# A NETWORK INTRUSION PREVENTION SYSTEM (NIPS) FOR HIGH-SPEED NETWORKS

A Thesis Submitted in fulfillment
of the requirements for the degree of
Master of Science

by

**Shimrit Tzur-David**

Supervised by

Prof. Danny Dolev

School of Engineering and Computer Science
The Hebrew University of Jerusalem
Jerusalem, Israel

September 2005

# Acknowledgments

First, I would like to deeply thank my advisor Prof. Danny Dolev, for his great support and for believing in me. I thank Danny for his most helpful guidance, his brilliant ideas and for spending many hours on my work.

I am especially thankful to Dr. Tal Anker who's original ideas, advices and enthusiasm about this work, made the whole project possible. His valuable suggestions and elucidations help me to significantly improve this research quality.

Special thanks go to Yaron Weinsberg for his endless support. I'm grateful for his precious time and efforts. I really enjoyed the time that we worked together. I thank Yaron for carefully reading my drafts, designated unclear points and help to improve this thesis with his original ideas and insights.

Thanks also to Elan Pavlov, Ariel Daliot and my brother, Asaf David for proof-reading my drafts and for their helpful comments. I'm grateful to all the members of the DANSS lab for their support and friendliness.

Last, but not least, I would like to thank my husband and partner Danny for his unending support. His endless love and encouragement make it possible for me to carry out this project.

This work has received sponsorship from Marvell Semiconductor. Any opinions, findings, recommendations or conclusions expressed in this thesis are those of the author and do not necessarily reflect the views of the sponsors.

# Abstract

*In the late 1990s, as hacker attacks and network worms began to affect the internet, Intrusion Detection systems were developed to identify and report attacks. Traditional Intrusion Detection technologies detect hostile traffic and send alerts but do nothing to stop the attacks. Network Intrusion Prevention Systems (NIPS) are deployed in-line with the network segment being protected. All data that flows between the protected segment and the rest of the network must pass through the NIPS. As the traffic passes through the NIPS, it is inspected for the presence of an attack. Like viruses, most intruder activities have some sort of signatures. Thus, the pattern matching algorithm resides at the heart of the NIPS. The pattern matching algorithm typically uses a large predefined and complex signatures set and deeply inspects the packets for attacks. When an attack is identified, the NIPS discards or blocks the offending data from passing through the system. There is an alleged trade-off between the accuracy of detection and algorithmic efficiency. Both are paramount in ensuring that legitimate traffic is not delayed or disrupted as it flows through the device. Thus, the pattern matching algorithm must be able to operate at wire speed, while at the same time detect the main bulk of intrusions. With networking speeds doubling every year, it is becoming increasingly difficult for software based solutions to keep up with the line rates. This thesis presents a NIPS design with a novel pattern matching algorithm. The algorithm uses a Ternary Content Addressable Memory (TCAM) and is capable of matching multiple patterns in a single operation. The algorithm achieves line-rate speed of several order of magnitude faster than current works, while attaining similar accuracy of detection. Furthermore, our system is fully compatible with Snort's rules syntax [Sno], which is the de facto standard for intrusion prevention systems.*

# Table of Contents

**Acknowledgments** **1**

**Abstract** **2**

**Thesis Overview** **8**

***Part I - Pattern Matching Algorithm*** **11**

**1 Introduction** **12**

**2 Notations and Definitions** **14**

2.1 General Definitions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14

2.2 Snort Specific Notations . . . . . . . . . . . . . . . . . . . . . . . . . . . 15

2.2.1 Rule's Syntax . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16

2.3 ClamAV Anti Virus . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17

**3 Related Work** **18**

3.1 Software Based Pattern Matching . . . . . . . . . . . . . . . . . . . . . . 18

3.1.1 KMP Algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . 18

3.1.2 BM Algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19

3.1.3 AC Algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20

3.2 Hardware Based Pattern Matching . . . . . . . . . . . . . . . . . . . . . 21

3.2.1 Parallel Bloom Filters . . . . . . . . . . . . . . . . . . . . . . . . 21

3.2.2 Network Processor Pattern Matching . . . . . . . . . . . . . . . . 22
3.2.3 TCAM Pattern Matching . . . . . . . . . . . . . . . . . . . . . . . . 23

**4 Rotating TCAM (RTCAM) Pattern Matching 25**

4.1 TCAM String Search Algorithm . . . . . . . . . . . . . . . . . . . . . . 29
4.1.1 Patterns with Negations . . . . . . . . . . . . . . . . . . . . . . . . 33
4.1.2 Long Patterns Optimization . . . . . . . . . . . . . . . . . . . . . . 34
4.1.3 Cross-Packets Attacks . . . . . . . . . . . . . . . . . . . . . . . . . 34
4.1.4 False Positives . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
4.2 Window Size Considerations . . . . . . . . . . . . . . . . . . . . . . . . 36
4.2.1 Effects of the TCAM Width on the Shift Value . . . . . . . . . . . 38
4.3 Pure TCAM Costs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39
4.4 The Trie Solution . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
4.4.1 Trie Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
4.4.2 The Algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
4.4.3 Comparison to the Pure TCAM Algorithm . . . . . . . . . . . . . 42
4.4.4 Paralleling The Trie . . . . . . . . . . . . . . . . . . . . . . . . . . 43
4.5 The Prefixes Solution . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
4.6 The 2-TCAMs Solution . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
4.7 Possible Attack . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46

**5 Porting RTCAM to SRAM 48**

5.1 Data Structures . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49
5.2 Hashed-Based Pattern Matching Algorithm . . . . . . . . . . . . . . . . 50
5.3 Dealing with Short Patterns . . . . . . . . . . . . . . . . . . . . . . . . 50

***Part II - Classification Engine Component* 54**

**1 Introduction 55**

**2 Related Work** 57
2.1 Snort . . . 57
2.2 Fortigate Appliance . . . 58
2.3 SecureSoft Absolute IPS NP5G, NP10G . . . 59

**3 The IDS** 60
3.1 Data Structures . . . 61
3.2 FTP Example . . . 64
3.2.1 Data Structures . . . 64
3.2.2 Control Flow . . . 66

***Part III - Simulation*** **68**

**1 Experimental Results** 69
1.1 Results on ClamAV Pattern Set . . . 70
1.1.1 Test Results . . . 70
1.2 Results on Snort Pattern Set . . . 71
1.2.1 Shift Average Results . . . 72
1.2.2 Scanning Time Results . . . 75

**2 Conclusions** 79
2.1 Future Work . . . 80

**Appendices** 81

**A Light Bulb Example** 81

**Bibliography** 83

# List of Figures


4.1 Part 1: Data Structures . . . . . 29
4.2 Part 1: TCAM Size Requirement . . . . . 36
4.3 Part 1: Signatures Lengths . . . . . 37
4.4 Part 1: The Trie . . . . . 41

5.1 Part 2: Constructing The Hash Value . . . . . 52

3.1 Part 2: Enhanced Classification Engine Architecture . . . . . 60
3.2 Part 2: GFC and SFD . . . . . 61
3.3 Part 2: Event Packet Classifier Operation . . . . . 63
3.4 Part 2: FTP Finite State Machine . . . . . 64

1.1 Part 3: TCAM Size Requirement . . . . . 73
1.2 Part 3: TCAM Size vs. Shift Average . . . . . 75
1.3 Part 3: Memory Accesses . . . . . 76
1.4 Part 3: TCAM Accesses . . . . . 76
1.5 Part 3: Effect of Memory Ratio on Scan Rate . . . . . 78

A.1 Sample Light Bulb Finite State Machine . . . . . 81

# List of Tables

4.1 Part 1: 2-TCAM solution - TCAM size requirements . . . . . . . . . . . . 45

3.1 Part 2: FTP EPC . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 65
3.2 Part 2: FTP PDT . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 65
3.3 Part 2: FTP EAT . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66

1.1 Part 3: TCAM width Impact on Shift Average . . . . . . . . . . . . . . . . 74
1.2 Part 3: Sum of TCAM accesses vs. Sum of SRAM accesses . . . . . . . . 77

A.1 Light-Bulb PDT . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 81

# Thesis Overview

A recent security survey [Van] has revealed that the top eight threats experienced by those surveyed were viruses (78 percent of respondents), system penetration (50 percent), denial of service (40 percent), insider abuse (29 percent), spoofing (28 percent), laptop theft (22 percent), data/network sabotage (20 percent) and unauthorized insider access (16 percent). Although viruses were the most significant threats faced by the respondents, 66 percent of the companies said that they perceive system penetration to be the largest threat to their enterprises. The survey also revealed that despite the fact that 86 percent of the companies used firewalls, the companies did not feel secure against penetrations.

A typical firewall will allow or deny incoming packets based on the port that the TCP or UDP request is arriving on. It is designed to deny clearly suspicious traffic but is also designed to allow some traffic through. This behavior has a a major disadvantage, as any packet is allowed through an open port in the firewall.

Many exploits take advantage of weaknesses in the very protocols that *are* allowed through the perimeter firewalls and once the web server has been compromised, this can often be used as a springboard to launch additional attacks on other internal servers. Once a “rootkit” or “back door” has been installed on a server, the hacker has unfettered access to that server at any point in the future.

The inadequacies inherent in current defences has driven the development of a new breed of security products known as *Intrusion Detection Systems (IDS)*. These systems are evolving and *Intrusion Prevention* capabilities are gradually integrated into these products in order to discard or block the offending data from passing through the system. *Intrusion Prevention Systems (IPS)* are proactive defense mechanisms designed to detect malicious

packets within normal network traffic. These systems stop intrusions by blocking the offending traffic automatically before it does any damage rather than simply raising an alert as, or after, the malicious payload has been delivered.

Within the IPS market place, there are two main categories of products: *Host IPS* and *Network IPS*. The host IPS relies on agents installed directly on the system being protected. It binds closely to the operating system kernel and services, monitoring and intercepting system calls to the kernel or APIs in order to prevent attacks as well as log them. The Network IPS (NIPS) combines features of a standard IDS, an IPS and a firewall. NIPS is sometimes known as an *In-line IDS* or *Gateway IDS (GIDS)*. Most NIPS products are basically IDS engines that operate in-line and are thus dependent on protocol analysis or pattern matching to recognize malicious content within individual packets (or across groups of packets). This thesis focuses on network IPS devices.

NIPS systems are usually comprised of two major components: a pattern matching engine and a complementary packet classification engine. The pattern matching engine's input is a received packet and its output is a set of patterns which are a subset of a set of signatures. The signatures are usually provided as byte arrays that identify worms, viruses and protocol specific keywords. The classification engine tracks each connection traversing the packet processor and ensure the packets are valid.

Today's pattern matching algorithms must be able to operate at wire speeds. With networking speeds doubling every year, it is becoming increasingly difficult for software based solutions to keep up with the line rates. This has underscored the need for specialized hardware-based solutions that can operate faster than 2 Gbps.

There is a number of challenges to the implementation of a NIPS. These challenges all stem from the fact that the NIPS device is designed to work in-line, presenting a potential choke point and a single point of failure. If an in-line NIPS device fails, it can seriously impact the performance of the network. Even if the NIPS device does not fail altogether, it still has the potential to act as a bottleneck, increasing latency and reducing throughput as it struggles to keep up with up to a Gigabit or more of network traffic. As an integral element of the network fabric, the NIPS device must perform like a network switch. NIPS

must meet stringent network performance and reliability requirements as a prerequisite to deployment. NIPS that slows down traffic, stops good traffic, or crashes the network, is counter-effective.

This thesis presents the design of a hardware based NIPS device comprised of a pattern matching module and a classification engine module. These modules can be used to build a dynamic and stateful inspection NIPS for high speed networks.

The thesis is comprised of three parts. **The first part** presents the pattern matching component. **The second part** presents the classification engine, which uses the pattern matching component as a ”black box”. **The last part** describes our NIPS simulation and its experimental results.

# Part I
# Pattern Matching Algorithm

# Chapter 1

# Introduction

The pattern matching algorithm is a building block in any intrusion detection system. There is a need to move beyond filtering traffic based just on the information contained in data packet headers to monitor active connections. Well known internet worms like Nimda, Code Red and Slammer contain a string of bytes as signature. The location of a signature in the packet payload is not deterministic, so the algorithm must be able to detect patterns of different lengths starting at arbitrary locations. The pattern matching algorithm allows to deeply dig into traffic flows to spot hidden attacks on targets like Web, e-mail, and DNS servers. The algorithm must be able to operate at wire speeds that are doubling every year.

The string matching algorithm we focus on, functions by analyzing the text using a search window and then systematically shifting the window along the text. This is known as the “sliding window” mechanism. The patterns length may be smaller or wider than the window width. We make use of the *bad character heuristic* in order to reduce the number of comparisons required. A brute force approach requires $O(mn)$ comparisons, where $m$ is the pattern length and $n$ is the text length. In the bad character approach the algorithm needs to preprocess the set of patterns (with a fixed length of $m$) to create a *shift table*. The shift table contains shift-values for each character from the patterns alphabet. The algorithm starts by comparing the rightmost character in the text sliding

window with the $m$'th character of the pattern. If the mismatching character appears in the search window, the search window is shifted so that the mismatching character is aligned with the rightmost position of the mismatching character in the search window. If the mismatching character does not appear in the search window, the search window can be shifted by its width. In the case of a mismatch, the shift table entry of the mismatching character determines how further we can skip the search window. When a match occurs, an exact match algorithm is invoked to efficiently compare the text and the set of patterns.

The string matching algorithm has the following characteristics:

1. The algorithm matches multi-pattern strings of various sizes.
2. The algorithm runtime complexity is independent of pattern length or count.
3. Many of the processing of the algorithm and data structures can be done offline.
4. The algorithm can be easily implemented in hardware (ASICs).
5. The algorithm's worst-case performance still enable processing packets in several Gbps.
6. Decreasing the algorithm's space complexity is a major goal.
7. The algorithm can be easily ported to TCAM and/or to regular memory (SRAM).

We devise two algorithms: One solely based on a TCAM and second using a standard memory.

Chapter 2 of this part presents several common algorithms that are used in various systems. Most of the algorithms are software oriented but some can be implemented in hardware. Chapter 3 describe s the problem and gives some notations. Chapters 4 and 5 present our two algorithms, the TCAM-based algorithm and the Hash-based algorithm. Both algorithms are efficient and operate at wire speed.

# Chapter 2

# Notations and Definitions

This chapter provides the necessary terms and notations, which are commonly used in the field of intrusion detection systems. We also briefly survey the very popular Snort IDS [Sno] and the widely used GPL anti-virus library, ClamAV [Cla].

## 2.1 General Definitions

DEFINITION 2.1.1 *Define a* ***pattern*** *$P$ to be a string of characters from an alphabet $\Sigma$ which needs to be identified within the input text. Define a* ***sub-pattern*** *$P_s$ to be a sub-string of a pattern $P$.*

DEFINITION 2.1.2 *Define a* ***search window*** *to be a sequential part of the input text within which a sub-pattern is looked for.*

DEFINITION 2.1.3 *Define a* ***string-matching algorithm*** *as follows: We assume that the text is an array $T_{[1..n]}$ of length $n$ and that the pattern is an array $P_{[1..m]}$ of length $m$. We further assume that the elements of $P$ and $T$ are characters drawn from a finite alphabet $\Sigma$. For example, we may have $\Sigma = 0, 1$ or $\Sigma = a, b, .., z$. We say that pattern $P$ occurs with shift $s$ in text $T$ (or, equivalently, that pattern $P$ occurs at position $s+1$ in text $T$) if $0 \leq s \leq n-m$ and $T_{[s+1..s+m]} = P_{[1..m]}$ (that is, if $T_{[s+j]} = P_{[j]}$, for $1 \leq j \leq m$). If $P$ occurs*

*with shift s in T, then we call s a valid shift; otherwise, we call s an invalid shift.*

The string-matching problem is the problem of finding valid shifts, which a given pattern $P$ occurs in a given text $T$. The extended problem of finding multiple patterns in a given text is called "multiple pattern matching".

DEFINITION 2.1.4 *Define a* ***multiple pattern string-matching algorithm*** *as follows: The text is an array $T = t_{[1..n]}$ of length n and the set of patterns is $P_j$, where $1 \leq j \leq r$ (the patterns may have different lengths). The algorithm goal is to output the positions of all occurrences of the* patterns *in the text*

Since we use Snort and ClamAV rules set in our simulation, we overview them both and present their specific notation.

## 2.2 Snort Specific Notations

Snort [Sno] is an open source Network Intrusion Detection System (NIDS), which is available free of cost. Snort uses rules stored in text files that can be modified by a text editor. Snort comes with a rich set of pre-defined rules to detect intrusion activity and it is possible to add more rules to the set at will.

Each Snort rule can contain header and content fields. The header part checks the protocol, source and destination IP address and port. The content part scans packets payload for one or more patterns. Rules with more than one pattern are called *correlated rules*. Rules can also contain negation patterns. Negation of patterns stands for no occurrence of the pattern. The matching pattern may be in ASCII, HEX or mixed format. HEX parts are between vertical bar symbols "j". An example of a Snort rule is:

```
alert tcp any any -> 132.65.200.24/32 111 (content:
"idcj|3a3b|j"; msg: "mountd access";)
```

The above rule looks for a TCP packet, with any source IP and port, destination IP: 132.65.200.24 and port 111. To match this rule, packet payload must contain pattern

"idcj|3a3b|j", which is ASCII characters "i", "d" and "c", and also bytes "3a" and "3b" in HEX format.

In contrary to previous works, our solution is Snort compatible. We handle correlated rules and negation patterns.

### 2.2.1 Rule's Syntax

Snort uses a simple, lightweight rules description language that is flexible and quite powerful. There are a number of simple guidelines for writing Snort rules.

Snort rules are divided into two logical sections, the rule header and the rule options. The rule header contains the rule's action, protocol, source and destination IP addresses and netmasks and the source and destination ports information. The rule option section contains alert messages and information on which parts of the packet should be inspected to determine if the rule action should be taken.

Snort defines a set of rule options. Rule options form the heart of Snort's intrusion detection engine. All Snort rule options are separated from each other using the semicolon character. Rule option keywords are separated from their arguments with a colon character.

This section provides some of the more important keyword definitions that are used to specify where in the text the string matching algorithm should operate. They also enable writing correlated rules with position relation among them.

DEFINITION 2.2.1 *Define a* ***rule*** *to be a set of patterns with some correlation among them. A rule is matched only if all its patterns are also matched with the expected correlation among them.*

DEFINITION 2.2.2 *Define* ***offset*** *to be the position in the text where to start searching for a pattern. It specifies how far into a packet the string-matching algorithm should ignore before starting to search for the specified pattern relative to the beginning of the packet.*

For example, an offset of 5 would tell the string-matching algorithm to start looking for the specified pattern after the first 5 bytes of the payload.

DEFINITION 2.2.3 *Define* ***depth*** *to be the position in the text where to stop searching for a pattern. It specifies how far into a packet the string-matching algorithm search for the specified pattern.*

For example, an depth of 5 would tell the string-matching algorithm to only look for the specified pattern within the first 5 bytes of the payload.

DEFINITION 2.2.4 *Define* ***distance*** *to be the number of characters the string matching algorithm should ignore before starting to search for the specified pattern relative to the end of the previous pattern match.*

Distance can be thought of as exactly the same thing as offset, except it is relative to the end of the last pattern match instead of the beginning of the packet.

DEFINITION 2.2.5 *Define* ***within*** *to be the number of characters in which the string matching algorithm should search for the specified pattern relative to the end of the previous pattern match. It specifies the maximum number of bytes between two pattern matches.*

Within can be thought of as exactly the same thing as depth, except it is relative to the end of the last pattern match instead of the beginning of the packet.

## 2.3 ClamAV Anti Virus

ClamAV [Cla] is an anti-virus toolkit for UNIX, initially designed for e-mail scanning on mail gateways. It provides a flexible and scalable multi-threaded daemon, a command line scanner and an advanced tool for automatic database updating via Internet. The package also includes a virus scanner shared library. ClamAV includes a virus database that currently contains nearly 27,000 signatures (version 0.82). Although this number is smaller than those of major commercial virus scanners, which detect anywhere from 65,000 to 120,000 viruses, the number of viruses recognized by ClamAV has been steadily increasing [MDWZ04]. All ClamAV signatures are simple patterns with no keywords or any other options.

# Chapter 3

# Related Work

## 3.1 Software Based Pattern Matching

Since a string matching algorithm is an essential building block for numerous applications it has been extensively studied [CLR90, WZN92]. This section briefly describe some of the best known software based algorithms which are: Knuth-Morris-Pratt [KMP77, MN98], Boyer-Moore [BM77, CCG$^{+}$99], and Aho-Corasick [AC75, CW79].

### 3.1.1 KMP Algorithm

The naive algorithm forgets all information about previously matched symbols after the pattern shifts. Thus, it is possible that it re-compares a text symbol with different pattern symbols again and again. This leads to its worst case complexity of (nm) (n: length of the text, m: length of the pattern). The algorithm of Knuth, Morris and Pratt [KMP77, MN98] makes use of the information gained by previous symbol comparisons. It never re-compares a text symbol that has matched a pattern symbol. As a result, the complexity of the searching phase of the Knuth-Morris-Pratt algorithm is in O(n).

However, a preprocessing of the pattern is necessary in order to analyze its structure. The preprocessing phase has a complexity of O(m). Since $m \leq n$ (text length $\leq$ pattern length), the overall complexity of the Knuth-Morris-Pratt algorithm is in O(n). In order to

understand how this algorithm works, we define the following:
Let $\Sigma$ be an alphabet and $x = x_0 \ldots x_{k-1}$, a string of length $k$ over $\Sigma$.

DEFINITION 3.1.1 *Define* prefix *of x to be a substring u with* $u = x_0 \ldots x_{b-1}$ *where* $b \in 0, \ldots, k$*, i.e. x starts with u.*

DEFINITION 3.1.2 *Define* suffix *of x to be a substring u with* $u = x_{k-b} \ldots x_{k-1}$ *where* $b \in 0, \ldots, k$*, i.e. x ends with u.*

DEFINITION 3.1.3 *A prefix u of x or a suffix u of x is called a* proper prefix or suffix*, respectively, if* $u \neq x$*, i.e. if its length b is less than k.*

DEFINITION 3.1.4 *Define a* border *of x to be a substring r with* $r = x_0 \ldots x_{b-1}$ *and* $r = x_{k-b} \ldots x_{k-1}$ *where* $b \in 0, \ldots, k-1$

A border of x is a substring that is both proper prefix and proper suffix of x. Its length b is the width of the border. The shift distance is determined by the widest border of the matching prefix of $p$. If the length of the matching prefix is $j$ and the length of the widest border is $b$, the shift distance is $j - b$.

### 3.1.2 BM Algorithm

The basic idea of the Boyer-Moore [BM77, CCG+99] algorithm is that more information is gained by matching patterns from the right than from the left. This allows to reduce the number of the needed comparisons.

We denote the pattern to search in the string as *pat* and the patterns length as *patlen*. *pat* is aligned with the string such that the first character of *pat* is aligned with the first character of string. We will call the *patlen*'th character of the string *char*. The algorithm uses three main observations:

1. if *char* is known not to occur in *pat*, then there is no possibility of an occurrence of *pat* starting at string positions *1* to *patlen*.

2. This observation is a generalization of the first one. If the last occurrence of *char* in *pat* is $delta_1$ characters from the right end of *pat*, *pat* can be slide down $delta_1$ positions without checking for matches. If *char* does not occur in *pat*, $delta_1$ is *patlen*. In the case where the last character of *pat* matches *char*, there is a need to determine if the previous character of the string matches the previous character of *pat*. The algorithm continues with this examination until it reaches the first character of pat or until it notices a mismatch. In the later case, *m* is the number of matched characters until the algorithm recognized the mismatch.

3. In the case where the algorithm matches part of *pat* (until some mismatched character), it shifts *pat* by $m + k$ characters where *k* depends on where *char* occurs in *pat*. If the last occurrence of *char* in *pat* is from the right to the mismatched character (within the characters the algorithm already matched), the algorithm does not gain anything from $delta_1$ so $k = 1$. In this case the algorithm shifts pat in 1 + m characters. On the other hand, if the last occurrence of *char* is from the left side of the mismatched character, the algorithm can set *k* to $delta_1(char) - m$, so pat can be shifted in $delta_1(char)$ characters.

Both algorithms build a shift table that contains shift-values for each character from the patterns alphabet. They use the shift table to avoid back tracking and to shift the text when possible. The search time for an *m* bytes pattern in a *n* bytes text is $O(n+m)$. If there are *r* patterns, the search time is $O(r(n+m))$, which grows linearly with *r*.

### 3.1.3 AC Algorithm

The Aho-Corasick [AC75] algorithm matches multiple patterns simultaneously. It preprocesses the patterns and builds a finite-state-machine which can process the text in a single path in $O(n)$ time. The behavior of the pattern matching machine is dictated by three functions: a goto function *g*, a failure function *f* and an output function $output$. *g* maps a pair consisting of a state and an input symbol into a state or a $fail$ message. Each pattern has a track in the pattern matching machine, *g* simply proceeds to the next state

with the given input or fails in case there is no such a transition. $f$ maps a state into a state. $f$ is consulted whenever $g$ reports $fail$. $f(s) = 0$ for all states $s$ of depth 1. To compute $f$ for the states of depth $d$, the algorithm performs the following actions: for each state $r$ with depth $d-1$

- If $g(r,a) = fail$ for all $a$, do nothing
- Otherwise, for each symbol $a$ such that $g(r,a) = s$, do the following:
  - $state = f(r)$
  - Execute $state \Longleftarrow f(state)$ until $g(state,a) \neq fail$.
  - $f(s) = g(state,a)$.

Certain states are designated as output states which indicate that a pattern has been found. Whenever the pattern matching machine encounters one of these states, $s'$, $output$ emits the set $output(s')$.

The problem with this technique is the exponential state explosion (see [FKL04]).

## 3.2 Hardware Based Pattern Matching

Packet inspection application, must be able to operate at wire speeds. With networking speeds doubling every year, it is becoming increasingly difficult for software based solutions to keep up with the line rates. This has underscored the need for specialized hardware-based solutions which are portable and operate at wire speed. This section presents some of the best known hardware based algorithms.

### 3.2.1 Parallel Bloom Filters

The Parallel Bloom Filters algorithm [DKSL03, TKD03] can handle thousands of patterns but it uses a bloom filter for each possible pattern length. A Bloom filter is a data structure that stores a set of signatures compactly by computing multiple hash functions on each

member of the set. Each Bloom filter computes $k$ hash functions of each pattern in its set and produces $k$ hash values ranging from 1 to its corresponding patterns length. It sets the $k$ bits in a $m$ bits vector. It repeats this process for each pattern in its set. Each Bloom filter scans a substring of its corresponding length of the streaming data and detects suspicious signatures. If all the k hash functions give the same values as some of the patterns, the bloom filter declares this pattern as suspicious and the *analyser* determines if the string is indeed a member of the set or a false positive. Multiple engines can be instantiated to monitor the data, thus the byte stream can be advanced by more than one byte at a time. With four parallel engines, the algorithm can push four bytes in a single clock cycle and the throughput is over 2.46Gbps. The fact that each pattern's length requires a separate Bloom filter is a limiting factor, especially when dealing with very long virus definitions that can be thousands of bytes long.

### 3.2.2 Network Processor Pattern Matching

The work of Liu, Huang and Chin [LHCK04] uses a shift based algorithm that uses a network processor enhanced with a memory based hashing engine. It uses a prefix sliding window (PSW) of length $w$, which shifts from the left most byte to the rightmost byte of the text $T$. The shift value is determined as follows: if the $w$ sequential bytes covered by the PSW contains $k$ bytes of pattern $P_j$ such that $PSW_{w-k}...PSW_{w-1} = a_0...a_{k-1}$ where $1 \leq k \leq w$, the algorithm can shift by $w-k$ bytes. If there is no such pattern, the algorithm can shift by $w$ bytes. Their solution focuses on single patterns and compositions of patterns without any correlation among them. Their algorithm is also optimized for patterns of length of 4. The motivation behind their "optimization" was the memory requirement that were a function of the width of the patterns. The integer value of the bytes in the PSW is used as the entry address for looking up the skip distance (the skip table size is $(2^8)^{width}$). At that time, analysis of Snort's rules pattern supported their assumption since the majority of the patterns was indeed of length of four (which is *not* the case anymore). This algorithm can get a shift average of around 2.

### 3.2.3 TCAM Pattern Matching

Most memory devices store and retrieve data by addressing specific memory locations. As a result, this path often becomes the limiting factor for systems that rely on fast memory access. The time required to find an item stored in memory can be reduced considerably if the stored data item can be identified for access by the content of the data itself rather than by its address. Memory that is accessed in this way is called content-addressable memory (CAM) [ACS03]. CAMs can be binary or ternary. A Ternary CAM, TCAM, can store three binary values for every bit: zero, one and ”dont care”. This extra feature enables more advanced algorithms to exploit the memory.

The work of Lakshman, Yu and Katz [FKL04] present a TCAM based pattern matching algorithm. A TCAM key is constructed for every byte in the packet. The width of the key is configurable and equals the TCAM width. Each row in the TCAM presents a pattern that is needed to be matched against. The algorithm repeatedly extracts a key comprised of $w$ consecutive bytes from the packet. Once the TCAM reports a hit (a matched TCAM row has identified) the algorithm reports the matched pattern. This process is performed for every byte in the packet. If the packet length is $n$, the algorithm has $n$ TCAM lookups. Assuming TCAM lookup time is 4 ns, this algorithm yields a matching speed of $8 \times n/4n = 2$ Gbps rate. This algorithm maintains a matching table in order to recover the long patterns. This table stores all the valid combination of prefixes and suffixes patterns. For any combination it stores the prefix index, the suffix index and the distance between the two. If a combination yields a new valid prefix, it is also added to the table as a new prefix. The algorithm also maintains a Partial Hit List (PHL). When it matches a prefix pattern it records it in this list. At every TCAM hit, the algorithm checks if it matches a prefix or suffix. If it matches a prefix, it simply adds it to the PHL but if it matches a suffix, it needs to check if a combination of the matched pattern with one of the prefixes in the PHL yields a valid combination (pattern hit or valid prefix). Since the lookup process requires searching the matching table to check wether the combination is valid, they trade space for speed. The matching table is a three-dimensional array. The total memory consumption for this array is $w \times a \times b$, where $w$ is the TCAM width, $a$ is the

number of prefixes, $b$ is the number of suffixes and they are both equal to $\sum_i(\lceil m_i/w\rceil - 1)$ where $m_i$ is the length of pattern $i$. Needless to say, most of the entries in this array are just empty. Another drawback of their solution is the fact that it relies solely on TCAM memory. The increased updates to signatures suggest that designing a pattern matching engine solely based on TCAM memory can be quite expensive. For example, at the time that the paper was published, populating ClamAV 2.3 viruses definition within a TCAM memory required 240KB[1] (TCAM width of 128 Bytes). In today's ClamAV version[2], it requires 4.8MB, which is very expensive in today's TCAM prices. Note that Snort's signatures are fewer (around 3100, version 2.3.2) and can be realized in a TCAM of about 150KB (for TCAM width of 64 Bytes).

They define a scan ratio as the total scanning time (including memory lookups) vs. the time spent on TCAM lookups only. Their simulation shows that 60% of the packets have a scan ration of 1, meaning that there are no memory hits at all and 80% of the packets have a scan ration of 1.2. The max scan ratio for all packets is less than 2 resulting a scan rate of 1 Gbps.

[1]Version 0.15, 1768 definitions.
[2]Version 0.82, 27,000 definitions.

# Chapter 4

# Rotating TCAM (RTCAM) Pattern Matching

This section presents an innovative TCAM based pattern matching algorithm, called RTCAM. The algorithm is intuitive and easy to understand. In chapter 5 we introduce a port of the RTCAM algorithm to regular memory that uses the same logic. We begin by introducing the data structures which are used by the RTCAM algorithm.

**TCAM sub-patterns table** - this table contains the $r$ patterns divided by $w$ which is the TCAM width. TCAM entries, which contain a part of a split pattern with length less than $w$ bytes, are prepended (padding at the prefix) with the suffix of the previous part in order to reduce the number of false matches (as done in [FKL04]). If the length of a complete pattern is less than $w$, the pattern is appended with *don't care* characters.

Each TCAM entry has a corresponding TCAM Rule Table Entry *(TRTE)* which is defined in a regular memory region called: *TCAM Rules Table*. The bad character heuristic is applied here by creating a corresponding shift value in the TRTE that states the number of bytes we can shift if a match occurs. Entries of the split patterns contain a shifting value of 0 since this indicates a possible full pattern match. Since a pattern may start at an arbitrary point within a given search window, a set

of shifted sub-patterns (and their corresponding shift values) should be created for every possible sub-pattern which is a prefix of the full pattern [1]. If the shift value is equal to $w$ then the mismatching character does not appear in any of the patterns.

A shifted sub-pattern is created by shifting each prefix sub-pattern to the right (losing the rightmost character and adding *don't care* at the left) and increasing the shift value starting from 0.

**Example:** Assume our patterns set is comprised of: { *"abcdefghij", "cgi", "cdef", "filename", "ab"* } and the TCAM width is 4. Patterns of length greater than 4 are split into sub-patterns and patterns of length smaller than 4 will be padded. Patterns with the same prefix are ordered according to their length in a descending order (enabling longest prefix match first scheme).

Split patterns= {*"abcd","efgh","ghji","cgi?","cdef","file","name","ab??"*}.

The shifted sub-patterns are created as follows: The shift value of the pattern *"abcd"* equals to 0 where the shift value of *"?abc"* is 1, the shift value of *"??ab"* is 2 and the shift value of *"???a"* is 3. Note that the last pattern in the split patterns set (*"ab??"*) is treated in the same way even though its length is smaller than the width, thus, it is yielding the following shifts: $[(ab??,0);(?ab?,1);(??ab,2);(???a,3)]$. The last TCAM entry contains all don't care and the shift value is the width=4. The shifted sub-patterns are ordered according to their shift values in an ascending order after the real patterns. The last entry in the TCAM composed of *w don't care* signs.

**TCAM Rules Table Entry (TRTE)** - each table entry corresponds to a TCAM row and contains the additional information needed for the follow up pattern matching algorithm. Each entry contains the following fields:

***shift offset*** - indicates how many bytes we can safely skip. If this value is 0 then a matching sub-pattern was found in the text. If this value is not 0, then the

[1]Note that this is done only for pattern prefixes and NOT for all sub-patterns, thus the TCAM space is not wasted.

fact that we have a match in the TCAM means that we can shift the text by this field value (this will reduce the average search time).

***Inclusion Patterns*** - a list of pointers to patterns list nodes that contains patterns that are matched by inclusion. Since the TCAM entries are descendent ordered by their length, patterns with length smaller than the TCAM width must be tracked when a longer pattern is matched. This list provides this information. In the example above, the TRTE for the sub-pattern prefix *"abcd"* must include the pattern *"ab"*.

***Associative Patterns*** - a list of pointers to patterns nodes that start with the matched sub-pattern (TCAM row content).

**Patterns List** - This list is ordered[2] by the pattern identifier (sub-patterns which belong to the same rule are directly linked). This list is meaningful only when the shift value is 0 and the TCAM row matched a prefix of a pattern. A rule can be comprised of several patterns which are linked together from the root pattern (e.g. a set of correlated patterns in Snort comprising a single Snort rule [Sno]). Rules do not share a similar text pattern since each pattern in the list contains a context information needed for validating the correlation (e.g, packet offset etc.).
Each entry contains the following:

***pattern*** - the pattern's string

***id*** - the pattern's id

***len*** - the pattern's length

***root*** - a boolean value indicating that this pattern is a root in the rule (root means that this pattern is the first in the rule).

***next*** - a pointer to the next pattern in the rule. The next pointer of the last pattern in each rule is equal to null (including the next pointers in the patterns of the simple rules).

[2] The reason is explained later on.

***offset*** - how far into a packet the search algorithm should ignore before starting to search for the specified pattern relative to the beginning of the packet.

***distance*** - *distance* specifies the minimum number of bytes between two consecutive matches. i.e. how far into a packet the search algorithm should ignore before starting to search for the specified pattern relative to the end of the previous pattern match. This keyword is relevant to correlated patterns.

***within*** - *within* specifies the maximum number of bytes between two consecutive matches. i.e. how far into the packet the algorithm should search for the specified pattern relative to the end of the previous pattern match. This keyword is relevant to correlated patterns.

***depth*** - how far into the packet the algorithm should search for the specified pattern.

**Matched Patterns List** - each entry contains the matched patterns and its corresponding end position in the text. In case of a match, if the pattern is not a root and is a related pattern (depends on previous match), the algorithm checks if the previous pattern (by id) is already in the list. In addition, the algorithm should check if the pattern position is compatible with the four keywords (*offset, distance, within and depth*). If the two conditions hold, the algorithm inserts the pattern to the list. Some of the correlated rules have no relation among the patterns, e.g., the patterns can appear in the text at any order. In this kind of rules, there is no root and the patterns ids do not indicate the appearance of the patterns in the text. In order to deal with these rules, it is insufficient to look for the previous pattern (by id) in the table. For that reason we also maintain a rules table.

**Rules Table** - each entry corresponds to a single rule end contains the following:

***patNum*** - the number of patterns in the rule

***patIdxArray*** - an array of size *patNum*, whenever the algorithm hits one of the rule's pattern, it marks the corresponding entry. If all the entries are marked, we found an attack.

The data structures used in the pattern matching process are presented in figure 4.1.

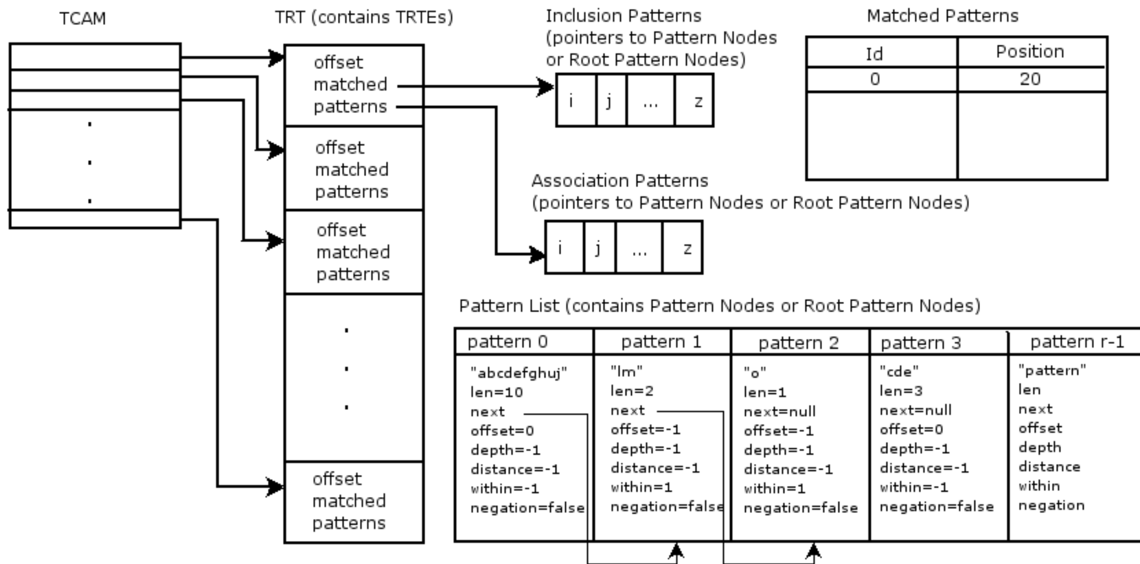



Figure 4.1: Part 1: Data Structures

## 4.1 TCAM String Search Algorithm

The matching procedure is quite simple. Initially the text (pattern) is aligned with the TCAM. A string of *width* bytes is fetched and inserted as a key to the TCAM. The TCAM Rule Table is indexed with the TCAM matched entry and the shift is retrieved. If the shift value N is not equal to 0, the text position is shifted right by N. If it is 0, then we have a possible pattern match and we need to look at patterns pointers. We first follow the TCAM rule entry's *associative list* which points to the patterns that contain the corresponding TCAM value as their prefix. For each pattern in the list, we check the following:

**The *offset* value** - if it is greater than the current position, do nothing.

**The *depth*** value - if it is lower than the $current position + len - 1$, do nothing (the pattern should end within depth bytes).

**The *len* value** - If the *len* value is less or equal to the TCAM width, the algorithm sets the matched flag to true, otherwise we extract the next *w* bytes from the text and use them as a key to the TCAM. In continuance to our example, suppose the text contains: *"abcdefghijklmnop"*. The first match is *"abcd"* which gives a shift value of 0. Since in pattern $(Id = 0)$ $Len > 4$, we extract the next 4 bytes from the text and enter the TCAM again. *"efgh"* is matched again (shift value is 0). We now have a residue of 2, so we'll have to take 2 additional bytes from the text in order to match *ghij* (from text position minus 2 bytes). Checking the TCAM again yields a match with shift value equals to zero which concludes the process and we can set the matched flag as true. Note that getting a shift 0 is not enough. The algorithm needs to check that the entry it gets is the one that corresponded to the sub-pattern of the specific pattern. In order to achieve that, we add an id to each TRTE. At the first match (the first *w* byte of the pattern), the algorithm stores the TRTE id. The next *w* bytes of the patterns has to have a consequent TRTE id. The algorithm checks this id for each sub-pattern match. If all the sub-patters were matched, the algorithm sets the matched flag to true.

If this is a related pattern (has *distance* or *within*) the algorithm checks if a previous pattern id appears in the *matched patterns list*. If yes, it also checks *distance* and *within*.

**The *distance* value** - if the current position is lower or equal to the end position value of the previous pattern in the *matched patterns list* entry plus *distance*, do nothing.

**The *within* value** - if the current position plus *len* minus *one* is greater than the end position value of the previous pattern in the *matched patterns list* entry plus *distance* and *within* values, do nothing.

If the two conditions hold or if the pattern is not related, the algorithm inserts the pattern to the *matched patterns list* and marks the pattern entry in the *patIdxArray*.[3]. If

[3] patIdxArray located in the rule's entry in the *rules table*

all the entries in the *patIdxArray* in the rule entry are marked, we found an attack. The algorithm can now continue with the next list element in the *associative list*.

We also need to check the *inclusion list*, for each pattern in this list the algorithm repeats these operations. After checking all the patterns also in the *inclusion list*, we can move the text position by one. After the packet has been fully consumed, we can empty the *matched patterns list*.

The algorithm is given at 1.

The *checkSubPatterns* method in the algorithm, takes the next *width* bytes from the packet and checks the TCAM for a match with $offset = 0$. It also has to compare the TRTE id with the previous sub-pattern match. The method returns *true* if all the sub-patterns are matched.

**Algorithm 1** TCAM Pattern Matching

```
1: T(Packet) = {T_i, 1 ≤ i ≤ n}
2: pos ⇐ 1
3: shift ⇐ 0
4: width ⇐ default
5: while pos ≤ n − width do
6:    key ⇐ T[pos,..,pos+width−1] {construct a TCAM search key}
7:    shift ⇐ TRT[matchedRow].offset
8:    if shift is 0 then {TCAM exact or partial matched rule}
9:       for all current = TRT[matchedRow].AssociativePatterns.next ≠ null do
10:         if pos < current.offset then {check offset}
11:            continue
12:         end if
13:         if pos + current.len > current.offset + current.depth then {check depth}
14:            continue
15:         end if
16:         if current.len ≤ width then {exact match!}
17:            matched ⇐ true
18:         else
19:            matched ⇐ checkSubPatterns(current) {iteratively checks for sub-patterns}
20:         end if
21:         if matched = true then
22:            if current.distance ≠ −1 OR current.within ≠ −1 then {related pattern}
23:               if MatchedList.contains(current.id - 1) then {previous pattern in matched list}
24:                  prevPos ⇐ MatchedList.get(current.Id − 1).pos
25:                  if pos ≤ prevPos + current.distance then {check distance}
26:                     continue
27:                  end if
28:                  distance ⇐ Max{current.distance, 0}
29:                  if pos + current.len − 1 > prevPos + distance + current.within then {check
                     within}
30:                     continue
31:                  end if
32:               else
33:                  continue
34:               end if
35:            end if
36:            MatchedList.add(current.id, pos + current.len)
37:            Rules[ruleId].markEntry(current.id)
38:            if Rules[ruleId].allMatched() then
39:               FOUND AN ATTACK
40:            end if
41:         end if
42:      end for
43:      for all current = MatchedTable[matchedRow].patterns.next ≠ null do
44:         repeat lines 10 to 41
45:      end for
46:   else
47:      pos ⇐ pos + shift
48:   end if
49: end while
```

### 4.1.1 Patterns with Negations

Some of Snort rules contain negated patterns. Negation of patterns stands for no occurrence of the pattern. Negation patterns always come in correlated rules, for example: $content$ : "$MAILFROM$"; $content$ :!"$|0A|$";. This rule means that if the algorithm has found the string MAIL FROM and has **not** found the hex value of 0A in the same packet, it found an attack. There can be a relation between the patterns, for example, $content$ : "$Username$"; $content$ :!"$Password$"; $within$ : 30. This rule means that if the algorithm has found the string $Username$ and the string *Password* does not appears within 30 bytes, it found an attack.

In order to handle this negated patterns, we maintain *matched negation patterns list* which is a matching list for the negation patterns (we can use the *matched pattern table* with a negation flag at each entry). Each time the algorithm matches a negation pattern, it adds the pattern to *matched negation patterns table*. If the algorithm matches a related pattern it checks if the previous pattern appears in the *matched pattern table*. If the matched pattern is related to a negation pattern, the algorithm looks for the previous pattern at the *matching negation patterns list*. If the previous patterns appear at the negation list, the algorithm has to check the following: if the *within* value of the pattern is $w$ and the *distance* value is $d$, the algorithm checks that the match is not within $w$ bytes from previous match and that the match is not far $d$ bytes from previous match. Note that these conditions are inverted to the conditions the algorithm checks once it matches a non-negation pattern.

In contrast to rules that do not contain negation, which the algorithm can decide before the end of the packet if there was an attack, at this kind of rules the algorithm needs to check the rules entries at the end of the packet to determine if there was an attack. The $patNum$ field in each rule entry in the *rules table* represents the number of the non-negation patterns in the rule. In addition to $patNum$, each entry holds a $negPatNum$ field that represents the number of negation patterns in the rule. At the end of each packet scan, the algorithm checks if the number of non-negation patterns it matches is equal to $patNum$. If so, it also checks the number of the negation patterns it matches, if it is not equal to $negPatNum$, it found an attack.

### 4.1.2 Long Patterns Optimization

We can treat a long pattern whose length exceeds the width of the TCAM, as several patterns. Using the *distance* and *within* keywords, we can split one pattern into two patterns without making any change to the algorithm. The second pattern will have 0 as its distance value and its within value will be equal to its length. For example, we can split the pattern: *abcdefgh* into the two following patterns: *abcd* (all the keywords of the original pattern remains in the first sub-pattern) and *efgh* with distance:0 and within:4. In this way, we enforce the second part of the pattern to appear after the first part (distance:0) and it has to be right after it (within:4). The reason we add this optimization is because it enables us to skip the *checkSubPatterns* sub-routine in our algorithm. Instead of proceeding with checking the packet with the sub-patterns and then return to the original position, we can ignore the rest of the pattern, knowing we will check it when we reach the right position. This approach is better when the patterns are long, as in ClamAV.

At long patterns, we might proceed into the packet in order to check the sub patterns and finally discover that there is no match. For example, suppose that the TCAM width is 32, the algorithm matches a prefix of pattern $P$ and $P$'s length is 100. We also assume that the last byte of $P$ is different than the corresponding byte in the text. After matching $P$'s prefix, the algorithm has to shift all the length of $P$ in order to decide if there is an attack. The algorithm wastes precious time since at the end of the process it has to return to the position where it first matched $P$'s prefix. Although, the probability to match $w$ bytes of the pattern without having a complete match is pretty low and is inversely proportional to the value of $w$, sending a lot of prefixes in a packet can be foiled as if it was an attack.

### 4.1.3 Cross-Packets Attacks

The cross packets attacks problem stands for rules that start at the end of one packet and continue at the next packet of the same flow. This problem can be solved easily for simple patterns. The algorithm maintains a *last-width-match* field for each flow. When the algorithm gets a match for the last *width* bytes of a packet, it saves these bytes at this

field. When it starts scanning a packet, the algorithm checks this field, if it is not null, the algorithm prepends these bytes to the payload of the scanned packet and continues the algorithm exactly as described earlier. This solution assumes that the packets arrive to the algorithm in the correct order; for more information see section 2.1.

### 4.1.4 False Positives

There are two types of the false positives problem. The first type occurs when the TCAM matches a pattern of correlated rule and after some processing the algorithm discovers that there is no attack. The second type occurs when the algorithm matches a patterns corresponded to one protocol, say HTTP, and it scans a packet related to another protocol, say FTP. This problem can slow down the speed of the inspection mechanism significantly. Ideally, the algorithm should not get a match unless there is a real attack.

We can eliminate the second kind of the false positives problem by adding some of the rule's flow data to each TCAM entry. For example, we can add a protocol bitmap that contains a bit for each main protocol. If the pattern appears in both FTP and HTTP rules, both bits are turned on. When the algorithm scans the packet, it adds the protocols bitmap to the search key with the flow protocol bit turned on. In this way, it matches only patterns that appears at rules that are applicable to the flow's protocol (or at rules that don't specify a protocol at all).

We generalize this by having a hash function whose input is some additional data related to the pattern (gathered from the rule or from the flow) and whose output is the addition to the TCAM key. The algorithm uses this function twice: the first time, when creating the TCAM table, the algorithm prepends each entry with the generated key and the second time, when the packet is received the algorithm uses the flow identifier, which can be obtained at wire speed, to create the hash key when constructing the key to enter the TCAM.

## 4.2 Window Size Considerations

The TCAM width is the most influential factor and is configurable. Suppose the TCAM width is $w$; there are $k$ patterns, each one of length $m_i$. Each pattern is cut to $\lceil m_i/w \rceil$ pieces. The total TCAM memory requirement of our algorithm is $w^2 \times \sum \lceil m_i/w \rceil$ bytes. Note that short patterns, or suffixes patterns are padded to the TCAM width so that the TCAM memory size increases with $w$. Figure 4.2 shows the TCAM size required in order to accommodate all the patterns in Snort and in ClamAV signature databases.

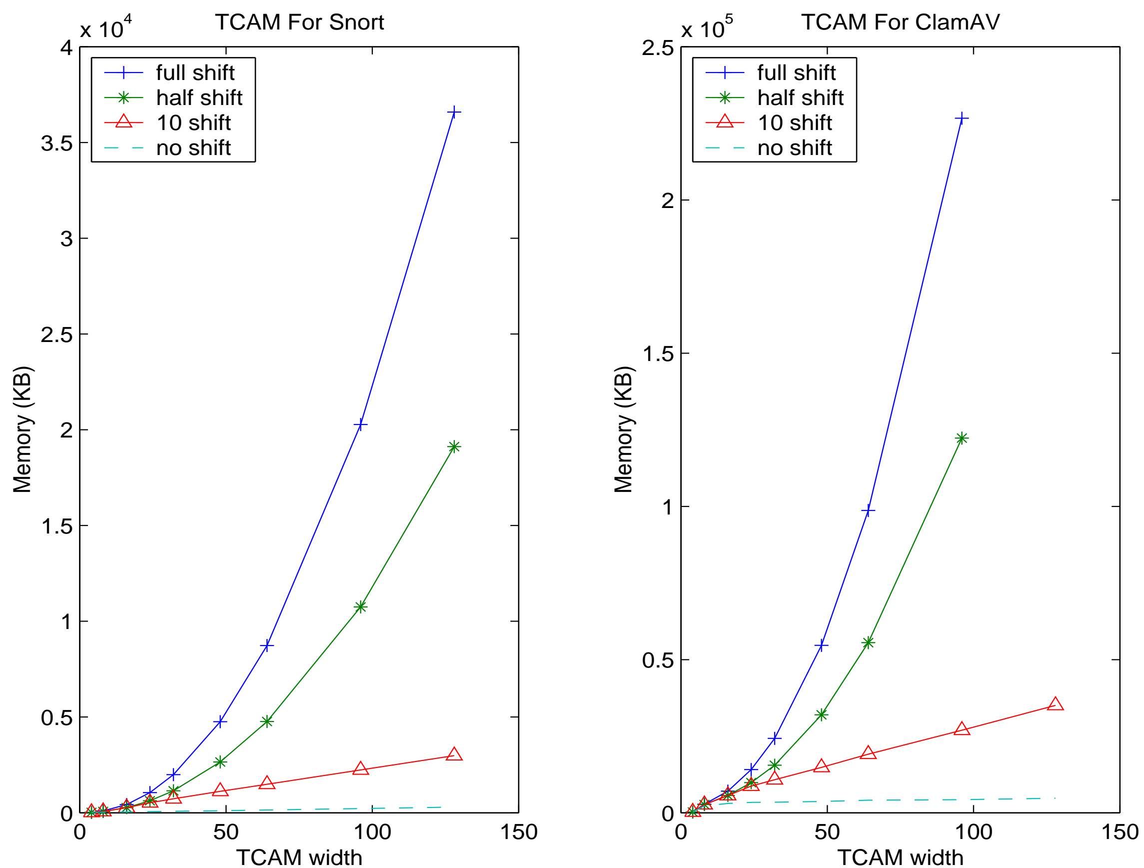



Figure 4.2: Part 1: TCAM Size Requirement

Figure 4.3 shows the amount of signatures (in each database) that can be covered by choosing a specific window size. We can see that for Snort, a TCAM width of 128 bytes covers all patterns. The higher the TCAM width, the less false positives there are. In addition, we minimize the need to check for sub-pattern matches that might complicate and decelerate the algorithm. Another major point is the effect of the TCAM width on the shift values. Since the TCAM contains all the shifted patterns, a wide TCAM results in a high shift value.

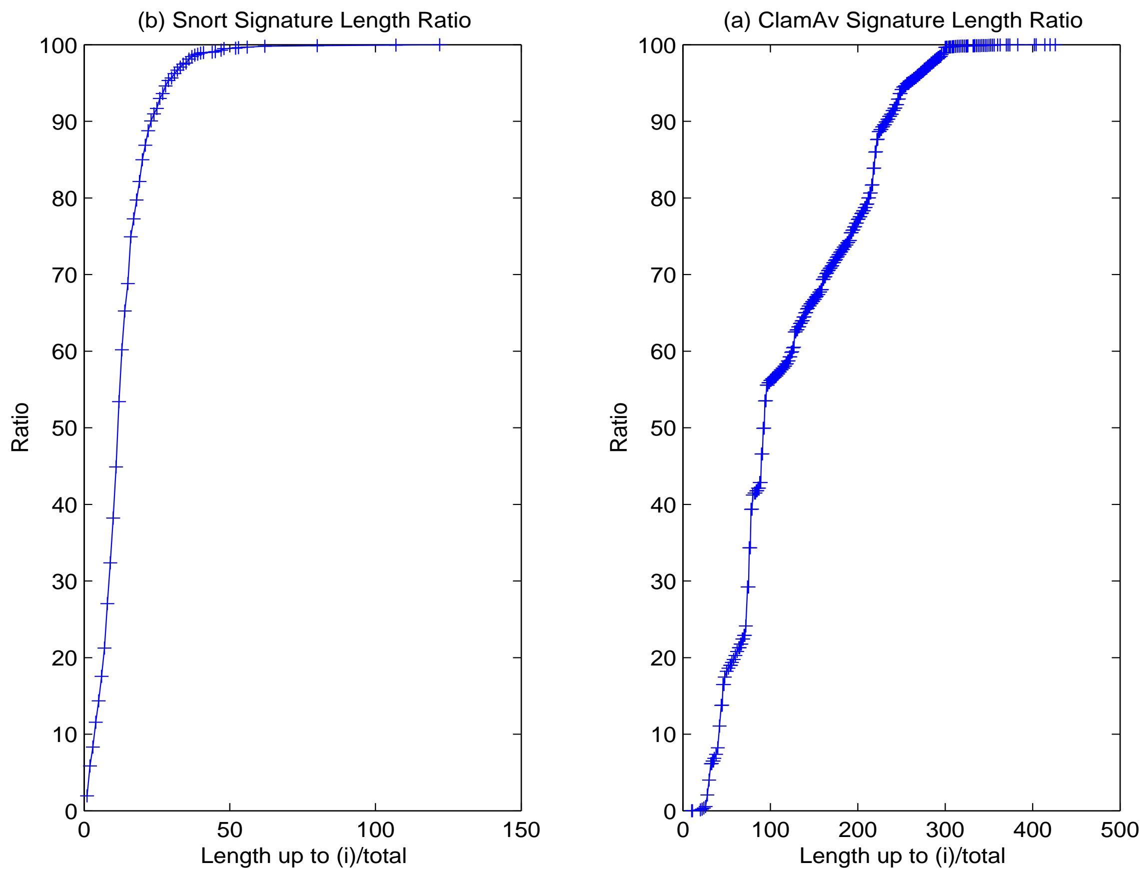



Figure 4.3: Part 1: Signatures Lengths

### 4.2.1 Effects of the TCAM Width on the Shift Value

Assume the packet has a random payload; there are $(2^8)^w$ different possibilities for each $w$ bytes from the payload. We know the shift value of each TCAM entry (the number of *don't care* signs that prepend the pattern). In order to calculate the shift average, we need to calculate the probability to hit each one of the TCAM entries.
We define two relations: antecessor-descendant and intersecting patterns.

**Antecessor-Descendant** - let patterns $P_i$ and $P_j$ be different. If for each character at position $t$, either $P_i[t] = P_j[t]$ or $P_i[t]$ is equal to the *don't care* sign, then we say that $P_j$ is a descendant of $P_i$.

For example the pattern *??cd* is a descendant of the pattern *?bcd*. Note that the shift value of $P_i$ is less or equal to the shift value of $P_j$ (there are never any *don't care* signs in the middle of a pattern).

**Intersecting patterns** - let patterns $P_i$ and $P_j$ be different and do not fulfil the antecessor-descendant relation. If for each character at position $t$, either $P_i[t] = P_j[t]$ or if either $P_i[t]$ or $P_j[t]$ is equal to the *don't care* sign, then we say that $P_i$ and $P_i$ have intersecting patterns.

For example the intersection of the patterns *??cd* and *xy??* is the pattern *xycd*.

Now, for each entry we calculate the number of the potential TCAM hits it can have. If the number of the *don't care* signs in the pattern $P_i$ is $K_i$ (they can appear either at the beginning of $P_i$ or at its end), then the initial number of the potential hits is $(2^8)^{K_i}$. Suppose that the shift value of $P_i$ is $S_i$ ($S_i \in 0, 1, 2, \ldots w$), then we can say that $P_i$ contributes the sum $(2^8)^{K_i} \times S_i$ to the total shift sum. This number is not precise.

Consider the two patterns *??cd* and *?bcd*, *??cd* contributes $(2^8)^2 \times 2$ to the total shift sum and *?bcd* contributes $(2^8)^1 \times 1$ to it. Note that there are $2^8$ patterns that we count twice. This is the reason we calculated the descendants for each pattern. We decrease from the sum $(2^8)^{K_i} \times S_i$ that $P_i$ (the antecessor) contributes to the shift sum, the amount $\sum(2^8)^{K_j} \times S_j$ for each descendant $P_j$ of $P_i$.

This is still not enough. Consider the two patterns $??cd$ and $xy??$. These patterns do not fulfil the antecessor-descendant relationship, but they still count the pattern *xycd* twice. This is the reason we also calculated the intersecting patterns. We also decrease the number of the intersecting patterns from the sum of the pattern that its shift value is greater, $??cd$, in the given example.

Finally, if $S_i$ is the shift value and $S_i^n$ is the number of patterns that yield $S_i$ as a shift value, then the total average shift value is $\frac{\sum_{S_i \in 0..w} S_i^n \times S_i}{(2^8)^w}$.

We calculated the shift average for each TCAM width, the results show that the average shift value is $w/2$. Assuming TCAM lookup time is 4ns, hence the total time to scan an $n$ bytes packet is 4(n/(w/2)) ns. This yields a matching speed of $\frac{8 \times n}{4(n/(w/2))} = w$ Gbps[4]. Thus, we would like to increase the TCAM width but there is a clear tradeoff between window size, required memory, speed and cost.

## 4.3 Pure TCAM Costs

In order to implement the pure TCAM solution we need to have all Snort patterns and their shifted patterns inside the TCAM (the patterns are divided by predetermined length - the TCAM width). As shown in figure 4.2, in order to implement the current solution while the TCAM width is 32, we need 1990KB of TCAM memory. TCAM costs much more than conventional memory so this implementation might not be acceptable by some of the IPS devices manufacturers. We would like to reduce the required TCAM memory. There are several ways to do so, each with well defined tradeoffs. Our first solution is an optimization of the algorithm described in section 4.1. The basic idea is to reduce the TCAM size by the price of reducing the average shift value. Instead of having the patterns and all their shifts inside the TCAM, we insert to the TCAM only half of the shifted patterns. For each pattern, instead of creating w-1 shifted patterns, we create only the first $w/2$ shifted patterns. This optimization decreases the shift value as shown at figure 4.2 but we still need 1148KB of TCAM for width$=32$ which may still not be practical. We

[4]This speed corresponds only to the TCAM lookups.

can create even less shifted patterns, for example, we can create no more than 10 shifted patterns for each pattern, i.e. the minimum of $w/2$ and 10. At this case, we decrease the needed TCAM memory to 733 KB which might be more reasonable. As explained at section 4.2.1, this optimization directly influences the shift average resulting in decreasing the algorithm rate. In this solution, the shift value can not exceed $w/2$. In other words, if we fall in to the last entry in the TCAM the shift value is $w/2$ and not $w$ as this entry would have returned at the original solution.

We would like to find a way that keeps the average shift value high but still uses less TCAM memory than the original solution. We also want to have wide enough TCAM so we can avoid false-positives and keep the search speed on several GBps.

## 4.4 The Trie Solution

In this solution we combine, in addition to the TCAM memory, a trie that will supply us the shift values. The trie can be implemented on a regular memory. All the shifted patterns are removed from the TCAM therefore, this solution satisfies the TCAM size constraints.

### 4.4.1 Trie Overview

A Trie is an ordered tree data structure that is used to store an associative array where the keys are strings. Unlike a binary search tree, no node in the tree stores the key associated with that node; instead, its position in the tree shows what key it is associated with. All the descendants of any one node have a common prefix of the string associated with that node and the root is associated with the empty string. Values are normally not associated with every node, only with leaves and some inner nodes that happen to correspond to keys of interest. A trie can be seen as a deterministic finite automaton, although the symbol on each edge is often implicit in the order of the branches.

Looking up a key of length $m$ takes at the worst case $O(m)$. Also, the simple operations tries use during lookup, such as array indexing using a character, are fast on real machines. Since the keys are not stored explicitly, only an amortized constant amount of space is

needed to store each key. Since the key is not present, tries are most useful when the keys are of varying lengths and we expect some key lookups to fail.

### 4.4.2 The Algorithm

We use the same principles as the Aho-Corasick [AC75, CW79] solution. The input to the trie is all the prefixes to the patterns (without their shifted patterns). The trie's height can be different than the TCAM width. The trie height determines the size of the prefixes that are inserted to it. We call this size the *trie width*. For each pattern prefix from the input, there is a path in the trie. There are, of course, less nodes than *the number of prefixes* $\times w$, since there are a lot of patterns that share the same prefix, these patterns also share the same nodes in the trie. In the root node, there is a self-loop with the *don't-care* sign, this loop gives us the shift values. Figure 4.4 illustrates the trie for $w = 4$ and the prefixes *ABCD* and *XYZW*:

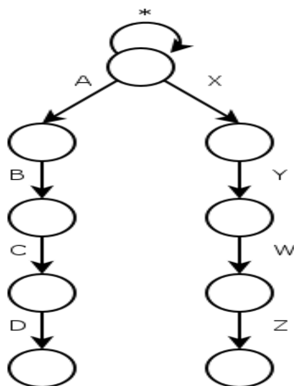



Figure 4.4: Part 1: The Trie

At the RTCAM solution we had the patterns *ABCD, ?ABC, ??AB, ???A, XYZW, ?XYZ, ??XY, ???X, ????*. Each pattern from this list can be represented using the trie, the shift value is the number of steps that we stepped in the root. The height of the trie is exactly the trie width. The algorithm is very similar to the RTCAM algorithm, we will emphasize the parts where the two algorithms are different.

Initially the text is aligned with the TCAM. A string of the size *TCAM width* is fetched and inserted as a key to the TCAM. At the same time, a string of the size *trie width* is fetched and inserted as a key to the trie. The *TCAM width* can be different than the *trie width*, in fact, *TCAM width*, most of the time, is greater than the *trie width*. Thus, we can use a reasonably small trie even though the TCAM width is wider. If the TCAM finds a match, we continue with the TCAM algorithm. In case there is no match, we use the shift value we get from the trie in order to shift more than one step in the text.

We extend this approach and remove from the TCAM all patterns whose length is smaller than the trie width. Since, the match for these patterns can be found in the trie, there is no need to put them in the TCAM. In order to do this we need to add a pointer to the patterns table from each node in the trie that represents a pattern match. The trie width can be determined by the time it takes the TCAM to return an answer, we want the TCAM and the trie to simultaneously finish.

In current memory technologies, memory access speed is 5 times faster than TCAM access speed. Therefore, the trie width can be 5 and both engines (TCAM and trie) can work freely in parallel.

We will now compare this solution with the RTCAM algorithm.

### 4.4.3 Comparison to the Pure TCAM Algorithm

It seems that if we can have the trie width to be large enough, the solution will be as good as the RTCAM algorithm. The truth is that by using less TCAM we lose a little in the rate of the algorithm.

For example, if we look at the trie in figure 4.4 and the key is *ABRR*, in the RTCAM algorithm we would get 4 as the shift value but in the trie we will start at *ABCD* path and after two steps we will not be able to proceed. As we described the trie algorithm till now, the returned shift value is 0.

We add two optimizations to the algorithm - the first: if we start in some path in the trie and after few steps we can not proceed, we can return 1 as a shift value, meaning: there is no match from the first character of the key. The second optimization: If in one search we

can make $w$ comparisons and we found out after $k$ steps $(k < w)$ that there is no match, we can continue to search $w - k$ characters. In this case, we will return to the root of the trie with the next position in the text.

In the given example, since after taking two steps (reading $A$ and $B$) we couldn't proceed in the trie, we can search two more characters. We will go back to the root with the key *BR* (we have only two more comparisons to make). Therefore, the returned shift value is 2.

Even if the key was *RABR*, we would step once in the root (shift value is one), then we would start to proceed in the *ABCD* path, we would get stuck after two more steps. In this stage, we return to the root and the key is $B$. The shift value is 2 again.

### 4.4.4 Paralleling The Trie

If we enter to the trie in parallel, we can insert several keys at the same time. Let's assume that we have $k$ tries working in parallel and the trie width is $w$. The first key is $Text[pos, pos + w - 1]$, the second key is $Text[pos + 1, pos + w]$ and the $k'th$ key is $Text[pos + k - 1, pos + k + w - 2]$. In this way we can gain a better shift value. Each entrance to the trie can give us a shift value between 0 to $w$. If the $i'th$ key gives $w$ as a shift value, the $i + 1'th$ key has to give at least $w - 1$, the $i + 2'th$ key $w - 2$ and so on. For that reason, the most significant value is the one that returned from the $k'th$ key. The maximum returned shift value is $k - 1 + shift[k]$ unless some of the keys give 0 as a shift value.

Generally, if $t$ keys returned shift value that is greater than 0 before the first key which gave 0, the total returned shift value is $t - 1$ + shift[t]. If $t/neqk$, $shift[t] = 1$. The maximum number that we can get as a returned shift value from all the k keys is $k - 1 + w$.

For example, suppose the text is *ABRRRRR* and we have 4 engines that work in parallel, the first key is *ABRR* and the returned shift value is 1, the second key is *BRRR* and the returned shift value is 4, the third key is *RRRR* and the returned shift value is 4 and the fourth key is *RRRR* and the returned shift value is also 4. The total returned shift value from all the engines is $3 + 4 = 7$, which is the maximum value we can get.

With this extension, we miniaturize the ”less-shift” problem we described, since even

if the $i'th$ key gives a smaller shift value as it would have give in the RTCAM algorithm, the keys that follows it will find the real shift. The problem remains for the $k'th$ key.

## 4.5 The Prefixes Solution

In this solution, the TCAM contains only the prefixes of the patterns and their full shifted patterns. Each entry points to a list of suffixes which is stored in a regular memory. When there is a match, the algorithm continues the comparison on the regular memory (the bus is usually 4 bytes wide so we can compare 4 bytes at a time). In contrast to the trie algorithm, the TCAM does not need to wait for the regular memory result, it can continue to the next position. Since more than one sequential TCAM match can occur, the second match might wait for the first match to finish, therefore, the regular memory lookup can be a bottle neck.

By having few engines working in parallel, we can finish the suffix comparison within one TCAM lookup which will solve the problem completely.

## 4.6 The 2-TCAMs Solution

This solution contains three separated TCAMs $T_1$, $T_2$ and $T_3$ and two different sizes, $w_1$ and $w_2$. The first TCAM, $T_1$, has a wider width, $w_1$, and it contains all patterns whose size is greater than $w_2$ (without their shifted patterns). The patterns are divided by $w_1$ as in the RTCAM algorithm. The second and third TCAMs, $T_2$ and $T_3$, have a narrower width, $w_2$, and they contain patterns whose size is less than $w_2$ and all the shifted patterns (of both TCAMs). $T_2$ and $T_3$ are exactly the same.

The key for $T_1$ is $T_{[pos,..,pos+w_1-1]}$ and the two keys for $T_2$ and $T_3$ are $T_{[pos,..,pos+w_2+1]}$ and $T_{[pos+w_2,..,pos+2\times w_2-1]}$. By having all the three TCAMs work in parallel, we simulate the scenario where in one hit at the wide TCAM we hit twice the narrow TCAM.

There is no constraint on the relation between $w_1$ and $w_2$, however, if $w_1$=$2\times w_2$, we could achieve the same result as if we had one TCAM at width $w_1$ that contains all the patterns and their shifted patterns.

Let $S_1$ denote the shift value that is returned by the $T_2$ and $S_2$ the shift value that is returned $T_3$. If $S_1 < w_2$, $S_1$ will be the actual shift value, otherwise ($S_1 = w_2$), the actual shift value will be $w_2$ + $S_2$. If $S_1 < w_2$, we lose the benefit of having two copies of the narrow TCAM. By making some probabilistic assumptions we can achieve better results. Since we assume a gaussian distribution, 68% of the data elements are within one standard deviation of the mean, 95% are within two standard deviations and 99.7% are within three standard deviations.

Let *mean* denote the shift average results by TCAM width $w_2$ and *sigma* the standard deviation, we can use this information to determine the key start position of $T_3$. If *pos* is the starting position of the key to $T_2$, the second key can start at position $pos + mean - n \times sigma$. So, if we take $n = 2$, in 95% of the cases, $pos + mean - n \times sigma$ will be lower or equal to $S_1$. In the remaining 5%, where $S_1$ is lower than $pos + mean - n \times sigma$, the actual shift value for the first TCAM will be $S_1$, no matter what is the value of $S_2$.

The larger $w_1$ is, the less patterns divisions we have (less false positives). The larger $w_2$ is, the bigger shift value we get and the faster our algorithm is. If for example, we take $w_1$ to be 32 and $w_2$ to be 12, we will need to divide only 5% of Snort's patterns, we will get a shift average of  6 and we will need only 472KB of TCAM memory. If we'll take $w_1$ to be 64, around 0.001% of the patterns will have to be divided and we will need only 500KB of TCAM memory.

Table 4.1 shows the TCAM memory size for two different combinations of $w_1$ and $w_2$.

| TCAM size requirement | | |
|---|---|---|
| $w_1$ | $w_2$ | *TCAM size* |
| 12 | 32 | 472 |
| 12 | 64 | 500 |

Table 4.1: Part 1: 2-TCAM solution - TCAM size requirements

The main advantage of this solution is that we still gain a high shift average with a practical size of TCAM. The TCAM separation causes less divisions of patterns which reduce the search speed. Another advantage is that the short patterns don't waste a lot of space in the TCAM since $w_2$ is relatively small. There are two drawbacks, one - we double

$T_2$ and second - since we use a smaller TCAM, we will have more false positives than in the RTCAM solution which may slow down the algorithm.

## 4.7 Possible Attack

Since we save all the matched patterns in the *matched patterns list*, attackers can send packets that will enlarge this list without the NIPS intercepting the attack. For example, if there is a rule of the form: content:"ab"; content:"cd"; within:10; and the packet payload contains 500 appearances of "ab", as it was described till now, we will have at least 500 entries in our *matched list*. We can reduce the list length by keeping to the following principles: One, if we matched a pattern of a correlated rule and the next pattern in the rule contains only the *distance* keyword, we can save only the first match of this pattern. Second, if the next pattern in the rule contains only the within keyword, we can save only the last match of this pattern. This is significantly decreases the size of the Matched List table but it does not solve the problem completely. There is more than one way to completely solve the problem:

- The first way is dropping the packet to the CPU - we can decide that if the length of the *matched list* exceeds some predetermined number, we drop the packet to CPU treatment.

For the next two solutions we need to store in each entry of the *matched list* another field called *removePos*. The value of this field is the current position of the match plus *within* value of the next pattern (if exists). If the current position is greater than this value, we can remove the entry from the list. The following two solutions use this value to minimize the length of the list.

- Heap - we can use a heap ordered by the 'removePos' value in an ascending order. At each position we'll check the head of this heap and remove it if its value is smaller than the current position.

- Different Task - a different task runs in the background, its role is to check the 'removePos' field of each entry and remove those whose value is smaller than the current position.

# Chapter 5

# Porting RTCAM to SRAM

Due to the high price of TCAM memory and the increasing number of signatures, a TCAM oriented solution may not be acceptable to the industry. In order to elevate the problem, we propose an alternative algorithm using a hash function that creates fingerprints of the packet payload which are then compared with the patterns signatures (previous works that are using hashing technique can be found in [KR81] and [WM93, WM91]).

For brevity we assume that all patterns have a length[1] of $m$ which is **larger** than the sliding window of $width$ bytes. The hash value is calculated on the sliding window and is used as a key later on.

As in the RTCAM algorithm, we use a shift table. Instead of creating a shift table that maps a single character to a shift value, we create shift values for a block which contains $B$ characters. Using blocks we reduce the amount of false matches. When a packet is processed, a block of characters is extracted from the search window (right to left) which are used as an index to the shift table. The shift value is retrieved from the shift table. If it is zero, then an exact match must be performed, otherwise we can shift the text.

In order to minimize the search space, the patterns are linked using a hash value. We calculate a hash value using the sliding window's text and locate the patterns bucket in the hash-table. The algorithm iterates the relevant patterns as done in the RTCAM algorithm.

[1] We will later show how the algorithm is used for matching patterns of smaller lengths.

## 5.1 Data Structures

In order to implement the hash-based solution we maintain several data-structures. We start by describing each one.

**Sub-Patterns Hash Table (HT)** - replaces the TCAM in the TCAM based algorithm. The table resides in SRAM and contains the $r$ patterns divided by $w$ which is the search window's width. Entries with less than $w$ bytes are prepended (padding at the prefix) with the suffix of the previous part. A key is calculated on the sub-pattern and the sub-pattern is placed in the table accordingly.

**Shift Table** - we keep a shift table for a block of characters $B$ (as in the algorithm presented by Manber [WM93, WM92, WM91]). The block of characters is used while preprocessing the patterns to construct the shifting table. Within the sliding widow, we look at the text, $B$ characters at a time. For simplicity[2] assume that the table size is $\Sigma^B$, where $\Sigma$ is our alphabet. Each entry corresponds to a distinct substring of length $B$.

Let $X = \{x_1, x_2, .., x_B\}$ be a string corresponding to the $i$'th entry of the shift table. There are two cases: either $X$ appears somewhere in one of the patterns or not. If $X$ does not appear in any of the patterns, we store $m - B + 1$ in the corresponding shift entry, otherwise, we find the rightmost occurrence of $X$ in any of the patterns that contain it; suppose it is in $P_j$ and that $X$ ends at position $q$ of $P_j$. Then we store $m - q$ in the table.

If the shift value is greater than zero, we can safely shift. Otherwise, it is possible that the current substring we are looking at in the text matches some pattern in the pattern list. To avoid comparing the substring to every pattern in the pattern list, we use the previous defined hash table (that minimizes the number of patterns to be compared).

[2] The table can be easily compacted by hashing the $B$ characters and setting the shift value to be the minimum of the values corresponding to the same bucket.

**Patterns Table** - an array of patterns ordered by a *patternID*.

**Matched Patterns List** - each entry contains the matched patterns and its corresponding end position in the text.

## 5.2 Hashed-Based Pattern Matching Algorithm

The algorithm follows the same logic as the RTCAM algorithm. A string of *width* bytes is fetched, the string's rightmost *block* bytes are inserted as a key to the shift table and the shift is retrieved. If the shift value N is not equal to 0, the text position is shifted right by N. If it is 0, then we have a possible pattern match and we need to look at the patterns pointers. We follow the hash entry's list, which points to the patterns that have the same key as the matched string. The rest of the algorithm is exactly the same as the RTCAM solution. The algorithm is given at 2.

## 5.3 Dealing with Short Patterns

There is a major difficulty in designing a unified algorithm that deals with long patterns and short ones since the performance is typically influenced by the overhead caused by the short patterns.

It is important to note that ClamAV signatures are quite long (an average of 124 bytes), where Snort's signatures are shorter (average is only 12 bytes).

In order to deal with short patterns, we need to pad them to the width on which the hash is applied. Since there is no way to pad short patterns with *don't care* signs as we did in the TCAM based solution, a real pad must be constructed. The pad can be usually extracted from the flow context. We remind the reader that the pattern matching algorithm is invoked in the context of a specific packet. This packet is a part of a flow which is identified by the protocol identifier, ports, source and destination IPs etc.

The pad is constructed twice, once, when creating the hash function (the flow is extracted from the rule) and the second time, when constructing the search key (the flow data

**Algorithm 2** Hash-Based Pattern Matching

```
1: T(Packet) = {T_i, 1 ≤ i ≤ n}
2: pos ⇐ 1;
3: shift ⇐ 0
4: width ⇐ default
5: B ⇐ default
6: while pos ≤ n − width do
7:     block ⇐ T[pos+width−B,..,pos+width−1]
8:     shift ⇐ SHIFT_TABLE[block]
9:     if shift is 0 then
10:        key ⇐ hash(T[pos,..,pos+width−1]) {construct a fingerprint}
11:        for all current = hash[matchedRow].next ≠ null do
12:            if pos < current.offset then {check offset}
13:                continue
14:            end if
15:            if pos + current.Len > current.Offset + current.Depth then {check depth}
16:                continue
17:            end if
18:            if current.len ≤ width then {exact match!}
19:                matched ⇐ true
20:            else
21:                matched ⇐ checkSubPatterns(current); {iteratively checks for sub-patterns}
22:            end if
23:            if matched = true then
24:                if current.distance ≠ −1 OR current.within ≠ −1 then {related pattern}
25:                    if MatchedList.contains(current.Id - 1) then {previous pattern in Matched List}
26:                        prevPos ⇐ MatchedList.get(current.Id − 1).pos
27:                        if pos ≤ prevPos + current.Distance then {check distance}
28:                            continue
29:                        end if
30:                        distance ⇐ Max{current.distance, 0}
31:                        if pos + current.len − 1 > prevPos + distance + current.within then {check
                           within}
32:                            continue
33:                        end if
34:                    else
35:                        continue
36:                    end if
37:                end if
38:                MatchedList.add(current.Id, pos + current.Len)
39:                Rules[ruleId].markEntry(current.id)
40:                if Rules[ruleId].allMatched() then
41:                    FOUND AN ATTACK
42:                end if
43:            end if
44:        end for
45:    else
46:        pos ⇐ pos + shift
47:    end if
48: end while
```

is constructed from the received packet).

Most of the short rules specifically state their flow identifier so actually a patterns is more than just the text - it is part of a flow. It is rather intuitive that as the pattern length increases the flow identifiers become general (i.e., any source ip etc.).

A padding technique must be used in order to create a key for locating short patterns. A random pad is the most trivial solution. This method is suitable for patterns with almost no flow identifiers. Using the flow identifier to create a pad is better practice since it reduces the amount of false positives while comparing the hash value with the fingerprints. For example, if we are currently parsing an FTP packet, the hash function will use the protocol (sub-protocol) as a pattern padding thus reducing the amount of false matches.

The following figure illustrates the hash calculation for short patterns as the packet received:

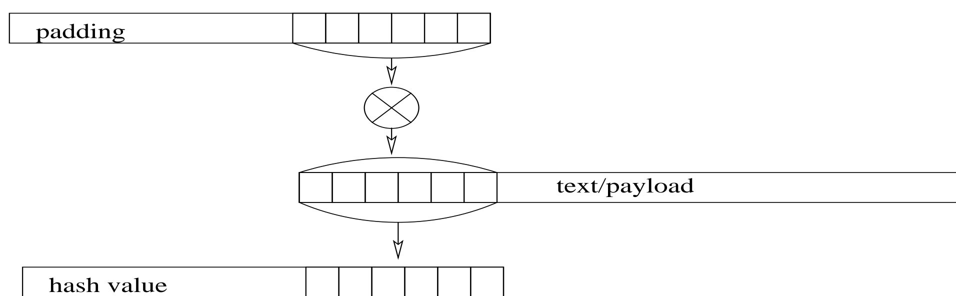



Figure 5.1: Part 2: Constructing The Hash Value

In order to locate short patterns fingerprints we must continuously calculate the hash for increasing sizes of text (pattern length) up to the hashing window size. Calculating the hash value should be done quickly. For example, one can create a full pad using a hash function and only use a simple XOR operation on the text. In order to recalculate the next hash we can XOR the text again, increase the text window width and XOR again with the pad.

The problem with this solution is the need to calculate $w-1$ hashing keys. Suppose that $w=4$, the text key is *ABCD* and the patterns are *XYZW, D*. Since *D* is a short pattern, its

key in the hast table is padded. We need to locate $D$ in our text. If $T_i$ is the result of padding the i'th character from the text, we need to create the keys: $T_{pos}BCD$, $T_{pos}T_{pos+1}CD$ and $T_{pos}T_{pos+1}T_{pos+2}D$. The hash search with the last key will find the pattern $D$.

In order to deal with this problem we can use several hash engines in parallel, each engine calculates the key and searches the hash. This solution solves the described problem, it is simple and it can be easily implemented in hardware or with a network processor with several hash engines.

# Part II

# Classification Engine Component

# Chapter 1

# Introduction

Stateful classification engine is usually a network layer operation. Unlike static packet filtering, which examines a packet based on the information in its headers, stateful inspection tracks each connection traversing the packet processor and ensures the packets are valid.

An example of a stateful device is a firewall. A firewall may examine not just the header information but also the contents of the packet in order to determine more about the packet than just information about its source and destination. A stateful inspection firewall also monitors the state of the connection and compiles the information in a state table. Therefore, filtering decisions are based not only on administrator-defined rules (as in static packet filtering) but also on context that has been established by prior packets that have passed through the firewall.

An additional example is a virus scanning application, which filters packets that may contain viruses (or unwanted cookies etc.), a content based switch which switches packets according to either XML tags in the packet payload or the URL within them etc. The "state" needed here is due to the possibility that the object that is searched for in the packet stream may well cross packet payload boundary (i.e., it is carried in more than one packet).

Like viruses, most intruder activities have some sort of signatures. Signatures may appear in different parts of a data packet depending upon the nature of the attack. For

example, one can find signatures in the IP header, transport layer header (TCP or UDP header), application layer header or payload. Usually IDS depends upon signatures to find out about intruder activities. Some vendor-specific IDS require updates from the vendor to add new signatures when a new type of attack is discovered. In other IDSs, e.g. Snort, one can update signatures manually.

Our presented classification engine is dynamic and stateful for high speed networks. It is capable of looking within the application payload and making decisions on the significance of that data based on its content. The engine classifies the packet using the string-matching technology described at the first part of this thesis.

Chapter 2 presents some of the best known hardware-based IDSs in the industry. Chapter 3 describes our algorithm and the interactions with the string-matching algorithm.

# Chapter 2

# Related Work

Stateful inspection provides an analysis of packets at the network layer as well as other layers in order to assess the overall packet. By combining information from various layers, the engine is better able to understand the protocol it is inspecting. This also provides the ability to create virtual sessions in order to track connectionless protocols. The reality of modern application demands and capabilities requires more intimate knowledge of the application payload. The engine must not only maintain the state of the underlying network connection but also the state of the application utilizing that communication channel.

While current stateful firewall technology provides for tracking the state of a connection, most current firewall products offer limited analysis of the application data.

The best known IPSs are Fortigate-800 appliance [For], Secure Soft Absolute IPS NP5G, NP10G [Sec] and Snort [Sno, NR03]. Since Snort is an open source, high-performance IDS it is very common and documented.

## 2.1 Snort

Snort is an open source Network Intrusion Detection System (NIDS), which is available free of cost. It is a host based IPS, implemented fully in software. Snort uses rules stored in text files that can be modified by a text editor. Rules are grouped in categories. Rules

belonging to different categories are stored in separate files. These files are included in a main configuration file called "snort.conf". Snort reads these rules at start-up and builds internal data structures to apply these rules to the received data traffic. Finding signatures and using them in rules is a tricky job, since the more rules one uses, the more processing power is required to scan the data in real time. It is important to implement as many signatures as one can, using as few rules as possible. Snort comes with a rich set of pre-defined rules to detect intrusion activity and it is possible to add more rules at will.

In earlier days, Snort used brute force pattern matching which was very slow. The first thing done to boost performance was implementing a partial Boyer-Moore pattern matching algorithm [BM77]. After a couple of months a full implementation of Boyer-Moore was implemented. Next was the implementation of a 2-dimensional linked list with recursive node walking, which gave Snort a 200 to 500 percent performance increase. Finally the detection engine was rewritten to include a linked-list-of-function-pointers, also called a three-dimensional linked list. It is a Boyer-Moore like algorithm applied to a set of keywords held in an Aho-Corassick like keyword tree that overlays common prefixes of the keywords. This new algorithm takes the best characteristics of both the Boyer-Moore and Aho-Corasick algorithms [AC75]. Current Snort performance with one high end PC is 300-400 Mbit/s

## 2.2 Fortigate Appliance

Fortigate appliance is an edge security appliance that is installed at the edge of the network and provides security protection to the network from harmful traffic through the gateway.

Specific functions provided by the vendor include anti-virus scanning, including the ability to scan for viruses within VPN encrypted tunnels, stateful inspection firewalling, Web content filtering based on URL and/or keyword and phrase-based blocking and intrusion detection/prevention of over 1300 types of attacks. The appliance data itself (attack patterns, signatures, etc.) is automatically maintained and updated via continuous updates

from the vendor's FortiProtect Network. Primary differences in the many FortiGate models made available by the vendor are in terms of speed/capacity. FortiGate-3000 appliance provides 3G of firewall throughput and 1G intrusion detection speed. As for anti-virus and content filtering, Fortinet uses an ASIC-based architecture.

The 3000 NPG comes with three gigabit and three 10/100 Ethernet ports. Users can group ports and assign different security and content filtering policies to each, providing multiple security zones within the enterprise.

## 2.3 SecureSoft Absolute IPS NP5G, NP10G

The Absolute IPS series of SecureSoft [Sec] is a family of hardware appliances designed to detect and prevent attacks across multiple network segments at up to 8Gbps (maximum aggregate throughput). Three models are available covering various sizes of installation and ranging from 2-8Gbps bandwidth.

The NP5G protects multiple network segments at up to 4Gbps. The device supports four 10/100/1000Mbps ports for detection and protection and three additional ports for management. The NP10G provides maximum aggregate throughput of 8Gbps and maximum open sessions of 3,000,000.

All packet processing is performed by a dedicated network processor (NP) on board of the card, which provides very high level of performance. SecureSoft claims up to 8Gbps throughput while conducting deep packet inspection. SecureSoft uses a proprietary signature matching algorithm called BPM (Bi-parallel Matching) which ensured that all signatures (currently, 1700 of them) are matched in a single operation.

Since the resources found on the web doesn't tell if this throughput is the worse or average case, it is impossible to compare the two. We argue that a major drawback of this device is its lack of compatibility with Snort. The administrator is required to enter platform dependant rules manually. Since Snort rules syntax is becoming a standard, future intrusion prevention systems are encouraged to adopt it.

# Chapter 3

# The IDS

This part of the thesis specifies the requirements from the new classification framework. In the context of this thesis we will refer to the whole classification framework as "Enhanced Classification Engine (ECE)".

Figure 3.1 shows the general classification engine architecture. The packet is first received at the static flow classifier (SFC) component which only extracts information from the headers, a process that yields a flow descriptor. The packet is then transferred to the classification engine which uses the pattern-matching algorithms presented at 4.1.

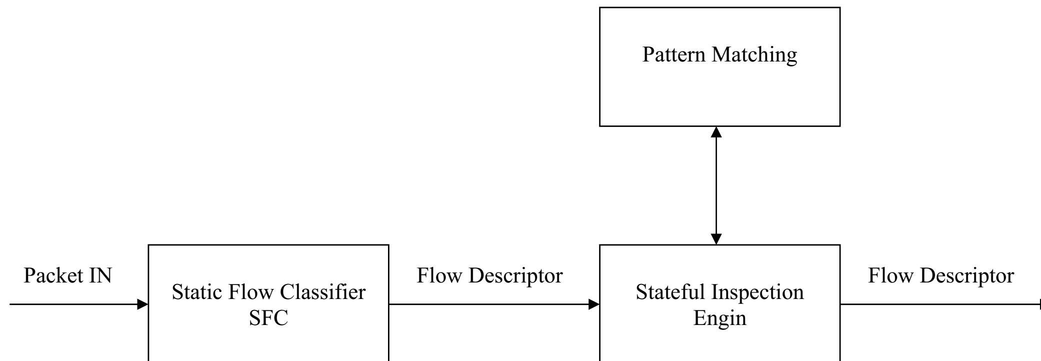
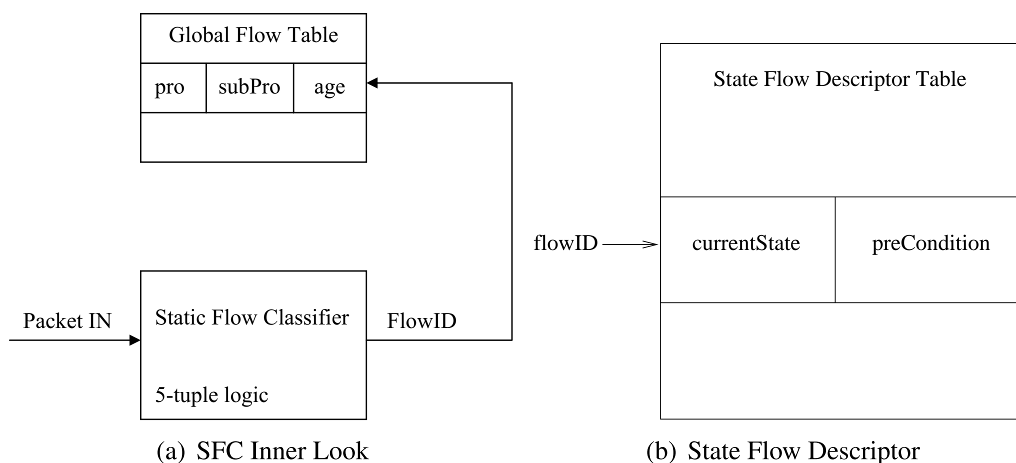



Figure 3.1: Part 2: Enhanced Classification Engine Architecture

## 3.1 Data Structures

We first describe the various data structures needed in order to implement the Enhanced Classification Engine. The first two tables are part of the implementation of the SFC component (see Figure 3.1).




(a) SFC Inner Look (b) State Flow Descriptor

Figure 3.2: Part 2: GFC and SFD

**Global Flow Table (GFT)** - contains the currently active flows. A flow is usually identified by 5-tuple: source IP, source port, target IP, target port and protocol (sometime the TOS/DSCP field is also used). The Static Flow Classifier returns an index to this table. Each GFT entry includes a *protocol* field (for example TCP), a *subProtocol* field (for example FTP) and an *age* field that may be used to destruct obsolete flows.

**Static Flow Classifier (SFC)** The SFC module performs the first stage of the classification process. The output of this process is a flowID index of the GFT data structure. Figure 3.2(a) presents an inner look at the static flow classifier. The flow descriptor is constantly used in order to identify the flow's protocol and state machine descriptions.

**State-Flow Descriptor Table (SDT)** Each packet flow follows a state machine (implementing a predefined protocol) according to the flow's protocol (or sub-protocol). The SDT contains entries regarding the current state within the state machine of each flow. Each entry contains a *currentState* and *precondition* (see fig 3.2(b)). The *currentState* indicates the current flow's state in the protocol's state machine. The field precondition is used to enable conditional state machine transitions. Every state transition is triggered by a protocol specific event. However, some events are expected only if a given previous event had occurred. This requirement is reflected in the precondition variable. The *preCondition* is used when indexing the PDT in order to find the next state. Quite a few protocol state machines can be compressed using this field thus avoiding creating many intermediate states.

**Protocol Description Table (PDT)** The PDT table contains the actual state machine for each protocol. Each table entry describes an edge (link) between two states according to the specific protocol's state machine. Since this table is accessed by searching it using a search-key, it is best implemented by a TCAM. The table can be written and updated at any given time (though we expect to update it rarely as protocols hardly change). The normal procedure would be that the SW installs the finite state machines that need to be tracked at initialization time. A state transition within a specific protocol is triggered by an external event (e.g., the reception of a packet). Upon the reception of such an event, the PDT is searched for the next state. The search is done using a search-key comprised by a *protocolID*, the *currentState* (which is taken from the SDT), the *preCondition* and the *eventID* which had just occurred. Each PDT Entry (PDTE) contains the following fields: *ProtocolID*, *currentState*, *eventID*, *preCondition* and *nextState*. Appendix A provides an example for the Protocol Description Table of a light-bulb on/off state machine.

**Event Packet Classifier (EPC)** The transition of the flow's state according to the protocol's state machine is triggered by an *eventID*. The EPC module provides the mapping between a protocol's packet and an *eventID*. For example, a TCP packet that contains a SYN flag should be mapped to a *TCP_SYN_EVENT* identifier. The EPC

uses the string matching algorithm as a tool to search for protocol specific patterns in the packet. The EPC operation is illustrated in Figure 3.3.

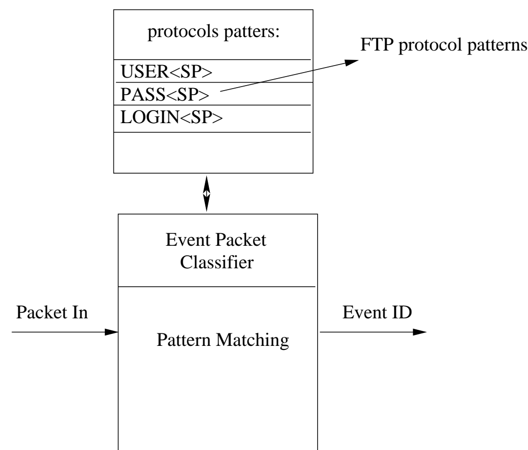



Figure 3.3: Part 2: Event Packet Classifier Operation

The pattern matching mechanism is used for two purposes. One is to find an event within the packet. The second purpose is for virus and intrusion detection. The lookup can be done concurrently or sequentially. In any case, the result is combined and if a virus was detected various actions can be taken (among them are dropping the packet with/without notifying the host CPU, or just trapping the host CPU with the relevant information) [1].

**Event Attribute Table (EAT)** The EAT entries extend each *eventID* with additional information. The EAT is a two dimensional table that is indexed by the *protocolID* and the *eventID*. The entry EAT[*protocolID*,*eventID*] contains a *postConditon* value. The *postConditon* value should be copied to the SDT as it will be used as a *preCondition* for the next state machine transition . Note that this mechanism has been introduced to simplify the creation of the protocol state machine and to decrease

[1] Note that if no cross-packet searches are assumed then the virus lookup (not Snort rules!) can be done at a previous stage (even before the initial packet classification into a flow) independent of the whole stateful classification process.

memory requirements (so enable them to reside on a TCAM).

**Protocol Associations Table (PAT)** The PAT purpose is to maintain associations between protocols and sub-protocol flows. For example, suppose an FTP flow has been initiated. If the FTP is closed by the user it is possible using PAT, to locate the related TCP data flow and modify it (for example decrease the flow's age field [2]).

## 3.2 FTP Example

The best way to describe the algorithm operations is through an example. The following section presents the data structure and the basic control flow for the FTP protocol.

### 3.2.1 Data Structures

The general FTP state machine is described in figure 3.4. A *command* is a general FTP command like: *USER*, *PASS*, *PORT*, *CDUP* etc. The *OK*, *OK_CONT* and *FAIL* are also FTP events which correspond to the server's response. We assume that the Enhanced Classification Engine is locates as an edge server thus receiving all of the requests and responses.

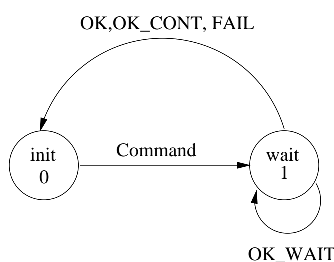



Figure 3.4: Part 2: FTP Finite State Machine

Assume that the EPC provides the mapping states in table 3.1. The *eventID* is returned from the EPC after consulting the string matching utility.

[2]This field can be a single aging bit.

| EPC Mappings | |
|---|---|
| *eventID* | *Pattern* |
| 10 | $USER < SP > \Phi \times LIMIT < CRLF >$ |
| 11 | $PASS < SP > \Phi \times LIMIT < CRLF >$ |
| 12 | $PORT < SP > \Phi \times 3, \Phi \times 3, \Phi \times 3, \Phi \times 3, portNum, portNum < CRLF >$ |
| 13 | $CDUP < CRLF >$ |
| 14 | $LIST < SP > \Phi \times LIMIT < CRLF >$ |
| 20 | $1\Phi\Phi < SP > \Phi < CRLF > \quad \Leftrightarrow \; OK_WAIT$ |
| 21 | $2\Phi\Phi < SP > \Phi < CRLF > \quad \Leftrightarrow \; OK$ |
| 22 | $3\Phi\Phi < SP > \Phi < CRLF > \quad \Leftrightarrow \; OK_CONT$ |
| 23 | $4\Phi\Phi < SP > \Phi < CRLF > \quad \Leftrightarrow \; FAIL$ |
| 24 | $5\Phi\Phi < SP > \Phi < CRLF > \quad \Leftrightarrow \; FAIL2$ |

Table 3.1: Part 2: FTP EPC

| Protocol Description Table | | | | |
|---|---|---|---|---|
| *ProtocolID* | *currentState* | *eventID* | *nextState* | *preCondition* |
| 2021 | 0 | 10 | 1 | $\Phi$ |
| 2021 | 0 | 11 | 1 | 10 |
| 2021 | 0 | 14 | 1 | 11 |
| 2021 | 0 | 14 | 1 | 12 |
| 2021 | 0 | 12 | 1 | 11 |
| 2021 | 1 | 21 | 0 | $\Phi$ |
| 2021 | 1 | 20 | 1 | $\Phi$ |

Table 3.2: Part 2: FTP PDT

The relevant Protocol Description Table is given in table 3.2.

Assume that the protocol ID is 2021. Notice that in order to process $eventID = 11$ (PASS) a previous event of 10 (USER) must hold. Also for $eventID = 14$ (LIST) the previous event must be 11 (PORT) or 12 (PASV) so we create two PDTEs. Also notice that there are cases where the state machine must introduce intermediate states in order to enforce the precondition. For example consider event $A \rightarrow B$ (A depends on B) and $C \rightarrow D$. If B is received at the initial state and sets the precondition to “B”, then D is received (and allowed) - the precondition is overridden by D to “D”. Finally, when A arrives, there is no trace that precondition B existed. In order to overcome this, we must introduce an

| Event Attribute Table | |
|---|---|
| *eventID* | *postCondition* |
| 10 | 10 |
| 11 | 11 |
| 12 | 12 |
| 13 | Φ |
| 14 | Φ |

Table 3.3: Part 2: FTP EAT

intermediate state in the state machine. It is important to note that the precondition and post condition is used as long as we gain memory space in the price of a slightly complicated state precedence rules.

In order to return to state 0 (to enable issuing a new command) we must get a response from connection peer and identify the corresponding flow. Thus, we must consider all of the packets relating to the specific flow in both directions (transmitted and received). We also have to identify the two sides of the flows as one conversation. This can be done by automatically updating the SFC and pointing from both flow entries to the same GFT entry.

The corresponding Event Attribute Table (EAT) is shown in table 3.3;

### 3.2.2 Control Flow

Let's assume that a new FTP packet is received, it contains the PORT command and the user has been already logged in (USER and PASS has been invoked). The following steps comprise the Enhanced Classification Engine operations.

1. SFC returns the relevant *flowID*, *protocol* and *subProtocol* (we assume that the TCP has been already checked according to an iterative classification assumption - see open issues).

2. Using the *flowID* as an index to the SDT we get the *currentState* that indicates the current flow's state in the protocol's state machine and the *preCondition* which holds

the current state's pre-condition that must be held in order to change the flow's state to the next one.

3. Using the sub-protocol (which contains 2021 in the FTP case) and the packet, we invoke the EPC module to identify the *eventID* which corresponds to the FTP (sub-protocol) packet. The invocation returns the desired *eventID* - in this case it is 12. The string matching algorithm is invoked in the context of this *flowID*.

4. Using the sub-protocol, *currentState* and the *preCondition* as indices to the PDT, we get the *nextState* if there is a match (in this case 1), otherwise the packet is illegal and may be discarded or not according to a predefined rule in some sort of a rule table. In this example, the *protocol* is 2021, the *currentState* is 0 and the *preCondition* is 11 since we assume that the user has already logged in.

5. Before the SDT's *currentState* is updated to the *nextState*, we locate the *eventID's* EAT entry to get the *postCondition*. If it is not $\Phi$, we copy the value to SDT's *preCondition* and then change the current state to the next one (*nextState*). In our case the *currentState* will be changed to 1 and the new *preCondition* will be set to 12 (so when event 14 will arrive its *preCondition* will be satisfied).

# Part III

# Simulation

# Chapter 1

# Experimental Results

We simulated our NIPS based on the pure TCAM solution since this is the basic algorithm and can be easily extended. Moreover, this solution's results are the basis of all the other solutions we have presented.

We experiment our simulation with two complex pattern sets. One, is a virus signature set from ClamAV [Cla], which contains (relatively long) simple patterns only. The second set is from Snort [Sno] intrusion detection with many correlated patterns. Our input packets were a real trace from the MIT DARPA project [MIT].

Since the width is the most influential factor on the solution's cost and performance, we examine our simulation with several TCAM widths. Choosing the right width is a very crucial decision when importing this solution to the industry.

We compare our results with Lakshman, Yu and Katz [FKL04] results since both algorithms are TCAM-based. We expected that our main improvement will be the TCAM accesses. At Lakshman, Yu and Katz algorithm, they access the TCAM for each byte in the text i.e. the shift value is constant and equals 1. As we analyzed it at 4.2.1, we expect to get an average shift value of around $w/2$, where $w$ is the TCAM width.

## 1.1 Results on ClamAV Pattern Set

Version 0.82 of ClamAV has 26987 simple patterns. The average pattern length is 124.1. Figure 4.2 shows the TCAM size required in order to accommodate all the patterns with different $w$ settings. As we can clearly see, as $w$ increases, TCAM space requirement increases too (even without the shifted pattern) since short or suffix patterns are padded. Figure 4.3 presents the amount of signatures that can be covered by choosing a specific window size. This figure shows that in order to cover most of the patterns in one TCAM hit we need a wide TCAM which directly results in large TCAM memory size and a very expensive solution. Thus, we stay with a reasonable TCAM width and divide the patterns into several pieces. We can also use the *long-patterns optimization* described at 4.1.2 to eliminate cases in which the algorithm proceeds into the packet in order to check the sub patterns and finally discover that there is no match.

### 1.1.1 Test Results

ClamAV is the easier case, there is a clear and direct proportion between the TCAM width and the performance. Since the average patterns length is high and the probability to match $w$ bytes of any of the long patterns without having a complete match is pretty low, the shift average is directly influenced by the TCAM width and the attacks in the packets set. Thus, we expect to have a higher shift average at ClamAV set.

Our simulation shows that when $w = 8$, we get a shift average of 7.53, when $w = 16$, the shift average increases to 15.73 and when $w = 32$, the shift average is 31.75. These results emphasize the benefit of having the shifted patterns in the TCAM. Let $d$ denote the deceleration caused by the SRAM accesses, with ClamAV data set and with TCAM width of 32, we can get a speed rate of around $\frac{63.5}{1+d}$ Gbps[1]. $d$ is influenced by two factors, one is the number of SRAM accesses, $S_n$, for each TCAM access and the second is the speed of each SRAM access.

We define a *memory ratio*, $M_r$, to be the relation between the TCAM access speed and

[1] the 1 at the denominator denotes the TCAM accesses time

SRAM access speed. For example, a memory ratio of 1 means that SRAM access speed is equal to TCAM access speed. The value 0.2 means that memory access speed is 5 times faster than TCAM access speed. With this notations, $d = S_n \times M_r$. Our results show that at 60% of the packets there are no attacks, at this case, $S_n = 1$ (one SRAM access for each TCAM access) resulting $d = M_r$. Since the values of the memory ratio are in the range of 0.2 to 1, the throughput average is in the range of $\frac{63.5}{1+0.2} = 52.9$ Gbps to $\frac{63.5}{1+1} = 31.75$. At ClamAV, at the average case, $S_n = 1$, so this range holds at this case as well.

## 1.2 Results on Snort Pattern Set

Snort's case is more complicated, Snort's patterns include a lot of short patterns (of size $\leq 4$). In most of the cases these patterns are part of correlated rules. Our algorithm matches these patterns, resulting in a very low average shift (around 2 independently of the TCAM width). Our simulation shows that excluding these patterns from the TCAM results in a very high shift average (more than $w/2$).

There are three optional solutions, one, to separate the short patterns and build two different mechanisms for short and long patterns. The short patterns mechanism can use a trie as described at 4.4. The problem with this solution is the need to synchronize between the two mechanisms. The second solution is to start look for the long patterns and just in a case of a match looking for the short ones. The drawback here is the need to search the packet backwards which will significantly slow down the algorithm rate. The last solution is to try to eliminate the TCAM hits by adding more information at each TCAM entry besides the pattern itself, as explained at 4.1.4.

We can add information such as protocol, sub-protocol, port etc. We created a hash function $h$ that it's input is the additional data $D$ and its output is a key $k$, $h(D) = k$. We add $k$ to each entry in the TCAM. By having a hash function (instead of just adding the additional data "as-is" to the TCAM), we reduce the length of the key we add to each TCAM entry. The assumption under this solution is the fact that most of the short patterns relate to specific flows and do not have to be checked for any received packet. We tried to

add the protocol, sub-protocol and the source and destination ports and we get extremely better results. We added a bitmap of one byte for the main protocols and sub-protocols. The port's key was calculated as follows: At the first time a pattern appears with a port, this port is the additional key to the TCAM entry, from now on, whenever the same pattern appears with a different port, we represent the port key as the range of the lowest port and the highest port.

For example, if a patterns appears with the ports: 2048, 3400 and 7777, the port key will be: 11???0??0?00?. The length of the port key is 2 bytes (for each port: source and destination).

Note, that there can be $D_1$ and $D_2$ such that $h(D_1) = h(D_2)$. Therefore, after matching the TCAM, the algorithm must check the flow's data. In order to be able to verify the flow's data, we save this data in each entry of the *rules table*. After getting a match, the algorithm gets the matched pattern's rule entry and verifies that the flow data of the matched pattern is identical to the flow data of the received packet.

With the hash extension we can get much better results as shown in figures 1.1. We can see that for $w = 16$, the average per packet is closely equal to the total average. Thus, the standard deviation is relatively low. This graph is pretty close to the gaussian distribution. The implication of this result is that the line speed is stable and it stands on around 12 Gbps (2 Gbps which is the basic TCAM rate multiplied by the total shift average.). Regarding the other graphs, we can see that the standard deviation is between 6 (where $w = 32$) to 27 (where $w = 128$). The high standard deviation implies that the line speed is unstable and it is vastly varies.

### 1.2.1 Shift Average Results

Table 1.1 presents the TCAM memory size requirement and the shift average value for each TCAM width. We can see that even with the addition of the flow data to the TCAM key, the shift average grows slowly when the TCAM width is greater than 24. The benefit of increasing the TCAM width is not as large as we thought. Initially a glance at figure 1.2 shows that as long as $w \leq 128$ we gain a raise in the shift value, but a closer look shows

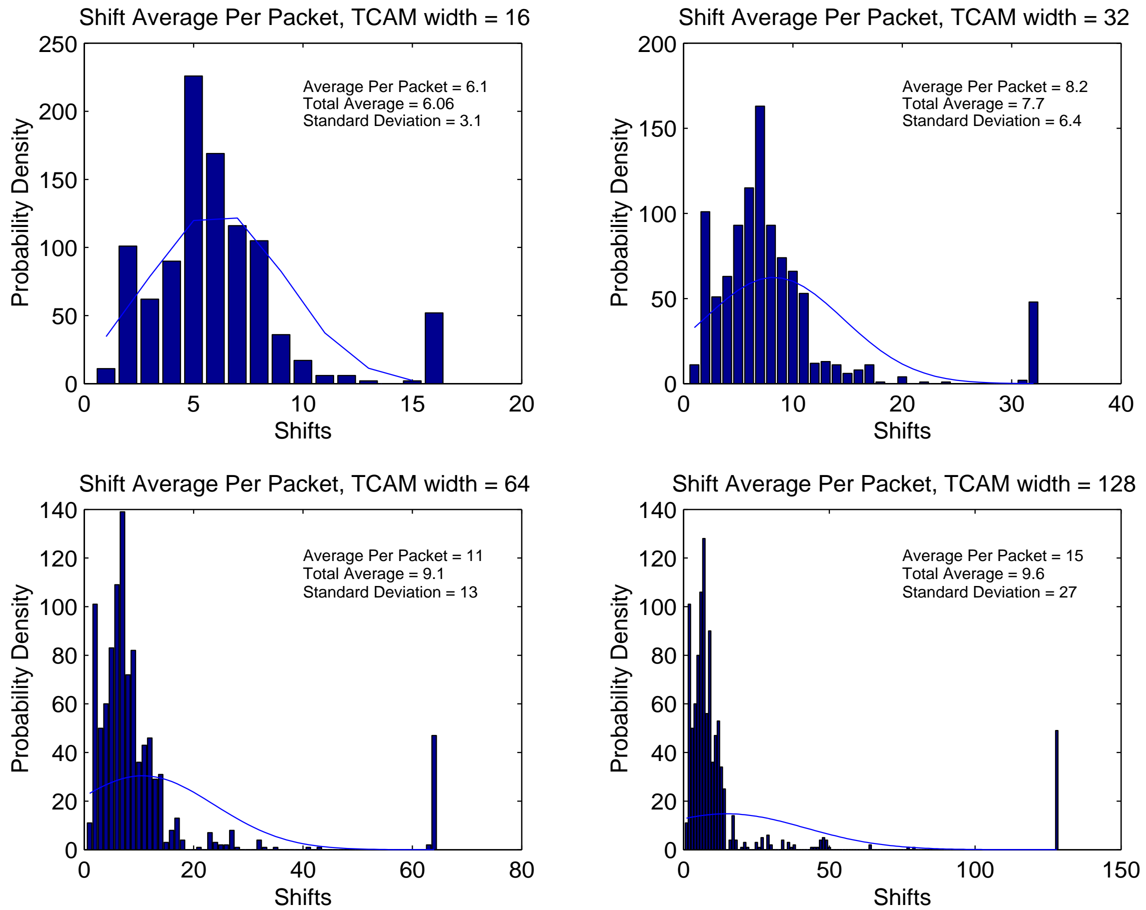



Figure 1.1: Part 3: TCAM Size Requirement

| TCAM width Impact on Shift Average with Snort Pattern Set | | |
|---|---|---|
| *TCAM width* | *TCAM memory size (KB)* | *Shift average value* |
| 4 | 26 | 2.62 |
| 8 | 99 | 4.42 |
| 16 | 443 | 6.06 |
| 24 | 912 | 7.41 |
| 32 | 1990 | 7.67 |
| 48 | 4760 | 8.31 |
| 64 | 8735 | 9.11 |
| 96 | 20273 | 9.17 |
| 128 | 36591 | 9.57 |

Table 1.1: Part 3: TCAM width Impact on Shift Average

that the tradeoff is not so simple. While $w \leq 24$, the shift average increases sharply but from $w = 24$ to $w = 128$, the benefit becomes moderated. In contrast to the moderated growth of the shift average, the TCAM memory size grows exponentially. Therefore, with the Snort pattern set the advisable TCAM width is 24.

The algorithm presented at Lakshman, Yu and Katz [FKL04] offers scan rate of 2 Gbps in the best case. Their algorithm achieves this rate when the scan ration is 1, meaning, there are no memory accesses at all.

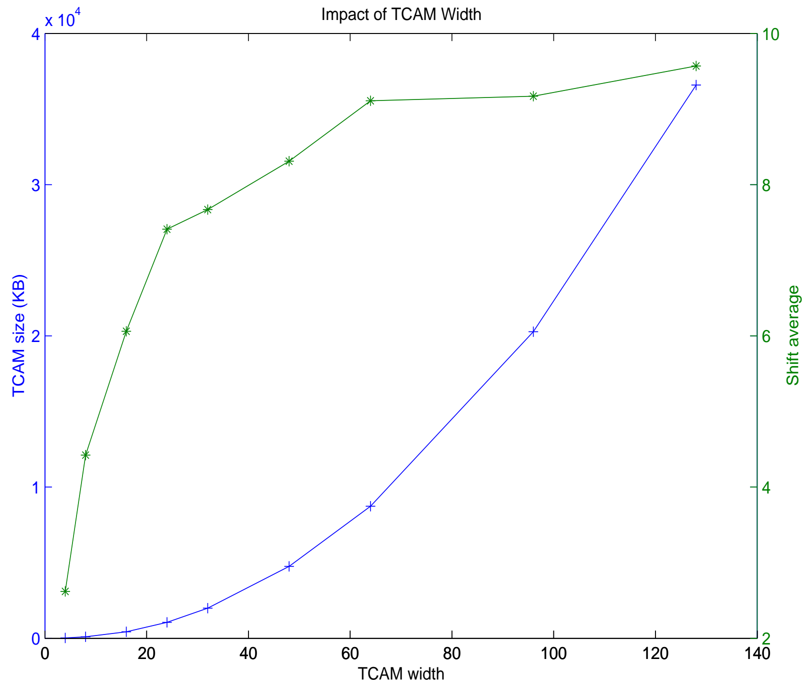



Figure 1.2: Part 3: TCAM Size vs. Shift Average

### 1.2.2 Scanning Time Results

The most influential factors on the scanning time are the TCAM accesses and memory access. In contrary to the algorithm presented at [FKL04] where a TCAM hit does not necessarily require a memory hit, our algorithm hits the memory at any TCAM hit in order to get the shift value.

Our main advantage over Lakshman, Yu and Katz algorithm is that our algorithm hits the TCAM significantly less times. Even though our algorithm hits the memory for each TCAM hit, the total number of memory hits is also significantly less than the one at Lakshman, Yu and Katz algorithm. Figure 1.3 shows our improvement over Lakshman, Yu and Katz algorithm in the amount of memory hits. Figure 1.4 presents the number of TCAM hits our algorithm required for different packets length. At TCAM accesses factor, our algorithm is significantly better than Lakshman, Yu and Katz algorithm.

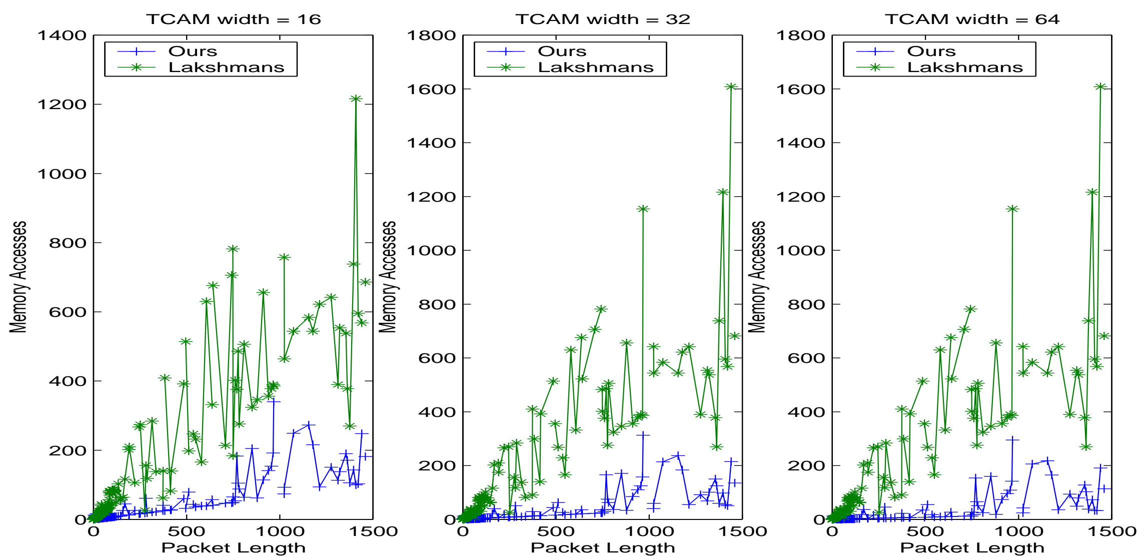



Figure 1.3: Part 3: Memory Accesses

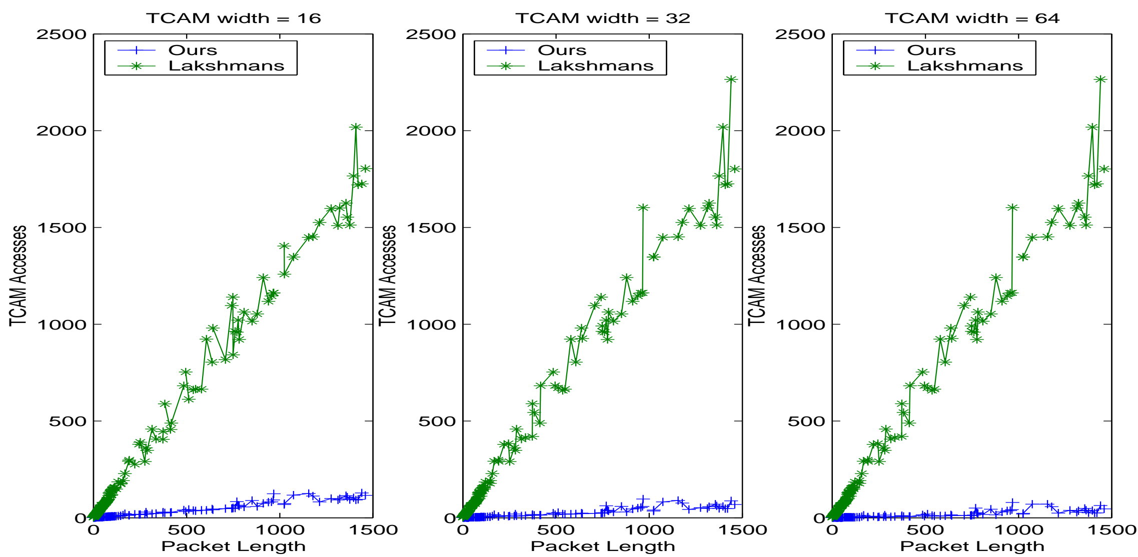



Figure 1.4: Part 3: TCAM Accesses

| TCAM accsees vs. SRAM accesses | | |
|---|---|---|
| *TCAM width* | *TCAM accesses* | *SRAM accesses* |
| 16 | 32048 | 50286 |
| 24 | 24156 | 45288 |
| 32 | 18983 | 37259 |
| 64 | 12765 | 31457 |
| 128 | 10218 | 28946 |

Table 1.2: Part 3: Sum of TCAM accesses vs. Sum of SRAM accesses

Running the simulation using MIT DARPA trace showed that for TCAM width of 24, 60% of the TCAM hits result in a shift value greater than 0. Since the RTCAM algorithm accesses the SRAM every TCAM lookup, the scan ratio (as defined in [FKL04]) is 2. The simulation results for this TCAM width show that the average shift value is 7.4. Since SRAM memory access speed is usually faster than the TCAM speed, in current memory technologies, the memory ratio is 0.2. Taking this figure, we can achieve a throughput of $\frac{2\times 7.4}{1+0.2} = 12.35$ Gbps. Table 1.2 shows that the relation between SRAM accesses and TCAM access is in the range of 1.5 to 2.8, thus, even at the average case (where $w = 24$) of all packets, our algorithm achieves a throughput of $\frac{2\times 7.4}{1+1.87\times 0.2} = 10.77$ Gbps.

Figure 1.5 shows the effect of different memory ratios on the achieved throughput. Note that even when the memory ratio is 1, we still get an average throughput of 7.4 Gbps.

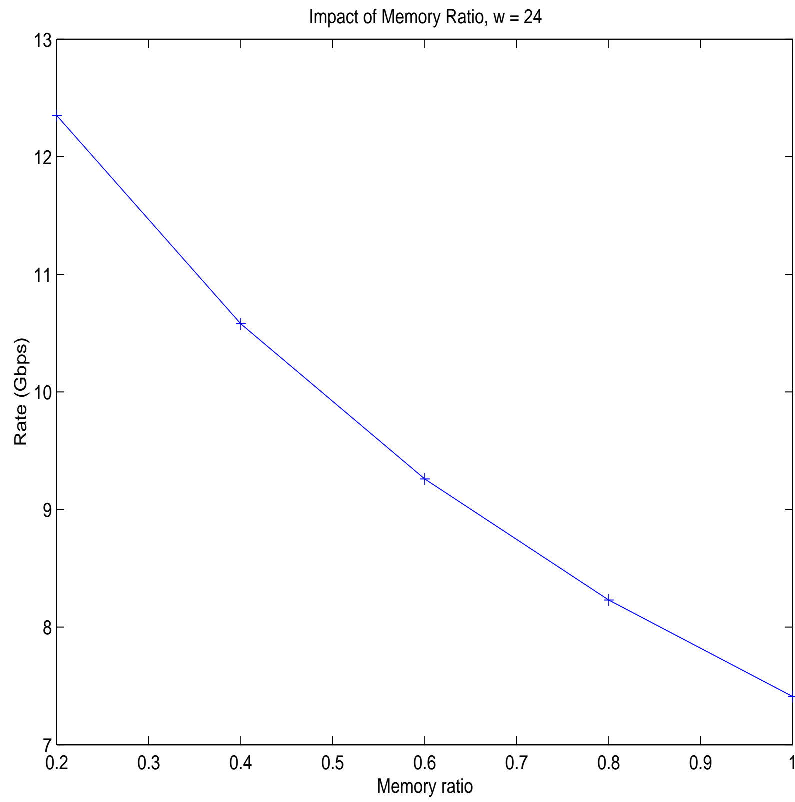



Figure 1.5: Part 3: Effect of Memory Ratio on Scan Rate

# Chapter 2

# Conclusions

In this thesis we have designed and implemented a NIPS system that is based on a novel pattern matching algorithm, called RTCAM. We have also designed a fully dynamic and configurable classification engine that uses the same RTCAM module as a building block. We have shown that our solution is adequate for NIPS device implementors as it achieves line-speed rates. Specifically, for about 60% of real network traffic, an average line-speed of 12.35 Gbps can be achieved[1]. This NIPS has several major advantages over existing NIPS devices. First, the achieved line-rate speed is of several orders of magnitude faster than related works and we gain it without loosing the algorithm accuracy. Second, as opposed to other solutions, our system is fully compatible with Snort's rules syntax. This is an important advantage as Snort is becoming the de facto standard for intrusion detection and prevention systems. We have created a simple tool that is capable of importing Snort's database by a mouse click.

[1] Using a RTCAM of 912KB.

## 2.1 Future Work

We have already initiated a lab project for prototyping the hash-based algorithm using FPGA. We would like to be able to compare the performance of the FPGA and the RT-CAM solutions. Another important research area is cross packets inspection. The intuition says that the amount of support for this problem is proportional to the amount of memory available on the intrusion detection device. Still, we would like to explore the various possibilities for dealing with this problem and to provide some experimental results. Last, we plan to design an integrated RTCAM circuit that will automatically compare the provided key with the rotations of each pattern in the TCAM (using dedicated circuitry). This will significantly reduce the amount of TCAM memory needed by the algorithm.

# Appendix A

# Light Bulb Example

Consider the state machine for a light bulb protocol presented in figure A.1 (assume that the *protocolID* is 100). The state machine can be expressed using two PDTEs as shown in table A.1.

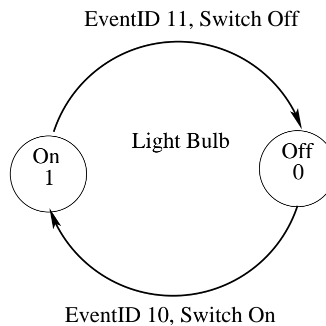



Figure A.1: Sample Light Bulb Finite State Machine

| Protocol Description Table | | | | |
|---|---|---|---|---|
| *ProtocolID* | *currentState* | *eventID* | *nextState* | *preCondition* |
| 100 | 0 | 10 | 1 | $\Phi$ |
| 100 | 1 | 11 | 0 | 10 |

Table A.1: Light-Bulb PDT

The *preCondition* field for the “switch-off” was added (not mandatory) in order to enforce that we only turn-off the light bulb if we received a previous event of “switch-on”. If the *preCondition* is omitted and a “switch-off” event is received when the light is off, then the default action is triggered which is probably just ignore. The *protocolID* is used for indexing the table thus avoiding irrelevant protocols state machine.

# Bibliography


[AC75] A. V. Aho and M. J. Corasick. *Efficient String Matching*. *Communications of the ACM*, 18(6):333–340, June 1975.

[ACS03] I. Arsovski, T. Chandler, and A. Sheikholeslami. *A Ternary Content-Addressable Memory (TCAM) Based on 4T Static Storage and Including a Current-Race Sensing Scheme*. *IEEE Journal of Solid-State Circuits*, 38(1), January 2003.

[BM77] R. S. Boyer and J. S. Moore. *A Fast String Searching Algorithm*. 20(10):762–772, October 1977.

[CCG+99] M. Crochemore, A. Czumaj, L. Gasieniec, T. Lecroq, W. Plandowski, and W. Rytter. *Fast Practical Multi-Pattern Matching*. *Inf. Process. Lett.*, 71(3-4):107–113, September 1999.

[Cla] ClamAV Anti-Virus. http://www.clamav.net/.

[CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[CW79] B. Commentz-Walter. *A String Matching Algorithm Fast on the Average*. *In Proc. 6th Int. Coll. on Automata, Languages and Programming (ICALP'79), LNCS*, 71:118–132, July 1979.

[DKSL03] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood. *Deep Packet Inspection Using Parallel Bloom Filters*. *Symposium on High Performance Interconnects (HotI), Stanford, CA, USA*, pages 44–51, August 2003.

[FKL04] Y. Fang, R. H. Katz, and T. V. Lakshman. *Gigabit Rate Packet Pattern-Matching Using TCAM*. In *ICNP*, 2004.

[For] Fortigate-800 Appliance. http://www.fortinet.com/.

[KMP77] D. E. Knuth, J.H. Morris, and V. R. Pratt. *Fast Pattern Matching in Strings*. *SIAM Journal of Computing*, 6(2):323–350, June 1977.

[KR81] R. M. Karp and M. O. Rabin. *Efficient Randomized Pattern-Matching Algorithms*. Technical report TR-31-81, Harvard University, Cambridge, MA, USA, December 1981.

[LHCK04] R. T. Liu, N. F. Huang, C. H. Chen, and C. N. Kao. *A Fast String-Matching Algorithm for Network Processor-Based Intrusion Detection System*. *Trans. on Embedded Computing Sys.*, 3(3):614–633, August 2004.

[MDWZ04] Y. Miretskiy, A. Das, C. P. Wright, and E. Zadok. *Avfs: An On-Access Anti-Virus File System*. In *Proceedings of the 13th USENIX Security Symposium (Security 2004)*, pages 73–88, San Diego, CA, August 2004.

[MIT] MIT DARPA Project Data Set. http://www.ll.mit.edu/IST/ideval/index.html.

[MN98] D. R. Musser and G. V. Nishanov. *A Fast Generic Sequence Matching Algorithm*. Technical report, Computer Science Dept., Rensselaer Pollytechnic Institute, Troy, NY, March 1998.

[NR03] M. Norton and D. Roelker. *Snort 2.0: High Performance Multi-Rule Inspection Engine*. http://www.cs.cuc.edu/ droelker/docs/Multi-Rule-Inspection.pdf, April 2003.

[Sec] SecureSoft Absolute IPS NP5G, NP10G. http://www.securesoft.com.

[Sno] Snort Project. http://www.snort.org/.

[TKD03] D. E. Taylor, P. Krishnamurthy, and S. Dharmapurikar. *Longest Prefix Matching Using Bloom Filters*. *ACM SIGCOMM*, 03:201–212, August 2003.

[Van] Vandyke software-related survey. http://www.vandyke.com/.

[WM91] S. Wu and U. Manber. *Fast Text Searching with Errors*. Technical Report TR-91-11, University of Arizona, Department of Computer Science, June 1991.

[WM92] S. Wu and U. Manber. *Agrep – A Fast Approximate Pattern-Matching Tool*. In *Proceedings USENIX Winter 1992 Technical Conference*, pages 153–162, San Francisco, CA, January 1992.

[WM93] S. Wu and U. Manber. *A fast Algorithm for Multi-Pattern Searching*. Technical Report TR-94-17, Department of Computer Science, University of Arizona, May 1993.

[WZN92] B. W. Watson, G. Zwaan, and Mrs F. Van Neerven. *A Taxonomy of Keyword Pattern Matching Algorithms*. Technical Report 27, Faculty of Computing Science, Eindhoven University of Technology, The Netherlands, January 1992.