4 The Wu-Manber algorithm

The Wu-Manber algorithm [6] was developed by Wu and Manber for use in agrep and glimpse — two text searching applications. The algorithm is base on a Bad

Character Heuristic similar to Boyer-Moore. The design of the algorithm concentrates on typical searches rather than on worst-case behavior. This allow it to do some engineering decisions that make the algorithm significantly faster than other algorithm in practice.

The algorithm starts by procomputing two tables, a Bad Character shift table constructed by pre-processing all the patterns instead of only one, and a hash table. When the bad character shift fails, the first two characters of the string are indexed into a hash table to find a list of pointers to possible matching patterns. These patterns are compared in order to find any matches and then the input is shifted ahead by one character and the process repeats.

Figure 2.5 shows an excerpt of the modified Wu-Manber data structure, assuming that the strings to be searched for include *cat*, *car*, *bar*, *foo*, and *for*. Note that character such as *x* and *z* which do not appear in any of the strings have the maximum shift values. In comparison, characters in the middle of the strings have reduced shift values, and those that are at the end of the strings, such as *r* and *o* must be resolved by indexing into the hash table.

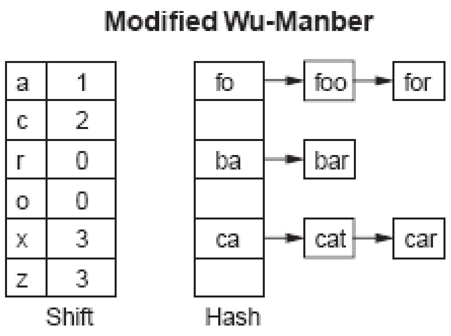
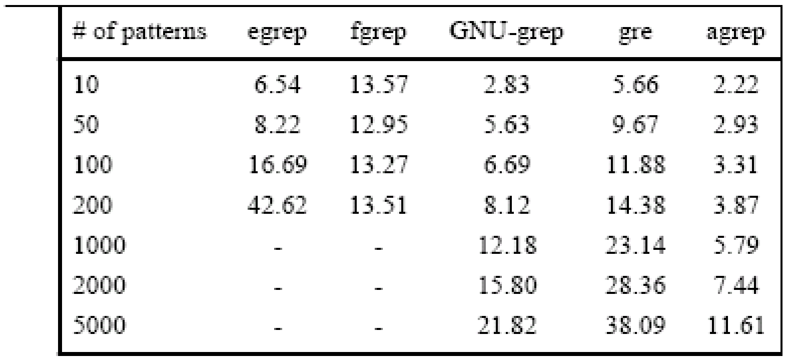


Figure 2.5 Modified Wu-Manber

Figure 2.6 compares the Wu-Manber algorithm, labeled *agrep*, against four other search routines: the original egrep and fgrep, GNU-grep version 2.0, and gre. The patterns were words of sizes ranging from 5 to 15 with average size slightly above 6. The original egrep and fgrep could not handle more than few hundreds patterns.



# of patterns	egrep	fgrep	GNU-grep	gre	agrep
10	6.54	13.57	2.83	5.66	2.22
50	8.22	12.95	5.63	9.67	2.93
100	16.69	13.27	6.69	11.88	3.31
200	42.62	13.51	8.12	14.38	3.87
1000	-	-	12.18	23.14	5.79
2000	-	-	15.80	28.36	7.44
5000	-	-	21.82	38.09	11.61

Figure 2.6 A comparison of different search routines on a 15.8MB text

The performance of Wu-Manber was originally measured using text files and various sets of patterns. Overall the algorithm achieves a worst-case performance that is no better than naïve string matching, but the average case performance is among the best of all multi-pattern string matching algorithms. In the worst case the algorithm requires for every character of input a memory access to the shift and hash table, followed by as many string compares as there are patterns to be matched (this can only happen if the hash fails).

2.5 Summary

In addition to algorithms which introduce above, there are some String Matching Algorithms. For example, K.M.P., which is similar to Boyer-Moore but is starting from leftmost, and AC-BM, which combines Aho-Corasick and Boyer-Moore to adopt their excellence.

These algorithms have used by a little Network Intrusion Detection System software, Snort and ClamAV, etc. Snort adopted Boyer-Moore and modified Wu-Manber as its string matching algorithm. And ClamAV picked Aho-Corasick. So, software seems to choose a suitable algorithm itself. However, some of these algorithms are so complex that they can't work well in hardware, and others cost a lot of hardware resources.

Chapter 3

Software-based Network Intrusion Detection Systems

This chapter includes the brief of some well-known Software-based Network Intrusion Detection Systems. We are going to give an introduction of their architecture and describe their working principles. Then we will show some examples about their rule sets.

3.1 Introduction

With each passing day there is more critical data accessible in some form over the network. Today any publicly accessible system on the Internet will be rapidly subjected to break-in attempts. So, Network Intrusion Detection Systems have become widely recognized as powerful tools for identifying, deterring and deflecting malicious attacks over the network. Essential to almost every Network Intrusion Detection System is the ability to search through packets and identify content that matched known attacks of its rule sets.

Network Intrusion Detection Systems are emerging as one of the most promising ways of providing protection to enterprise network, end users, and so on. The NIDS market has been estimated at $100 million by the Aberdeen Group, with expectations that it will double in 2004 and keep growing in future years.

3.2 Snort

3.2.1 Background

Snort fills an important “ecological niche” in the realm of network security: a cross-platform, lightweight network intrusion detection tool that can be deployed to monitor small TCP/IP networks and detect a wide variety of suspicious network traffic as well as outright attacks. Snort is available under the GNU (General Public License), and is free for use in any environment, making the employment of Snort as a network security system more of a network management and coordination issue than one of affordability.

Lightweight NIDS should be cross-platform, have a small system footprint, and be easily configured by system administrators who need to implement a specific security solution in a short amount of time. Lightweight NIDS are small, powerful, and flexible enough to be used as permanent elements of the network security infrastructure. Snort is well suited to fill these roles. Compare this with many commercial NIDS, Snort con be configured and left running for long periods of time without requiring monitoring or administrative maintenance, and can therefore also be utilized as an integral part of most network security infrastructures.

3.2.2 Architecture

Snort is a libpcap-based packet sniffer and logger that can be used as a lightweight network intrusion detection system. It features rules based logging to perform content pattern matching and detect a variety of attacks and probes, such as buffer overflows, stealth port scans, CGI attacks, SMB probes, and much more. The detection engine is programmed using a simple language that describes per packet tests and actions. Ease of use simplifies and expedites the development of new exploit detection rules.

Snort’s architecture is focused on performance, simplicity, and flexibility. There are three primary subsystems that make up Snort: the packet decoder, the detection engine, and the logging and alerting subsystem.

The Packet Decoder

The decode engine is organized around the layers of the protocol stack present in the supported data-link and TCP/IP protocol definitions. Each subroutine in the decoder imposes order on the packet data by overlaying data structures on the raw network traffic. Snort provides decoding capabilities for Ethernet, SLIP, and raw (PPP) data-link protocols.

The Detection Engine

Snort maintains its detection rules in a two dimensional linked list of what are termed Chain Headers and Chain Options, looks like Figure 2.7. These are lists of rules that have been condensed down to a list of common attributes in the Chain Headers, with the detection modifier options contained in the Chain Options.

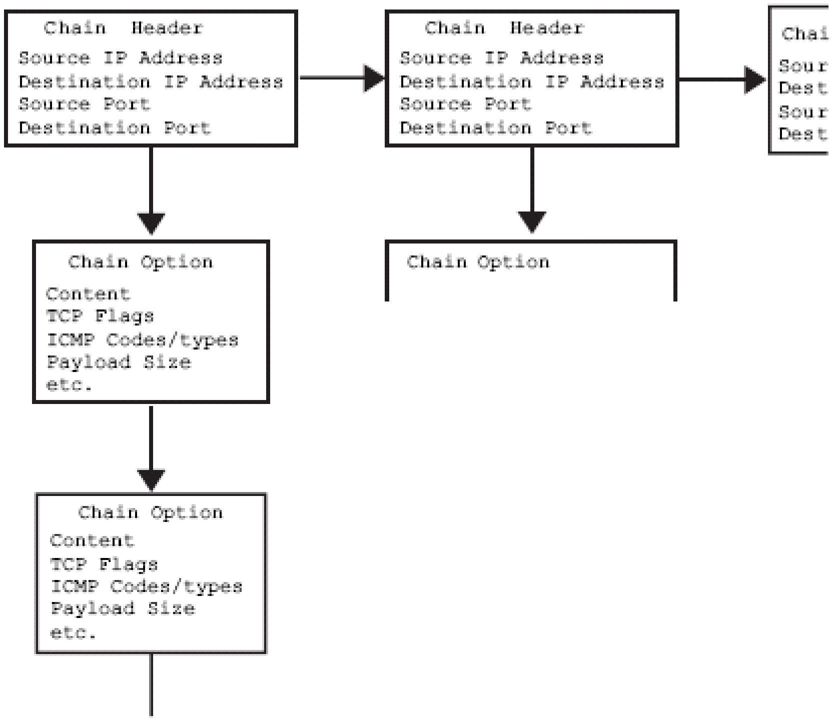


Figure 3.1 Rule Chain logical structure

The Logging/Alerting Subsystem

There are currently three logging and five alerting options.

Log packets:

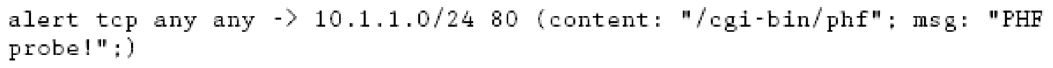
1. In their decoded.
2. In human readable format to an IP-based directory structure.
3. In tcpdump binary format to a single log file.

Alerts:

1. The syslog alerts are sent as security/authorization messages.
2. WinPopup alerts allow event notifications to be sent to a user-specified list.
3. Full and Fast alerting. There are two options for sending the alerts to a plain text file.
4. Completely disable alerting.

3.2.3 Snort Rules

Snort rules are simple to write, yet powerful enough to detect a wide variety of hostile or merely suspicious network traffic. Snort also interprets keywords enclosed in parentheses as “option fields”. Option fields are available for all rule types and may be used to generate complex behaviors from the program, such as in Figure 3.2.



```
alert tcp any any -> 10.1.1.0/24 80 (content: "/cgi-bin/phf"; msg: "PHF
probe!";)
```

Figure 3.2 Option allow increased rule complexity.

The rule in Figure 3.2 would detect attempts to access the PHF service on any of the local network’s web servers. If such a packet is detected on the network, an event notification alert is generated and then the entire packet is logged via the logging mechanism selected at run-time.

3.3 ClamAV

3.3.1 Background

Clam AntiVirus is a GPL (GNU Public License) anti-virus toolkit for UNIX. The main purpose of this software is the integration with mail servers (attachment scanning). The package provides a flexible and scalable multi-threaded daemon, a command line scanner, and a tool for automatic updating via Internet. The programs are based on a shared library distributed with the Clam AntiVirus package, which you can use with your own software. Most importantly, the virus database is kept up to date. It is the most widely used open-source anti-virus scanner available. Currently, it has a digital signature database of 20,712 viruses, worms and trojans. Then it also built-in supports for many kinds of compressed files, mail files, and compressed portable executable files.

3.3.2 Data Structure

ClamAV stores its automaton in a trie data structure. To quickly perform a lookup in this trie, ClamAV uses a 256 element array for each node. It also modifies Aho-Corasick such that the trie has a height of two, and the leaf nodes contain a linked list of possible patterns. ClamAV fixes its trie depth to two because its database contains polymorphic viruses whose prefixes are as shout as two bytes.

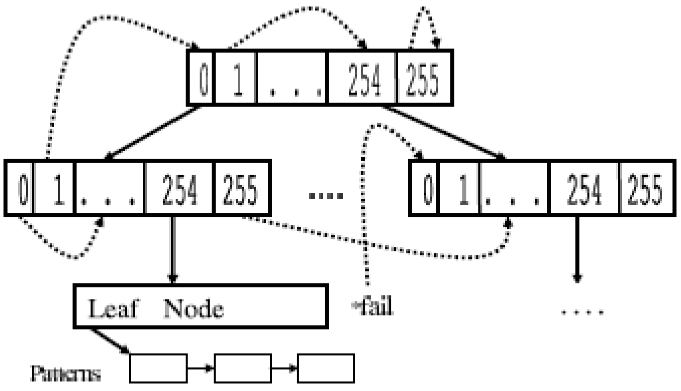


Figure 3.3 Part of the trie structure used by ClamAV. Success transitions are shown with solid lines, and failure transitions are represented with dashed lines.

See the Figure 3.3. As the linked lists get longer, the performance of ClamAV suffers from the cost of traversing the linked list. Unfortunately, each digital signature overtakes two bytes at least. As a result, ClamAV doesn't scale well with large databases.

3.3.3 Improves the Scalability

As a result of the depth of the trie is the key point of the scalability of ClamAV. There was two recent studies which explored methods of breaking this restriction. By implementing work-arounds to grow the trie’s depth to four, ClamAV developers doubled scanning performance. But its memory consumption went from 11MB to 90MB. Besides, researchers at Stony Brook also explored techniques of increasing the trie’s depth and achieved speeds as high as 3.13 times that of ClamAV on certain files [7].

3.4 Compare ClamAV with Snort

Snort uses a string matching engine that performs very well for their rule sets. It uses a modified version of the Wu-Manber algorithm. Other variations of Snort use Boyer-Moore for rule sets with less than 10 signatures and Wu-Manber for others. The approach taken by Snort cannot be applied to virus scanning. First, the number of virus signatures is much larger. Second, while Snort can divide its signature into rule sets due to the nature of the threats they guard against, virus scanning systems cannot do the same. Even in the relatively small category of macro viruses, there are over 7000 signatures. Thus, virus scanning applications have to use algorithms that scale to a large number of signatures.

Chapter 4

Hardware-based Network Intrusion Detection Systems

This chapter will introduce former related work in FPGA. It contains their architectures and functions. Of course, it will include their characteristic in the final section.

4.1 FPGA versus ASICs

Given the processing bandwidth limitations of the General Purpose Processors (GPP), which can serve only a few hundreds Mbps throughput, Hardware-based NIDS is an attractive alternative solution. Generally, ASICs (Application Specific Integrated Circuits) programmable security co-processors are expensive, complicated, and although they can support higher throughput compared to GPP, they do not achieve impressive performance.

On the other hand, FPGAs (Field Programmable Gate Arrays) are more suitable for using on NIDS, because they are reconfigurable. And FPGAs provide hardware speed and exploit parallelism. By vendor's CAD tools, FPGA-based systems can be entirely changed with only the reconfiguration overhead, just keeping the interface constants. This characteristic of reconfigurable devices allows updating or changing the rule sets, adding new features, even changing systems architecture, without any hardware cost.

There are some features of FPGA implementation against ASICs.

Advantages:

a) Fast and cheap procedure for implementing hardware.

b) Fast functional verification.

c) Low cost of low-volume production.

d) Improved time-to-market.

e) Re-configurability in the field.

Disadvantages:

a) Non-optimal utilization of silicon area.

b) Signal delay and power consumption are high.

c) Routing problems could limit flexibility.

d) Potential clock-skew problems.

4.2 Related Work 1 － Content Pattern Match Unit

4.2.1 Brief

In [7], they used the rule sets which is called Hogwash (A search engine based on Snort), about 95 percent of all the known Internet attacks using 105 signature rules of most common attacks, to build their FPGA based NIDS. The rules contain information to search through packet layers 2 through 4 first. If the packet header information matches the rule criteria, an exhaustive pattern search is performed on the payload (layers 5-7). Such a pattern search is done for all the rules with matching header information.

4.2.2 Parallel Design

Figure 4.1 is a block diagram of the architecture. Each rule unit implements the logic for a single Snort rule signature. Packet data is passed to the rule units through a 32-bit bus. Effective use of the hardware pipeline with optimized combinational logic between each stage shortened the critical path of the design.

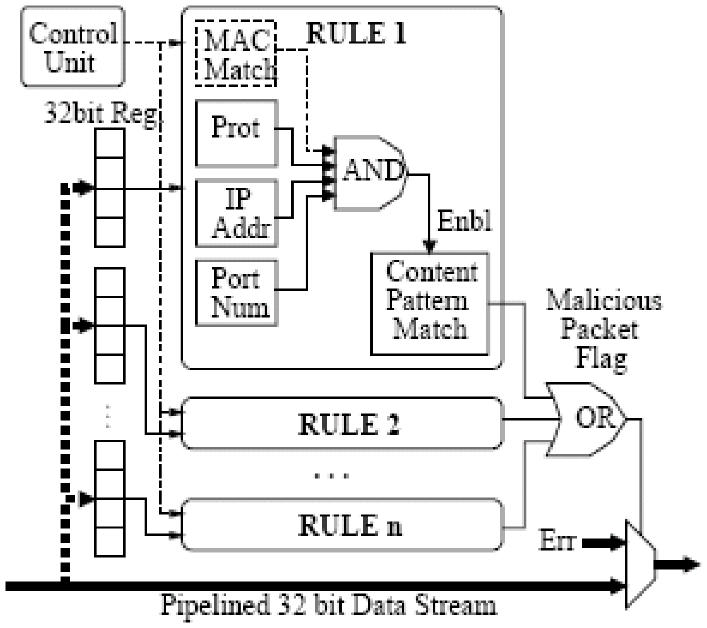


Figure 4.1 Parallel Datapath of NIDS in Reconfigurable Hardware

The simplicity of the design easily shows its resemblance with the software execution path of Hogwash. The header information of each packet is compared with the predefined header data. If the header information matched the rule, the payload is sent to the content pattern match unit where the predefine pattern is searched. However, unlike the software implementation, all the rule chains are matched in parallel to achieve predictable high performance.

If a given packet is detected to be malicious, a flag is raised. In a system with sufficient buffer memory, the flag can be used to drop the packet. However, without any external memory, this flag can be used to corrupt the rest of the packet payload going through the pipeline; thereby causing the packet to be dropped at the receiving node.

4.2.3 Content Pattern Match Unit

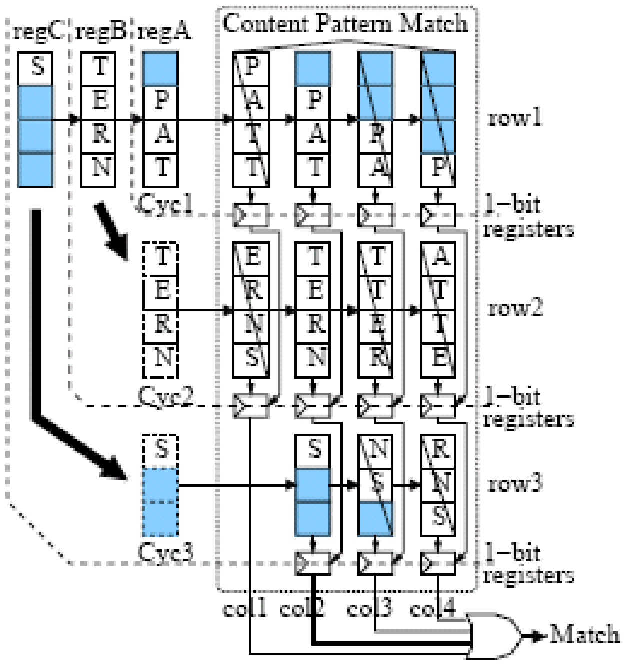


Figure 4.2 Pattern Match Example

The content pattern match unit is the main function of the design. In Figure 4.2, each column of each row has four 8-bit comparators work together to match four consecutive bytes of data simultaneously. In this example with data “pattern”, each row has four comparators groups which contain four different strings with different byte offsets. The match results are passed to the 1-bit registers located below all the comparators. Output from 1-bit register controls the subsequent 1-bit comparison result. When the OR gate receives 1 from one register, the Match flag would be set to 1 to represent that this unit has confirmed a match with the incoming string.

4.3 Related Work — Pipelined Comparator

4.3.1 Brief

This architecture of an FPGA-based NIDS includes blocks that match deader fields rules, and blocks that perform text match against the entire packet payload [8]. In this architecture they focus on making the pattern match module as fast as possible, and assume that header field rules are relatively straightforward to implement at high speed since they involve a comparison of a few numerical fields only.

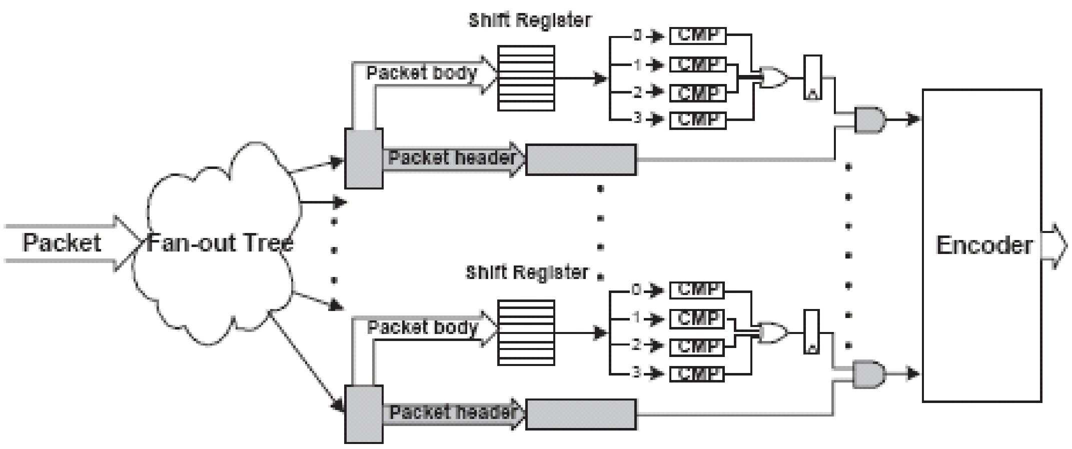


Figure 4.3 Envisioned FPGA NIDS

In Figure 4.3, we can the architecture of the Envisioned FPGA NIDS. Packets arrive and are fan-out to the matching engines. *N* parallel comparators process *N* characters per cycle (four in this case) to achieve higher processing throughput, and the matching results are encoded to determine the action for this packet. Shaded is the header matching logic that involves numerical field matching. The results of the system:

a) An indication that there was indeed a match.

b) The number of the rule that did match.

4.3.2 Pipelined Comparator

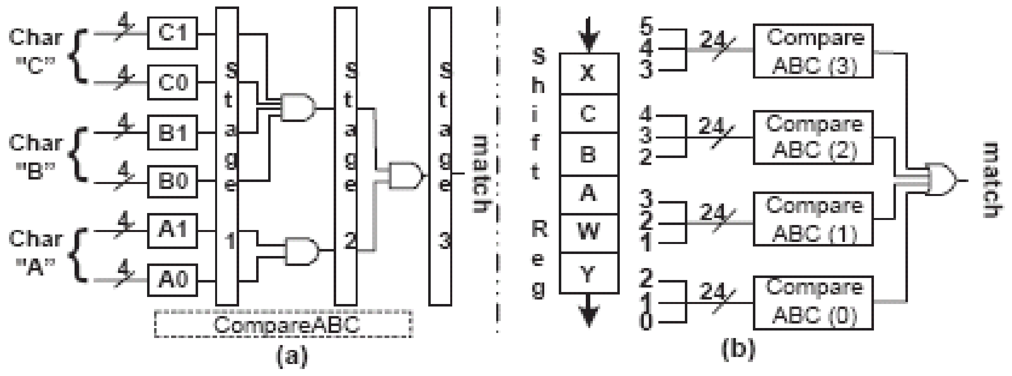


Figure 4.4 (a) Pipelined comparator, which matches pattern "ABC". (b) Pipelined comparator, which matches pattern "ABC" starting at four different offsets.

In Figure 4.4 (a), it shows a pipelined comparator that matches the pattern "ABC". In the first stage, comparator matched the 6 half bytes of the incoming packet data. In the following two stages, the partial matches transfer results (1 or 0) to AND gates to produce the overall match signal. In Figure 4.4 (b), it depicts the connection of four comparators that match the same pattern shifted by zero, one, two and three characters (indicated by the numerical suffix in the comparator label). Comparator comparator_ABC(0) checks byte 0 to 2, comparator_ABC(1) checks bytes 1 to 3 and so on.

4.4 Summary

In addition to approach that mentioned before, there are some researches in reconfigurable hardware about NIDS. Literature [9] shows that the approaches using reconfigurable hardware essentially involve building a finite automaton for a string to be searched, generating a specialized hardware circuit using gates and flip-flops for this finite automaton, and then instantiating multiple such finite automata in the reconfigurable chip to search the incoming data in parallel.

Most proposed FPGA-based NIDS use finite automata (either deterministic or non-deterministic) [9, 10] to perform the text search. These approaches are employed mainly for their low cost, which is reported to be is a few logic elements per search pattern character. But the operation of finite automata is limited to one character per cycle operation. To achieve higher bandwidth researchers have proposed the use of packet-level parallelism, whereby multiple copies of the finite automata work on different packets at lower rate.

However, the common characteristic of these approaches is that the on-chip hardware resource consumption (gates and flip-flops) grows linearly with the number of characters to be searched. Secondly, these methods require the FPGA to be reprogrammed to add or delete individual strings from the database. Any change in the database requires the recompilation, regeneration of the finite automaton, re-synthesis, re-place and route of the circuits, even though it costs less than ASICs.

Chapter 5

The Proposed Approach

In this chapter, we will propose our approach. It contains the architecture and its detail function. And we will explain why we do this.

5.1 Conception

In some researches [7, 8, 11, 12], they implemented a few rule sets of Software-based NIDS (for example, Snort) in the reconfigurable hardware. For a certainty, they increased throughput of their system, even reaching the level of giga-bits per second. However, nowadays most of NIDS software has thousands of rules, and even more than ten thousands of rules in some NIDS software, like ClamAV. But by the restricted number of logic gates in the reconfigurable hardware, they can not put all rule sets on FPGA.

Therefore, if we want to implement a lot of rules of software-based NIDS on reconfigurable hardware and also keep higher throughput, we have to use memories to store large rule sets.

We take ClamAV's digital signatures as our rule sets. Because its signatures are hexadecimal data, in our design we do not need to support a unit that decodes incoming data’s header information.

Gen.12 Tricks-A2=0231944201d1c24e79f7

Above string is one of ClamAV signature. Left of the sign “=” is the rule’s name, and its digital signature is formed hexadecimal in the right. In this example, the signature is composed by ten bytes. But in some signatures they may be composed by more than one hundred bytes. Following section, we will introduce our architecture.

5.2 Our Proposed Architecture

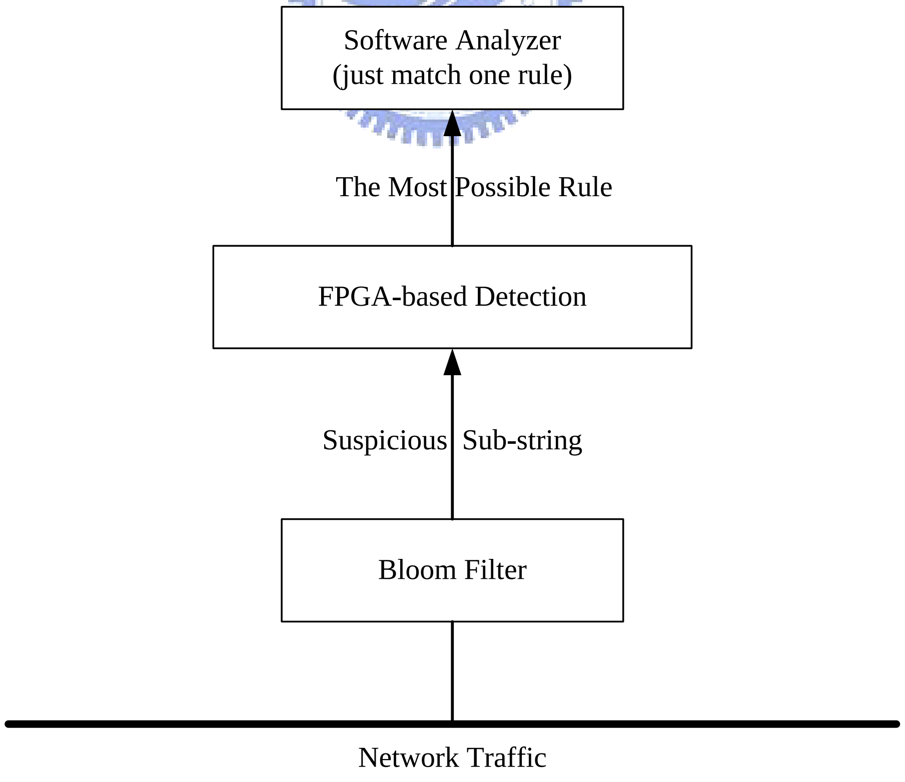


Figure 5.1 Architecture

Figure 5.1 shows our architecture. In the architecture, there are three part of it. We introduce their function briefly below:

Bloom Filter

It collects fixed length incoming data, and then according to rule sets, using hash function to filter suspicious sub-strings. If hash result is noting, then shifts one byte and be continuous.

FPGA-based Detection

Detecting the suspicious sub-string to decide if it is false positive or not. If it is not false positive, points out which rule it may be. Although there are few chances to receive continuous suspicious sub-strings, we still need a small buffer to prevent it.

Software Analyzer

When it is active, we hope that it just need to pick only one rule to scan. In the case of which rule, it will get the information from FPGA-based Detection.

In [13], they just have bloom filter and analyzer. When bloom filter gets something, it will transmit the suspicious sub-string to analyzer. Afterwards, the analyzer search sub-string in its database. Although they may choose some suitable hash function to decrease false positive, it still may happen. From above, we can see that searching in database and false positive are its cost.

So, we add "FPGA-based Detection" between bloom filter and analyzer. The FPGA-based Detection has some functions in this. First, it will check the suspicious sub-string again to confirm that if it is false positive or not. If it is not false positive, in this step, we almost can say that it must be suspicious. Second, according the address of memory it reads, we can view the address as serial numbers of the rule. And then, software analyzer bases on this serial number to find out the corresponding rule to match. In this situation, software analyzer just has to pick the designate rule directly, and do matching. For this reason, the FPGA-based Detection can save a lot of software operating time to increase efficiency.

5.3 FPGA-based Detection

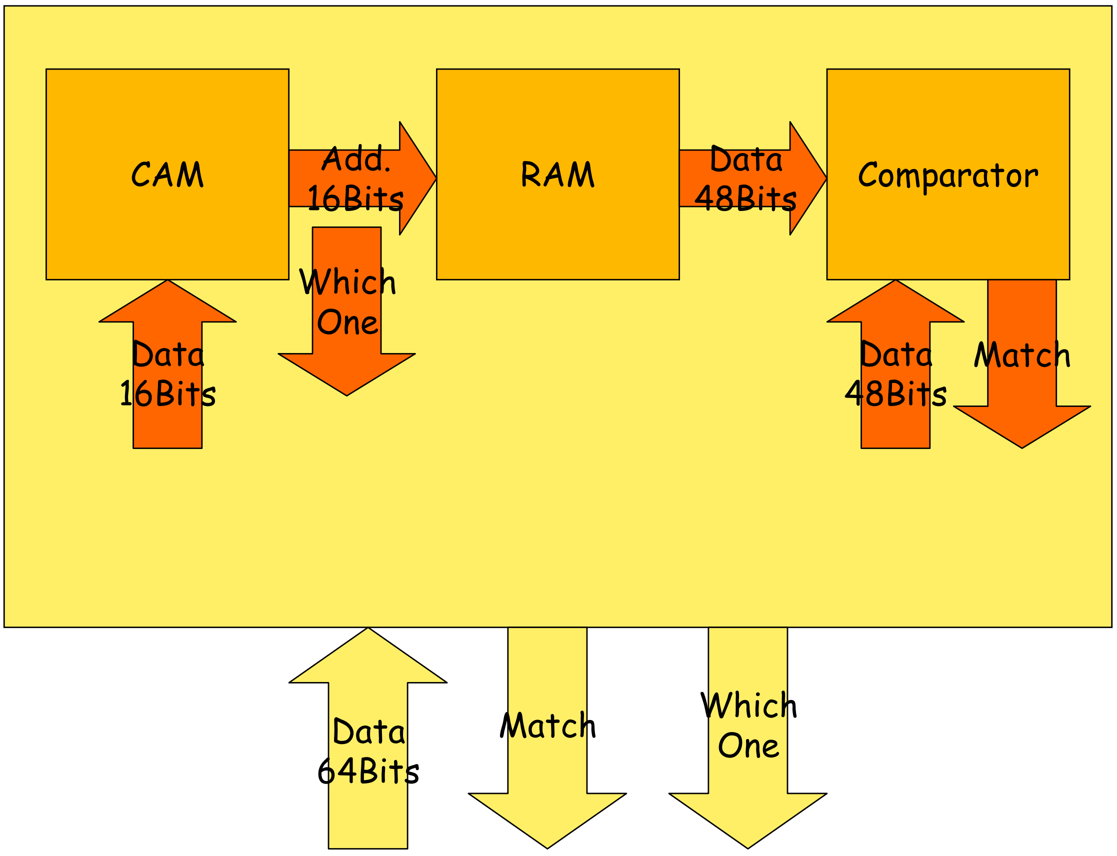


Figure 5.2 FPGA-based Detection

In Figure 5.2, there are CAM, RAM, Comparator in our FPGA-based Detection. In the design, we base on 64Bits (8bytes) data. Because the length of signatures in ClamAV is different, we have to cut them for fixed length to deal easily. Following we will introduce functions of these units.

5.3.1 Function of Each Unit

Content Addressable Memory (CAM)

An item stored in memory can be identified much faster by its content than by its address. Content Addressable Memory, works in this way—the system supplies the data, and CAM returns the address—making it ideal for high-speed search applications. In Figure 5.3 is about CAM's functionality.

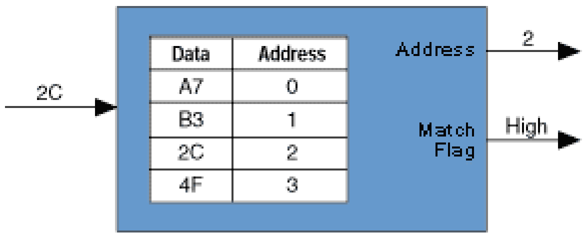


Figure 5.3 CAM, when using CAM, the system supplies the data, and CAM returns the address.

We use CAM's characteristic to get the address of signatures, so we do not search signatures in memory from top to bottom in worst case. But in APEX20K, which the FPGA we use, only provides 4096-words X 12-bits. In order to achieve the result of 16bits-CAM, we use some logic element to design a simple unit to co-operate. So, when we get 64-bits incoming data, pick 16-bits into CAM to get the corresponding address. And the other 48-bits waits for comparator.

Random Access Memory (RAM)

We read the partial signatures from RAM by using addresses that get from CAM. The 16-bits address can point out 65536 partial signatures. However, because we produce addresses by 16-bits of fixed length signatures, it may happen that some signatures have the same 16-bits, i.e. they have the same address.

We see the following Table 5.1 first, it stores four signatures, 01123456, 01234567, 01345678, 01456789. In the Address column, they all represent in binary, and the left 16-bits are the same with CAM's output. And in content column, except that the leftmost is binary, they represent in hexadecimal.

Address	Content
0000000000000001000	0123456
0000000000000001001	0234567
0000000000000001010	0345678
0000000000000001011	1456789

Table 5.1 The content of RAM

To solve the problem that mentioned before, we use some tricks. Each 16-bits address out from CAM, we combine 3-bits with it. So we bottom on this 19-bits address to read data. The first signature's 3-bits is 000 in the address. If another signature has the same 16-bits, its 3-bits is 001. And if the signature is the last one in the same 16-bits address, the leftmost bit of content sets 1, which be called Finish Bit. Previously signatures set 0.

Comparator

It receives 48-bits partial signature from RAM, and compares with other 48-bits of incoming data. If they match, view the address as the serial numbers of the signature. When comparison is failed, and the Finish Bit of current signature is not 1, the address will be plus 1 to read the next signature into Comparator to compare again. However, with the same 16-bits there are few signatures more than eight. In this situation, we will send prefix of these signatures to software to do complete matching.

5.3.2 Flow Chart

There is a flow chart in the following Figure 5.4. Here, we will relate it.

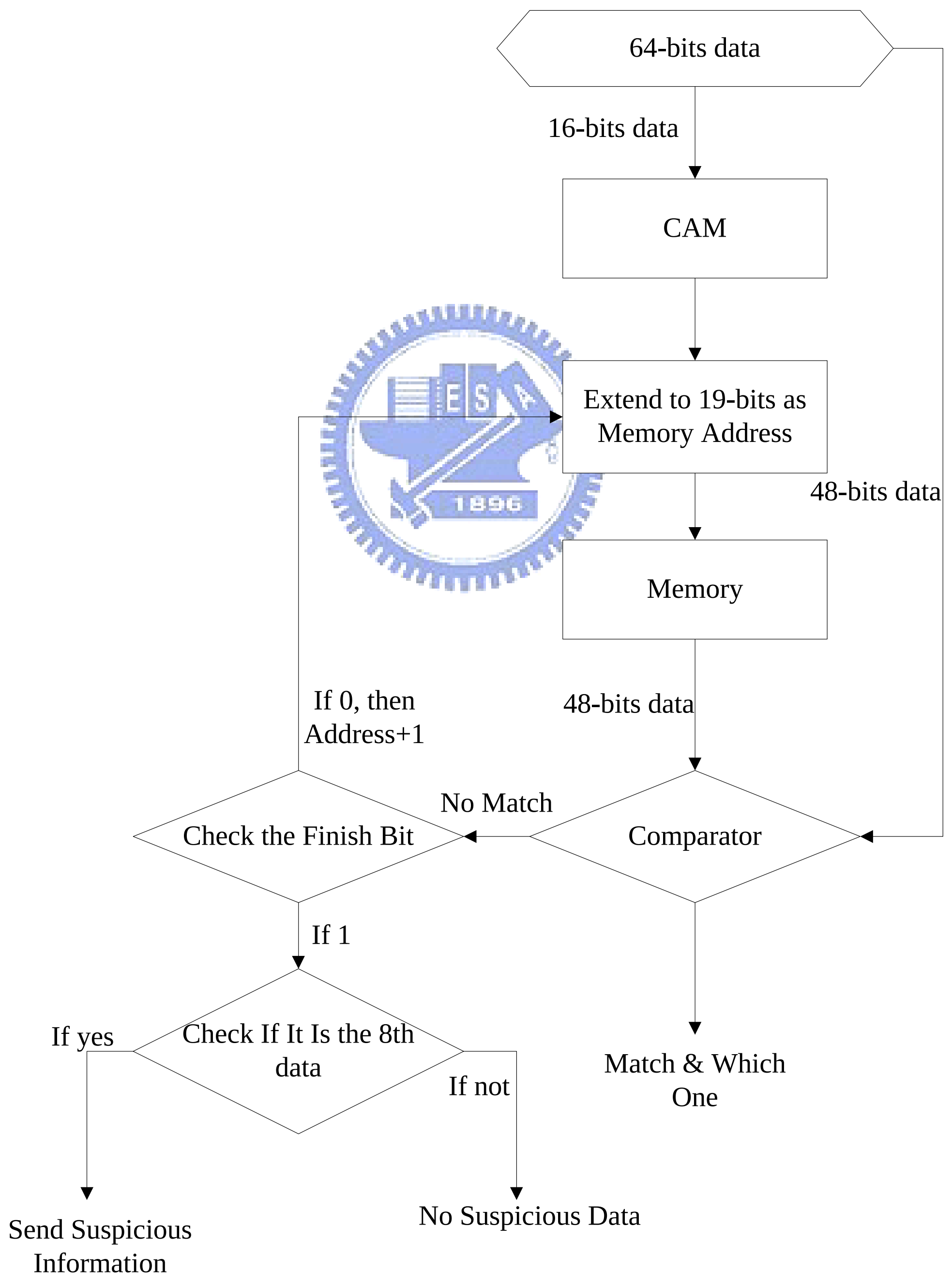


Figure 5.4 Flow Chart

When receiving suspicious 64-bits data from bloom filter, FPGA-based Detection will take 16-bits of incoming data into CAM to find out the corresponding address. Then we extend the address to 19-bits by 3-bits “000”, because there are some signatures that have the same 16-bits possibly.

Using this 19-bits to read the corresponding other 48-bits signature from memory, it will send them to Comparator to compare with the other 48-bits incoming data. If it matches, then sends now address which is as serial number of the signature to software analyzer. If it does not match, and Finish Bit is 0, it means that in the same 16-bits address prefix of memory still has partial signatures, then now address will be plus 1 to read the next data to compare. If the Finish Bit is 1, and it is the eighth content of the same 16-bits address, it will send some information about the prefix of the incoming data to software analyzer. Then software uses this information to scan but not total database. Otherwise, it means that this incoming data is not suspicious.

Chapter 6

Comparison and Conclusion

In this chapter, we will show the results and compare our performance with others. And there is a conclusion of this thesis in final.

6.1 Results

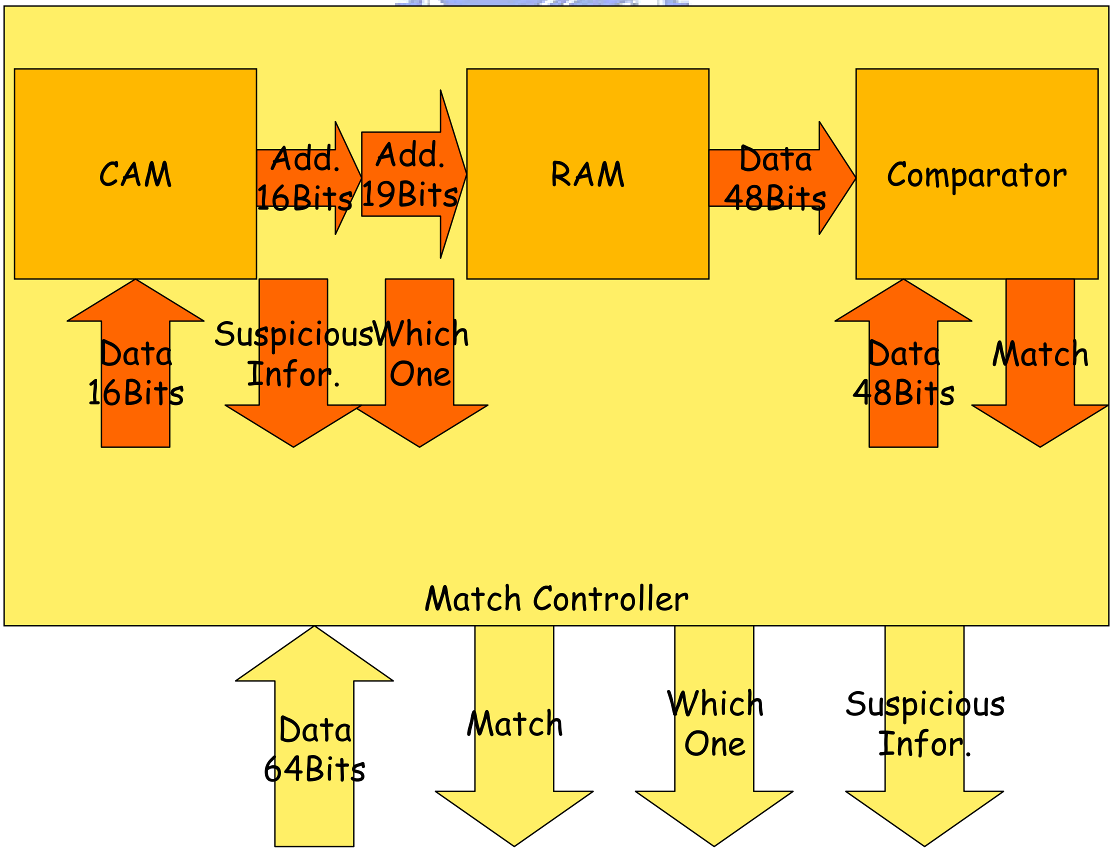


Figure 6.1 The Corrected Architecture

Based on some correction, there is the architecture in above figure. Compare to Figure 5.2, we add "extend to 19-bits address" and "Suspicious Infor." in it.

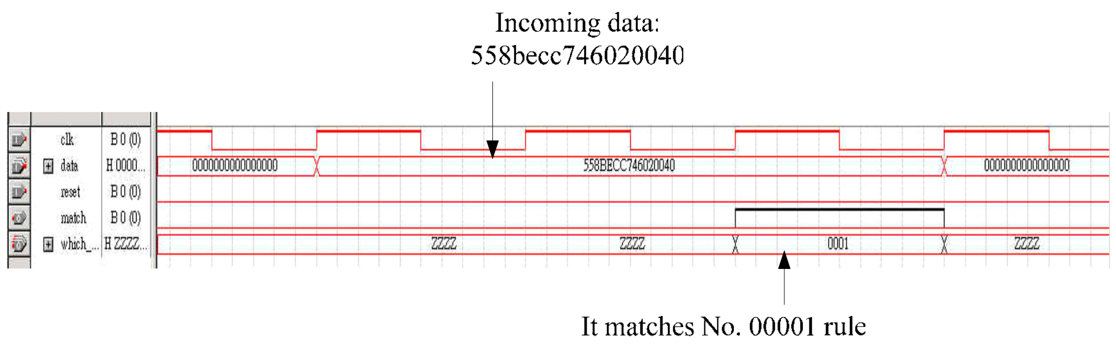


Figure 6.2 To match a non-unique 16-bits address incoming data

There are two waves in Figure 6.1. First picture shows that the system receives an incoming data "558becc746020040". Because it does not have the unique 16-bits address, it must wait for some time to get the result. And then it finds out that the incoming data matches the No. 00001 rule.

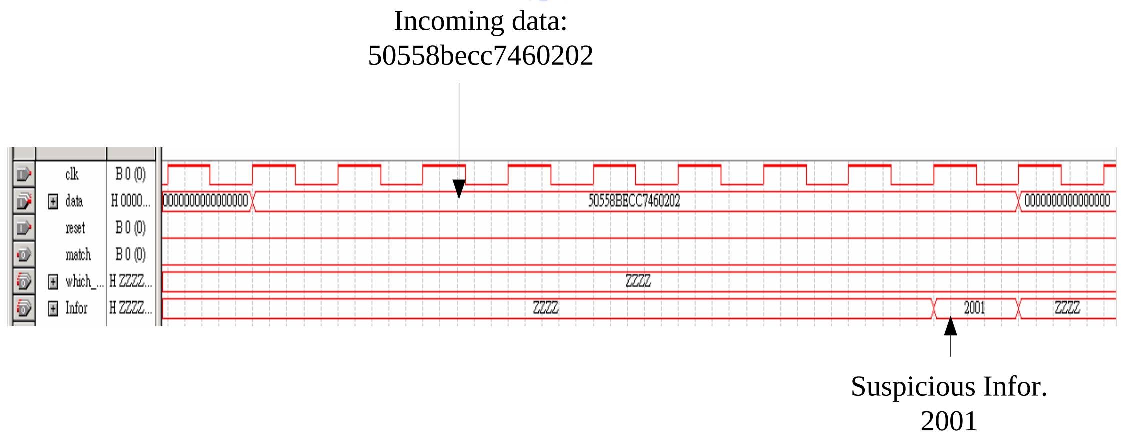


Figure 6.3 The Incoming Data that Need Suspicious Infor.

In Figure 6.3, the incoming data "50558becc7460202" is belong to "more than eight signatures in the same 16-bits address." So, when it can not find the matching signatures in memory, it will send "Suspicious Infor."

6.2 Comparison

Description	Device	Patterns X Characters	Logic Cells	Utilization	Throughput (Gbps)
Sourdis-Pnevmatikatos Discrete Comparators [8]	Virtex 1000	10 X 10 47 X 10.4	1728 8132	7% 33%	6.176 5.472
Sidhu et al. [9] NFAs/Reg. Expression	Virtex 100	(1 X) 9^4	280	11%	0.748
Lockwood [14] FSM+counter	VirtexE 1000	1 X 11	98	0.4%	3.808
Gokhale et al. [11] Discrete Comparators	VirtexE 1000	32 X 20	9722	39%	2.176
Young Cho et al. [7] Discrete Comparators	Altera EP20K	N/A	N/A	N/A	2.88
Our Approach	Altera EP20K	20000 X 8	711	2%	1.514 (2.432)

Table 6.1 Performance Comparison

In Table 6.1, there are some comparisons between our approach and others'. We can see that our design implements the most characters, in others' they only put fewer characters. It means that we can almost put all signatures in our design. In others', they put their rules on FPGA. Because FPGA has limited resources (Logic Elements or Logic Cells), it can not put more rules on itself. So, we use FPGA to increase indexing speed but not put signatures.

ESB (Embedded System Block) is one of component of Altera APEX 20K. It distributes over the APEX 20K discretely. And the CAMs we use are included in ESB. Because of the routing between APEX 20K ESB, throughput of our design is lower than others'. But it still achieves giga-bits rate.

In Table 6.1, we can see two statistics in our approach. In the parentheses, it means that more than half of signatures can achieve 2.432 giga-bits rate. The other means that the average throughput is 1.514 giga-bits rate in total signature.

6.3 Conclusion

Nowadays, there are many attacks in Network. In order to find out these suspicious traffics, we scan network traffic according to rules. But the numbers of these rules are increasing all the days. Therefore, we must improve the efficiency of NIDS. So, we choose reconfigurable hardware. But when we design the detection system based on FPGA, not only concern about how high throughput of this system can achieve but also take care its scalability. For example, if we can increase throughput to terabits per second, but the system just can scan tens of rule sets. This system still is useless. In this thesis, we achieve these targets. In our design we successful propose an architecture that can include a lot of signatures and has not bad throughput.

Bibliography

[1] M. Fisk and G. Varghese. An analysis of fast string matching applied to contentbased forwarding and intrusion detection. In Techical Report CS2001-0670 (updatedversion), University of California - San Diego, 2002.

[2] Roesch, M.: Snort – lightweight intrusion detection for networks. In: Proceedings of LISA'99: 13th Administration Conference. (1999) Seattle Washington, USA.

[3] T. Kojm. ClamAV. www.clamav.net, 2004.

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

[5] Boyer R. S., and J. S. Moore, "A fast string searching algorithm," Communications of the ACM 20 (October 1977), pp. 762-772.

[6] S. Wu and U. Manber. A fast algorithm for multi-pattern searching. Technical Report TR-94-17, University of Arizona, 1994.

[7] Young H. Cho, S.N., Mangione-Smith, W.: Specialized hardware for deep network packet filtering. In: Proceedings of 12th International Conference on Field Programmable Logic and Applications. (2002) France.

[8] I. Sourdis and D. Pnevmatikatos. Fast, Large-Scale String Match for a 10Gbps FPGA-Based Network Intrusion Detection System. In Proceedings of FPL2003, 2003.

[9] R. Sidhu and V. K. Prasanna, "Fast Regular Expression Matching using FPGAs," in IEEE Sysposium on Field-Programmable Custom Computing Machines (FCCM), (Rohnert Park, CA, USA), Apr. 2001.

[10] Moscola, J., Lockwood, J., Loui, R. P., Pachos, M.: Implementation of a content-scanning module for an internet firewall. In: Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines. (2003) Napa, CA, USA.

[11] Gokhale, M., Dubois, D., Dubois, A., Boorman, M., Poole, S., Hogsett, V.: Granidt: Towards gigabit rate network intrusion detection technology. In: Proceeding of 12th International Conference on Field Programmable Logic and Applications. (2002) France.

[12] Jin Hwan Park, K. M. George: Parallel String Matching Algorithms based on Dataflow. HICSS 1999.

[13] Sarang Dharmapurikar, Praveen Krishnamurthy, Todd S. Sproull, John W. Lockwood: Deep Packet Inspection using Parallel Bloom Filters. IEEE Micro 24(1): 52-61 (2004).

[14] J. W. Lockwood, "An open platform for development of net work processing modules in reprogrammable hardware," in IEC DesignCon'01, (Santa Clara, CA), pp. WB-19, Jan. 2001.

[15] F. Yu, R. H. Katz, and T. V. Laskhman, "Gigabit Rate Packet Pattern Matching with TCAM," UCB technical report, UCB//CSD-04-1341, July 2004.

[16] Nathan Tuck, Timothy Sherwood, Brad Calder, George Varghese: Deterministic Memory-Efficient String Matching Algorithms for Intrusion Detection. INFOCOM 2004.

[17] Kostas G. Anagnostakis, Spyros Antonatos, P. Markatos, Michalis Polychronakis: E^2xB: A Domain-Specific String Matching Algorithm for Intrusion Detection. SEC 2003: 217-228.

[18] Yevgeniy Miretskiy, Abhijith Das, Charles P. Wright, Erez Zadok: Avfs: An On-Access Anti-Virus File System. USENIX Security Symposium 2004: 73-88.