HIGH PERFORMANCE STRIDE-BASED NETWORK PAYLOAD INSPECTION

By

Xiaofei Wang

Submitted in partial fulfilment of the requirements

for the Degree of Doctor of Philosophy

Supervisors:

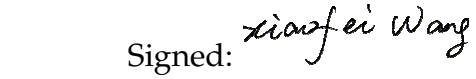
DR. XIAOJUN WANG AND PROF. BIN LIU

SCHOOL OF ELECTRONIC ENGINEERING

DUBLIN CITY UNIVERSITY

August 27, 2012

I hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of Doctor of Philosophy is entirely my own work, and that I have exercised reasonable care to ensure that the work is original, and does not to the best of my knowledge breach any law of copyright, and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

Signed: 

ID number: 58124705

Date: AUGUST 27, 2012

Abstract

There are two main drivers for network payload inspection: malicious data, attacks, virus detection in *Network Intrusion Detection System* (*NIDS*) and content detection in *Data Leakage Prevention System* (*DLPS*) or *Copyright Infringement Detection System* (*CIDS*).

Network attacks are getting more and more prevalent. Traditional network firewalls can only check the packet header, but fail to detect attacks hidden in the packet payload. Therefore, the NIDS with Deep Packet Inspection (DPI) function has been developed and widely deployed. By checking each byte of a packet against the pattern set, which is called pattern matching, NIDS is able to detect the attack codes hidden in the payload. The pattern set is usually organized as a Deterministic Finite Automata (DFA). The processing time of DFA is proportional to the length of the input string, but the memory cost of a DFA is quite large. Meanwhile, the link bandwidth and the traffic of the Internet are rapidly increasing, the size of the attack signature database is also growing larger and larger due to the diversification of the attacks. Consequently, there is a strong demand for high performance and low storage cost NIDS. Traditional software-based and hardware-based pattern matching algorithms are have difficulty satisfying the processing speed requirement, thus high performance network payload inspection methods are needed to enable deep packet inspection at line rate. In this thesis, Stride Finite Automata (StriFA), a novel finite automata family to accelerate both string matching and regular expression matching, is presented. Compared with the conventional finite automata, which scan the entire traffic stream to locate malicious information, the StriFA only needs to scan samples of the traffic stream to find the suspicious information, thus increasing the matching speed and reducing memory requirements.

Technologies such as instant messaging software (Skype, MSN) or BitTorrent file sharing methods, allow convenient sharing of information between managers, employees, customers, and partners. This, however, leads to two kinds of major security risks when exchanging data between different people: firstly, leakage of sensitive data from a company and, secondly, distribution of copyright infringing products in Peer to Peer (P2P) networks. Traditional DFA-based DPI solutions cannot be used for inspection of file distribution in P2P networks due to

the potential out-of-order manner of the data delivery. To address this problem, a hybrid finite automaton called Skip-Stride-Neighbor Finite Automaton (S^2N-FA) is proposed to solve this problem. It combines benefits of the following three structures:

1) Skip-FA, which is used to solve the out-of-order data scanning problem;

2) Stride-DFA, which is introduced to reduce the memory usage of Skip-FA;

3) Neighbor-DFA which is based on the characteristics of Stride-DFA to get a low false positive rate at the additional cost of a small increase in memory consumption.

Keywords: Network Intrusion Detection Systems, Deep Packet Inspection, Non-deterministic Finite Automaton, Deterministic Finite Automaton, Pattern Matching, File detection

Acknowledgements

I would like to thank the many people who have contributed to this thesis. Without them, this work would not have been possible.

First and foremost I want to thank my supervisors, Dr. Xiaojun Wang and Prof. Bin Liu, who have supported me throughout my thesis with their patience and knowledge. Throughout my years in university, they provided encouragement, sound advice, good teaching, and lots of good ideas.

I am greatly indebted to Dr. Chengchen Hu and Dr. Yang Xu. Your valuable contributions will always be treasured.

I would also like to thank Dr. Junchen Jiang, Yi Tang and Wei Lin for their assistance and guidance in getting my graduate career started on the right foot and providing me with the foundation for my research work.

Moreover, I would like to offer my sincere gratitude to the people (Huichen Dai, Ting Zhang and Keqiang He) in Tsinghua University and Brendan Cronin in Dublin City University who spend their precious time in reading this thesis.

Finally, and most importantly, I would like to thank my wife Jieyan. Her support, encouragement, quiet patience and unwavering love were undeniably the bedrock that enabled me to reach this point.

List of Acronyms

Acronyms

NIDS Network Intrusion Detection System

NFA Nondeterministic Finite Automata

DFA Deterministic Finite Automata

regex regular expression

PE file Portable Executable file

FSM finite state machine

LBM Length-based Matching

StriFA Stride-based Deterministic Finite Automata

DFS Depth First Search

ASIC Application-Specific Integrated Circuit

CAM Content Addressable Memories

QoS Quality of Service

DDoS Distributed Denial of Service

FPGA Field Programmable Gate Array

ASCII American Standard Code for Information Interchange

TCP Transmission Control Protocol

UDP User Datagram Protocol

SYN Synchronize

IP Internet Protocol

ISP Internet Service Providers

BL Bloom Filter

SL Stride Length

List of Publications

Paper:

- **Xiaofei Wang**, Junchen Jiang, Yi Tang, Yi Wang, Bin Liu and Xiaojun Wang, "StriD^{2}FA: Scalable Regular Expression Matching for Deep Packet Inspection"in *IEEE International Conference on Communications (ICC 2011)*, Japan, 2011.

- **Xiaofei Wang**, Chengchen Hu, Keqiang He, Junchen Jiang, Bin Liu and Xiaojun Wang, "Measurements on Movie Distribution Behavior in Peer-to-Peer Networks" in *IFIP/IEEE International Symposium on Integrated Network Management (IM2011)*, Dublin, 2011.

- **Xiaofei Wang**, Chengchen Hu, Yachao Zhou, Xiaojun Wang and Bin Liu, "Efficient Log System for Distribution Behavior Analysis in Peer-to-Peer Networks" in *CIICT 2010*, Wuhan, 2010.

- **Xiaofei Wang**, Junchen Jiang, Yi Tang, Bin Liu and Xiaojun Wang, "Extraction of Fingerprint from Regular Expression for Efficient Prefiltering" in *ICCTA 2009*, Beijing, 2009.

- **Xiaofei Wang**, Wei Lin, Yi Tang, Ashwin Lall, Bin Liu and Xiaojun Wang, "A Scalable Bloom Filter based Prefilter and Hardware-oriented Predispatcher" in *INFOCOM 2009 Student Workshop*, Brazil, 20 April 2009.

- Junchen Jiang, **Xiaofei Wang**, Keqiang He and Bin Liu, "Parallel Architecture for High Throughput DFA-Based Deep Packet Inspection" in *IEEE International Conference on Communications (ICC 2010)*, South Africa, 23 May 2010.

- Yi Tang, Junchen Jiang, **Xiaofei Wang**, Bin Liu and Yang Xu, "Independent Parallel Compact Finite Automatons for Accelerating Multi-String Matching" in GLOBECOM 2010.

- Wei Lin, **Xiaofei Wang**, Yi Tang, Derek Pao and Bin Liu, "Compact DFA Structure for Multiple Regular Expression Matching" in *IEEE International Conference on Communications (ICC 2009)*, June 14-18, Dresden, Germany.

- Wei Lin, **Xiaofei Wang**, YaXuan Qi, Derek Pao and Bin Liu, "High-Speed Memory-Efficient Network Intrusion Detection System" in *INFOCOM 2009 Student Workshop*, Brazil, 20 April 2009.

- Junchen Jiang, **Xiaofei Wang**, Yi Tang and Bin Liu, "SPC-DFA: A novel technique for multi-string matching acceleration" in *ANCS Poster 2009*, USA, 19 October 2009.

- Gao xia, **Xiaofei Wang** and Bin Liu, "SRD-DFA: Achieving Sub-Rule Distinguishing with Extended DFA Structure" in *SCC'09*, Sichuan, 2009.

SCI indexed Journals:

- **Xiaofei Wang**, Chengchen Hu, Yi Tang, Ting Zhang, Chunming Wu, Bin Liu and Xiaojun Wang, "Parallel Length-based Matching Architecture for High Throughput Multi-Pattern Matching" in *Chinese Journal of Electronics* 2011. (Accepted)

- Chengchen Hu, **Xiaofei Wang**, Xiaojun Wang, Keqiang He and Bin Liu, "Measurements on Movie Distribution Behavior in Peer-to-Peer Networks" in *IET Communications* 2011. (Accepted)

- Yi TANG, Junchen JIANG, **Xiaofei Wang**, Chengchen HU, Bin LIU and Zhijia CHEN, "Parallel DFA Architecture for Ultra High Throughput DFA-Based Pattern Matching" in IEICE TRANSACTIONS on Information and Systems. VOL.E93CD, NO.12 December 2010.

CONTENTS

List of Figures **iv**

List of Tables **vii**

List of algorithms **viii**

1 Introduction **1**

1.1 Research Background 1

1.1.1 Intrusion Examples 2

1.1.2 NIDS 3

1.1.3 DPI technology 5

1.1.4 String and Regular Expression Matching 6

1.1.5 NFA and DFA 9

1.1.6 CIDS and DLPS 11

1.2 Problem Statement and Work Description 13

1.3 Thesis Organization 15

2 Pattern Matching Algorithms **16**

2.1 Single-character based String Matching Algorithms 16

2.1.1 Aho-Corasick algorithm 17

2.2 Multi-Character Based String Matching Algorithms 20

2.2.1 Transition-distributed Parallel DFA 21

2.2.2 Bloom Filter-based String Matching Algorithm 22

2.2.3 Variable Stride DFA 28

2.3 Regular Expression Matching 31

2.3.1 k-DFA 31

2.3.2 Multi-Stride DFA . . . 32
2.3.3 Sampled DFA . . . 33
2.4 Comparison of Multi-character Matching Algorithms . . . 35

3 StriDFA: Stride DFA for String Matching 37
3.1 Introduction . . . 38
3.2 Motivation . . . 40
3.2.1 Traditional DFA in Multi-string Matching . . . 41
3.2.2 Stride-based DFA . . . 41
3.2.3 Proof of Correctness . . . 44
3.3 Architecture of StriDFA Matching Engine . . . 45
3.4 Benefits of StriDFA . . . 46
3.4.1 Increased Matching Speed . . . 46
3.4.2 Small Memory Requirement . . . 47
3.4.3 Easily Implemented on Existing Platforms . . . 47
3.5 Challenges . . . 48
3.5.1 Tag Selection . . . 48
3.5.2 Potentially Infinite Alphabet Set . . . 48
3.5.3 Rate of False Alarm . . . 48
3.5.4 Regular Expression Support . . . 49
3.6 Limit Alphabet Set by a Window . . . 49
3.7 Tag Selection Approach . . . 50
3.7.1 How tags "cover" a pattern . . . 52
3.7.2 Greedy algorithm for tag selection . . . 53
3.8 Verification module . . . 54
3.9 Evaluation . . . 55
3.9.1 Experiment setup . . . 55
3.9.2 Memory consumption and speedup . . . 56
3.9.3 Filter rate and false alarm rate . . . 56
3.10 Conclusion . . . 58

4 StriNFA and StriDFA for Regular Expression Matching 59
4.1 Introduction . . . 60
4.2 Problem Statement . . . 63
4.3 Stride Finite Automaton . . . 63
4.3.1 Building StriFA by DFA-based method . . . 63
4.3.2 StriNFA to StriDFA . . . 66
4.3.3 Correctness Proof . . . 68

4.4 Stride Finite Automaton 70
4.4.1 Building StriNFA by NFA-based method 70
4.4.2 StriFA-based Matching Architecture 74
4.5 Analysis and Optimization 74
4.5.1 Stride-Neighbor FA 75
4.5.2 Performance of StriFA 76
4.5.3 Conversion Complexity of StriFA 77
4.6 Evaluation 77
4.6.1 Trace characteristics 78
4.6.2 Throughput 79
4.6.3 Memory consumption 80
4.6.4 Speedup 84
4.6.5 Filter rate and false alarm rate 85
4.6.6 Performance on real traces 86
4.7 Conclusion 86

5 S^{2}N-FA: A Hybrid Finite Automaton for File Detection 87
5.1 Introduction 88
5.2 Related Work 90
5.2.1 Fingerprint Extraction 91
5.2.2 File Detection in CIDS and DLPS 91
5.3 Problem Statement 92
5.3.1 Out-of-Order Data Transmission in P2P network 92
5.3.2 Problem Formulation 94
5.4 Building Skip Finite Automata 95
5.4.1 A Basic Model: Skip-FA 95
5.4.2 Problem of Memory Usage 99
5.5 Skip-stride Finite Automaton 99
5.5.1 Building Skip-Stride Finite Automaton 100
5.5.2 Problem of False Positive 101
5.6 Building Skip-Stride-Neighbor Finite Automaton 102
5.6.1 Skip-Stride-Neighbor Finite Automaton 102
5.6.2 Analysis of Stride-Neighbor DFA 104
5.7 Experimental Results 105
5.7.1 Experiment Setup and Test Sets 105
5.7.2 Memory Usage 106
5.7.3 Matching Speed 106
5.7.4 Longest Overlap Percentage 108

5.8 Conclusion 110

6 Conclusions and Future Work **111**
6.1 Summary of thesis work 111
6.1.1 StriFA 112
6.1.2 S^2N-FA 112
6.2 Future work 113
6.2.1 Tag selection 113
6.2.2 Hardware implementation 114

Appendix **115**
I Convert Regex to NFA 115
II Restructure Traditional DFA to StriDFA 118
III Regular Expression Syntax 122

References **123**

LIST OF FIGURES

1.1 Network Intrusion Detection System. 4
1.2 Equivalence of regular expression, NFA and DFA. 7
1.3 NFA and DFA of regex `a.{n}bc`. 10

2.1 AC algorithm with pattern set {he, her, him, his}. 18
2.2 Extracting bit-level parallelism from the Aho-Corasick algorithm by splitting the state machine into 8 parallel state machines. 20
2.3 Transfer character set to the token sequences. 21
2.4 An example of TDP-DFA with w=3. 22
2.5 Deploy TDP-DFAs for detection in parallel. 22
2.6 Introduction of Bloom filter. 23
2.7 Jump-ahead Aho-Corasick NFA (JACK-NFA) 26
2.8 Winnowing sample text. The gray part is covered by the selected fingerprints. 28
2.9 Segment Pattern into Head/Core/Tail Blocks (k = 2, w = 4). 29
2.10 An example of a VS-DFA. 30
2.11 (a) NFA accepting (1) `ab.*cd` and (2) `ac+e`; (b) corresponding 2-NFA. 31
2.12 MS-DFA for patterns {"AABCDZGHIJ3A2B1C", "ABCDEFGHIJSTUVWXYZ", "0123456789Z", "6789KLMNOPYZABC"}. 33
2.13 Example: the regex `ab.*cd` is sampled (with θ = 2) to [ab].*[cd] and matched against a text of 16 bytes. 34
2.14 Traditional DFA and reverse DFA for pattern `ab*|ba`. 35

3.1 Traditional DFA of patterns "`reference`" and "`replacement`" . 41
3.2 Use tag to convert input stream into SL stream with tag '`e`'. 42

3.3 Convert patterns to the corresponding StriDFA. 43
3.4 StriDFA of "reference" and "replacement" 44
3.5 The overall structure of StriDFA. 45
3.6 The stride length stream with window w=5. 49
3.7 Frequency of appearance for each characters in Snort and ClamAV rule sets. 51
3.8 Overall speedup and memory usage of the StriDFA (P_1 and P_2) with different window sizes. 57
3.9 Overall speedup and memory usage of P_3 and P_4 with different window sizes. 57
3.10 Filter rate and false positive rate of the StriDFA (P_1 and P_2) with different window sizes. 58

4.1 Flow chart describing the steps to convert a regex to a StriFA via DFA. 63
4.2 Traditional NFA of regex .*abba.{2}caca. 64
4.3 Traditional DFA of regex .*abba.{2}caca and its corresponding Tag decision FA. 65
4.4 StriNFA of the regex .*abba.{2}caca before renumbering. 68
4.5 StriNFA and StriDFA of regex .*abba.{2}caca with tag = 'a' and w = 3. (The transitions back to state 1, 2 and 3 of the corresponding StriDFA are partly ignored for simplicity). 69
4.6 Flow chart represents how to convert regexes to StriNFA/StriDFA via NFA directly. 71
4.7 Traditional NFA and StriNFA of .*abba.{2}caca 71
4.8 Explanation of recursive steps. 72
4.9 StriNFA of the regex .*abba.{2}caca with tag ='a' and w = 3 before renumbering. 74
4.10 Architecture of StriFA-based multi-regex matching engine. 75
4.11 Stride-Neighbor DFA for P_1="reference" and P_2="replacement" with tag='e' and w=5. 76
4.12 Throughput of three different traces with different finite automaton. 80
4.13 Overall speedup and memory usage of StriFA with different window sizes. 84
4.14 Efficiency factor of StriDFA, Stride-Neighbor DFA of different window sizes. 85

5.1 CIDS and DLPS in the network. 89
5.2 The difference between LO matching (left) and LP matching (right). 93

5.3 The Skip-FA of signature $\mathcal{F}$. 95
5.4 Skip-Stride Finite Automaton of $\mathcal{F}$ with window w = 5. 101
5.5 Skip-Stride-Neighbor finite automaton with tag='e' and window w=5. 102
5.6 Efficiency factor of Skip-FA, Skip-Stride FA of different window sizes. 104
5.7 Number of Di-trans with different window sizes and Start State Table (SST) numbers with various threshold v. 108
5.8 The LO percentage of Skip-Stride FA and S^{2}N-FA in 20 consecutive packets. 109

A-1 NFA of `.*abba.{2}caca` 115
A-2 Original DFA of `.*abba.{2}caca`. 118
A-3 DFA of `.*abba.{2}caca` after renumbering. 118
A-4 Restructure DFA of `.*abba.{2}caca` by classifying transitions. . 118
A-5 StriNFA of `.*abba.{2}caca`. 119
A-6 StriDFA of `.*abba.{2}caca` after determination. 121
A-7 StriDFA of `.*abba.{2}caca` after renumbering and minimization. 121

LIST OF TABLES

1.1 Recent traffic volumes and growth rates on the global Internet. . . . 2
1.2 Regular expressions syntax. 8

2.1 Transition table of JACK-NFA. 27
2.2 Comparisons of multi-character matching algorithms. 36

3.1 Memory consumption between traditional DFA and StriDFA 47

4.1 An example to show the correctness of StriFA. 68
4.2 Worst case comparisons of NFA, DFA [68], StriNFA and StriDFA. . 77
4.3 Real data samples 79
4.4 Comparison between Traditional NFA/DFA, k-DFA and StriNFA/StriDFA 82

5.1 Starting State Table of the Skip-FA. 97
5.2 Starting State Table of Skip-Stride Finite Automaton. 101
5.3 Starting State Table of Skip-Stride-Neighbor finite automaton. . . . 102
5.4 Basic characteristics of test sets 106

A.1 Detailed regular expression syntax. 122

LIST OF ALGORITHMS

1 Algorithm of SL extraction 50
2 Algorithm of tag selection 53
3 Algorithm of transforming Tag decision FA to StriNFA via DFA . . . 67
4 Algorithm of transforming Tag decision FA to StriNFA via NFA . . . 73
5 Building S^{2}N-FA of signature $\mathcal{F} = c_i, \ldots, c_n$ with window = w 103

CHAPTER 1

Introduction

1.1 Research Background

The Internet, invented in 1965, is probably the most important and revolutionary invention of the last century [1]. Beginning with military purposes, the Internet rapidly evolved into a very complex tool for the purpose of resource sharing and communication among scientists. Now the Internet has become a critical infrastructure for global communication, information sharing and content publishing.

The Minnesota Internet Traffic Studies (MINTS) project analyses a large set of publicly available data sources and concludes that current traffic volume per month is between 7.5 and 12 exabytes[1], while the annual Internet traffic growth rates are in the region of $40 \sim 50$ percent [2]. The recently released Visual Networking Index forecast of Cisco Systems shows global IP traffic of 11 exabytes per month, growing at a compound annual growth rate of 40 percent between 2008 and 2011 [3]. The Japanese Internet has been studied by Kenjiro Cho and his colleagues using aggregated data from many local ISPs and they report seeing

[1] An exabyte is a unit of computer storage equal to one quintillion bytes. The unit symbol for the exabyte is EB. 1 EB = 10^{18} bytes.

Table 1.1: *Recent traffic volumes and growth rates on the global Internet.*

Study Name	Traffic Volume (exabytes/month)	Annual Growth Rate
MINTS	7.5–12	40–50%
Cisco VNI	11	40%
Cho et al.	0.7 (Japanese domestic)	40%
ATLAS	9	45%

around 40 percent growth per annum since 2005 for peak traffic rates at domestic Internet exchange points [4]. The ATLAS Internet Observatory is a collaborative research project from Arbor Networks, the University of Michigan and Merit Network. Believed to be the largest Internet monitoring infrastructure in the world, its results represent the first global traffic engineering study of Internet evolution[5]. Table 1.1 describes all the above statistics [6], from which the average Internet traffic growth rate can be found to be around 40% per annum.

The Internet brings enormously convenient functionality, however, it also creates the possibilities for hackers to steal secret information and protected data.

1.1.1 Intrusion Examples

Calculations based upon a total number of 250,000 unique samples every day collected in recent years, indicate that a new piece of malware is created on average every 1.5 seconds [7].

On May 4, 2000, one mass-mailing worm named *LoveBug* was originally distributed in an email with a subject line "I love you" [8]. The email contained the message "kindly check the attached LOVELETTER from me" and an attached file called LOVE-LETTER-FOR-YOU.TXT.VBS. Once opened, the malicious VBScript contained in the attachment could automatically send to every email address in

the recipient's address book a copy of the message. *LoveBug* went on to affect 45 million computer users worldwide. It was estimated that the so-called *LoveBug* email virus had caused some $10 billion in losses in as many as 20 countries. As a result of the *LoveBug* virus, legislation in the Philippines was changed and some highly effective legislature was established to combat online crime [9].

On August 11, 2003, a computer worm named *Blaster* spread on computers running Microsoft operating systems: Windows XP and Windows 2000 [10]. The worm could spread without users opening attachments by simply spamming itself to large number of random IP addresses. Another version was programmed to start a SYN flood [11] to website *http://windowsupdate.com*, thereby creating a distributed denial of service attack (DDoS) [12]. According to a statistics from Microsoft, over 25 million unique computers were identified as being infected by this worm [13]. So far, damages resulted from the *Blaster* worm are estimated to be at least $525 million. On August 12, 2007, hackers attacked the United Nations' official website and some webpages on the site were breached [14]. RBS WorldPay, the payment-processing arm of the Royal Bank of Scotland, was compromised to raise the amount of funds available on the compromised cards and boost their daily withdrawal limits. In 2010, the hacker swindled about $9.5 million in less than 12 hours [15].

The increasing intrusion attacks not only cause significant productivity and economic losses, but also influence politics and people's daily life.

1.1.2 NIDS

A Network Intrusion Detection System (NIDS) involves the deployment of monitoring devices, or *Sensors*, throughout the network, which capture and analyze the traffic as it traverses the network. The Sensors detect malicious and unauthorized activities in real time and can take action when required. They also detect distributed denial of service (DDos) attacks, worms, and viruses. Packets that do

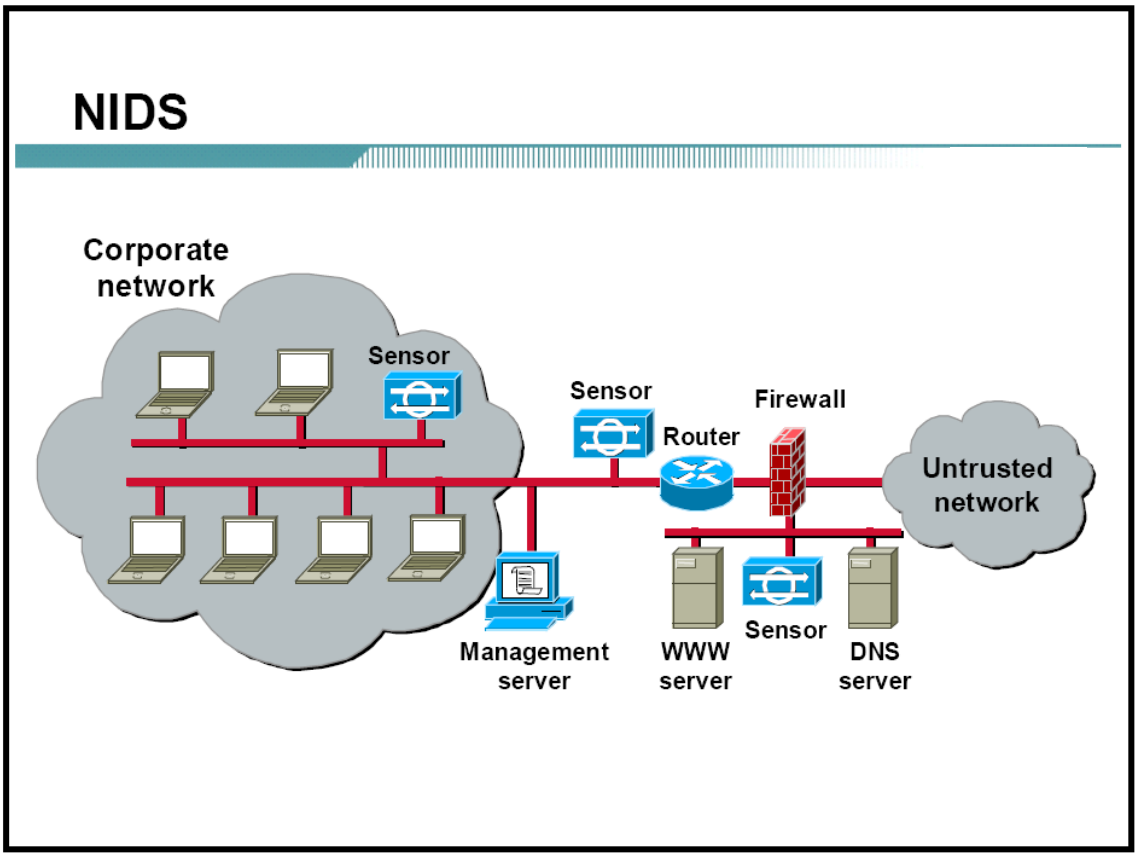


Figure 1.1: *Network Intrusion Detection System.*

not comply with security policies of NIDS will not be allowed into the protected network. Figure 1.1 shows how NIDS devices are configured to detect security threats.

The problem starts from the fact that networks today are becoming faster and faster. The network bandwidth is doubled every six months [16]. As a consequence, DPI and NIDS must have high processing throughput so that they will not be the bottleneck of the network [17]. Another important factor of NIDS is the implementation cost. NIDS can be software programs that run on the General Purpose Processor of the inspection machine or a hardware unit that conducts this specific work. In both cases, implementation cost must be as low as possible. The performance of a NIDS is affected by the scalability and flexibility of the matching system. A NIDS is a system that will be constantly updated with new rules to support the detection of new viruses and threats. For this reason, a NIDS

must be scalable and flexible so that when the rules increase, the implementation cost will not grow, and performance will not diminish significantly.

1.1.3 DPI technology

Security has become one of the most serious concerns on the Internet. NIDS has been adopted by enterprises to defend against attacks. There is an increasing demand for network devices that are capable of examining the content of data packets in order to enhance network security and provide application-specific services. A packet is the basic unit of data that is transmitted across an IP network, and the technology that inspects those packets beyond the transport header is called Deep Packet Inspection (DPI).

From an IP network's perspective, a packet is composed of three basic elements.

$$\{IP \quad header \quad [Protocol \quad header \quad (Content)]\}$$

- IP header: An IP header is the outermost portion of the packet where information such as the source and destination IP addresses reside (analogous to the address information on a postcard). It also contains other useful information about the packet, such as the packet length and the priority level on delivery.

- Protocol header: The protocol header describes the type of protocol used to transmit the packet. This header has traditionally been used as a basic firewall filtering method, which is essential for basic Internet security. Virtually all home, enterprise, and government networks employ firewalls on their Internet gateways today. Protocol headers are also routinely used in private and public IP networks to help classify network traffic so that each traffic type can be given the performance characteristics it needs.

- Content: This is also called packet payload. The payload portion of the

packet that contains the actual content or data being transmitted. Since there may exist some malicious data in the payload, some network packet payload analysis technologies are used to detect the payload content.

The traditional packet inspection technology only detects the packet header information so as to identify application bytes through the port number. However, illegal applications can use hidden or spoofed port numbers to avoid inspection and monitoring. By enabling the examination of the entire payload, DPI gives unprecedented "visibility" into deeper levels of network traffic to identify viruses and remedy vulnerabilities like data leakage and unauthorized access.

DPI is a crucial technique in today's Network Intrusion Detection System. Nearly 70% of processing time is consumed by DPI-based matching engines in NIDS [18]. DPI technologies are widely used in intrusion detection systems for deterring, detecting and stopping malicious attacks over the network. Nearly all intrusion detection systems have the ability to search through packets and identify contents that match known attacks.

1.1.4 String and Regular Expression Matching

A *string* is a sequence of characters over a finite alphabet Σ [19]. For instance, "reference" is a string over $\Sigma = \{c, e, f, n, r\}$. A *regular expression (regex)* describes a set of strings. Given a text T and a pattern P over some alphabet $\sum$, the string-matching problem is to find all occurrences of the pattern P in the text T. Classical string searching algorithms are based on character comparisons.

Regular expressions (regexes) can be recognized as an expression that specifies a set of strings. Due to their power of description, generalization and flexibility, regexes have been widely used in various aspects of computer science. Typically, regexes are not evaluated directly, but are translated into Nondeterministic Finite Automata (NFAs) or Deterministic Finite Automata (DFAs) for matching. Their translation map is given in Figure 1.2 [20]. A large number of algorithms

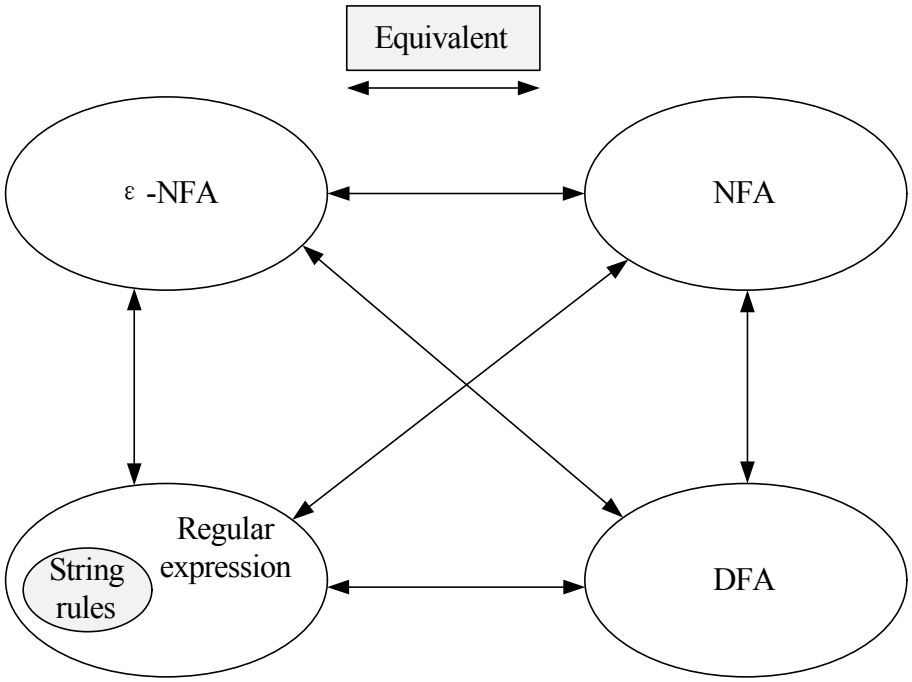


Figure 1.2: *Equivalence of regular expression, NFA and DFA.*

have been proposed to construct finite automata from regexes. The work in [21] presents a taxonomy of finite automata construction algorithms. One of the construction algorithms was proposed by K. Thompson [22], which is also called "structural induction" in textbooks [20] and considered to be more readable than Thompson's original paper.

A description of the regular syntax is given below [23].

Regex notation can be used to represent languages. Let Σ be an alphabet.

- $\emptyset$ is a regex representing the empty language;
- ε is a regex representing the language $\{\varepsilon\}$;
- $a \in \Sigma$ is a regex representing the language $\{a\}$;

if r and s are regexes representing languages $L(r)$ and $L(s)$, respectively, then

- R_1, R_2 is a regex representing $L(R_1)L(R_2)$;
- $R_1 + R_2$ or $R_1|R_2$ is a regex representing $L(R_1) \cup L(R_2)$;
- R_1^* is a regex representing $L(R_1)^*$; and

- nothing else is a regex.

Operator precedence, from high to low $*, \cdot, +$(or |), can be altered by parentheses.

Examples:

$(R_1 + R_2)^*$	all strings of R_1's and R_2's
$(R_1 + R_2)^*(R_1 + R_2R_2)$	all strings of R_1's and R_2's ending with R_1 or R_2R_2
$(R_1R_1)^*(R_2R_2)^*R_2$	even number of R_1's followed by an odd number of R_2's

The regex syntax is given in Table A.1.

Table 1.2: *Regular expressions syntax.*

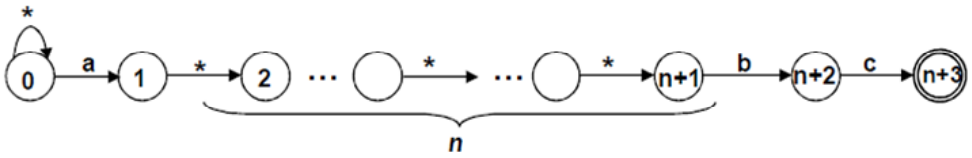
Symbol	Description	Example
.	Matches any single character except newline	a. matches aa, ab, ac
*	Repeats the previous item zero or more times.	a* matches a, aa, aaa, . . .
^	Matches beginning of line.	^a matches ab, ac or aa
$	Matches end of line.	a$ matches ba, ca or aa
+	Repeats the previous item once or more.	a+ matches a, aa, aaa
?	Matches **zero** or more instances of previous item.	abc? matches ab or abc
|	Causes the regex engine to match either the part on the left side, or the part on the right side.	a|b|c matches a, b or c
[ ]	Matches a single character that is contained within the brackets.	[abc] matches a, b, or c
[^]	Matches a single character that is not contained within the brackets.	[^abc] matches any character other than a, b, or c
{n}	Repeats the previous item exactly n times.	a{3} matches aaa
{n,m}	Repeats the previous item between n and m times.	a{1,3} matches a, aa or aaa

1.1.5 NFA and DFA

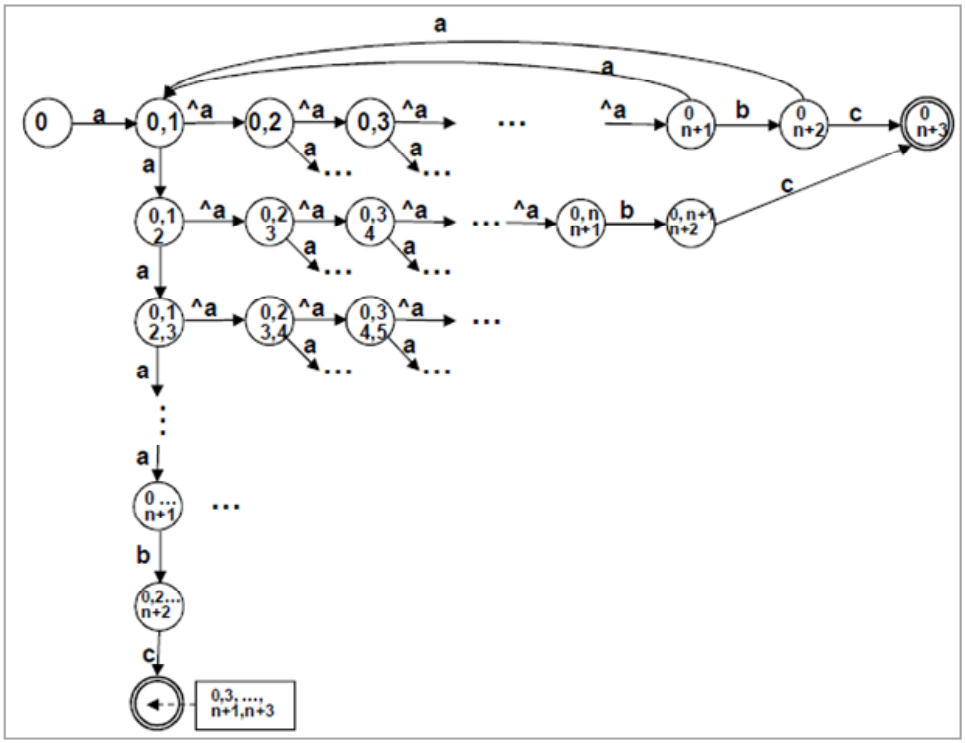
Finite automata are are also known as state machines which are the natural formalism for regexes. As described in the previous subsection, there are two main categories: NFA and DFA.

There exist three major approaches for transforming regular expressions into finite automata. The first approach, due to Thompson [22], is to transform a regex into a ε-NFA. This approach is simple and intuitive, but may generate many ε-transitions. Thus, the resulting ε-NFA can be unnecessarily large and the further transformation of it into a DFA can be rather time and space consuming. ε-NFAs have almost the same properties as NFAs, except that ε-NFAs have ε-transitions while NFAs do not. In the rest of this thesis, the terminology "NFAs" is also used to refer to ε-NFAs for simplicity. The second approach transforms a regex into a NFA without ε-transitions. This approach is due to Berry and Sethi [24], whose algorithm is based on Brzozowski's theory of derivatives [25] and McNaughton and Yamada's marked expression algorithm. Berry and Sethi's algorithm has been further improved by Brüggemann-Klein [26], Chang and Paige [27]. The third approach is to transform a regex directly into an equivalent DFA [25, 28]. This approach is very involved and can be replaced by two separate steps: (1) regular expressions to NFA using one of the above approaches and (2) NFA to DFA. Since NFAs tend to be time-inefficient, rate-sensitive applications usually deploy DFA-based implementations for regular expression matching. The most popular algorithm to convert NFAs into DFAs is "subset construction" [20]. The detail steps can be found in Appendix I.

The DFA generated by the subset construction algorithm is usually not state-minimized. For DFAs which accept the same language, there is an equivalent state-minimized DFA. In the process of DFA minimization, states are divided into several equivalent groups; the states within each group are equivalent. Therefore, states in the same group can be merged together into one state so that the



(a) NFA of regex `a.{n}bc`.



(b) DFA of regex `a.{n}bc`.

Figure 1.3: *NFA and DFA of regex `a.{n}bc`.*

number of states is reduced. Many different DFA minimization algorithms have been proposed [29]. A well-known and easy-to-understand algorithm, which is introduced in almost all textbooks, is the table-filling algorithm [20], with a time complexity of $O(n^2)$. The most efficient minimization algorithm currently known is Hopcroft's algorithm, whose time complexity is reduced to $O(nlogn)$ [30].

DFA is fast and has deterministic matching performance but suffers from the memory explosion problem. NFA, on the other hand, requires less memory but suffers from slow and non-deterministic matching performance.

When the NFA is converted into a DFA, it may generate O(Σ^n) states[2]. For

[2]Here n denotes the state number of NFA.

example, given regex `a.{n}bc`, the corresponding NFA is shown in Figure 1.3(a) and DFA is shown in Figure 1.3(b). It is easy to find that the DFA is more complicated than the NFA. In fact, the DFA of regex `a.{n}bc` has about 1000 times states more than its corresponding NFA if n=40.

Given R, the finite set of regexes, a finite automaton is extended to 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where:

- Q is a finite set of states;
- Σ is a finite set of input symbols;
- δ is a transition function: for a NFA, δ is a mapping $Q \times \Sigma \rightarrow 2^Q$, where 2^Q is the power set of Q; for a DFA, δ is a mapping $Q \times \Sigma \rightarrow Q$;
- q_0 is the single start state;
- $F \subseteq Q$ is a set of final states.

1.1.6 CIDS and DLPS

Technologies such as instant messaging software (Skype, MSN) or BitTorrent file sharing methods, allow the convenient sharing of information between managers, employees, customers and partners. In contrast to traditional client server file sharing mode, each peer acts as a supplier and a consumer at the same time. Peer to Peer (P2P) mode has contributed to a tremendous portion of today's file sharing traffic. It is claimed that P2P file sharing systems generated as much as 43-70% of the total Internet traffic in each continent around the world during 2008 and 2009 [31].

However, in spite of the convenient file access, file sharing technologies such as P2P applications (BitTorrent, eMule, vuze) and instant messaging tools (Skype, MSN) also create two major potential security problems in today's networks: the

leakage of personal information or confidential documents and the distribution of copyright infringement files.

It is not a secret that the majority of files being shared over BitTorrent are movies that are likely being shared illegally. Movie producers are now more than ever worried about the potential loss of revenues due to movie files sharing in the P2P networks. Voltage Pictures [32], producers behind the Oscar-winning film, "The Hurt Locker" [33], filed a lawsuit against illegal file sharers. The suit claimed that 5,000 BitTorrent users violated copyright laws and is one of the largest lawsuits filed against individuals, with users receiving letters to pay a $1,500 fee to settle. Considerable efforts have been devoted to investigate technology solutions to reduce content availability in P2P file sharing systems. Movie producers' wise option is to cooperate with network service providers to control the distribution of movie files and introduce online download services with copyright. In P2P networks, the illegal sharing of movies, music and software causes significant financial losses. The value of unlicensed software is calculated to have hit $51.4 billion in 2009. Similarly, as a consequence of global piracy of sound recordings, the U.S. economy is estimated to lose $12.5 billion in total output annually, and as a result of sound and recordings piracy, the U.S. economy loses 71,060 jobs [34] annually. In fact, the benefits of protecting copyright include additional jobs, new local revenues, and additional revenues.

In order to prevent the leakage of sensitive information via the file sharing system, a *Data Leakage Prevention System* (*DLPS*) could be applied at the entrance of the enterprise network (e.g., gateway or edge router) to filter the outgoing information for sensitive content. Similarly a *Copyright Infringement Detection System* (*CIDS*) is required to solve the problem of copyright violations by Internet service providers (ISP) and movie producers.

Traditional Deep Packet Inspection (DPI) technologies cannot be used directly in DLPS or CIDS. With the purpose of increasing the speed of distribution of a re-

source file, most P2P systems split the file into fixed-size pieces, except for the last piece, enabling P2P clients to download data from multiple peers simultaneously [35]. However this advantage leads to a problem for file content detection as the file is transferred in an out-of-order manner. The downloaded file is composed of random fragments passing through the gateway. After receiving all the out-of-order packets, the original file can be reassembled by the P2P application. Traditional DPI works by applying the pattern matching methods to the reassembled payload. Obviously, it is impractical to cache all the packets during the distribution of large-scale files. Therefore, traditional DPI technologies cannot be used to identify the files transmitted via P2P applications.

1.2 Problem Statement and Work Description

Since the rise of widespread broadband Internet access, malicious software and virus applications have been designed to destroy files on hard disks, or to corrupt file systems. Malicious traffic, although only a very small part of the whole traffic in the network, can bring terrible consequences. Traditional network traffic inspection methods have serious limitations since they only check packets header information without checking payload information.

New applications such as real-time deep packet inspection require high-speed regex matching, since regex has better flexibility in representing attacks than fixed strings. Unfortunately, the number of regexes in pattern rulesets is increasing to several thousands, which requires a memory-efficient solution.

The current NIDSes have the following two challenges:

- There are a wide varieties of attacks. Thus each packet needs to be matched against thousands of attack signatures. For example, SNORT [36] is a popular open-source intrusion detection system, with millions of downloads up to date. It can be configured to perform protocol analysis, and content

searching and matching, on real-time traffic to detect a variety of worms, attacks and probes. The SNORT network intrusion detection system has 4356 rules as of v2.9 06 Apr, 2011, and each contains attack signatures.

- The widespread use of regex is due to their expressive power and flexibility for describing protocol patterns. For example, all protocol signatures in the L7-filter [37] are written in regex. Multiple regexes can be combined together into either NFAs or DFAs, which provides a constant time process for checking each byte of a packet payload. A regex is powerful, however, it is time-consuming to generate the corresponding DFA with very significant memory cost, even if a regex can be transformed to a DFA successfully. The requirment of memory storage reaches 15GB after the combination of about 700 rules in Snort NIDS [38].

In order to use DFAs practically in NIDS devices with limited memory resources, a large amount of work have been conducted on how to reduce memory consumption [39, 40, 41]. However, there has been much less focus on increasing the matching speed. Only a few studies focused on multi-characters matching algorithms to speed up the matching process.

To accelerate regex matching and enable deep packet inspection at line rate, Stride-based Matching, a novel acceleration scheme for regex matching, is proposed in this thesis. It converts the original byte stream into a much shorter integer stream. The integer stream is used to make a match to achieve a higher throughput than matching the input stream byte-by-byte. The Stride-based Matching method opens the door for a new class of regex matching accelerating algorithms.

1.3 Thesis Organization

The purpose of this thesis is to give a detailed overview in the area of pattern matching and payload inspection, and more importantly to present the idea of the Stride-based matching algorithm, and finally to illustrate the main outlook of the future work.

In Chapter 2, previous work related to pattern matching is discussed. StriDFA, a novel acceleration scheme, is presented in Chapter 3 for multi-characters string matching. Chapter 4 illustrates how to perform regex matching with StriNFA and StriDFA. In Chapter 5, a hybrid finite automaton called Skip-Stride-Neighbor Finite Automaton (S^2N-FA) is proposed to solve the out-of-order data scanning problem. Planned future work is listed in Chapter 6.

CHAPTER 2

Pattern Matching Algorithms

String matching algorithms can be classified into either single pattern string matching or multiple pattern string matching algorithms. In single pattern string matching, the packet is searched for a single string at a time. In multiple pattern string matching, the algorithm searches the packet for a set of strings simultaneously. Single pattern string matching algorithms are inefficient and cannot be used in real NIDS, thus multiple pattern string matching algorithms are discussed in this thesis. The multiple pattern string matching algorithms can be further classified into single-character based string matching algorithms and multi-character based string matching algorithms.

2.1 Single-character based String Matching Algorithms

Two of the most popular algorithms for single-character based string matching algorithms are those published by Aho and Corasick (AC) [42] and Boyer and Moore (BM) [43] almost 30 years ago. The Aho-Corasick algorithm constructs a finite state machine (FSM) for detecting all occurrences of a given set of patterns

by processing the input in a single pass, performing a state transition for each input character. The Boyer-Moore algorithm, which is basically a single-pattern matching solution, exploits two heuristics (named "bad character" and "good suffix") to skip portions of the input stream in order to improve the average performance. Concepts similar to Aho-Corasick and Boyer-Moore can be found in many other pattern-matching algorithms, such as the algorithms by Commentz-Walter [44], and more recently, the Aho-Corasick-Boyer-Moore (AC-BM) algorithm proposed by Silicon Defense [45] which combines the Boyer-Moore and Aho-Corasick algorithms. Another new algorithm is the Setwise Boyer-Moore-Horspool algorithm by Fisk *et al.* [46], whose average case performance is better than Aho-Corasick and Boyer-Moore. These algorithms greatly improve multi-pattern matching speed. However, it is still below the line rate required for network deployment.

2.1.1 Aho-Corasick algorithm

The Aho-Corasick (AC) algorithm [42] is based on finite state automata (FSA). Before the matching process, the string pattern set is complied into an FSA. Then the input steam is scanned byte-by-byte by the FSA to find all the matching string patterns in it. During the preprocessing stage, three functions are generated, which are the "goto" function, the "failure" function and the "output" function. The "goto" function outputs the next state given the current state and input character. The "failure" function outputs the next state when the input character cannot continue to match a string pattern from the current state. The "output" function outputs the pattern ID when one of the string patterns has been matched by the current state. The AC automata built with the string pattern set {`he, her, him, his`} is shown in Figure 2.1. The matching process of the AC algorithm is: from the initial state 0, one character is fetched from the input stream each time. According to the current state and input character, the next state can be

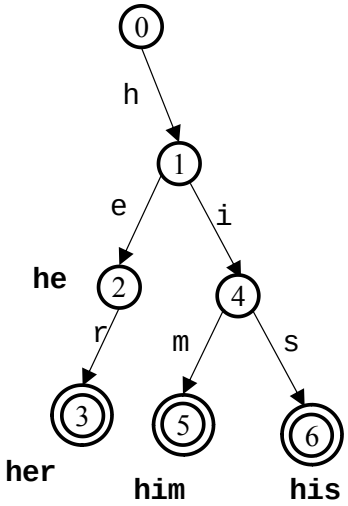


Figure 2.1: *AC algorithm with pattern set {he, her, him, his}.*

obtained with the "goto" function or "failure" function. If the returned result of the "output" function of a state is not null, it means a string pattern is matched at this state and the pattern ID can be found. For example, if the input text is "`helloher`", the matching process is shown below.

$0 \xrightarrow{h} 1 \xrightarrow{e} 2 \xrightarrow{l} 0 \xrightarrow{l} 0 \xrightarrow{o} 0 \xrightarrow{h} 1 \xrightarrow{e} 2 \xrightarrow{r} 3$ (matched by pattern "`her`")

The time complexity of the AC algorithm is $O(n)$, where n is the length of the input text. The processing time is unrelated to the number and the length of the strings in the pattern set. Whether string pattern P is included in input text T or not, every character in T must be processed by the AC automata. Therefore, the time complexity of the AC algorithm is always $O(n)$ in the best case or the worst case. Considering the preprocessing time, the total time complexity is $O(M + n)$, where M is the total length of all the string patterns. The space complexity (amount of memory required) is $O(M)$.

The Aho-Corasick algorithm is the classic algorithm for searching for multiple patterns simultaneously. Generally speaking, the Aho-Corasick algorithm uses the structure of a finite automaton that accepts all strings in the set. The automaton is structured so that every prefix is represented by only one state, even if the prefix begins multiple patterns. When the next character in the text is not

one of the expected next characters in the pattern, a failure edge is linked to the state representing the longest prefix of a pattern that is also the proper suffix of the current state.

One of the advantages of the Aho-Corasick and Commentz-Walter algorithms is they could match multiple patterns simultaneously. The process time is only related to the length of the input text. They both pre-process the patterns and build a finite automaton which can process the input packet in $O(n)$ time where n is the length of the input text. Although both algorithms are fast, they suffer from an exponential state explosion.

2.1.1.1 AC algorithm based on bitmap compression and path compression

Tan *et al.* proposed a bit-splitting finite state automaton algorithm [47], where each finite state machine (FSM) is split into eight corresponding bit FSMs. Each bit-split FSM processes one bit of the input character. There are only two possible values of the transition conditions which are '0' and '1' from a state to its next state in a bit FSM.

The construction process of bit FSMs is as follows. For an AC automata D, the eight bits of the ASCII code of the input character for each state is used to construct the corresponding binary FSMs, which are represented by $B_0, B_1, \ldots, B_7$. For a bit FSM B_i, starting from the initial state of the original AC automata D, the next states are classified into two categories. The transition condition of B_i is '1' if the i^{th} bit of the input character is '1', and the transition condition of B_i is '0' if the i^{th} bit of the input character is '0'. The new states in B_i will be generated according to the above two conditions.

For example, for the pattern set {`he, she, his, hers`}, the original AC automata is shown on the left of Figure 2.2.

The bit FSM B_3 representing the fourth (here i=3 and i start from 0) bit of the character in the pattern which is shown on the middle of Figure 2.2. The

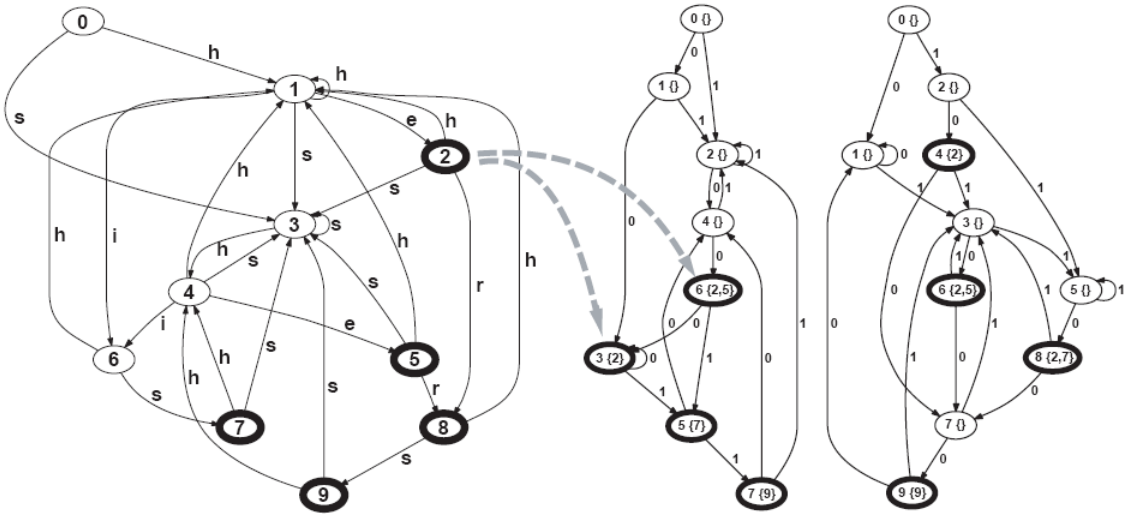


Figure 2.2: *Extracting bit-level parallelism from the Aho-Corasick algorithm by splitting the state machine into 8 parallel state machines.*

ASCII code of character 'h' is "01101000", and the ASCII code of character 's' is "01110011". From the state 0 to state 1 with the transition condition 'h' in the original automata D, the corresponding fourth bit of 'h' is '0', so a new state is generated in B_3, and the transition condition from state 0 to state 1 in B_3 is '0'. Similarly, the fourth bit of 's' is '1', a new state is generated in B_3, and the transition condition from state 0 to state 2 in B_3 is '1'. Similarly the other FSM B_i ($0 \leq i \leq 7$) can be generated separately. Each i bit of input character will be sent to the corresponding FSM B_i to make a match. If all B_i can be matched, then this input character can be matched. Otherwise it cannot be matched if any one of the B_i cannot match the input bit.

2.2 Multi-Character Based String Matching Algorithms

Traditional single-character DFA is a state machine which consists of a finite set of states and directional transitions. Each transition has a specific input character on which a transition can occur from one state to another. Multiple-character DFA also consists of a set of states and transitions; however, the transitions are not limited to a single character.

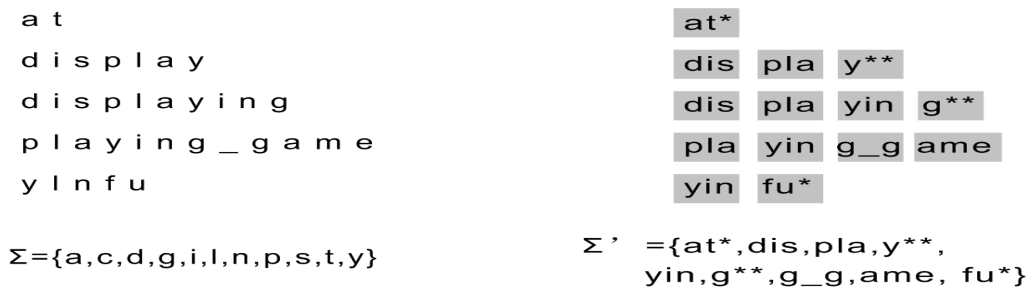


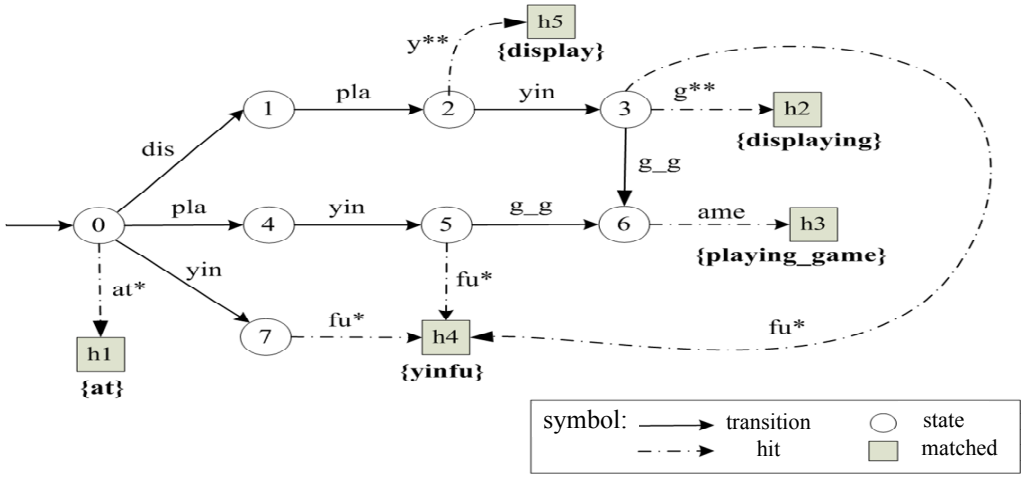
Figure 2.3: *Transfer character set to the token sequences.*

2.2.1 Transition-distributed Parallel DFA

Lu *et al.* [48] proposed a memory-efficient multiple-character-approaching scheme consisting of multiple parallel deterministic finite automata (DFAs), called transition-distributed parallel DFAs (TDP-DFA). Parallel DFAs with overlapping input windows are used to achieve the goal of processing multiple characters in each clock cycle. TDP-DFA meets the size limitation of embedded memory, and can be implemented on-chip with current ASIC technology.

In TDP-DFA, w characters (eg., $w = 3$) are regarded as a token, and then each signature is decomposed into one or more tokens. Note that, as shown in Figure 2.3, the appropriate number of wildcards, i.e., '*'s may be padded to make the length of the signature an exact multiple of w. A corresponding NFA can be constructed and converted to a DFA for detecting the token sequences. The DFA structure can be found in Figure 2.4. As in the case of Figure 2.5, the boundary of a single input window may not be aligned with the starting character of the pattern, hence w DFAs need to be deployed for detection in parallel.

The advantage of TDP-DFA is it can process more than one input character at a time so as to achieve a higher matching speed. The disadvantage of TDP-DFA is that w DFAs are used for matching in parallel which means the requirement of memory cost of the TDP-DFA is w times larger than the original one.



For simplicity, some transitions are not shown.

Figure 2.4: *An example of TDP-DFA with w=3.*

2.2.2 Bloom Filter-based String Matching Algorithm

There are some hardware-based string matching techniques using Bloom filters, which can detect strings in streaming data without degrading network throughput [49, 50].

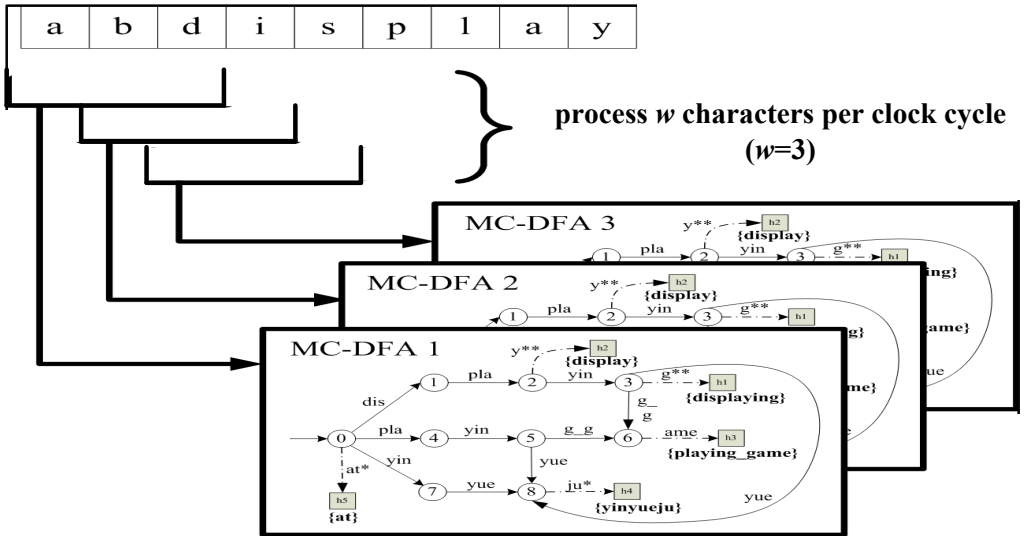


Figure 2.5: *Deploy TDP-DFAs for detection in parallel.*

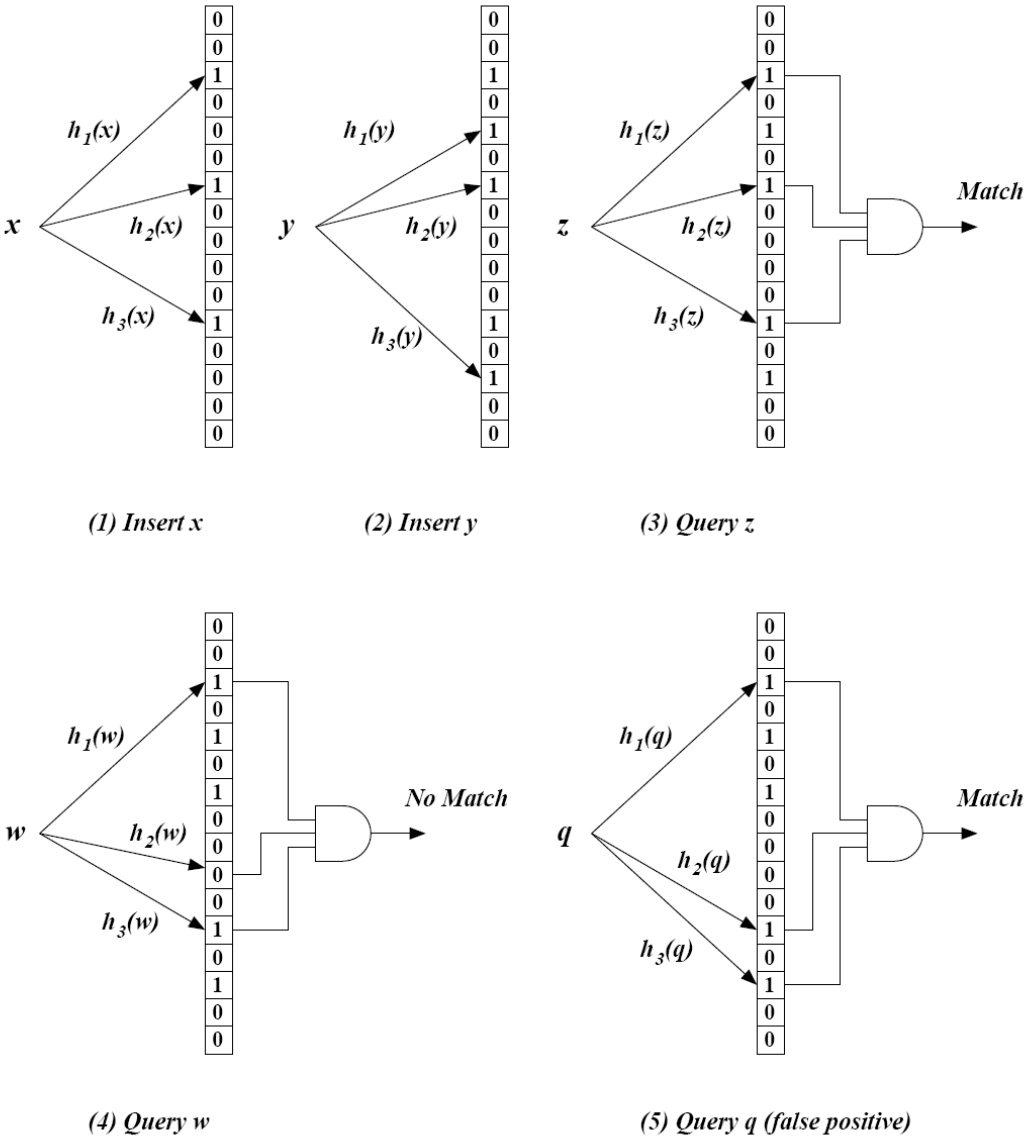


Figure 2.6: *Introduction of Bloom filter.*

2.2.2.1 Bloom Filter

A Bloom filter is a data structure that contains a set of patterns compactly by computing multiple hash functions on each member of the set.

The hash functions in a Bloom filter should be independent and as fast as possible. Figure 2.6 (1) and (2) illustrate Bloom filter programming. Two strings, x and y are programmed in the Bloom filter with $k = 3$ hash functions and $m = 16$ bits in the array. The bit will be set to 1 at all these positions generated by the hash functions. Note that different strings can have overlapped bit patterns.

k hash values are generated using the same hash functions used to program the filter with the input string s. The bits in the m-bit vector at the locations corresponding to the k hash values are checked. If at least one of the k bits is 0, then the string is declared to be a non-member of the set[3] (Figure 2.6(4)). If all the bits are found to be 1, then the string is a potential membership of the set (Figure 2.6(3)). If all the k bits are found to be set and s is not a member of S, then it is said to be a false positive.

The ambiguity in membership comes from the fact that the k bits in the m-bit vector can be set by any of the n members of S. For instance, in Figure 2.6(5), q maps to all the bits which were set by x and y. Although $q \notin S$, the filter shows a match which is a false positive. Thus, finding a bit set does not necessarily imply that it was set by the particular string being queried. However, finding a 0 bit certainly implies that the string does not belong to the set; if it were a member, then all k-bits would have been set when the Bloom filter was programmed.

2.2.2.2 False Positive Probability

If m is the number of bit vector. The probability that a random bit of the m-bit vector is not set to 1 by a hash function is simply $1 - \frac{1}{m}$. The probability that it is not set to 1 by any of the k hash functions is $(1 - \frac{1}{m})^k$. If n strings are inserted,

[3]The set of signatures is denoted as S.

the probability becomes $(1 - \frac{1}{m})^{nk}$ which a certain bit is still 0. The probability that this bit is 1 becomes $1 - (1 - \frac{1}{m})^{nk}$. For a string to be detected as a possible member of the set, all k bit locations generated by the hash functions need to be 1. The probability that this happens, f, is given by

$$f = (1 - (1 - \frac{1}{m})^{nk})^k$$

For large values of m, the previous equation reduces to

$$f \approx (1 - e^{\frac{-nk}{m}})^k$$

This probability is independent of the input string and is termed the false positive probability. The false positive probability can be reduced by choosing appropriate values for m and k for a given size of the member set, n. It is clear that the size of the bit-vector, m, needs to be much larger than the size of the string set, n. For the given ratio $\frac{m}{n}$, the false positive probability can be reduced by increasing the number of hash functions, k. In the optimal case, when false positive probability is minimized with respect to k, the following relationship can be calculated:

$$k = \frac{m}{n} \ln 2$$

In practice, the false positive probability can be adjusted according to different situation. If a low false positive is required, then the parameters k, n and m can be changed by the above theoretical analysis.

2.2.2.3 Jump-ahead Aho-Corasick NFA

S. Dharmapurikar *et al.* proposed an algorithm called Jump-ahead Aho-Corasick NFA (JACK-NFA) [50]. The basic idea of JACK-NFA is to compare k characters at a time.

As Figure 2.7 shows, each string is firstly segmented into k character segments

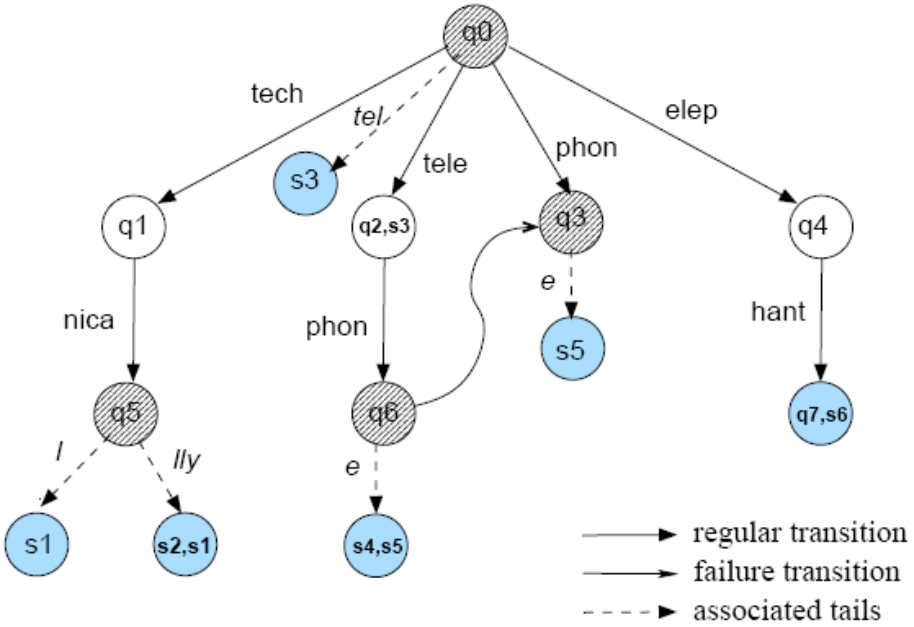


Figure 2.7: *Jump-ahead Aho-Corasick NFA (JACK-NFA)*

($k = 4$ for the purpose of illustration) and the left over portion is called a *tail*. Each of these k-character segments is treated as a symbol. There are six unique symbols in the given example namely {`tech, nica, tele, phon, elep, hant`} and the tails associated with these strings are {`l, lly, e`}. The tails "`l`" and "`lly`" are attached to state q_5. However, the tails do not create any state to which the automaton jumps. Tails simply indicate the completion of a string.

To execute pattern matching, the matching engine starts from state q_0 and looks at k characters (bytes) from the text. If this k-character symbol matches with any of the valid symbols associated with the state q_0 then a transition is made to the corresponding state and the next k characters are scanned. At each state, while the k characters are considered, all the prefixes of these k characters are checked to see if any of them matches any of the tails associated with that state. If at any state a match is found for a tail then the corresponding string associated with the tail will be reported.

There exists a byte alignment problem which means that for multi-byte scanning that every character of the input stream should have the chance to be examined as the first character. This requires duplicate matching modules to return the

correct matching results. For example, given the input string "`abctechnical`", the matching engine will detect the string as "`abct echn ical`". "`abct`" cannot be matched by any 4-character segment associated with state q_0, the automaton will never jump to a valid state causing the machine to miss the detection. Similarly next 4-character segment "`echn`" cannot be matched either. However, there is a transition from q_0 to q_1 by accepting 4-character segment "`tech`" which appears in the input string. To avoid the byte alignment problem, 4 matching modules are used to match the input string in parallel. The first matching module detects "`abct echn ical`" and the second matching module detects "`bcte chni cal`". Similarly, the third machine scans it as "`ctec hnic al`" and finally the fourth machine scans it as "`tech nica l`". The first three matching modules will miss the detection while the final one will report a match.

A hash table is used for representing the JACK-NFA (shown in Table 2.1). Each table entry consists of a pair $< state, substring >$ which corresponds to an edge of this NFA. If the next k-character segment can be found with the current state in the table, we can jump to next state to keep on processing.

Table 2.1: *Transition table of JACK-NFA.*

$\langle state, substring \rangle$	Next State	Matching Strings	failure chain
$\langle q_0, tech \rangle$	q_1	NULL	q_0
$\langle q_0, tele \rangle$	q_2	s_3	q_0
$\langle q_0, phon \rangle$	q_3	NULL	q_0
$\langle q_0, elep \rangle$	q_4	NULL	q_0
$\langle q_1, nica \rangle$	q_5	NULL	q_0
$\langle q_2, phon \rangle$	q_6	NULL	q_3, q_0
$\langle q_4, hant \rangle$	q_7	s_6	q_0
$\langle q_0, tel \rangle$	NULL	s_3	NULL
$\langle q_3, e \rangle$	NULL	s_5	NULL
$\langle q_5, l \rangle$	NULL	s_1	NULL
$\langle q_5, ly \rangle$	NULL	s_2, s_1	NULL
$\langle q_6, e \rangle$	NULL	s_4, s_5	NULL

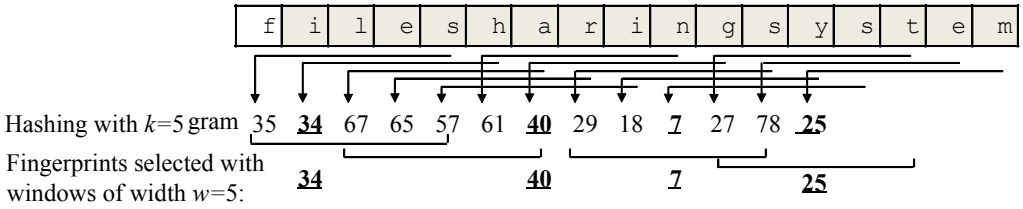


Figure 2.8: *Winnowing sample text. The gray part is covered by the selected fingerprints.*

2.2.3 Variable Stride DFA

In [51], Nan *et al.* presented a variable-stride multi-pattern matching algorithm in which a variable number of bytes from the data stream can be scanned in one step. The winnowing idea [52] is developed for document fingerprinting in a novel manner to develop a variable-stride scheme that increases the system throughput considerably while also decreasing memory usage by an order of magnitude over previously proposed algorithms.

2.2.3.1 Winnowing Algorithm

The winnowing algorithm is a document fingerprinting scheme, which ensures that the matched parts between the files and the input are preserved in the matching results of the processed files and input (consisting of fingerprints). In Figure 2.8, the text is firstly transformed into a sequence of hashes of the k-grams ($k = 5$ in Figure 2.8), so each hash value represents a k-character block. Then by sliding a window of width w ($w = 5$ in Figure 2.8) from left to right, a specific hash value is selected from each window, e.g., the smallest hash value is selected in each window[4]. In this example, four fingerprints are selected from a text of length 17 while almost all contents are represented in the fingerprints selected (the gray segment). In general, the gaps between fingerprints can be widened with larger window size w, and it is possible to have more of the contents covered by selecting a fingerprint with large k.

The winnowing algorithm could solve the byte alignment problem (explained

[4]If there is more than one smallest hash value, the rightmost one will be chosen.

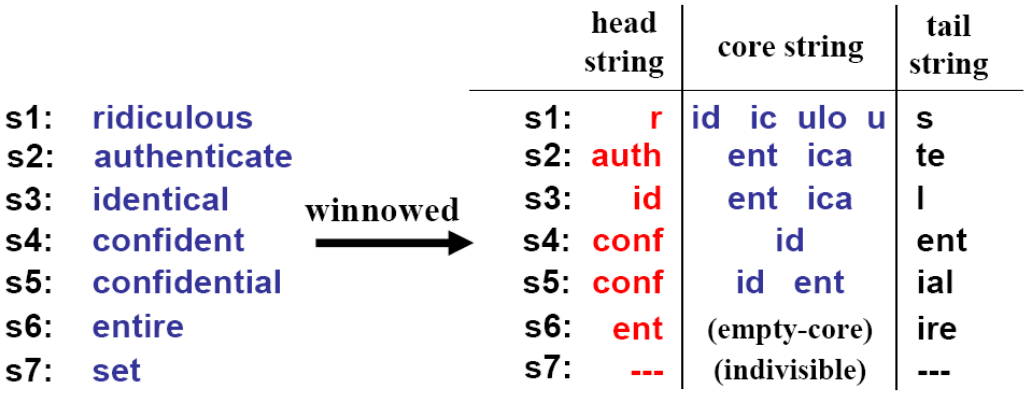


Figure 2.9: *Segment Pattern into Head/Core/Tail Blocks ($k = 2$, $w = 4$).*

in subsection 2.2.2.3) because it has a "self-synchronization" property. This algorithm ensures that irrespective of the context in which a pattern appears in the input stream, it is always segmented into the same sequence of blocks (with some head and tail pattern exceptions). These unique blocks can be used as atomic units to construct a variant DFA which is called VS-DFA here. The basic idea of VS-DFA is to compare these unique blocks of input string with the corresponding blocks of the pattern.

Pattern segmentation examples are shown in Figure 2.9. If two blocks `ent` and `ica` have been matched consecutively, there is potentially a match to S_2. To verify the match, all the system needs to do is to retrieve the w-byte prefix characters and $(w + k - 2)$-byte suffix characters in the data stream to compare against S_2's head block and tail blocks.

As shown in Figure 2.10, all the segment strings are used to construct VS-DFA. The key feature of VS-DFA is that its state transitions are based on core blocks rather than individual characters. A VS-DFA pattern matching system consists of two modules: the winnowing module and the state machine module. The incoming data stream (packet payload) is first segmented into variable sized blocks by the winnowing module and pushed into a First-In-First-Out (FIFO) queue. Then the blocks are fetched from the queue and fed into the state machine one by one. The state machine runs the VS-DFA and outputs the matching results.

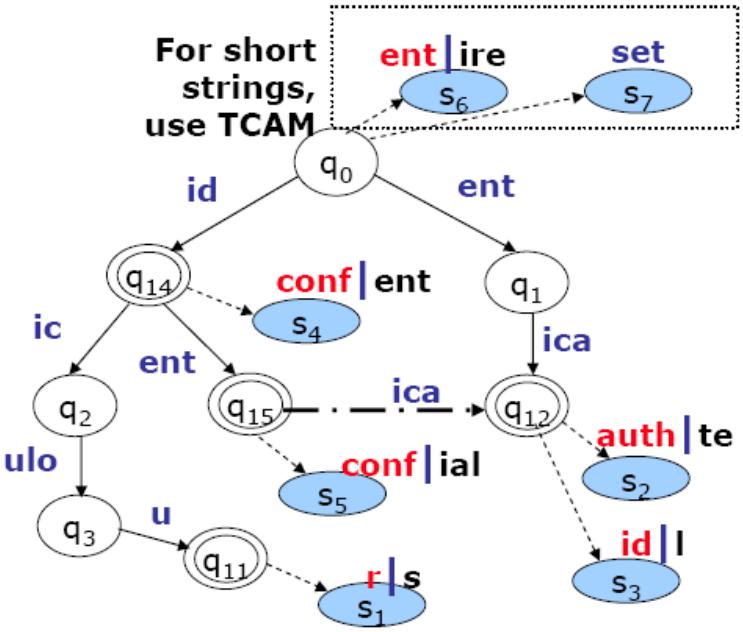


Figure 2.10: *An example of a VS-DFA.*

The incoming traffic data will be sent to the same fingerprinting scheme so as to get the correctly sized block of characters. These blocks is used to feed to the VS-DFA. The VS-DFA matching procedure and the fingerprinting operation can be done simultaneously.

VS-DFA could process more than one input characters in a block at a time so as to achieve a higher matching speed. But there are some disadvantages about VS-DFA. It cannot be used for regex matching and not suitable for software implementation. This method is sensitive to both rule set and input string, with greatly reduced throughput in the worst cases. A malicious stream can continuously produce short strides, leading to degraded matching throughput close to that of the single-character matching solutions; moreover, a large portion of the patterns would be left to be matched by other brute-force matching hardware resources.

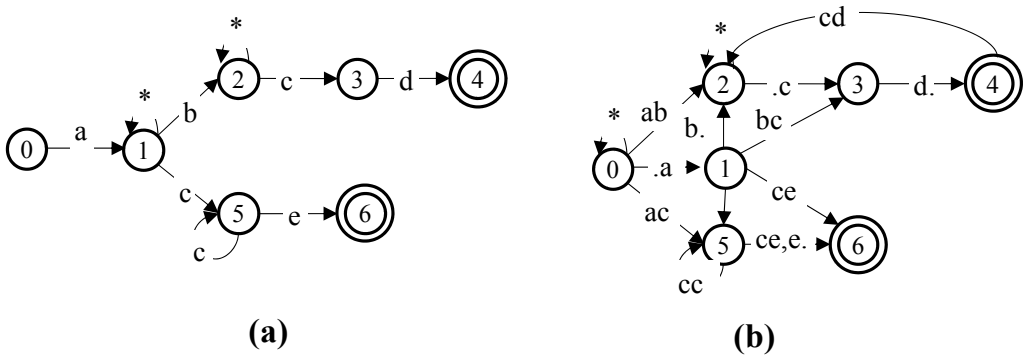


Figure 2.11: *(a) NFA accepting (1)* `ab.*cd` *and (2)* `ac+e`*; (b) corresponding 2-NFA.*

2.3 Regular Expression Matching

Regular expression matching was initially studied as a topic in automata theory and formal theory in the context of theoretical computer science [20]. To accelerate the regex matching in real-world systems, the problem has been intensively studied in practical scenarios in recent years. Vulnerability signatures have recently been proposed as an alternative to regex, but a high speed regex matching subsystem [53] is still required.

2.3.1 k-DFA

Michela Becchi and Patrick Crowley proposed k-NFA and k-DFA [54], meant to process k input characters at a time. Figure 2.11 shows an example of 2-NFA of `ab.*cd` and `ac+e`. Their work proposes acceleration techniques that rely on multiplying the amount of bytes (strides) processed per cycle, with the obvious problem of memory blow-up (due to the exponential growth of edge numbers with the stride size).

There are two problems of k-NFA/k-DFA: alphabet set explosion and transition explosion. One problem of k-NFA/k-DFA is the Alphabet set explosion. For the 2-NFA in Figure 2.11 (b), assuming that $\sum$ is the ASCII alphabet (256 elements), $\sum^2$ would have 256^2, that is 65536 elements. Similarly, if the alphabet

of the original DFA is $\sum$, the non-compressed k-DFA will have $|\sum|^k$ outgoing transitions per state. The other problem is transition explosion. For each state, the possible outgoing transition number is $|\sum|^k$. Because of the not appropriate explosion, it is impractical for implementation in real system.

2.3.2 Multi-Stride DFA

Vespa *et al.* [55] presented a multiple-stride pattern-matching architecture that requires a small storage and non special-purpose hardware. The basic idea is to group DFA states/transitions into three coarse-grained and variable-sized blocks, so that each individual block can employ different-specific methods to optimize storage requirements and performance. The blocks are naturally identified based on basic observations of DFA characteristics: prefix, linear trie and state dependencies.

An example of multiple-stride DFA is used to explain the basic idea of Multi-Stride DFA (MS-DFA). If the patterns are {"AABCDZGHIJ3A2B1C", "ABCDEFGHIJSTUVWXYZ", "0123456789Z", "6789KLMNOPYZABC"}, we can divide the pattern into tree parts according to the relationship among the patterns which is shown in 2.12. These three parts are used to generate as the label of transitions and stored in three different memory blocks.

Matching starts in the primary block at state 0. The primary block stores transitions for the first m bytes of each pattern (here $m = 10$). If a transition is followed in the primary block, control point moves to the secondary block. If a secondary block transition fails and a ternary block transition exists for the current state, then control moves to the ternary block. A matching transition in the ternary block moves control back to the secondary block. Failure in the ternary block moves control back to the primary block [55].

If there is a shared prefix in two patterns, then the first character except for this prefix is stored in the ternary block. Then the remaining bytes of each pattern

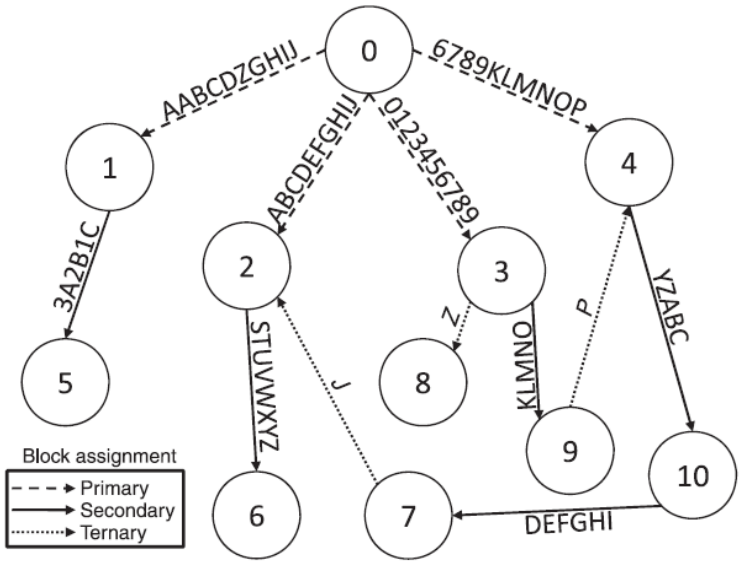


Figure 2.12: *MS-DFA for patterns {"AABCDZGHIJ3A2B1C", "ABCDEFGHIJSTUVWXYZ", "0123456789Z", "6789KLMNOPYZABC"}.*

are stored in the secondary block. Next, longest prefix match transitions are created and stored in the secondary and ternary blocks. For example, the substring "6789" appears at the end of pattern "0123456789Z". At the same time, it is the beginning of the pattern "6789KLMNOP". The rest substring of "6789KLMNOP" except for the same prefix is "KLMNO". It will be stored in the secondary block. The rest substring of pattern "0123456789Z" except for the same prefix is "Z" which is stored in the ternary block.

MS-DFA is a memory-based DFA run from software only. Sometimes it is a little hard to find two regexes sharing the same prefix. So MS-DFA cannot work well in regex matching.

2.3.3 Sampled DFA

A recent method [56] introduced sampling techniques to accelerate regex matching, which allows skipping a large portion of the input stream, thus processing fewer bytes[5]. The price to pay is a small probability of false alarms, which require a confirmation stage. Therefore, the author proposed a double-stage match-

[5]called sampled DFA or θ-DFA, where θ is the sampling period.

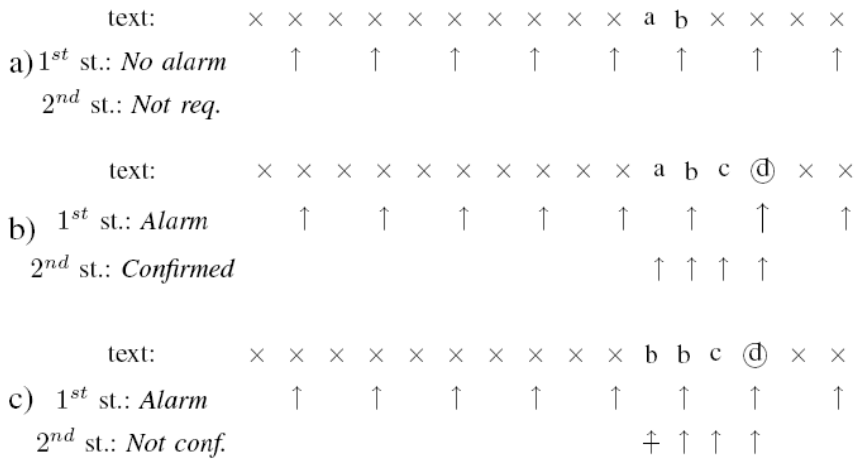
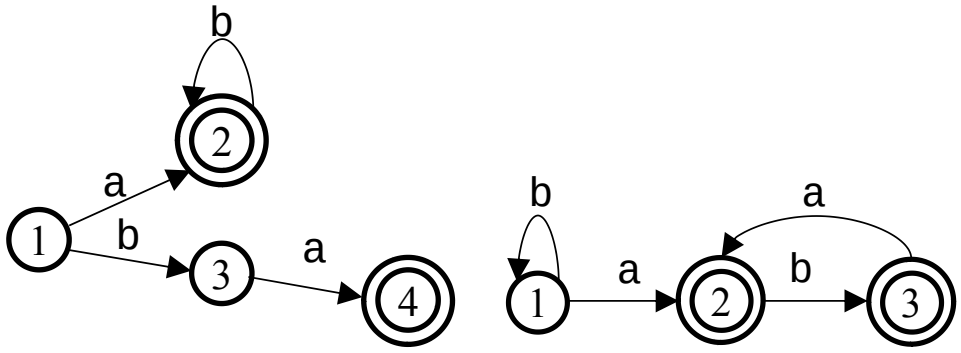


Figure 2.13: *Example: the regex* `ab.*cd` *is sampled (with* $\theta = 2$*) to [ab].*[cd] and matched against a text of 16 bytes.*

ing scheme providing two new different automata (sampled DFA and reverse DFA [57]). The basic idea of sampled DFA is to extract the fixed string of patterns as the new pattern to match the payload text, thus reducing the number of characters to be checked. Sampled DFA works as a filter like the other kinds of filter engine, such as Bloom filter. The effectiveness of this method is justified by the fact that in most cases, the first sampled lookup is enough to a classify packets, while every few packets only require a second stage of processing.

Figure 2.13 shows the principles of the matching scheme, with the example regex `ab.*cd`. A sampled DFA is used to make the match (that matches [ab].*[cd]) and a regular non-sampled one is used to make the confirmation ([xy] means that both characters 'x' and 'y' trigger a transition). For example in Figure 2.13 c), the input string "bbcd" will be checked 'b' and 'd' first with $\theta = 2$. It can be matched by the sampled DFA [ab].*[cd], then the other characters 'b' and 'c' will be checked. It cannot be matched which means it is a false alarm. The first check is performed on the text by using the sampled DFA; if a match is found, then the second stage is triggered. Whenever the sampled regex is matched, the non-sampled text has to be checked to confirm the match.

Reverse DFA is first proposed in [57]. The idea is to use dual finite automata



(a) Traditional DFA for pattern `ab*|ba`.

(b) Reverse DFA for pattern `ab*|ba`.

Figure 2.14: *Traditional DFA and reverse DFA for pattern* `ab*|ba`.

to scan regular expressions faster. For a given regular expression `ab*|ba`, the traditional DFA is shown in Figure 2.14 (a). According to the construction method in [57], the corresponding reverse DFA can be found in Figure 2.14 (b). Assuming the input string is "abbb", it can be matched by the traditional DFA at state 2. The reverse string of "abbb" is "bbba" and the reverse string can also be matched by the reverse DFA (at state 2).

2.4 Comparison of Multi-character Matching Algorithms

JACK-NFA is based on the Bloom filter and VS-FA uses winnowing as its basic function, both of which are not suitable for software implementation. VS-FA is used for string matching, but it cannot be employed for regex matching. MS-FA is software only and cannot be used for regex matching. k-DFA is fast, but has a transition explosion problem. θ-DFA works for constant input bytes and has bad performance for some regexes in worst cases.

StriDFA can be implemented in both software and hardware matching systems. It has no byte alignment problem and can be used for regex matching. In the next chapter, the details of StriDFA will be explained.

Table 2.2: *Comparisons of multi-character matching algorithms. '+' means more memory usage is needed while '-' means less memory requirements.*

	Hardware support	Software support	Regex support	Transition explosion	Speedup	Memory usage	Variable stride	Byte alignment problem	Σ
JACK-NFA	Y	N	Y	N	Y	+	Y	Y	256
VS-FA	Y	N	N	N	Y	+	Y	N	256
MS-FA	N	Y	Y	N	Y	+	Y	Y	256
k-DFA	Y	Y	Y	Y	Y	+	Y	Y	256^k
θ-DFA	Y	Y	Y	N	Y	-	Y	N	256
StriDFA	Y	Y	Y	N	Y	-	Y	N	$\frac{256}{w}$

CHAPTER 3

StriDFA: Stride DFA for String Matching

With rapid advancement in Internet technology and usage, some emerging applications in data communications and network security require matching of huge volumes of data against large signature sets with thousands of strings in real time. Being the most widely deployed, firewalls ensure data transfer from trusted sources to destinations by inspecting the packet headers. However, viruses, spam, intrusions and many other forms of malicious content can still outplay firewalls by hiding themselves in the payload of packets. A Network Intrusion Detection System (NIDS) has the ability to inspect both packet headers and payloads to identify attack signatures in order to protect Internet systems. Pattern matching is a key component, as well as the main bottleneck in NIDS to achieving the requirement of real-time processing at wire speed.

Most of the multi-string matching algorithms are derived from the classic Deterministic Finite Automata (DFA). An attractive feature of the DFA algorithm is that it can solve the string-matching problem in a time linearly proportional to

the length of the input stream, and the computation time is independent of the number of strings in the signature set. A major disadvantage of the DFA algorithm is the high memory resources required to store the transition rules of the underlying deterministic finite automaton.

In this chapter, **StriDFA**, a new high speed string matching algorithm is presented. It is based on DFA and uses a stride-based matching approach.

3.1 Introduction

Research in packet processing has been focused on *longest prefix matching (LPM)* to select an entry from the routing table over the last decade. LPM is a search for the longest prefix among those that match an initial substring of a given network address. For example, given an IP forwarding table that consists of a prefix with entry {110, 10001, 11011, 1101}, if the given network address is 11011111, a search of the IP forwarding table will yield matches in the first, third and fourth entries, and the prefix of the third entry (11011) will be selected as the LPM search result because it is the longest of the matching entries [58].

In the recent years, LPM has been used in multi-field packet classification, firewall and quality-of-service (QoS) applications. With the increasing volume of network threats, network operators have started to control the flow of data depending on the information in the packet. Therefore, firewall policies which operate on just the packet header are no longer sufficient. Policies to look deeper in the packet payload to detect if the data is unmalicious are required. This gives rise to another class of search processes: searching for a set of patterns in the input streaming data in network. These patterns can be signature strings or regular expressions for detecting viruses, intrusions, spams or Internet worms. This problem is commonly known as multi-pattern matching and regular expression matching for network security[6].

[6]Regular expression matching will be described in next chapter.

In multi-pattern matching, an input data stream T (which is alternatively called *text*) is compared with a set of strings $P = \{P_1, P_2, \ldots, P_m\}$ (called *patterns*). The aim is to find all the occurrences of any strings of P_i in T. The strings in P are pre-processed and used to build a "machine". Then the streaming data is fed to this machine which then reports matching strings whenever one is found. Basically, string matching can be abstracted as a LPM problem.

Conventional multi-string matching algorithms are impractical for today's network packet inspection [42, 47]. Due to the large pattern database and high speed data detection requirement, an effective detection engine must be able to search for a set of patterns simultaneously, rather than iteratively performing single-pattern matching [59].

Many multi-string matching algorithms have been proposed in the past [51, 60, 61], most of which are derived from the classic Deterministic Finite Automata (DFA). The worst case performance of these algorithms is deterministic, linear to the length of the input stream, and independent of the rule set size. There are two main problems with most existing DFA matching approaches. One is the slow matching speed because, only one input character can be processed at a time; the other problem is the huge memory consumption if too many patterns are combined into an automaton. In this study, a stride-based DFA algorithm (StriDFA) is proposed to solve the above problems, while maintaining the advantage of DFA. Instead of feeding the matching engine with single-byte characters, we feed the StriDFA with the "distance"s between every two adjacent special (*tag*) characters[7]. If the distances can be matched, then there is a possibility that the input stream maybe matched.

StriDFA can achieve faster matching speed than the original DFA with less memory consumption. The main contributions in this chapter are summarized as follows:

[7]The selection of the *tag* character will be discussed in Section 3.7.

- A novel multi-string matching acceleration scheme, StriDFA, is proposed. Its main idea is to convert the original byte stream into a much shorter integer stream and then match the integer stream with a variant of DFA, called StriDFA.
- The formal construction method of StriDFA and the *tag* selection algorithm is described.
- Implement a general instance of StriDFA. It is demonstrated that this instance achieves both space and time efficiency and can be expediently migrated to existing platforms. Approximately 10 times speedup is achievable while the memory cost is also smaller than traditional DFA.

The rest of this chapter is organized as follows. Section 3.2 presents the overall structure of StriDFA and how it works with a simple example. Section 3.3 gives the architecture of StriDFA matching engine. The advantages of StriDFA are presented in Section 3.4 and its problems are listed in Section 3.5. Section 3.6 explains how to get a smaller alphabet set using a sliding window. Section 3.7 describes the tag selection method. The verification module is outlined in Section 3.8. Section 3.9 reports the experimental results on the performance of StriDFA. Section 3.10 concludes this chapter.

3.2 Motivation

In this section, the main idea of StriDFA is demonstrated with an example. For the sake of simplicity, the example used here only considers string matching, which is a special case of regular expression matching. In the next chapter, a general solution will be presented to cover both string matching and regular expression matching.

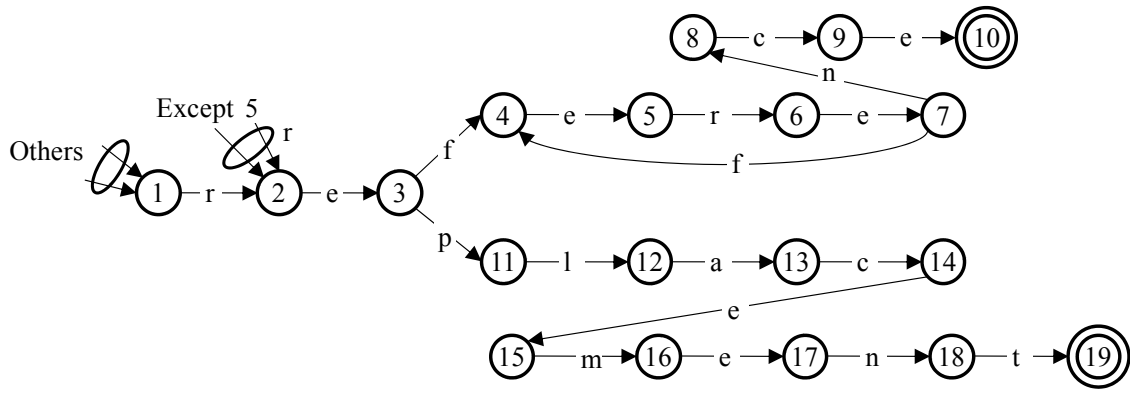


Figure 3.1: *Traditional DFA of patterns "`reference`" and "`replacement`". Some default transitions are omitted for simplicity.*

3.2.1 Traditional DFA in Multi-string Matching

Suppose we have two patterns to match: "`reference`" (P_1) and "`replacement`" (P_2). The conventional scheme is to first covert the patterns to a DFA, which is shown in Figure 3.1. The matching is performed by sending the input stream to the automaton byte by byte. If the DFA reaches any of its accept states, we say a match is found. It is easy to see that the number of states to be visited during the processing is equal to the length of the input stream (in units of bytes) and this number determines the time required for the matching process (each state visit requires a memory access, which is a major bottleneck in today's computer systems).

In this scheme, I want to reduce the number of states to be visited during the matching process. If this objective is achieved, the number of memory accesses required for detecting a match can be reduced, and consequently, the pattern matching speed can be improved. One way to achieve this objective is to reduce the number of characters sent to the DFA.

3.2.2 Stride-based DFA

Instead of comparing the input stream character by character with patterns in the rule set; I pick **tag** characters from the input stream and feed the "fingerprint"

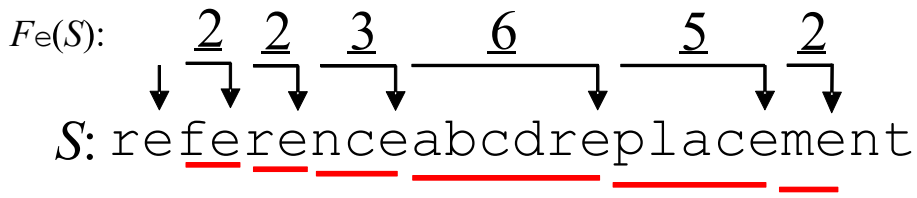


Figure 3.2: *Use tag to convert input stream into SL stream with tag 'e'.*

of these tag characters to the automaton for the matching examination. Since the fingerprint is normally much shorter than the original input stream, the number of state visits required by the matching process can be significantly reduced.

Here, I use distance (or the number of characters) between adjacent tags (denoted as "stride lengths" or step sizes) as the fingerprint. Stride lengths extracted from the rule set are compared with stride lengths extracted from the input strings for coarse grained matching.

For example in Figure 3.2, character 'e' is selected as the tag[8]. Stride lengths extracted from the rule set are compared with stride lengths extracted from the input strings for coarse grained matching.

Definition 1. *Stride Length (SL) is the distance between every two adjacent tags.*

In our scheme, instead of feeding the automaton with single-byte characters, we feed the new SL automaton (StriDFA) with the "distance"s (called *stride lengths (SL)*) between two adjacent tags we find in the input stream.

Definition 2. *A convertor converts the original input stream to its corresponding SL stream.*

Definition 3. *Let $F_x(S)$ denote the SL stream of S when using x as the tag.*

Consider the example in Figure 3.2, the input stream `referenceabcdrepla cement` to be fed into the SL automaton (StriDFA) is $F_e(S)$ = 2 2 3 6 5 2 where 'e' denotes the tag character in use. The underscore is used to indicate a SL, to distinguish it as not being a character.

[8] A more detailed definition of tag is given in Section 3.7.

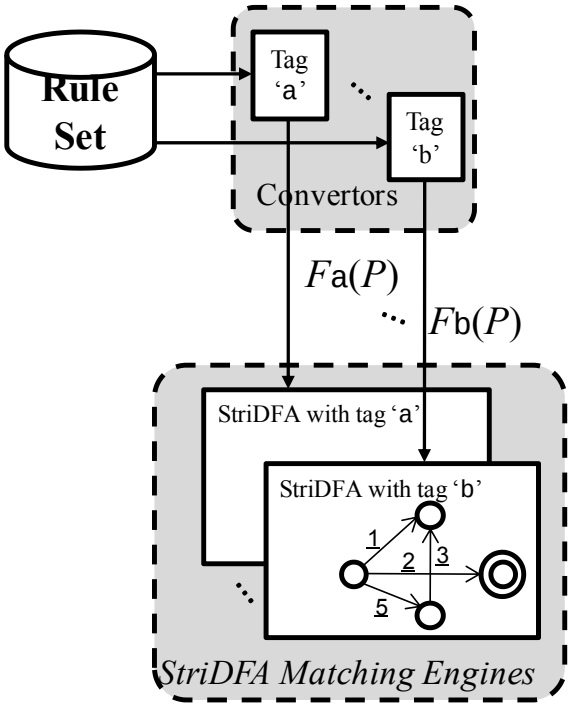


Figure 3.3: *Convert patterns to the corresponding StriDFA.*

Clearly, the volume of processing to be performed by the SL DFA is reduced compared with the original DFA. The original DFA needs to process 24 input characters, while the new SL DFA only needs to process 6 input SLs.

Of course, the DFA needs to be modified in order to handle the input "stride" (the new DFA variant is called as StriDFA). The construction of StriDFA in this example is very simple. What we need to do is first convert the patterns to SL sequences. Then the SL sequences are used to construct StriDFA according to the traditional DFA construction method. Figure 3.3 describes how to construct StriDFA from the original rule set.

As shown in Figure 3.4, the SL of patterns P_1 and P_2 are $F_e(P_1)$ = $\underline{2}\ \underline{2}\ \underline{3}$ and $F_e(P_2)$ = $\underline{5}\ \underline{2}$ with tag 'e'. After obtaining the SLs, I can then use the classical DFA construction algorithm to build the StriDFA.

The original DFA and its corresponding StriDFA associated with the pattern P_1 and P_2 are given in Figure 3.1 and Figure 3.4, respectively. Note that the transitions in the StriDFA are labeled with SLs rather than characters.

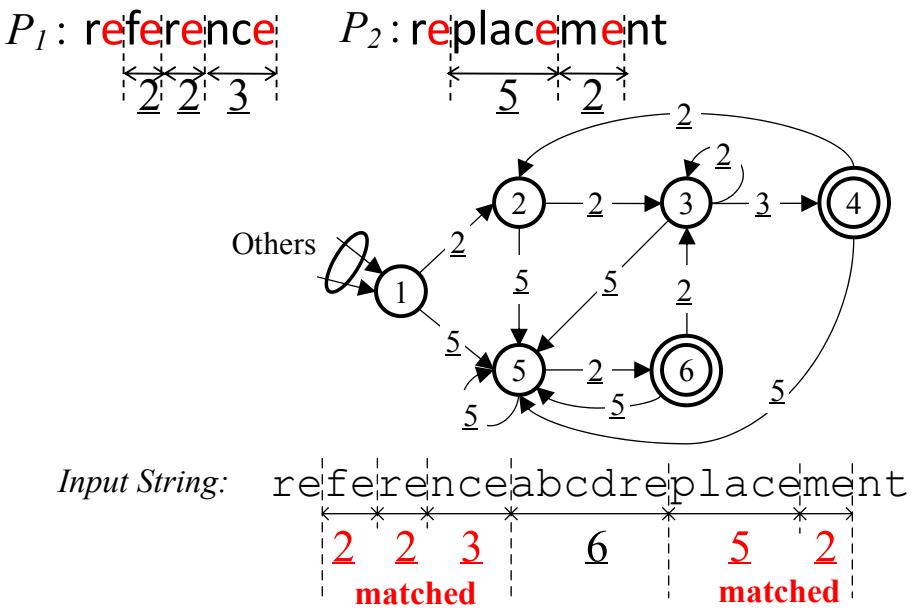


Figure 3.4: *The sample StriDFA of patterns "`reference`" and "`replacement`" with character 'e' as tag.*

3.2.3 Proof of Correctness

In this section, the correctness of StriDFA when making a match is proved. The correctness here means for any given input stream, if the original DFA can be matched, the corresponding StriDFA can be matched too; if the StriDFA cannot be matched, then the original DFA cannot be matched either.

Lemma 1. *If StriDFA cannot be matched, then the corresponding original DFA cannot be matched either.*

Proof. Denote A = {DFA can be matched}, then $\overline{A}$ = {DFA cannot be matched};

B = {StriDFA can be matched}, then $\overline{B}$ = {StriDFA cannot be matched}.

$A \rightarrow B$ will be proved firstly. Assume the original DFA can be matched at the final state of pattern $P = p_1 p_2 \cdots p_m$ by input string T. Then there always exist an i in T that $t_i t_{i+1} \cdots t_{i+m-1} = p_1 p_2 \cdots p_m$. Specifically, it means $t_i = p_1, t_{i+1} = p_2, \cdots t_{i+m-1} = p_m$.

According to the definition of $F_x(S)$ in Definition 3, $F_x(t_i t_{i+1} \cdots t_{i+m-1}) = F_x(p_1 p_2 \cdots p_m)$, that is, the stride length sequences of P have the same stride length sequences from the input string T. In other words, if the original DFA

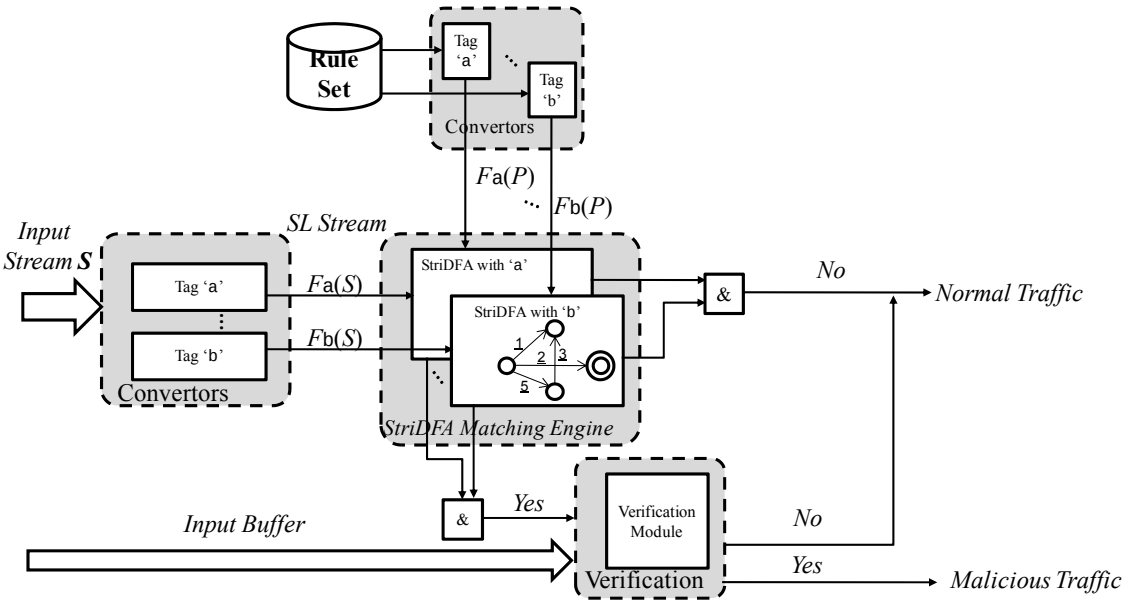


Figure 3.5: *The overall structure of StriDFA.*

can be matched then StriDFA can be matched by the input string T ($A \rightarrow B$).

If a statement is true, the contrapositive is also logically true. Here the statement is that if the original DFA can be matched then the corresponding StriDFA can also be matched is true which has been proved ($A \rightarrow B$). So the contrapositive statement is also true: $\overline{B} \rightarrow \overline{B}$. Finally we have proven that if StriDFA cannot be matched, then the corresponding original DFA cannot be matched either. □

3.3 Architecture of StriDFA Matching Engine

The architecture of a Stride-based DFA (StriDFA) matching system is shown in Figure 3.5, which consists of three main components: SL convertor, StriFA matching engine and verification module.

1) The SL convertors convert the input byte stream into multiple stride-length (SL) streams according to different predetermined tags.

2) The core is a *StriDFA-based* matching engine whose function is to match the input against regex rules, similar to that of a traditional NFA or DFA.

3) Finally the verification phase is used if a potential match is found by all

StriDFAs. The input buffer will be sent to make an accurate match.

In the next two sections, the advantages and challenges of StriDFA-based matching architecture will be analyzed.

3.4 Benefits of StriDFA

Instead of matching the input stream byte by byte, the lengths between specific characters are used to find a potential match in the StriDFA method in order to achieve a high throughput. There are many advantages of using StriDFA.

3.4.1 Increased Matching Speed

The variable stride length DFA matching system proposed in this chapter is a rapid intrusion-detection system. The traditional DFA matching system is like the current airport security system, if it takes one minute to do a full scan of each passenger and his/her carry-on luggage (analogous to packets in NIDS), then 10 passenger would need 10 minutes for full scanning, if one passenger carries illegal items, this will be detected. Our StriDFA is like a new security scan system, which can quickly scan the 10 passengers and their carry-on luggage in just a minute instead of 10 minutes required by the traditional security scan system. There will be no false negatives, so security is not compromised. There will be false positives, which can be further checked using the traditional security system for a full scan. Assuming StriDFA detects two passengers carrying suspicious items, the other eight passengers can be allows to go through the security check point quickly. The two suspected passengers are required to go through further checks. This further check will identify the passenger who carries illegal items, and the other passenger will be identified as a false alarm, and will be allowed through the security check point. The false positive rate can be controlled at a low level, so that most passengers can go through the security check quickly without

Table 3.1: *Memory consumption between traditional DFA and StriDFA for "`reference`" (P_1) and "`replacement`" (P_2) with $w = 5$.*

Finite Automaton	State No.	Transition No.	$\sum$
Tradition DFA	19	19	256
StriDFA	6	13	5

wasting time waiting.

Instead of feeding the automaton with single-byte characters, the stride stream of the input stream will be sent to the StriDFA matching engine. The lengths between properly chosen tags are relatively long in real trace. Normally the average SLs of real trace are larger than 200. Therefore, a high matching speed can be achieved by a StriDFA matching engine.

3.4.2 Small Memory Requirement

Compared with traditional string matching methods, the StriDFA is more compact in memory usage for two reasons. Firstly, the number of states is generally less than its corresponding traditional DFA (*e.g.*, StriDFA in Figure 3.4 has 13 less states than the traditional DFA in Figure 3.1). Secondly, as shown in Table 3.1, the fanout of each state is controlled by the window size[9], which is generally far smaller than the fanout of a traditional DFA (256 in standard ASCII). With fewer states and a more compact state-transition table, the memory requirement is greatly reduced compared with traditional DFA methods.

3.4.3 Easily Implemented on Existing Platforms

Unlike sophisticated methods or those needing auxiliary memory ([62]), StriDFA can be easily implemented on existing hardware or software, since the StriDFA has exactly the same logic structure as a traditional DFA. Therefore, throughput can be further improved when high performance hardware-based architecture in FPGA or TCAM is used.

[9]The detail of window is explained in Section 3.6.

3.5 Challenges

With increased speed, small memory consumption and straighforward implementation on existing platforms, the advantages of StriDFA are evident. However, StriDFA also has some challenges.

3.5.1 Tag Selection

One of the challenges for StriDFA is how to choose an appropriate tag. In both the rules and the incoming traffic, the frequency probabilities of different characters vary. The problem of choosing an appropriate tag from a rule set is outlined later in this chapter: a definition of how tags cover regex rules is first given in subsection 3.7.1 and a greedy algorithm for tag selection is then proposed in subsection 3.7.2.

3.5.2 Potentially Infinite Alphabet Set

The distance between any two tags in the input stream can be 100, 200 or even larger than one thousand. This leads to the problem of stride lengths being arbitrarily large. Furthermore, the stride length between two tags in a regex which contains a wildcard (i.e., '.*') may be arbitrarily large. With an unlimited SL stream feeding the corresponding StriDFA, the alphabet set is infinitely large. In fact, it cannot construct a StriFA with arbitrary large stride lengths because there may exist infinitely outgoing transitions.

3.5.3 Rate of False Alarm

Since the SL stream is a highly compressed form of an input stream, part of the information is left out before being sent to StriDFA. Therefore, it is only a potential match if StriDFA reports a match, causing a false positive. For example, if given two strings T="efe" and T'="ere", we have $F_{\text{e}}(T') = \underline{2} = F_{\text{e}}(T)$. If the stride

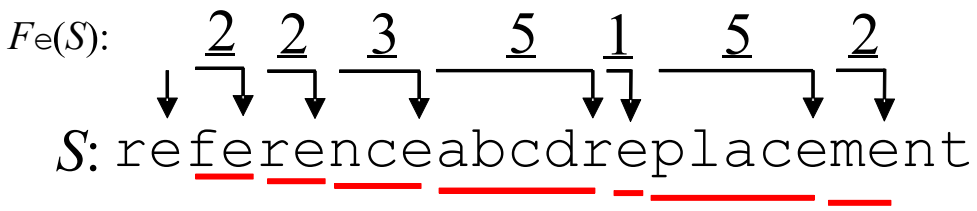


Figure 3.6: *The stride length stream with window w=5.*

length $\underline{2}$ is matched, it is not sure if it is T or T' that has been matched. For this reason we have to add a verification module to confirm any potential match.

3.5.4 Regular Expression Support

The example here only discusses the case of string signatures. However, it is not trivial to convert a regular expression signature into its corresponding stride lengthes, due to regex's powerful use of multiple wildcard characters (e.g., '.', '*'), length restrictions ('?', '+') and groups of characters ('[]', '^'). Finding a method to preserve the expressiveness in StriFA raises a significant challenge. For example, given a regex `.*abba.{2}caca`, the strides among different tags are undecidable because '.*' can match any character any number of times. In the next chapter (Chapter 4), a general solution is presented for both string matching and regular expression matching.

3.6 Limit Alphabet Set by a Window

To solve the problem of the large alphabet set, a fixed size sliding window is adopted (a similar application can be found in [51]). The window works in the following way (see Figure 3.6): if a tag is not found within a window width since the last tag, then the last character of the window is marked as a fingerprint anchor (a fingerprint anchor is not a tag, but is treated like a tag to get the SL from the previous tag), the window size w is sent to StriDFA and the character following the fingerprint anchor is set to be the beginning of the next window. In this

manner, any SL sent to a StriDFA must be in a finite alphabet set $\Sigma = \{1, \ldots, w\}$.

3.7 Tag Selection Approach

One of the problems for StriDFA is how to choose an appropriate tag. Since in both the rules and the incoming traffic, the frequency probabilities of different characters vary, it is a challenge to choose an appropriate tag from a rule set.

Minimizing false positive rate while preserving other performances (i.e., throughput and memory usage) is analyzed in this section. High false positive rate leads to frequent use of the verification module, degrading the overall throughput. Although the idea and core mechanism is simple and straightforward, StriDFA is a very complicated system as a whole. Several optimizations have been adopted in the StriDFA-based matching system which is described here. As shown in Figure 3.7, the discrepancy of character frequency is different in the Snort and ClamAV rule set.

Essentially, the optimization of reducing false positive rate is a balance be-

Algorithm 1: Algorithm of SL extraction

Input: $String\,T : t_1 t_2 \cdots t_n$, window size w, tag x
Output: $StrideLengthstream : SLs$

$SLs \leftarrow \emptyset, counter \leftarrow R, i \leftarrow 1, isFirstTag \leftarrow false, pos \leftarrow 0, \ell \leftarrow 0$
while $i \leq n$ **do**
    if $t_i = x$ **then**
        if $isFirstTag \neq false$ **then**
            $\ell \leftarrow i - pos$
            while $\ell > w$ **do**
                $SLs \cup \{w\}$
                $\ell \leftarrow \ell - w$
            end
            $SLs \cup \{\ell\}$
        else
            $isFirstTag \leftarrow true$
            $pos \leftarrow i$
end
return SLs

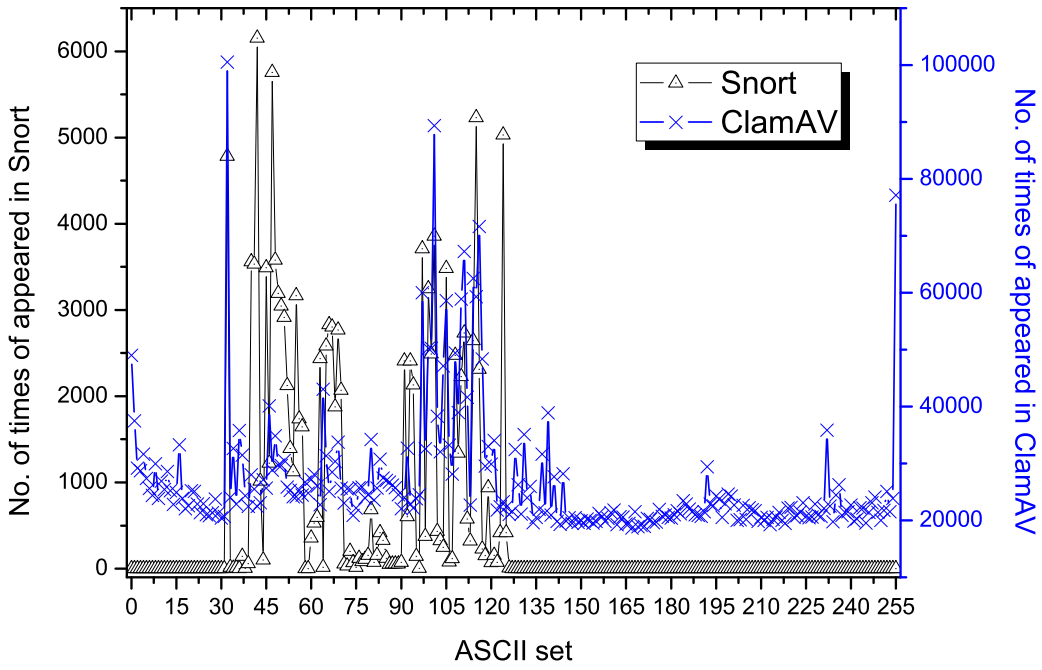


Figure 3.7: *Frequency of appearance for each characters in Snort and ClamAV rule sets. X-axis is the ASCII code (from 0 to 255) of the ASCII table set.*

tween two extremes. On the one hand, to reduce the false positive rate to zero, all possible characters should be used as tags which the convertor checks (that is, 256 in the worst case) and builds a StriDFA for each of them; however, this will degrade speedup to one (as every input character invokes an access to some StriDFAs) and lead to large memory consumption (linear with the number of tags). On the other hand, to achieve a high throughput and low memory usage, it is expected to use as few tags as possible; however, the possibility of false positives can increase significantly since a large number of the input characters are left out[10].

To strike a balance, a small group of proper tags is selected so that each rule can be "covered" by some tags in the group. In the following two subsections, first the definition of how a pattern is "covered" by some tags is given , and then the selection of tags is addressed to achieve both space-time efficiency and lower

[10]The last character in the window is marked as a fingerprint anchor if there is no tag in the window.

false positive rate.

3.7.1 How tags "cover" a pattern

The character which has most frequent occurrences in a pattern can extract more SLs. Intuitively, more SLs could express more information for the original pattern. For example, considering pattern P_1="`reference`", $F_e(P_1)$ = $\underline{2}$ $\underline{2}$ $\underline{3}$ and $F_r(P_1)$ = $\underline{4}$. $F_e(P_1)$ has more SLs when using 'e' as the tag, while $F_r(P_1)$ only has one SL with tag 'r'.

Definition 4. *$Freq(c, P)$ refers to the number of occurrences of character c in pattern P over the length of pattern P.*

$$Freq(c, r) = \frac{\#c \text{ in } P}{|P|}$$

Here $\#c$ in P is the number of occurrences of c in pattern P and $|P|$ is the length of pattern P. According the above definition, it is easy to get $Freq(e, P_1) = \frac{4}{9}$ = 0.44 and $Freq(r, P_1) = \frac{2}{9}$ = 0.22.

In the work described in this thesis, the most frequent character in a pattern is chosen to be a tag for this pattern. Of course, other strategies can be used for tag selection according to different situations of the rule set.

Then the definition of how tags "cover" the rules is given as follows.

Definition 5. *A pattern P is covered by a set of tags, TAG, if the frequency of TAG in P exceeds a predefined* threshold *θ.*

For example, $Freq(e, P_1) = 0.44$ while $Freq(r, P_1) = 0.22$ in the above example. If $\theta = 0.3$, then P is only covered by 'e' (the default $\theta = 0.3$). The θ has been tested from 0.1 to 0.5, the result shows that 0.3 could work well. Actually, people could change θ according to different situation. The reason of using $Freq(c, P)$ to select tags is twofold. First, $Freq(c, P)$ has the additive property; that is, the frequency of a set of characters in pattern P is the sum of $Freq(c, P)$ over all c in the character set. So it is very simple to calculate the frequency of a set of characters in all rules. Second, with higher $Freq(c, P)$, the possibility of false positive is

lower because more parts (*i.e.*, more characters) of the pattern are checked by the chosen set of tags.

3.7.2 Greedy algorithm for tag selection

The objective of tag selection is that every rule is covered by some chosen tags. However, this task is non-trivial, since each rule can be covered by several combinations of characters (*i.e.*, multiple combinations of tags are available), and in fact to select the least number of tags is known to be an NP-hard problem [63].

A greedy algorithm is used to select tags. With greedy algorithm, people can make whatever choice seems best at the moment and then solve the subproblems that arise later. Actually, the users could try their own algorithms to select tags. This is also a future work to test the performance among different tag selection algorithms. A character is selected as a tag if its covers the maximum number of remaining uncovered rules. This step is repeated until all rules are covered, as presented in Algorithm 2.

Algorithm 2: Algorithm of tag selection

Input: $Freq : \Sigma \times R \mapsto [0,1], \theta$ /* Σ: alphabet set (e.g., ASCII), R: regex rule set, θ: the frequency threshold */
Output: $TAG \subseteq R$

$TAG \leftarrow \emptyset, UNCOV \leftarrow R$
while $UNCOV \neq \emptyset$ **do**
    $c \leftarrow \text{maxarg}_{c \in \Sigma} \sum_{r \in UNCOV} Freq(c, r)$
    /* get the character c which has maximum $Freq(c, r)$ of the rules in the rule set $UNCOV$. */
    $TAG \leftarrow TAG \cup \{c\}, \Sigma \leftarrow \Sigma \setminus \{c\}$
    /* add c to the tag set TAG and delete c from θ. */
    $UNCOV \leftarrow UNCOV \setminus \{r | \sum_{c \in TAG} Freq(c, r) > \theta\}$/* delete the rules covered by c. */
end
return TAG

3.8 Verification module

Since the SL stream is a highly compressed form of an input stream, part of the information is left out before being sent to StriDFA. We have to make an exact match in the verification module to confirm all the potential matches.

When the StriDFA reports a possible match, the verification module is triggered to start the exact match. Instead of matching the whole buffer, only part of the input stream needs to be sent to the verification module. Considering that the memory consumption of NFA is much less than the corresponding DFA, NFA can be used in the verification module.

If the verification module also reports a match, it means there is a real match from the input stream. Otherwise the input stream is recognized as normal traffic because it cannot be matched by any rules of the rule set.

Definition 6. *$L_{head}(P)$ is the length from the first character of P to the first tag. $L_{tail}(P)$ represents the length from the last tag of P to the end of the pattern. Similarly, $L_{StriDFA}$ is sum of all the stride lengths passed during matching the initial state to the final state in StriDFA.*

Property 1. $L_{head}(P) \leqslant |P|; L_{tail}(P) \leqslant |P|$.

L_{head} and L_{tail} can be calculated by the pattern itself directly. For instance, for a given pattern P_2="`replacement`", $L_{head}(P_2)$ = 1 (there is only one character 'r' before the first tag 'e') and $L_{tail}(P_2)$ = 2 (there are two characters 'nt' after the last tag 'e').

Specially, $L_{StriDFA}$ records the path of the StriDFA during the matching until the final state is reached. According to the above definition of $L_{StriDFA}$ in definition 6, given the input string `eabcreplacement` (the SLs which feed to the StriDFA are 5 5 2), we have: $L_{StriDFA}(P_2)$ = 5 + 5 + 2 = 12.

When the StriDFA reports a possible match at position t in the buffer, the verification module is triggered to start an accurate backward match from position $t + L_{tail}(P_2)$ backward to position $L_{tail}(P_2)$ + $L_{StriDFA}(P_2)$ + $L_{head}(P_2)$ = 15 charac-

ters. Instead of matching the whole buffer, there are only 15 characters that need to be fed to the verification engine.

3.9 Evaluation

Messages on the Internet are broken down into small units called packets. Each packet contains a header and a data field. Similar to NIDS, the StriDFA-based matching engine opens and reads the data field in real time to do the state-aware inspection (as shown in Figure 3.5). The instance is implemented on a software platform, and in particular, StriDFAs are realized using standard finite automata transition-table storage.

The performance of a StriDFA-based matching system, including throughput, memory size, filter rate and false positive rate, is influenced by four aspects: (a) the input stream (trace), (b) the string patterns, (c) the window size and (d) the selected tags. The memory consumption is influenced by (b)(c) and (d); speedup is influenced by (a)(c) and (d); and the overall throughput is influenced by (a)(b)(c) and (d). Similarly the filter rate and false positive rate have a relationship with (a)(b)(c) and (d).

3.9.1 Experiment setup

1,000,000,000 characters (5,000,000 lines, 200 characters for each line) are generated randomly and are used as input stream. The patterns P_1 and P_2, previously defined, are employed to match the input stream to compare the performance between a traditional DFA and the StriDFA.

Two classic quotes are selected from the book "Harry Potter" as patterns to search over 600 famous novels in the history. The patterns are "`It does not do to dwell on dreams and forget to live, remember that.`" (P_3) and "`it really is like going to bed after a very, very long day. After all, to the well-organized mind, death is but the next`

`great adventure.`" (P_4) [64]. According to the definition of $F_x(S)$ in Definition 3, $F_e(P_3)$ = 15 9 13 9 4 2 3. Similarly the stride length sequences of P_4 is $F_e(P_4)$ = 12 12 6 6 6 17 13 3 11 10 14 3 6 7 5.

The experiments are carried out on two desktop PCs, each has eight 3.8GHz CPUs and 12 GB memory.

3.9.2 Memory consumption and speedup

Because there is no explosion caused by the wildcards (like ".*") in string matching, the memory consumption is not a big problem for string matching. Figure 3.8 shows that the memory usage decreases as the window size increases until w=5. Larger than 5, the memory usage stays at about the same level. This is because the stride length sequences are getting longer if a smaller window size is used. For example, $F_e(P_2)$ = 5 2 with w = 5 while $F_e(P_2)$ = 3 2 2 with w = 3. Obviously, 3 2 2 need more states and transitions when constructing a DFA. However, if w is bigger than the maximum stride length of a pattern, then the stride length sequences will not change with further increase of the window size. $F_e(P_2)$ = 5 2 with w = 10, 15, ...

Similarly as shown in Figure 3.9, the memory usage stays stable in P_3 with w = 15 and P_4 with w = 20.

The process of converting rule sets to SLs and constructing StriDFA can be done offline before the matching[11]. The overall speedup here means the ratio between the matching time of traditional DFA and the processing time of StriDFA + the time for the verification procedure. As shown in Figure 3.8 and Figure 3.9, the overall speedup is around 8 times.

3.9.3 Filter rate and false alarm rate

Figure 3.10 shows the false positive rate staying the same (0.42%) when $w \geq 5$. This is because the structure of StriDFA for P_1 and P_2 does not change when $w \geq 5$

[11]The detail converting complexity will be analyzed in subsection 4.5.3.

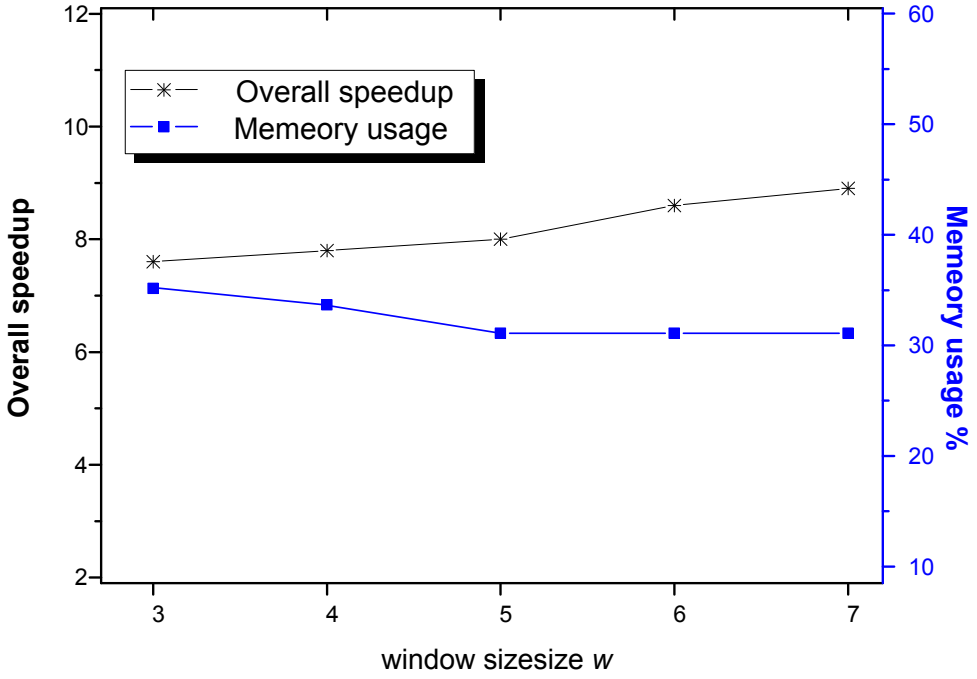


Figure 3.8: *Overall speedup and memory usage of the StriDFA (P_1 and P_2) with different window sizes.*

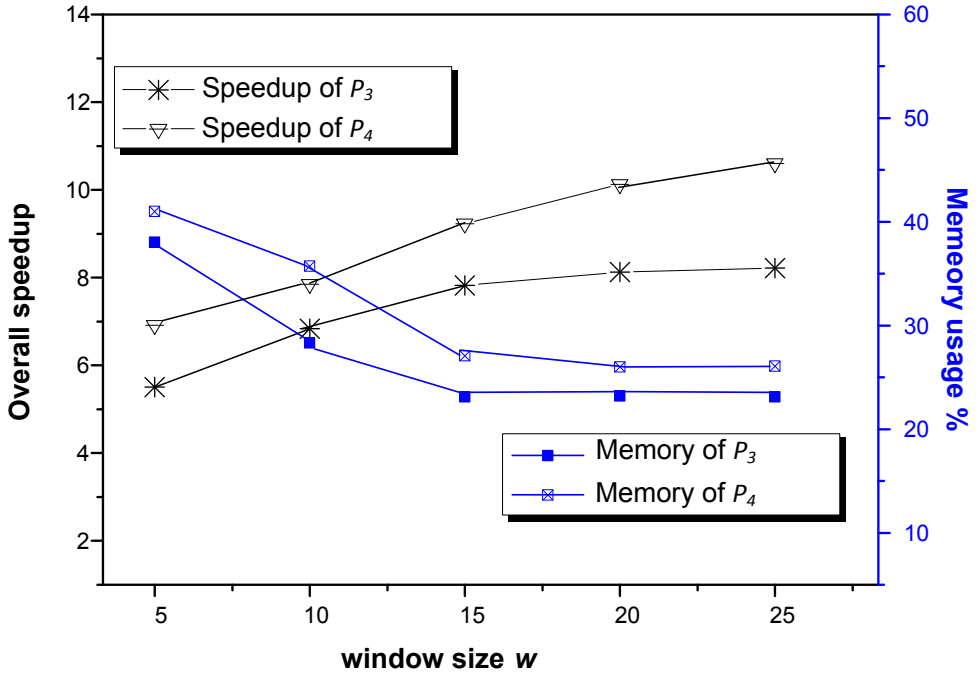


Figure 3.9: *Overall speedup and memory usage of P_3 and P_4 with different window sizes.*

because the maximum SL of P_1 and P_2 are smaller than 5. Since P_1 and P_2 are short patterns, the stride length sequences of P_1 and P_2 have a high probability of being matched by the incoming stream. P_1 is matched 23,373 times during the test and the possible number of matches for P_2 is 46,036. However, the false

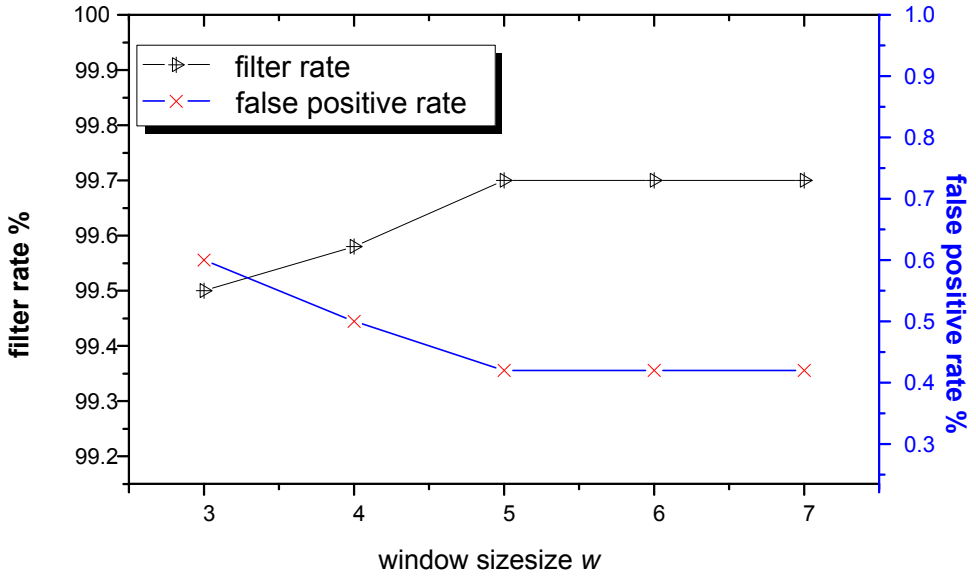


Figure 3.10: *Filter rate and false positive rate of the StriDFA (P_1 and P_2) with different window sizes.*

positive rate is 0 for P_3 and P_4. It is easy to understand that with longer stride length sequences, the false positive matches are fewer.

3.10 Conclusion

A novel acceleration scheme for multi-string matching is proposed in this Chapter. This novel Stride-based matching scheme converts the original byte stream into a much shorter integer stream and then matches it with a variant of DFA, called StriDFA. The formal construction of StriDFA was given, which transforms any set of patterns to a corresponding StriDFA. Also, the method used to produce the SL sequences is described, so that the false positive rate can be reduced to an acceptable level. The experimental results show that a StriDFA-based matching architecture achieves an 8 to 10 times increase in speed with less memory consumption than the traditional DFA.

CHAPTER 4

StriNFA and StriDFA for Regular Expression Matching

Deep Packet Inspection (DPI) has become a key component in Network Intrusion Detection Systems (NIDSes) where every packet of the incoming data stream needs to be compared with all the patterns in the current attack database byte-by-byte using either string matching or regular expression matching. Regular expressions matching, despite its flexibility and efficiency in attack identification, brings significantly higher computation and storage complexities to NIDSes, making line-rate packet processing a challenging task.

In this chapter, Stride Finite Automata (StriFA), a novel finite automata family, is presented to accelerate regular expression matching. Compared with conventional finite automata, which scan the entire traffic stream to locate malicious information, a StriFA only needs to partially scan a traffic stream to find the suspicious information. The presented StriFA technique has been implemented in software and evaluated based on different traces. The simulation results show that the StriFA acceleration scheme offers an increased speed over traditional NFA/DFA while at the same time reducing the memory requirement.

4.1 Introduction

Deep Packet Inspection (DPI) has been widely deployed in modern Network Intrusion Detection System (NIDS) to detect attacks and viruses based on patterns stored in a database. Examples include Snort [36], ClamAV [65], and security applications from Cisco Systems [66]. The format for writing these patterns is either strings or regular expressions (regex). To support increasingly complex services, regular expressions have been used to replace strings in DPI because of their better expressiveness and flexibility. However, regular expression matching brings significantly high computation and storage complexities to NIDS, which can prohibit its usage in many of the applications that require high processing speed with limited memory. Designing a regex matching engine that achieves both time and space efficiency is a significant challenge.

Deterministic finite automaton (DFA) and Non-deterministic finite automaton (NFA) are two typical finite automata used to implement regular expression matching. DFA is fast and has deterministic matching performance, but suffers from the memory explosion problem [67]. NFA, on the other hand, requires less memory, but suffers from slow and non-deterministic matching performance. Therefore, neither DFA nor NFA is suitable for independently implementing high-speed regular expression matching in environments where fast memory (e.g., cache or on-chip memory) is very limited.

Recently, much research work has focused on improving the speed and/or reducing the memory cost of regular expression matching [48, 61, 68]. These schemes can be roughly classified into two categories: (1) single-byte stride[12] scanning and (2) multi-byte stride scanning. Traditional NFA and DFA along with some of their variations, such as HybridFA [69], CDFA [67], D^{2}FA [39], CD^{2}FA [70] and XFA [38, 62], scan only one character at a time and belong to the first category. The main research focus in these schemes is to (i) reduce the

[12]If n characters can be scanned in one step, we say the stride or length of the step, is n.

number of active states of the automaton during matching, which in turn reduces the number of memory accesses, resulting in improved matching speed, or (ii) reduce the memory consumption by reducing the state number or transition number of the automaton. Schemes in the second category scan multiple characters at a time and therefore naturally provide better performance than those in the first category. However, most schemes in this second category suffer from two problems:

1) *memory blow-up problem* due to the exponential growth of transition numbers when the stride increases.

2) *byte alignment problem* which means that for multi-byte scanning that every character of the input stream should have the chance to be examined as the first character. This requires duplicate automata to return the correct matching results.

Some other schemes in the second category do overcome the memory blow-up and byte alignment problem, but can only be used to handle string matching rather than regex matching [51].

In this chapter, I undertake the problem of designing a variable-stride pattern matching engine that can achieve an ultra-high matching speed with a relatively low memory usage. More specifically, a Stride Finite Automata (StriFA) is proposed to handle both string matching and regular expression matching. Compared with other algorithms that also examine multiple characters at a time, StriFA is immune to the memory blow-up and byte alignment problems, and therefore requires much less memory. The proposed StriFA is a family of automata and language sharing the same concept which includes StriDFA (Stride Deterministic Finite Automaton), and StriNFA (Stride Nondeterministic Finite Automaton). StriDFA for multi-string matching has been introduced in Chapter 3. StriNFA and StriDFA for regular expression matching is described in this chapter.

Moreover, StriFA can be expediently deployed on existing hardware/software

platforms, as StriFA shares the same I/O interfaces and logic structure as traditional NFA/DFA built directly from the regex set.

In summary, the main contributions in this chapter are:

- The concept of StriFA, a novel acceleration scheme for both regex matching and string matching is proposed. The main idea of the scheme is to convert the original byte stream into a much shorter integer stream and then match the integer stream with a variant of NFA/DFA, called StriNFA/StriDFA, to identify the potential matches in the original input byte stream.

- The formal construction of StriNFA and StriDFA is described. Moreover, a formal algorithm is proposed to convert traditional NFA directly to StriNFA without the need of DFA. The memory explosion problem originally involved in the DFA construction can be avoided by this method.

- An improved version of StriFA which utilizes the "neighbor" information to reduce its false alarm rate at the cost of a small increase in memory is proposed.

- An implementation of a general instance of StriFA is demonstrated. This instance achieves both space and time efficiency and can be expediently migrated to existing platforms. Results show that approximately a 10 fold increase in speed is achievable while the memory cost is smaller than for traditional NFA/DFA.

The rest of the chapter is organized as follows. Section 4.2 presents the problem of constructing StriDFA with regex. Section 4.4 gives the formal construction of StriNFA and StriDFA. Analysis and optimization are addressed in Section 4.5. Section 4.6 reports the experimental results on the performance of StriFA. Section 4.7 concludes this chapter.

4.2 Problem Statement

In the previous chapter, for a given string pattern P_1 = `reference`, it is easy to extract the SL sequences of P_1 because the distance between tags are easily determined. However, this is a much more difficult task in regex matching.

Suppose the regex rule is `.*abba.{2}caca`. JFlex syntax for regex is used in this thesis, i.e., ".*" can match any character any times. By default, the 8-bit input character set is used for analysis. It matches any contiguous part of the input stream that starts with `abba`, followed by two arbitrary characters, and finally ends with `caca`. Using 'a' as the tag, `.{2}` two arbitrary characters which could be `[^a][^a]`, `a[^a]`, `[^a]a`, or `aa`. So possibly the SL stream of F_a(`abba.{2}caca`) = 3 4 2, 3 1 3 2, 3 2 2 2 or 3 1 1 2 2. If the window size w = 3, thus F_a(`abba.{2}caca`) = 3 3 1 2, 3 1 3 2, 3 2 2 2 or 3 1 1 2 2. The initial part of regex `.*abba.{2}caca` is ".*" which could match any characters any number of times. It is impossible to extract SL from the regex `.*abba.{2}caca` directly.

4.3 Stride Finite Automaton

4.3.1 Building StriFA by DFA-based method

A formal method is given here to construct a StriNFA from a regex. It involves several steps as shown in Figure 4.1:

1) building a NFA from a regex using a traditional method ;

2) transforming the NFA to the corresponding DFA;

3) restructuring the DFA to a tag decision finite automata (tag decision FA);

4) transforming the tag decision FA to StriNFA.

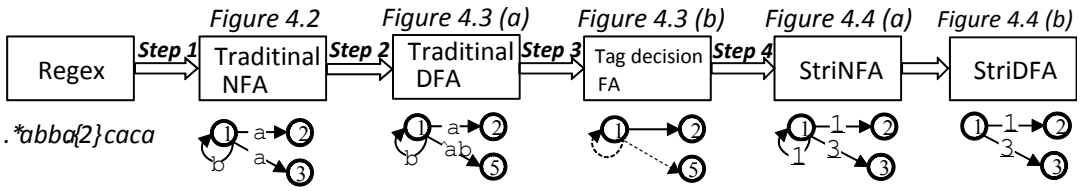


Figure 4.1: *Flow chart describing the steps to convert a regex to a StriFA via DFA.*

5) transforming the StriNFA to the final StriDFA (similar to step 2 that transform a NFA to DFA).

Step 1: Compile a regex to its corresponding NFA

The way of compiling a regex to a corresponding NFA is conventional (intensively studied in [71]). Figure 4.2 is the traditional NFA of the regex `.*abba.{2}caca`.

Step 2: Convert a NFA to its corresponding DFA

The method of converting a NFA to its corresponding DFA is clear. As discussed in subsection 1.1.5, the method that converts an NFA to a DFA is explained in [20]. Figure 4.3(a) shows the traditional DFA of the regex `.*abba.{2}caca`.

Step 3: Restructure a DFA to a Tag decision FA

In this step, each transition is drawn as a solid line if its label is the tag or is drawn as a dotted line otherwise. Then all labels are removed from transitions of the traditional DFA. The output structure is named as a tag decision FA (shown in Figure 4.3(b)) after transformation from Figure 4.3(a) using tag '`a`'.

Take Figure 4.3(a) as an example, the outgoing transitions of state 5 are: $5 \xrightarrow{[\hat{}ab]} 6$, $5 \xrightarrow{a} 7$ and $5 \xrightarrow{b} 8$ ([^ab] matches any character other than 'a' or 'b'. Details can be found in Table A.1). Depending on whether the label is the tag 'a', the outgoing transitions of state 5 are redrawn as $5 \dashrightarrow 6$, $5 \longrightarrow 7$ or $5 \dashrightarrow 8$ (as shown in Figure 4.3(b)).

Step 4: Transform Tag decision FA to StriNFA

In this step we generate a Stride non-deterministic FA (StriNFA) from the Tag decision FA (a directed graph consisting of solid and dotted transitions). Non-

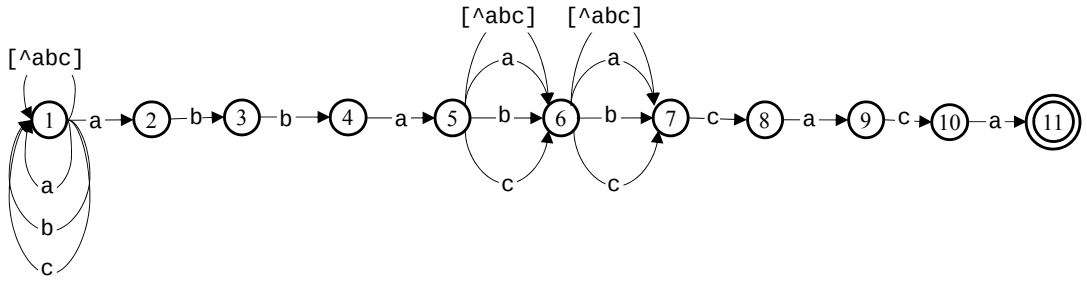
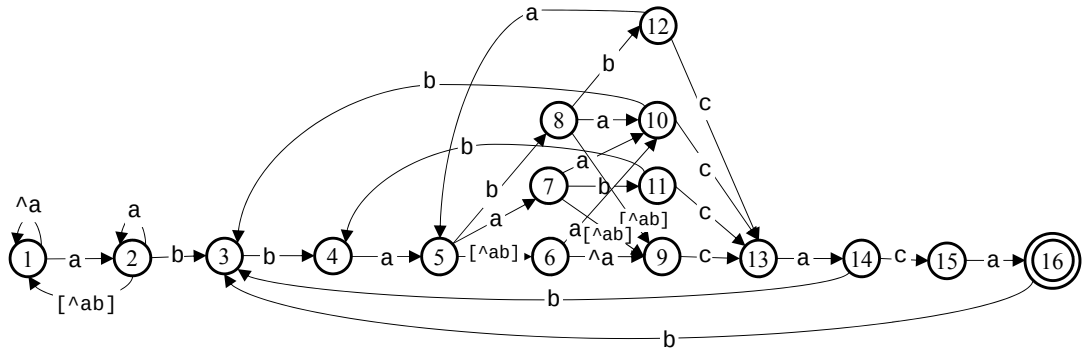
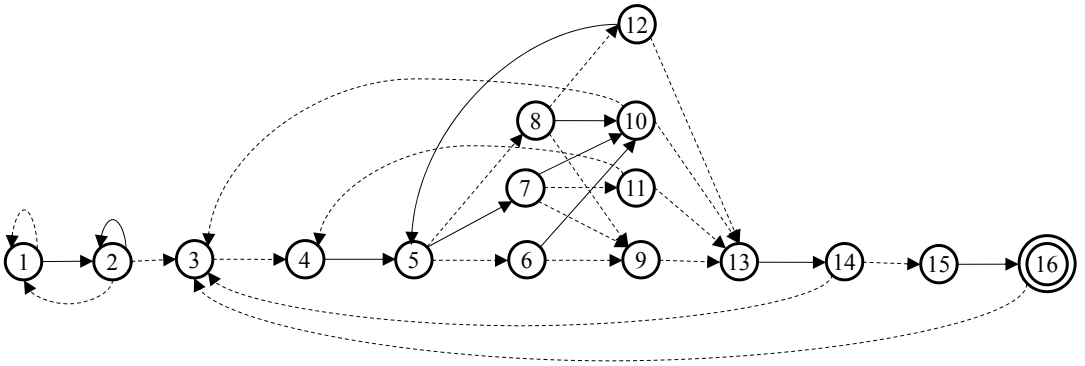


Figure 4.2: *Traditional NFA of regex* `.*abba.{2}caca`.



(a) Traditional DFA of regex `.*abba.{2}caca`.



(b) Tag decision FA of regex `.*abba.{2}caca`.

Figure 4.3: *Traditional DFA of regex `.*abba.{2}caca` and its corresponding Tag decision FA.*

deterministic means some states can have more than one outgoing transition labeled with the same SL (integer). To explain the method, the following steps are processed recursively: starting from any state p in a Tag decision FA,

- **case 1**: if a solid transition (pointing to state q) is reachable in l steps where $l \leq w$, add a transition labeled l from p to q with label l (stride length);
- **case 2**: otherwise if there is a all-dotted-transition path of length w to state q, then add a transition labeled w from p to q with label w.

The first case applies when the SL convertor produces one SL with value of $l(l \leq w)$, and the case 2 applies when the convertor finds no tag in the window and then uses the window size w as the SL. The basic operation can be done by a depth first search with a maximum depth of w. For the whole graph, the basic operation is processed iteratively, starting from the initial state to find all the reachable states and build transitions between them.

The pseudocode for transforming a Tag decision FA to a StriNFA is given in Algorithm 3. The function **move()** in line 14 takes a state and an outgoing transition type, and returns the state reachable by this transition.

Take state $\{2\}$ in Figure 4.3(b) as an example: starting with state $\{2\}$, the related transitions within w (here w=3) steps are shown in the following.

$$\{2\} \longrightarrow \{2\}$$
$$\{2\} \dashrightarrow \{1\} \longrightarrow \{2\}$$
$$\{2\} \dashrightarrow \{1\} \dashrightarrow \{1\} \longrightarrow \{2\}$$
$$\{2\} \dashrightarrow \{3\} \dashrightarrow \{4\} \longrightarrow \{5\}$$

According to the above recursive procedure in Step 4, if a solid transition (pointing to state q) is reachable in l steps where $l \leq w$, add a transition labeled l from p to q with label l. All the corresponding outgoing transitions of state $\{2\}$ are listed as follows.

$$\{2\} \xrightarrow{\underline{1}} \{2\}$$
$$\{2\} \xrightarrow{\underline{2}} \{2\}$$
$$\{2\} \xrightarrow{\underline{3}} \{2\}$$
$$\{2\} \xrightarrow{\underline{3}} \{5\}$$

Obviously, state 2 has closure transitions labeled with $\underline{1}$, $\underline{2}$ and $\underline{3}$. There is another transition between state 2 and state 5 labeled with $\underline{3}$. The above outgoing transitions can be found in Figure 4.4.

4.3.2 StriNFA to StriDFA

StriNFA is a special kind of NFA in which the labels are all integers. Transforming from StriNFA to StriDFA is equivalent to the procedure of transforming from

Algorithm 3: Algorithm of transforming Tag decision FA to StriNFA via DFA

1 **Procedure** $BUILD(NFA, w)$
2 **Input:** $NFA = (Q, \Sigma, \delta, q_0, F), w, tag$; /* q_0: initial state and F: set of final states, w: window size */
3 **Output:** StriNFA $A = (Q', \Sigma', \delta', q_0', F')$
4 $Q' = \delta' = F' = \emptyset, \Sigma' = \{1, \ldots, w\}$

5 $S_i = FindInitialState(Q, q_0)$

/* get initial state. */
6 $Explore(S_i, S_i, w, Q, \delta, F)$
7 **return** $A = (Q', \Sigma', \delta', q_0', F')$

8 **Procedure** $FindInitialState(Q, q_0)$; /* Find the initial state */
9 $S := q_0$
10 **for** $e = (S, S') \in D^+(S)$ **do**
/* $D^+(S)$: outgoing transitions set of the current state set S */
11 **if** e = $\rightarrow$ **then**
12 **return S′**
13 **else if** e = $\dashrightarrow$ **then**
14 $S' := move(S, \dashrightarrow)$
15 $FindInitiaState(Q, S')$
16 **end**

17 **Procedure** $Explore(S_0, S, depth, Q, \delta, F)$
18 **if** $depth = 0$ **then**
19 $Q := S \cup Q$
/* add a new state S to finite set Q */
20 $\delta := \delta \cup (S_0, w, S)$; /* add a new transition $S_0 \xrightarrow{w} S$ to transition function δ */
21 $Explore(S, S, w, Q, \delta, F)$
22 **else**
23 **for** $e = (S, S') \in D^+(S)$ **do**
24 **if** e = $\dashrightarrow$ **then**
25 $depth := depth - 1$
26 $S' := move(S, \dashrightarrow)$
27 $Explore(S_0, S', depth, Q, \delta, F)$
28 **else if** e = $\rightarrow$ **then**
29 $S' := move(S, \rightarrow)$
30 $\delta := \delta \cup (S_0, w - depth, S')$; /* add a new transition $S_0 \xrightarrow{w-depth} S$ to transition function δ */
31 **if** $S' \notin Q$ **then**
32 $Q := S' \cup Q$
33 $Explore(S', S', w, Q, \delta, F)$
34 **end**

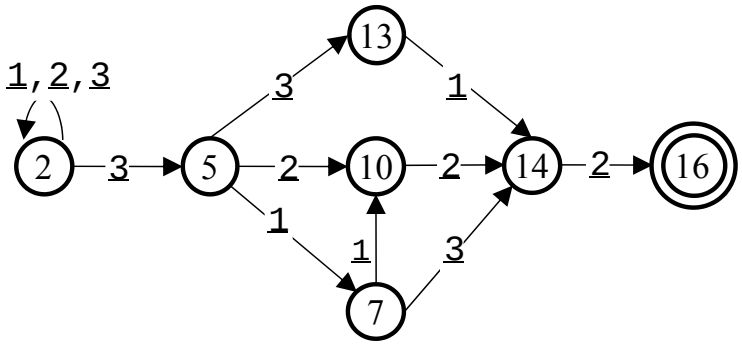


Figure 4.4: *StriNFA of the regex* `.*abba.{2}caca` *before renumbering.*

NFA to DFA. One of the traditional transform algorithms was proposed by K. Thompson [22], which is also called "structural induction" in textbooks [20].

As shown in Figure 4.5(a), the StriNFA is generated after state renumbering from Figure 4.4. The corresponding StriDFA can be constructed from StriNFA in Figure 4.5(b) by the above mentioned construction method. The detailed steps can be found from Step-41 to Step-70 in Appendix II .

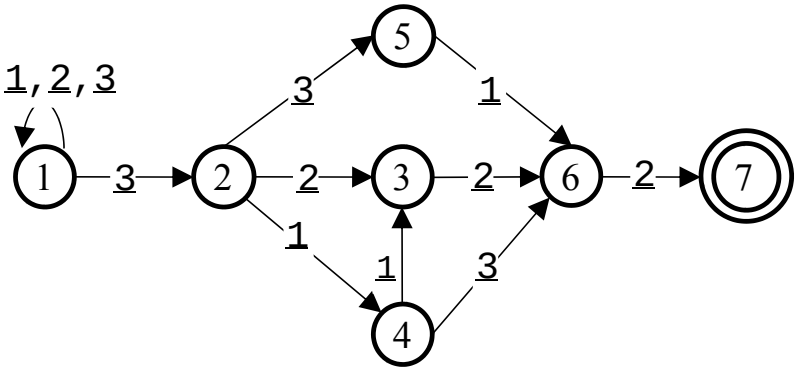
In Table 4.1 (default window size w=3), it is easy to see that, for the example input stream, if the traditional DFA can be matched, the StriFA can also be matched (the first three rows), while if the StriFA cannot be matched, then the traditional DFA cannot be matched either (the last row of the table). The correctness proof is given in subsection 4.3.3.

4.3.3 Correctness Proof

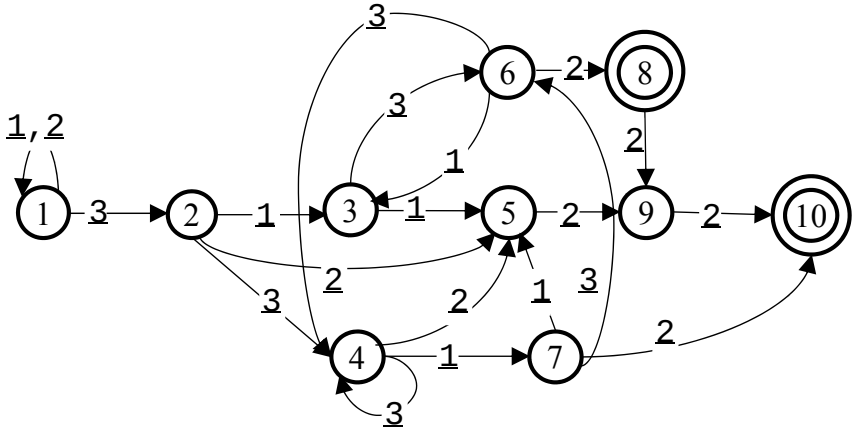
Generally, a false negative indicates that the intrusion detection system is unable to detect a genuine attack [72]. The false negative in StriFA detection system means that the traditional NIDS can find the intrusion while the corresponding StriFA detection system is unable to detect the intrusion. In this subsection, we

Table 4.1: *An example to show the correctness of StriFA.*

Input stream S	Traditional DFA	$F_e(S)$	StriNFA	StriDFA
caabbaaccaca	match at state 16	1 3 1 3 2	match at state 7	match at state 8
baabaabbabbcaca	match at state 16	1 2 1 3 3 1 2	match at state 7	match at state 10
caabbaaacaca	match at state 16	1 3 1 1 2 2	match at state 7	match at state 10
caabbaabccaca	cannot match	1 3 1 3 1 2	cannot match	cannot match



(a) StriNFA of the regex `.*abba.{2}caca` with tag = 'a' and $w = 3$.



(b) StriDFA of the regex `.*abba.{2}caca` with tag = 'a' and $w = 3$.

Figure 4.5: *StriNFA and StriDFA of regex `.*abba.{2}caca` with tag = 'a' and $w = 3$. (The transitions back to state 1, 2 and 3 of the corresponding StriDFA are partly ignored for simplicity).*

prove the correctness of traditional NFA/DFA and StriFA. The correctness here means there is no false negative in StriFA: if the StriFA cannot be matched, then the original NFA/DFA cannot be matched either. This is because if a statement is true, the contrapositive is also logically true. So we only need to prove if the original NFA/DFA can be matched, the corresponding StriFA can be also matched.

The input stream can be regarded as a long string. Let $A_i = a_1 a_2 \dots a_i$ be first i characters of the input sequence. After reading A_i, the traditional NFA/DFA reaches state set Q_i and StriFA reaches state set Q'_i. Denoting a_i as a tag, we want to prove the statement that if the original NFA/DFA can be matched, the corresponding StriDFA can be matched too, then the lemma $Q_i \supseteq Q'_i$ needs to be proved first.

Here the induction method is used to prove the correctness. Let a_{t_j} be the j^{th} tag of input sequence, so we only need to prove $Q_{t_j} \supseteq Q'_{t_j}$ holds for j =1, 2, ….. According to Algorithm 3, we have $Q_{t_1} \supseteq Q'_{t_1}$. This is because of that a_{t_1} is the first tag of the input sequence, it can trigger the initial state of StriDFA. Denote the corresponding matching state of a_{t_1} is q_{t_1} ($q_{t_1} \in Q_{t_1}$). Meanwhile, q_{t_1} is the initial state of StriDFA so that here we have $\{q_{t_1}\} = Q'_{t_1}$. After reading the first tag, the lemma holds. Suppose the lemma holds when a_{t_j} is the j^{th} tag: $Q_{t_j} \supseteq Q'_{t_j}$. For any state $q_{j+1} \in Q_{t_{j+1}}$, let q_j be the state in Q_{t_j} that reaches q_{j+1} by reading $a_{t_j} \ldots a_{t_{j+1}}$. By induction hypothesis, q_j is also in Q'_{t_j}. According to line 33 in Algorithm 3, q_{j+1} must be reachable in traditional NFA/DFA by reading $a_{t_j} \ldots a_{t_{j+1}}$, that is, $q_{j+1} \in Q'_{t_{j+1}}$. So we have proven that $Q_{t_{j+1}} \supseteq Q'_{t_{j+1}}$.

The statement that if the original NFA/DFA can be matched then the corresponding StriFA can also be matched is true. So we have proven that if StriFA cannot be matched, then the corresponding original NFA/DFA cannot be matched either. In other word, there is no false negative in StriFA.

4.4 Stride Finite Automaton

4.4.1 Building StriNFA by NFA-based method

In the above section 4.3, a construction method of StriNFA/StriDFA based on traditional DFA is proposed. When combining multiple regexes to a DFA, wildcards may cause the transitions of the corresponding DFA to grow exponentially [68]. StriFA cannot be generated conveniently using traditional DFA as an intermediate step. It is time consuming to generate the corresponding DFA and it costs huge memory usage even if a regex can be transformed to a DFA successfully. For example, the cost of memory storage will reach 15GB after the combination of about 700 rules in Snort NIDS [38].

As the processing steps shown in Figure 4.6, a new transform approach is presented to generate StriNFA/StriDFA via the traditional NFA directly.

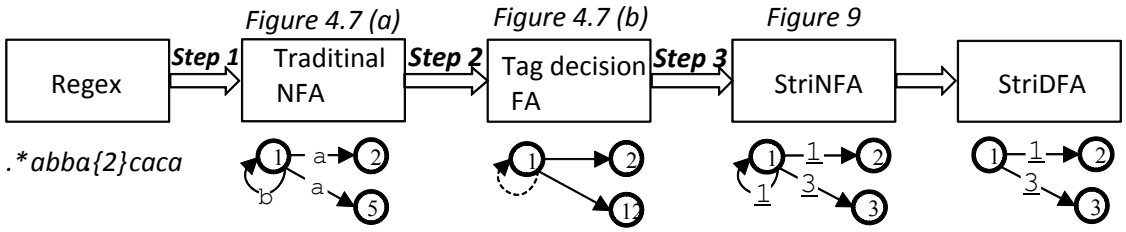


Figure 4.6: *Flow chart represents how to convert regexes to StriNFA/StriDFA via NFA directly.*

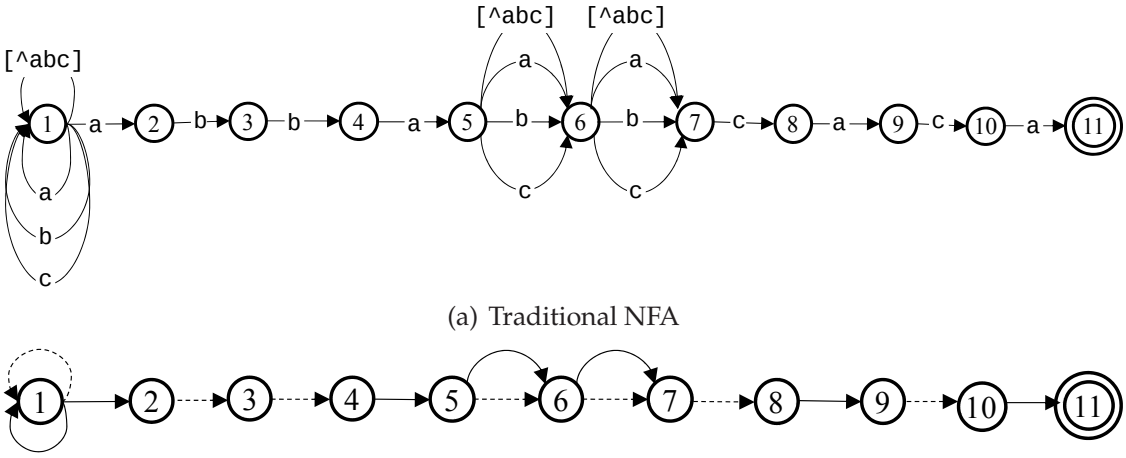
Step 1: Compile regex to corresponding NFA

The way of compiling a regex to a corresponding NFA is the same as the first step of section 4.3. Instead of converting a traditional NFA (Figure 4.7(a)) to the traditional DFA, a tag decision FA is directly generated from the traditional NFA.

Step 2: Restructure NFA to Tag decision FA

In this step, each transition is marked based on whether its label character is a tag (solid transition) or not a tag (dotted transition). And then all labels are removed from transitions of the traditional NFA. The tag decision FA in Figure 4.7(b) is generated after the transformation from the traditional NFA in Figure 4.7(a) using tag '`a`'.

Take Figure 4.7(a) as an example, the outgoing transitions of state 5 are: $5 \xrightarrow{a} 6$ and $5 \xrightarrow{[\hat{}a]} 6$. By checking if the transition label between state 5 and 6 equals to tag ('a') or not, the outgoing transitions of state 5 are redrawn as $5 \longrightarrow 6$ or $5 \dashrightarrow 6$ as



(a) Traditional NFA

(b) Tag decision FA using tag '`a`'

Figure 4.7: *Traditional NFA and Tag decision FA of regex* `.*abba.{2}caca`.

shown in Figure 4.7(b).

Step 3: Transform Tag decision FA to StriNFA

Unlike in a DFA, there maybe more than one outgoing transitions for an incoming character. So the initial state for StriNFA may be an initial set of states from its corresponding traditional NFA. As shown in Figure 4.8(a), the initial state set of the StriNFA is $\{1, 2\}$ when transforming from the traditional NFA.

The algorithm for transforming a tag decision FA to a StriNFA is almost the same as algorithm 3 except for some minor differences such as the initial state searching method. The initial state searching method is described in algorithm 4.

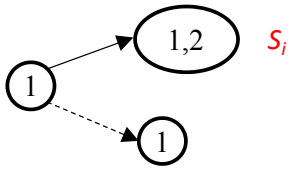
Similarly, take state set $\{1, 2\}$ as an example, start with state $\{1, 2\}$, the related transitions within w steps are shown as follows.

$$\{1,2\} \longrightarrow \{1,2\}$$

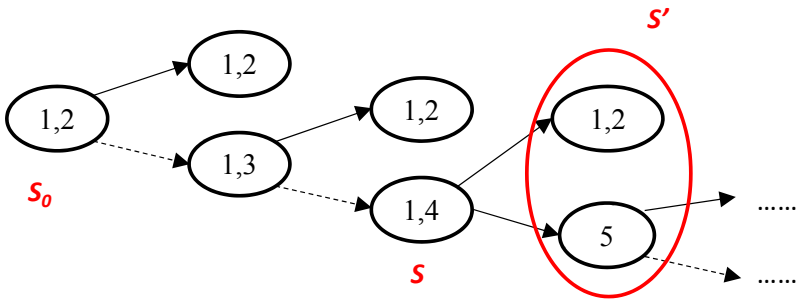
$$\{1,2\} \dashrightarrow \{1,3\} \longrightarrow \{1,2\}$$

$$\{1,2\} \dashrightarrow \{1,3\} \dashrightarrow \{1,4\} \longrightarrow \{1,2\}$$

$$\{1,2\} \dashrightarrow \{1,3\} \dashrightarrow \{1,4\} \longrightarrow \{5\}$$



(a) Initial state set.



(b) Recursive steps.

Figure 4.8: *Explanation of recursive steps.*

As for step 4 in section 4.3, the outgoing transitions of state set {1, 2} are generated as follows:

$$\{1,2\} \xrightarrow{1} \{1,2\}$$

$$\{1,2\} \xrightarrow{2} \{1,2\}$$

$$\{1,2\} \xrightarrow{3} \{1,2\}$$

$$\{1,2\} \xrightarrow{3} \{5\}$$

After the procedure of transforming a tag decision FA to a StriNFA, a StriNFA before state renumbering is obtained in Figure 4.9. Comparing Figure 4.9 with Figure 4.4, there is no difference between these two StriNFA structures apart from the state numbers. After renumbering the state number in Figure 4.9, the same StriNFA in Figure 4.5(a) is obtained. It means that we can get the StriNFA directly

Algorithm 4: Algorithm of transforming Tag decision FA to StriNFA via NFA

```
1  Procedure FindInitiaState(Q_i, q_0); /* Find the initial state set */
2  Input: NFA = (Q, Σ, δ, q_0, F), w, tag; /* q_0: initial state and F:
   set of final states, w: window size */
3  Q_i := ∅
   /* Initialize the initial state set Q_i                          */
4  for e = (S, S') ∈ D+(S) do
       /* D+(S): outgoing transitions set of the current
       state set S                                                   */
5      if e = --→ then
6          S' := move(S, --→)
7          FindInitiaState(Q_i, S') /* if --→ is found, then move to
               next state S' and keep on search                      */
8      else if e = → then
9          S' := move(S, →)
10         if S' ∉ Q_i then
11             Q_i := Q_i ∪ S'
               /* add a new state to initial state set              */
12 end
13 return Q_i
```

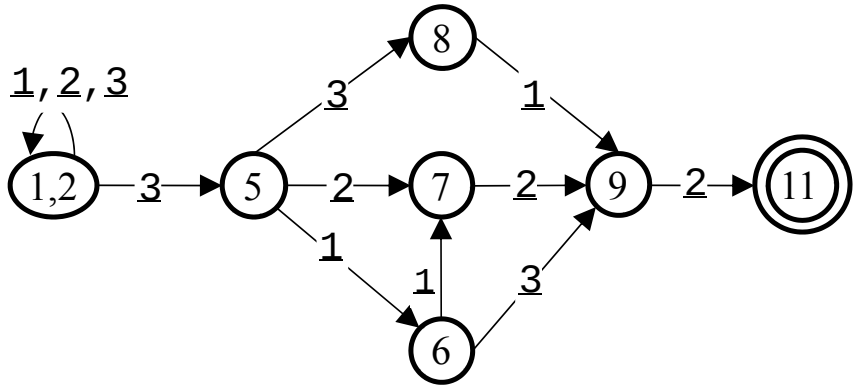


Figure 4.9: *StriNFA of the regex* `.*abba.{2}caca` *with tag ='a' and* $w = 3$ *before renumbering.*

from the traditional NFA without converting the traditional NFA to a DFA in the intermediate steps.

The procedure of converting a StriNFA to its corresponding StriDFA is exactly the same as the procedure in subsection 4.3.2.

4.4.2 StriFA-based Matching Architecture

There are some differences between the methods to build StriFA from string patterns and regex patterns. The architecture for multi-string matching can be found in Section 3.3. The multi-regex matching engine is shown in Figure 4.10.

4.5 Analysis and Optimization

The tradeoffs that are analyzed in this section involve maximizing filter rate and minimizing the false alarm rate while preserving other key performance indicators (i.e., throughput and memory usage) as far as possible. A low filter rate will trigger frequent use of the verification module, degrading the overall throughput. A high false alarm rate leads to waste of time in performing the reverse DFA match.

The filter rate is affected by both the characteristics of the incoming traffic and the structure of the StriFA while the false alarm rate is affected by the structure of StriFA. It is not conflicting to get a higher filter rate and lower false alarm rate. We

cannot change the characteristics of the incoming traffic, but we can make better use of it and change the structure of StriFA. The selection of tags and window size may affect the structure of StriFA and in turn can affect the filter rate and false alarm rate.

4.5.1 Stride-Neighbor FA

In order to reduce the false alarm rate, I propose a scheme called Stride-Neighbor FA.A ***neighbor symbol*** is defined as the character before the tag, or as the character before the last character of the window if a tag is not found within a window. For instance, in Figure 4.11, the characters before the tags are `f`, `r` and `c` respectively in P_1. Characters `f`, `r` and `c` are used as neighbor symbols to construct a neighbor DFA. Similarly, neighbor symbols `c` and `m` are extracted from P_2. The combination neighbor DFA of P_1 and P_2 is illustrated in Figure 4.11.

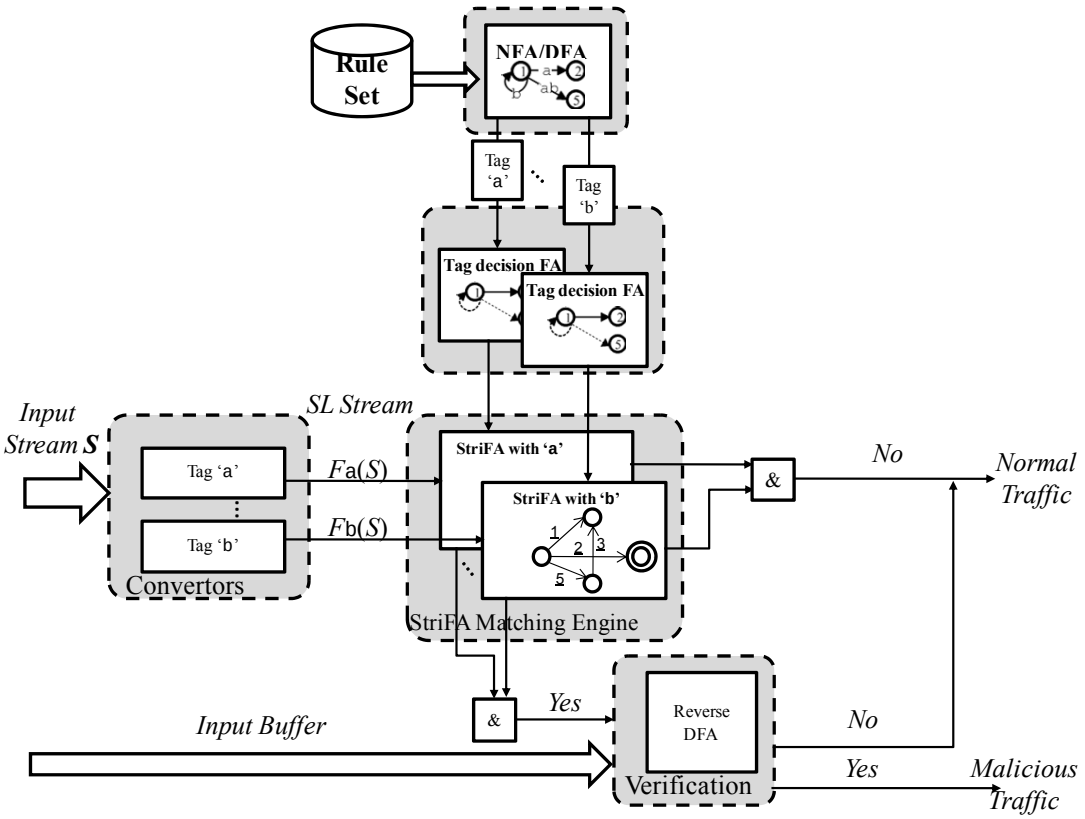


Figure 4.10: *Architecture of StriFA-based multi-regex matching engine.*

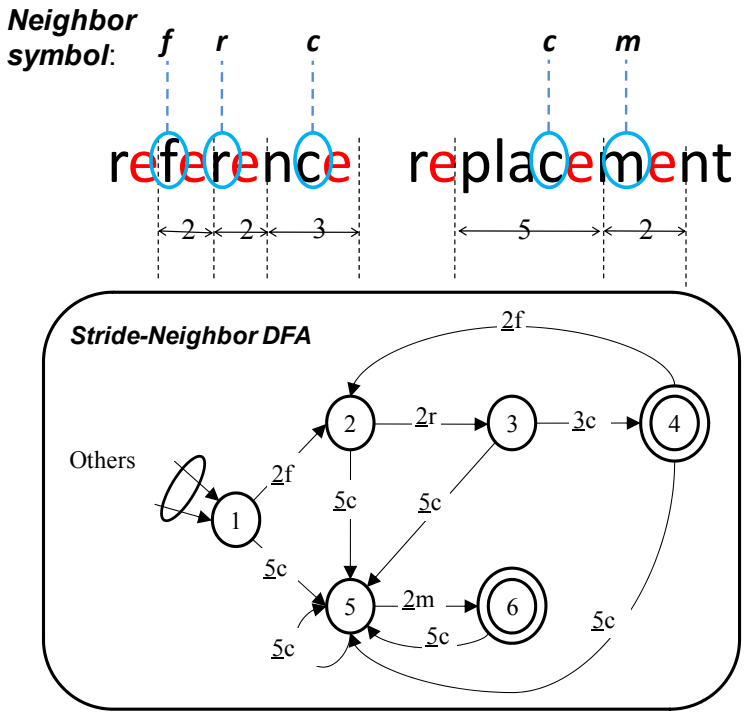


Figure 4.11: *Stride-Neighbor DFA for P_1="reference" and P_2="replacement" with tag='e' and w=5.*

The stride-neighbor DFA shown in Figure 4.11 uses a combination of each neighbor character with the corresponding stride length for StriDFA labels (e.g., $\underline{2}f$).

The basic idea of Stride-Neighbor FA is to reduce the false alarm rate by starting a preliminary step of exact matching, at a small memory consumption cost. Generally, the false alarm rate can be reduced by 94.1% when using Stride-Neighbor FA. Efficiency results can be found in subsection 4.6.5.

4.5.2 Performance of StriFA

Table 4.2 shows that a single regular expression of length n can be expressed by an traditional NFA with O(n) states. When the traditional NFA is converted to traditional DFA, it may generate $O(\sum^n)$ states. There is only one outgoing transition for each character from a DFA state, so the processing complexity of traditional DFA for each character is $O(1)$, while it is $O(n^2)$ for traditional NFA when all n states are active at the same time [68]. Considering StriDFA, there is

also one outgoing transition for each input SL from each StriDFA state, so the processing complexity of StriDFA for each input SL is $O(1)$. Suppose windows size = w, then the average stride length is $\frac{w}{2}$. The average state number is $\frac{n}{\frac{w}{2}}$, so the processing complexity of StriNFA is $O((\frac{n}{w})^2)$ when all $\frac{2n}{w}$ states are, in the worst case, active at the same time.

4.5.3 Conversion Complexity of StriFA

Let α denote the average fanout in each state. From step 10 to step 33 in Algorithm 3, we find that the time complexity of the recurse procedure in a state is α^w. Then the average time complexity of converting NFA/DFA to StriNFA is $n\alpha^w$. In practice, α is less than 3 in Snort rules ($\alpha = 2.6$ on average). So the complexity of converting to StriFA is acceptable. Furthermore, all the conversion from traditional NFA/DFA to the corresponding StriNFA/StriDFA can be done off-line. The rule set of the popular NIDS is updated once every one or two months, so we don't need to worry too much about the off-line complexity of StriNFA/StriDFA construction/updates.

4.6 Evaluation

Messages on the Internet are transmitted in small units called packets. Each packet contains a header and a payload. We extract the payload of each packet

Table 4.2: *Worst case comparisons of NFA, DFA [68], StriNFA and StriDFA.*

	One regular expression of length *n*		***m* regular expression compiled together**	
	Processing complexity	Storage cost	Processing complexity	Storage cost
NFA	$O(n^2)$	$O(n)$	$O(n^2m)$	$O(nm)$
DFA	$O(1)$	$O(\sum^n)$	$O(1)$	$O(\mid\sum\mid^{nm})$
StriNFA	$O((\frac{n}{w})^2)$	$O(\frac{n}{w})$	$O((\frac{n}{w})^2m)$	$O(\frac{nm}{w})$
StriDFA	$O(1)$	$O(w^n)$	$O(1)$	$O(w^{nm})$

and feed it to our StriFA-based matching engine. Similar to other DPI technologies, the StriFA-based matching engine opens and reads the packet payload in real time to do the state-aware inspection.

To evaluate the efficiency of the StriFA-based matching engine, an instance of it was implemented, which uses tags to produce the SL stream. The StriDFAs are built in the way as described in Section 4.4. The instance is implemented on a software platform, and in particular, StriFAs are realized using standard finite automata transition-table storage.

The performance of the StriFA-based matching system, including throughput and memory size, is also influenced by four aspects: (a) input stream (trace), (b) regex rules, (c) window size and (d) selected tags. The memory consumption is influenced by (b)(c)(d); speedup is influenced by (a)(c)(d); and the overall throughput is influenced by (a)(b)(c)(d).

4.6.1 Trace characteristics

Three real-life network traffic traces are employed to evaluate the system, they are the Darpa, Defcon and Tsinghua-trace respectively. The characteristics of the trace files are shown in the first part of Table 4.3. Defcon is from the Shmoo Group DefCon 17.0 Capture the Flag Contest [73]. Darpa is from the DARPA intrusion detection data sets collected by MIT Lincoln Laboratory [74]. The Tsinghua-trace was collected at the gateway node of Tsinghua University campus network. In Table 4.3, Row "# *of Packets*" denotes the number of packets; Row "# *of Conn.*" indicates the number of TCP connections; and Row "*Avg. Packet Len.*" lists the average packet size of each trace.

In additional to the above three trace types, a number of other traces (all taken from the World Wide Web) were also used to evaluate the performance.

- Webpages from CNN [75] were collected by Larbin [76] in different periods. Larbin is a web crawler intended to fetch a large number of web pages to

fill the database of a search engine.

- WebbSpam[77] consists of nearly 572,292 web spam pages with a size of 5.54GB, which was chosen additionally to demonstrate the performance of our architecture.

- Phishing Set mixes Phishing corpus[78] and other up to date Phishing samples.

The experiments were carried out on two desktop PCs, each has eight 3.8GHz CPUs and 12 GB memory.

4.6.2 Throughput

The matching system throughput here means how much input data stream can be processed in one second. We calculate all the active states in traditional NFA and DFA as well as StriNFA and StriDFA during the matching so as to get the number of clock cycles which are needed. As shown in Figure 4.12, the throughput differs for the various trace according to the different traffic characteristics. The throughput also differs based on the detection technique used (i.e., tradition NFA, traditional DFA, StriNFA and StriDFA). DFA is faster than NFA, and similarly StriDFA is faster than StriNFA. Here we also notice that the throughput of one specific automaton under different traces are not the same. For example, for

Table 4.3: *Real data samples*

Real Trace			
Trace	Defcon	Darpa	Tsinghua
# of Packets	8,405,211	5,078,652	4,714,151
# of Conn.	26,207	201,591	60,765
Avg. Packet Len. (B)	285.3	312.4	428.6
Avg. Minimum Stride	72.5	10.4	30.3
Real Web Pages			
Web pages	CNN	WebbSpam	Phishing Set
# of Pages	1,102,986	572,292	93,155
File sizes (GB)	12.737	5.541	1.019
Avg. Stride Len. (B)	48.283	55.419	34.564

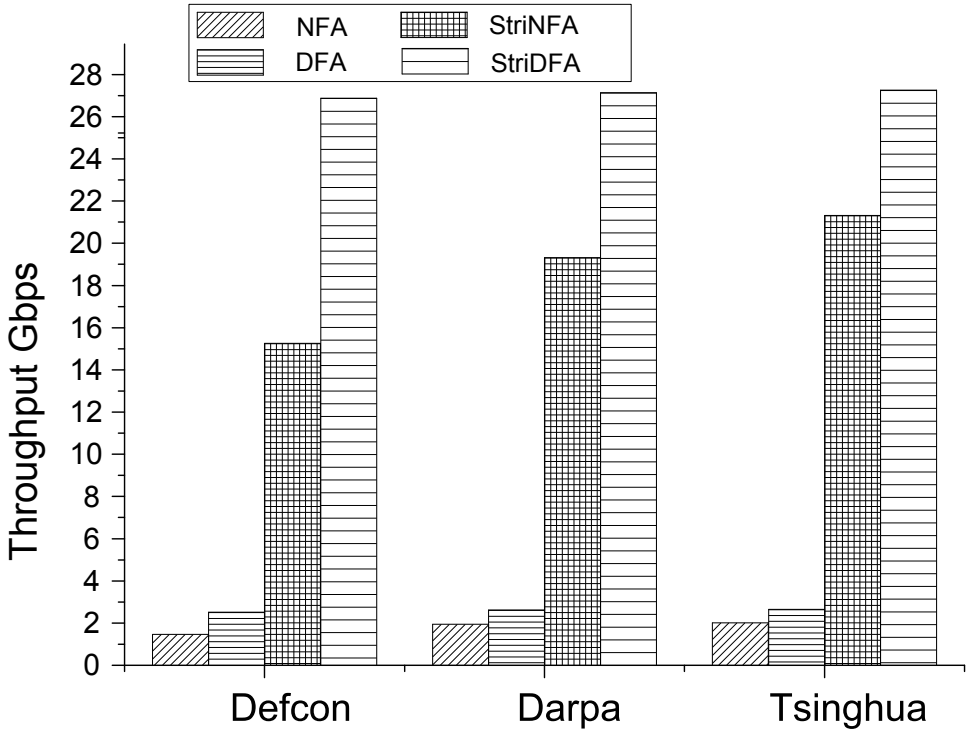


Figure 4.12: *Throughput of three different traces with different finite automaton.*

traditional NFA, we can find that the throughput of DefCon (1.46 Gbps) is slower than that of Darpa trace (1.95 Gbps). The Tsinghua-trace achieves the highest throughput which is 2.01 Gbps. Here the default window size is set to 140 (the reason for this is explained in subsection 4.6.5). The trace of DefCon contains various kinds of attack information and malicious data. The verification engine is triggered more frequently resulting in a slower overall throughput for the detection system. While the characteristic of Tsinghua-trace is different which contains daily life data. The Stride-based matching could filter most of the Tsinghua-trace so that the throughput is higher than other traces.

4.6.3 Memory consumption

The rule sets from the recent version of Snort (v2.9 as of 06 Apr, 2011) and ClamAV (v0.97.2 as of 11 Nov, 2010) were used. Traditionally, combining all (even part of) regex rules into one DFA requires a large amount of memory; for example more than 15 GB of memory is needed if 88 Snort rules are combined into

one single DFA[62] (the traditional DFA memory usage is in the second column of Table 4.4). For StriFA, the memory usage of different rule sets using various window sizes is investigated. The sliding window is chosen to control the largest SL which is sent to the StriFAs and to reduce the size of the state-transition table from 256 to window size w. The number of tags selected also affect the total memory size. The fifth to seventh columns in Table 4.4 show the total number of states and transitions in all StriFAs using different window sizes w.

Table 4.4: *Comparison between Traditional NFA/DFA, k-DFA and StriNFA/StriDFA*

Snort										
Memory			k**-DFA**		**StriNFA**			**StriDFA**		
factors	**NFA**	**DFA**	k=2	k=4	w=10	w=20	w=30	w=10	w=20	w=30
# State	1160	6469	8152	19314	471	397	314	671	447	264
# Transition	6194	1.73M	2.1M	3.5M	1342	1924	2459	8.5k	8.94k	9.92k
Alphabet set	256	256	2114	7024	10	20	30	10	20	30
# Active states / # Input chars	1.141	1	0.623	0.523	0.696	0.136	0.076	0.322	0.105	0.035
# Active states / # Input Strides	*null*	*null*	*null*	*null*	2.011	1.045	1.052	1	1	1
ClamAV										
Memory			k**-DFA**		**StriNFA**			**StriDFA**		
factors	**NFA**	**DFA**	k=2	k=4	w=40	w=100	w=160	w=40	w=100	w=160
# State	3246	18484	93162	184412	1041	771	612	3834	2101	1912
# Transition	6843	2.28M	3.78M	5.04M	2198	3511	4250	9.3k	10.5k	11.7k
Alphabet set	256	256	2673	9532	20	60	100	20	60	100
# Active states / # Input chars	1.391	1	0.631	0.546	0.079	0.045	0.032	0.035	0.021	0.017
# Active states / # Input Strides	*null*	*null*	2114	7024	2.373	1.61	1.08	1	1	1

The results are compared with the memory cost of another well-known DFA acceleration matching method *k*-DFA [54]. Although it was proposed with several specific memory cost optimizations, for a given DFA defined on an ASCII alphabet Σ, *k*-DFA will have nearly $|\Sigma|^k$ outgoing transitions in every state.

The overall memory usage of the Stride-based matching engine is the sum of memory usage of StriFA and NFA instead of DFA (see subsection 2.3.3). Reverse NFA plays the same role as reverse DFA. The only differences between reverse NFA and reverse DFA are the differences of traditional NFA and DFA.

The 2^{nd} and 3^{rd} column in Table 4.4 are memory factors of traditional NFA and DFA. Using the Snort rule set as an example, we can see the basic information of NFA and DFA, such as state number, transition number and the size of alphabet set. There are more states and transitions in DFA than NFA. The size of the alphabet set is the same, equal to 256. For one input character, there may exist more than one active state in NFA. However in DFA, every input character has only one determinate outgoing transition. This is why there is always one active state for an input character in DFA. On average there may trigger 1.141 active states for an input character in NFA. k-DFA [54] has transition explosion and alphabet set explosion problems, and so the author presented alphabet-reduction and default transition compression methods. The size of the alphabet set is still very large which is shown in sub-columns of the 4^{th} column. StriNFA has less states and transitions than traditional NFA. In the first sub-column of the 5^{th} column (when w=10), there are only 471 states and 1342 transitions. Because the size of the alphabet set is decided by the window length (explained in Section 3.6), the alphabet set size = w = 10. The number of transitions in the first sub-column of the 6^{th} column (w=10 of StriDFA) is 8500. Comparing with traditional DFA (1730000 in the 3^{rd} column), we can see that StriDFA reduce the number of transitions by 50.86%.

From Table 4.4 we can see the memory usage of StriNFA and StriDFA are

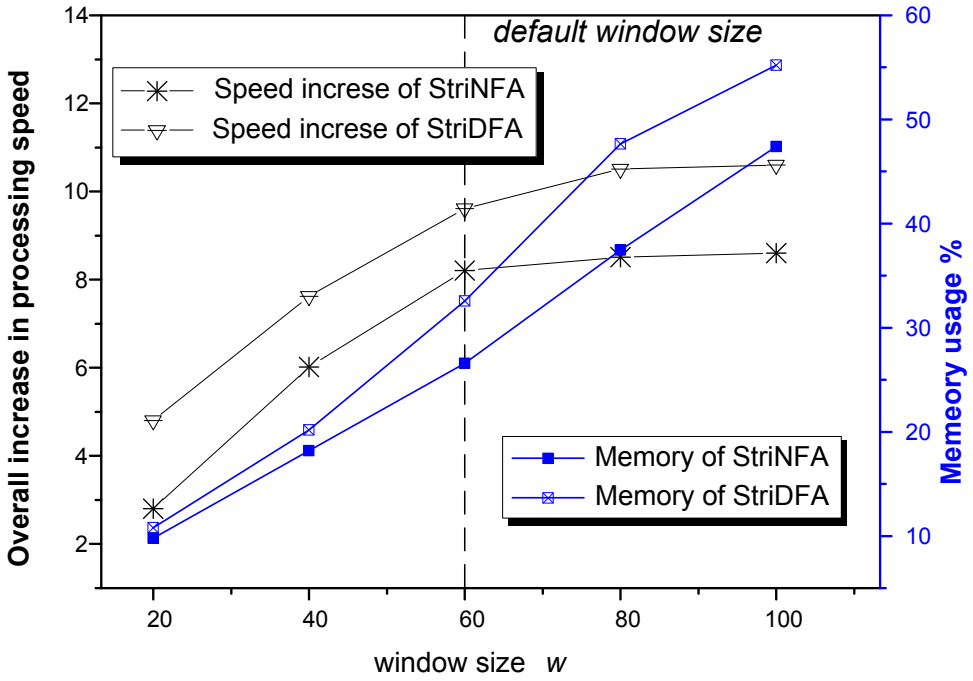


Figure 4.13: *Overall speedup and memory usage of StriFA with different window sizes.*

much smaller than traditional NFA/DFA. If reverse NFA is used in the verification module, the whole memory usage of StriNFA + reverse NFA can be smaller than traditional DFA. As shown in Figure 4.13, as the window size is increased, the overall memory usage becomes larger and larger. This is because each state has at most 256 outgoing transitions in traditional NFA/DFA while there are w transitions at most for a state in StriFA.

4.6.4 Speedup

When we traverse the text using a StriNFA, a number of transitions can be followed and a set of states become active. However, a DFA has exactly one active state at a time. StriNFA and StriDFA have different memory access when processing an input character. We evaluate the overall speedup = all the states accessed in traditional DFA / (all the states accessed in StriNFA or StriDFA + the accessed states number of reverse DFA). Figure 4.13 describes the overall speedup of StriNFA and StriDFA.

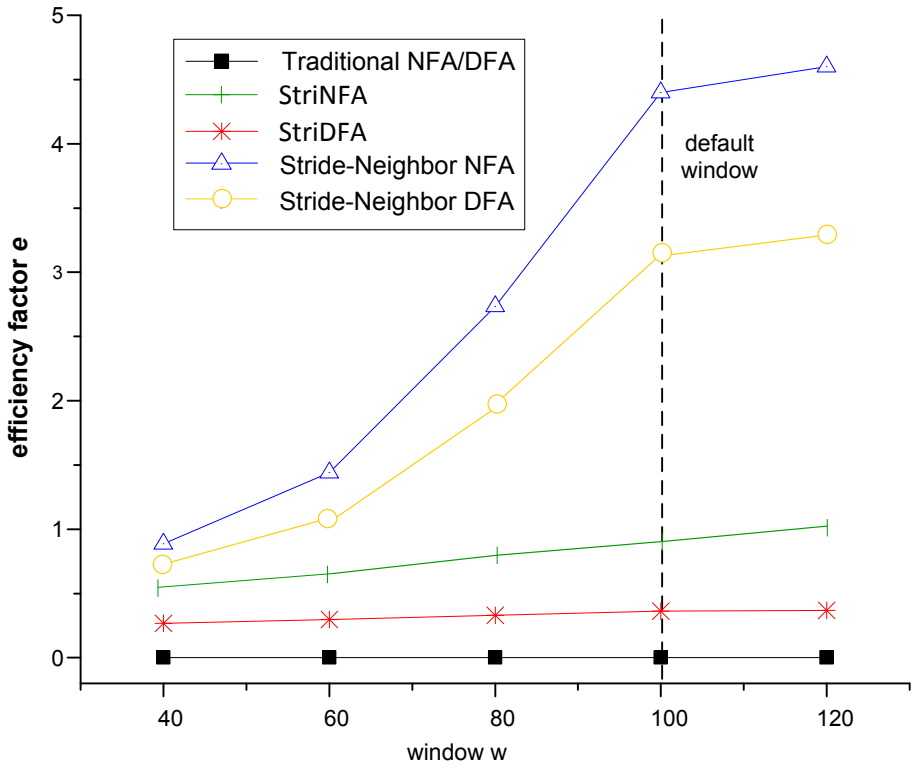


Figure 4.14: *Efficiency factor of StriDFA, Stride-Neighbor DFA of different window sizes.*

4.6.5 Filter rate and false alarm rate

Memory usage can often be reduced at the cost of increased false alarm rate. Therefore we define the Efficiency Factor as

$$e = \frac{\Delta w}{\Delta fp} = \frac{percentage\ of\ reduced\ memory\ usage}{increased\ false\ alarm\ rate}$$

That is, it measures the amount of memory saved per increased false alarm rate. We cannot make a simple combination of Traditional NFA/DFA and StriNFA/StriDFA since the efficiency factor is not good ($e < 1$). As shown in Figure 4.14, after employing Neighbor DFA, the efficiency factor of Stride-neighbor FA is much better than StriFA ($e > 3$ when w=100). A clear turning point can be found at w=100 after which the increase in efficiency factor decelerates with respect to the window size. Consequently the **default window size** is set to the window size at the turning point (here w=100).

4.6.6 Performance on real traces

In the StriFA-based matching engine, the length of the SL stream is only about 0.38% of the length of its original input stream. This is the main reason that the StriFA-based matching engine can achieve a much higher throughput than the NFA/DFA scheme. According to the experimental results of the Tsinghua-trace and some groups of ClamAV rules, in traditional NFA/DFA-based detection system, 13 matches can be detected. In StriFA-based detection system, there are 127 potential matches alarmed by StriFA while all 13 real matches are confirmed by the verification module. The same number of intrusion attacks can be found in both traditional detection system and our StriFA-based detection system.

4.7 Conclusion

In this chapter, StriFA: a novel regular expression matching acceleration scheme for complex network intrusion detection systems is presented. The main idea of StriFA is to convert the original byte stream into a much shorter integer stream and then matches the integer stream with a variant of DFA, called Stride Finite Automaton (StriFA). We provide the formal construction algorithm of StriFA which is able to transform an arbitrary set of regex to a StriFA. We also describe the method to produce a stride length stream so that false positives can be reduced to an acceptable level. The results show that our architecture can achieve about 10 fold increase in speed, with a lower memory consumption compared to traditional NFA/DFA while maintaining the same detection capabilities.

CHAPTER 5

S^2N-FA: A Hybrid Finite Automaton for File Detection

An essential requirement for today's business is the ability to access information and documents from anywhere, at any time, and to collaborate with anyone. Technologies such as instant messaging software (Skype, MSN) or BitTorrent file sharing methods, allow the convenient sharing of information between managers, employees, customers, and partners. This, however, leads to two major security risks when exchanging data between different people: firstly, leakage of sensitive data from a company and, secondly, distribution of copyright infringement products in Peer to Peer (P2P) networks. Traditional Deterministic Finite Automaton (DFA) based Deep Packet Inspection (DPI) solutions cannot be used for inspection of file distribution in P2P networks due to the potential out-of-order delivery of the data.

The basic idea of this chapter is to propose a hybrid finite automaton called Skip-Stride-Neighbor Finite Automaton (S^2N-FA). It combines the benefits of the following three structures: 1) Skip-FA is used to solve the out-of-order data scanning problem; 2) Stride-DFA is employed to reduce the memory usage of Skip-FA; 3) Neighbor-DFA is used to achieve a low false positive rate at the additional cost of a small increase in memory consumption. The proposed S^2N-FA is tested

with three different real traces and the experimental results show that S^{2}N-FA consumes about 60% less memory and achieves about 20 times increase in speed compared with Skip-FA.

5.1 Introduction

File sharing is one of the most important functions of today's Internet. It provides users with various accesses to digitally stored content, such as videos, audio, documents, and executable programs. Unlike the traditional client-server model where clients request resources and servers provide them, the P2P model lets every node (called a peer) play the role of both a server and a client. The flexibility and scalability of the model enables users to participate in large file sharing communities with more convenience.

However, file sharing technologies such as P2P applications (BitTorrent, eMule, vuze) and instant messaging tools (Skype, MSN) also create two major potential security problems in today's networks: the leakage of personal information or confidential documents and the distribution of copyright infringement files.

In order to prevent the leakage of sensitive information via the file sharing system, a *Data Leakage Prevention System* (*DLPS*) should be installed at the edge of the enterprise network (e.g., gateway or edge router) to filter the outgoing information for sensitive contents. Similarly a *Copyright Infringement Detection System* (*CIDS*) is required to solve the problem of copyright violations for Internet service providers (ISP) which could control the download speed of the clients who are using P2P applications. In Figure 5.1, a DLPS is deployed at the gateway of an enterprise network and a CIDS used in the ISP's networks.

Traditional Deep Packet Inspection (DPI) technologies cannot be used directly in DLPS or CIDS. With the purpose of increasing the speed of distribution of a resource file, most P2P systems split the file into fixed-size pieces, except for the last piece, enabling P2P clients to download data from multiple peers simultane-

ously [35]. However this advantage leads to a problem for file content detection because the file segments may be transferred out-of-order. The downloaded file is composed of fragments randomly passing through the gateway. After receiving all the out-of-order packets, the original file can be reassembled by the P2P application. Traditional DPI works by applying pattern matching methods to the reassembled payload. Unfortunately it is impractical to cache all the packets during the distribution of large files. Therefore, it cannot be used to identify the file being transmitted via P2P applications.

It is worth mentioning that although the concepts are similar, the content scanning method employed in CIDS and DLPS is different from that of the DPI in a Network Intrusion Detection System (NIDS) [36][65] (also shown in Figure 5.1).

Compared to the signature-based DPI in NIDS, content scanning in CIDS and DLPS has many unique characteristics and constraints as follows.

(1) In DPI, short patterns are used to match long input streams. However, the content scanning engine in DLPS stores the entire information of the digital files, and the payload of each packet is usually far shorter than those of the digital files.

(2) In DPI, we can apply pattern matching methods to the reassembled payload after reassembling the packets. On the contrary, what we have to face in CIDS and DLPS is the out-of-order data of the P2P transmission model.

In this chapter a high speed and memory-efficient structure, the *Skip-Stride-Neighbor Finite Automaton*(*S^{2}N-FA*) is proposed. This is a hybrid finite automaton

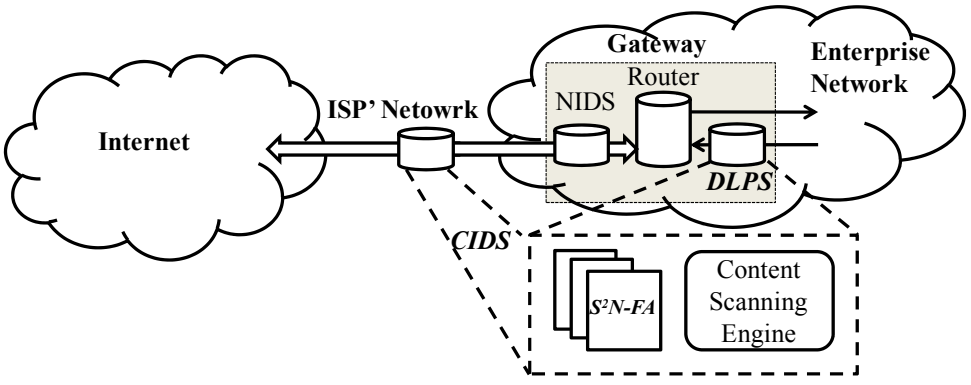


Figure 5.1: *CIDS and DLPS in the network.*

to extract multiple strides from the original files, so as to scan the incoming out-of-order packets with high-speed. Our contributions are summarized as follows:

1. The particular content scanning problem in CIDS and DLPS is formulated. To address this problem, a novel finite automata representation, named *Skip Finite Automata* (*Skip-FA*), is developed to detect the out-of-order packets carrying protected information by using special transitions to efficiently track the overlapping parts between packets' payloads of the sensitive files.

2. In order to reduce the memory consumption of Skip-FA, we present *Skip-Stride FA* to extract fingerprints from the original signature with increased speed and lower memory usage.

3. A neighbor DFA is used to reduce the false positive rate of Skip-Stride FA and a hybrid finite automaton S^{2}N-FA is constructed to accelerate the speed of matching with low memory consumption.

The rest of this chapter is organized as follows. The problem is formulated in Section 5.3. Section 5.4 provides the structure of Skip-FA and Section 5.5 describes the basic idea of StriDFA and how to build a Skip-Stride FA. The neighbor DFA is introduced and hybrid S^{2}N-FA are presented in Section 5.6. We evaluate the proposed hybrid automaton and discuss the results in Section 5.7. Finally, Section 5.8 concludes the chapter.

5.2 Related Work

DPI is intensively studied and is used for attack detection and for protocol recognition by matching the content of packets against a set of predefined short signatures [51, 39, 62, 79, 80]. The structure of traditional DPI is not suitable for scanning out-of-order data. The work related to file detection is studied as follows.

5.2.1 Fingerprint Extraction

A fingerprint (also called footprint) is a fragment of the pattern used to check packet payloads [81], just like a human fingerprint can identify a human being. If the fingerprint cannot be matched, then the human that the fingerprint represents certainly cannot be matched.

Nearly thirty years ago, M. Rabin first proposed the Rabin fingerprinting scheme to test small changes to the content, including adding or removing bytes [82]. U. Manber presented a tool, called sif [83], to identify similar files. The basic idea is to use fingerprints on several small parts of the file and have several fingerprints rather than just one.

Pucha *et al.* presented SET [84], a new approach to multi-source file transfers that obtains data chunks from sources of non-identical but similar files. The SET method employs a new technique called handprinting to locate these additional sources of exploitable similarity using only a constant number of lookups and a constant number of mappings per file.

5.2.2 File Detection in CIDS and DLPS

In file sharing systems (especially in P2P networks), piracy information detection has been studied and discussed in [85, 86, 87, 88]. However, most of these solutions follow the study of traffic behaviors and indirectly manipulate the packets' payload.

In addition, protection of sensitive files has been studied under data loss prevention in the field of security and privacy. In industrial society, software applications have been developed to prevent data loss at a user-level with tracking of work flow [89, 90, 91]. Many host-based features are exploited, such as detecting operations on marked files in the operating system. They work well in the end-user scenario but are not suitable for applications in gateway services due to their limited scalability in memory and throughput.

The core technology of DLP systems consists of some sophisticated algorithms that extract the fingerprints of sensitive data such as files, records, music and other content [92]. Some vendors deploy hash-based identifications to compare the hash values of transferred files with the database server [93].

To the best of our knowledge, the hybrid S^{2}N-FA is the first automaton that is capable of scanning out-of-order data with high speed and lower memory usage.

5.3 Problem Statement

5.3.1 Out-of-Order Data Transmission in P2P network

BitTorrent is a P2P protocol that enables fast download of large files using minimum Internet bandwidth. Unlike the traditional Client-Server downloading method, BitTorrent maximizes transfer speed by gathering pieces of the file you want from people who already have them and downloading these pieces simultaneously. The overall download speed is greatly improved by downloading multiple pieces at the same time from different peers, only part of the file is downloaded from each peer. However, this brings new challenges in file detection because traditional pattern-based matching approaches are not designed to handle out-of-order incoming packets.

Assume the content of the file (denoted as $\mathcal{F}$) is split into five pieces p_1, p_2, p_3, p_4 and p_5 by the BitTorrent protocol. One BitTorrent client may download $\mathcal{F}$ from other peers in the order of p_4, p_2, p_5, p_1 and p_3.

$$\{\mathcal{F} = p_1p_2p_3p_4p_5\} \xrightarrow[transfer\ to]{order:\ p_4p_2p_5p_1p_3} BTClient$$

Traditional DPI method is not powerful enough for P2P file detection. Fingerprint strings are usually extracted as patterns from the content of $\mathcal{F}$ in the traditional DPI method. For example, given $\mathcal{F}$ = "$referencementionsthereplacement\ collection$", one part of $\mathcal{F}$, R = "$replacement$" could be selected as one pattern to match $\mathcal{F}$, which means R can be used to match input $\mathcal{F}$.

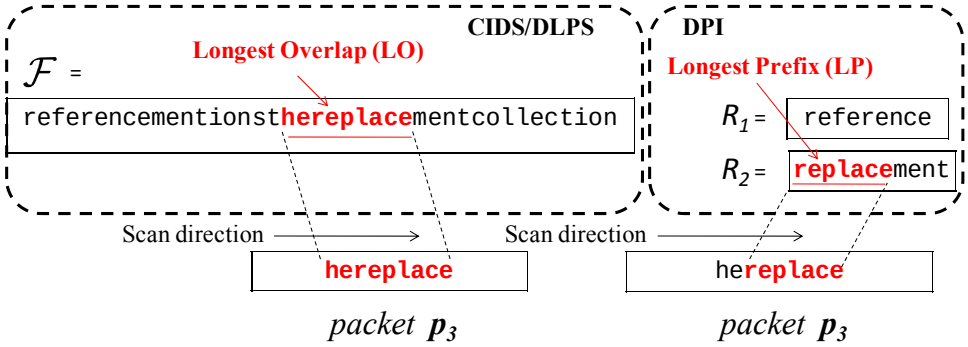


Figure 5.2: *The difference between LO matching (left) and LP matching (right).*

Due to the out-of-order delivery of pieces, traditional string matching and regular expression matching methods are not suitable. A sample split is shown below:

p_1 = "$reference$";

p_2 = "$mentionst$";

p_3 = "$hereplace$";

p_4 = "$mentcolle$";

p_5 = "$ction$"

Obviously, the following out of order sequence: p_4, p_2, p_5, p_1, p_3 cannot be matched by pattern R as shown in Figure 5.2. The problem is to find a method to determine if $\mathcal{F}$ is transmitted in an out of order network data stream. This requires matching against data that spans multiple packets, and yet does not require reassembling of the entire stream.

This is why we need to find a solution for P2P file detection. Regardless of the order of packets, a piece must be a part of the original file. So instead of matching pattern R, we can use the whole file $\mathcal{F}$ as the pattern to match the incoming packet. The automaton structure is illustrated in Section 5.4, and in Section 5.5 we explain how to reduce pattern's size from $\mathcal{F}$ to make it work in practice.

5.3.2 Problem Formulation

The content scanning engine of CIDS and DLPS stores the entire information of the digital files, and the payloads of packets are generally much shorter than the digital files. In practice, a packet contains header information that is not used in content scanning, therefore, the content scanning checks if any contiguous part in the payload of outgoing packets matches any contiguous part of the digital files. We formally describe the matching problem as follows:

Input: A set of n string rules (called *rule set*) $\mathcal{R} = \{R_1, \ldots, R_n\}$ and a string W (called *input*) over the alphabet set Σ, i.e., $W \in \Sigma^*, R_i \in \Sigma^*, 1 \leq i \leq n$. Assume the length of string R_i is l_i and the length of W is l. Let $R_i = a_{i,1} \ldots a_{i,l_i}$ and $W = b_1 \ldots b_l$.

Output: The longest substring of input W which also appears in some rule $R \in \mathcal{R}$ as a *substring*. An overlapping part is called an *overlap* and the longest one is called the *longest overlap*.

We call the problem *Longest Overlap* (*LO*) matching. In byte-by-byte processing style, we only need to keep the longest suffix of the input which is also an overlap. The left side of Figure 5.2 shows an example of LO matching in byte-by-byte processing style where "hereplace" is currently kept as the longest suffix which is also an overlap. By keeping these suffixes, we can easily find the LO since it must have once appeared as one of these suffixes. Thus, for simplicity, we refer to LO in our discussion as the longest suffix which is also an LO at any moment of the matching.

Instead of matching LO in the content scanning of CIDS or DLPS, DPI and IP lookup take the longest prefix (LP) matching as their basis. In longest prefix matching, we have the same input as LO matching, but the output is the longest *prefix*, rather than the longest *substring* of some rules which is also the suffix of the input string at some time of the matching. The right part of Figure 5.2 illustrates the LP matching where the current LP "replace" is a prefix of some rule. It is

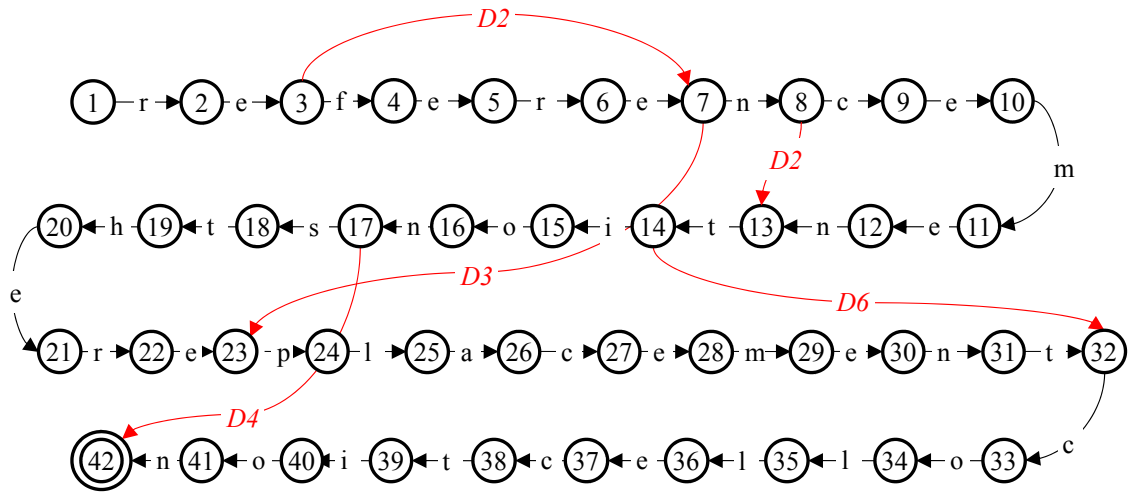


Figure 5.3: *The Skip-FA of signature $\mathcal{F}$={referencementionsthereplacementcollection}, the dotted lines show the Di-trans (D1-trans are ignored for simplicity).*

clear that LP matching is a *special case* of LO problem since a prefix itself is a substring. So in essence, the solutions developed for DPI cannot be employed in CIDS/DLPS directly.

5.4 Building Skip Finite Automata

We begin with a basic model, called Skip-FA, which tracks the overlap part completely by default transitions at the cost of unbounded delay time [94]. In the next section, we will introduce a technique to convert large binary files to short integer strings which can then be compiled to Skip-FA.

5.4.1 A Basic Model: Skip-FA

In Figure 5.3, a sample Skip-FA of $\mathcal{F}$ = "$referencementionsthereplacementcollecti on$" is given. As shown in the figure, a Skip-FA consists of forward transitions, called *basic transitions or basic-trans*, which link the states within the rule (e.g., the transition from state 1 to state 2) and dotted lines show transitions which are labeled with the combination of a symbol 'D' and an integer i ($i \geq 1$), called Di-trans.

To give the definition of Di-trans, let us denote the longest suffix of length i

of the prefix represented by state p as $Str(p, i)$. For a state q, we denote the set of characters on its outgoing basic-trans as $Out(p)$ (e.g., $Out(1) = \{\texttt{r}\}$ in Figure 5.3). The Di-trans are defined as follows:

Definition 7. ***Default transitions, Di-trans***

State p has a Di-trans, pointing to state q if and only if when:
q is the smallest one[13] *in $\Gamma = \{q' | q' > q, q'$ meets (i)(ii)(iii)$\}$, if $\Gamma \neq \emptyset$,*
where (i) $Str(p, i) = Str(q', i)$,
(ii) $Out(p) \neq Out(q')$,
(iii) $Str(p, i + 1) \neq Str(q', i + 1)$.

The intuition of condition (i) is that, we can implicitly keep a suffix of length i of the input string by tracking Di-trans. Conditions (ii)(iii) are used to reduce the number of Di-trans without distorting the intuition of (i). For example, for state 3, state 7 and 23 meet condition (i)(ii)(iii) since they share a 2-character suffix `re`. For a state q, there is possibly more than one destination state that meets (i)(ii)(iii), and the smallest one is selected to guarantee that only one state is chosen for the Di-trans of p. For example, we choose 7 as the destination of the D2-trans of state 3 by definition 7. In fact, from a state p and integer i, we can visit all states meeting (i)(ii)(iii) by tracking Di-trans. E.g., we can visit 7 by tracking D2-tran from state 3 and visit 23 by tracking D3-trans from state 7.

The equation $p \xrightarrow{Di} q$ means there is a Di-trans between state p and q which also means there are i same characters before state p and q. If the next character of state p cannot be matched, then state q is reached by transition Di for further matching. For example in Figure 5.3, we have (i)$Str(7, 3) = ere = Str(23, 3)$, (ii) $Out(7) = n \neq Out(23) = p$ and (iii) $Str(7, 4) = fere \neq here = Str(23, 4)$. So $7 \xrightarrow{D3} 23$ is generated (state 7 and 23 have the same 3 characters "ere" as prefix).

The Di-trans here are essentially an extension of failure transitions in the well-known Aho-Corasick finite automaton (AC-FA which is described in subsection 2.1.1). Designed for LP matching (see Subsection 5.3.2), AC-FA always records the overlap as a prefix of some rule; however, the overlap in the LO

[13]The states are ordered according to their states' numbers.

matching can start at any position, so every possible starting point shall be checked.

An index table is employed, called *Starting State Table* (SST) which is shown in Table 5.1. For a block B indexed by v characters, we store the minimum state number $\min\{q : Str(q, c) = B\}$ as its contents and the integer v is called the *lower threshold*. Given an input string, we match the first v-character block by searching for it in the SST and find the starting state of further matching. Table 5.1 shows an SST with lower threshold $v = 1$. The size of the index table is bounded by the size of the rule set, since the number of v-character blocks is equal to the number of unique v-character groups in the rule.

During the matching phase, we maintain a counter which is used to record the length of the current LO. It is initially set to zero and once SST successfully matches the first v-character block, it becomes v. The counter value is incremented only when a character is matched by a basic-tran or the SST. Note that the matching process is *deterministic* since for each state, there is at most one Di-trans for each specific i. Now, suppose that the incoming stream is composed of packets p_4, p_2, p_5, p_1, p_3, and we match it against $\mathcal{F}$ as the pattern by using the Skip-FA in Figure 5.3. At the first character $\mathtt{m}$ of first arriving packet p_4 ("$mentcolle$"), we

Table 5.1: *Starting State Table of the Skip-FA.*

Character	*Starting State*	*Character*	*Starting State*
a	26	***m***	11
c	9	***n***	8
e	3	***o***	16
f	4	***p***	24
h	20	***r***	2
i	15	***s***	18
l	25	***t***	14

look it up in the SST and find that we should start the match from state 11. At the same time, the counter is set to one, because `m` is successfully matched by SST. Similarly, after matching the second character `e`, the current state becomes state 12 and the counter is 2. State 13 is reached by `n` and the current counter is 3. Similarly `t` is matched by state 14 and counter is incremented to 4. When matching the fourth character `t` at state 14, we find that state 14 has no transition labeled `c`, so we use its D6-trans (since the counter value is 4 which is not bigger than 6), and reach state 32. The procedure is repeated in state 32 and transfer the current state to state 33 by successfully matching `c`, and the counter is 5 now. Similarly `o, l, l` are matched by state 35, 36, 37 respectively. Finally, `e` is successfully matched in state 37 and counter value is 9.

Steps of matching packet p_4="$mentcolle$":

$$p_4 : \xrightarrow[counter=1]{m \text{ by SST}} 11 \xrightarrow[counter=2]{\texttt{e}} 12 \xrightarrow[counter=3]{\texttt{n}} 13 \xrightarrow[counter=4]{\texttt{t}} 14 \xrightarrow[counter=4]{D6} 32 \xrightarrow[counter=5]{\texttt{c}} 33 \xrightarrow[counter=6]{\texttt{o}} 34 \xrightarrow[counter=7]{\texttt{l}} 35 \xrightarrow[counter=8]{\texttt{l}} 36 \xrightarrow[counter=9]{\texttt{e}} 37$$

The counter = 9 means that 9 characters have been matched by the signature. The length of p_4 also equals to 9, so we say the packet p_4 is matched by the signature. Similarly, the other packets matching procedure is as follows.

Steps of matching packet p_2="$mentionst$":

$$p_2 : \xrightarrow[counter=1]{m \text{ by SST}} 11 \xrightarrow[counter=2]{\texttt{e}} 12 \xrightarrow[counter=3]{\texttt{n}} 13 \xrightarrow[counter=4]{\texttt{t}} 14 \xrightarrow[counter=5]{\texttt{i}} 15 \xrightarrow[counter=6]{\texttt{o}} 16 \xrightarrow[counter=7]{\texttt{n}} 17 \xrightarrow[counter=8]{\texttt{s}} 18 \xrightarrow[counter=9]{\texttt{t}} 19$$

There are 9 characters in p_2 and counter = 9, so p_2 can be matched by the automaton.

Steps of matching packet p_5="$ction$":

$$p_5 : \xrightarrow[counter=1]{c \text{ by SST}} 9 \xrightarrow[counter=1]{D1} 33 \xrightarrow[counter=1]{D1} 38 \xrightarrow[counter=2]{\texttt{t}} 39 \xrightarrow[counter=3]{\texttt{i}} 40 \xrightarrow[counter=4]{\texttt{o}} 41 \xrightarrow[counter=5]{\texttt{n}} 42$$

There are only 5 characters in packet p_5 while counter = 5, so p_5 is also

matched.

Steps of matching packet p_1="$reference$":

$$p_1 : \xrightarrow[counter=1]{r \text{ by SST}} 2 \xrightarrow[counter=2]{\texttt{e}} 3 \xrightarrow[counter=3]{\texttt{f}} 4 \xrightarrow[counter=4]{\texttt{e}} 5 \xrightarrow[counter=5]{\texttt{r}} 6 \xrightarrow[counter=6]{\texttt{e}} 7 \xrightarrow[counter=7]{\texttt{n}}$$
$$8 \xrightarrow[counter=8]{\texttt{c}} 9 \xrightarrow[counter=9]{\texttt{e}} 10$$

Without the help of Di-trans, packet p_1 can be matched state by state until reaching state 10.

Steps of matching packet p_3="$hereplace$":

$$p_3 : \xrightarrow[counter=1]{h \text{ by SST}} 20 \xrightarrow[counter=2]{\texttt{e}} 21 \xrightarrow[counter=3]{\texttt{r}} 22 \xrightarrow[counter=4]{\texttt{e}} 23 \xrightarrow[counter=5]{\texttt{p}} 24 \xrightarrow[counter=6]{\texttt{l}}$$
$$25 \xrightarrow[counter=7]{\texttt{a}} 26 \xrightarrow[counter=8]{\texttt{c}} 27 \xrightarrow[counter=9]{\texttt{e}} 28$$

Beginning with state 20, after reading the 9 characters of packet p_3, the automaton stops at 28. The whole content can be found in the automaton.

5.4.2 Problem of Memory Usage

The memory usage of Skip-FA is closely related to the number of states, which is the size of the rule set, i.e., the total number of characters of the rules. According to the statistics from "thepiratebay" [95], the average movie size of the top 100 popular movies is 1139.89 megabytes. Similarly the average application file size of the top 100 popular applications is 1449.53 MB. It is impractical to use the content of the original file as a rule. To curb the problem of space explosion when dealing with large files, it is necessary to extract more compact representations from the original file.

5.5 Skip-stride Finite Automaton

In this section, a novel acceleration scheme is illustrated for pattern matching which converts the original byte stream into a much shorter stride-length stream (SL) and then matches it with a variant of DFA, called StriDFA as explained in Chapter 3 and Chapter 4. StriDFA is then combined with Skip-FA.

5.5.1 Building Skip-Stride Finite Automaton

Since StriDFA can reduce the memory consumption of the rules, we reconstruct Skip-FA based on StriDFA, called **Skip-Stride FA**.

According to the definition 3 of $F_x(S)$ in Subsection 3.2.2, $F_e(\mathcal{F})$ = 2 2 3 2 5 4 2 5 2 5 2 (here $\mathcal{F}$ = "$referencementionsthereplacementcollection$" and window size $w = 5$ is used). The stride length stream is used to construct StriDFA for $\mathcal{F}$. After reconstructing the StriDFA for $\mathcal{F}$ by the method outlined in Section 5.4, we finally get the Skip-Stride FA in Figure 5.4.

Instead of matching the stream byte by byte in Skip-FA, with Skip-Stride FA we can match the input stream with multiple stride length streams. Let's take the same example to illustrate how to use Skip-Stride FA to match the input stream. First we get the stride length stream of packet p_4 ("$mentcolle$") with window $w = 5$ according to the definition of $F_x(S)$ in Definition 3: $F_e(p_4)$ = 5 2. The first stride length is 5 and we look it up in the SST in Table 5.2 to find the corresponding start state 6. State 6 has no transition label with 2, so we go along its D2-trans (since the counter value is 1 which is small than 2), and reach state 9. Finally, 2 is successfully matched in state 10.

$$F_e(p_4) : \xrightarrow[counter=1]{5 \text{ by SST}} 6 \xrightarrow[counter=1]{D2} 9 \xrightarrow[counter=2]{2} 10$$

Similarly, the stride length sequences of p_2, p_5, p_1, p_3 can be calculated:

$$F_e(p_2) = F_e(\texttt{mentionst}) = 5$$
$$F_e(p_5) = F_e(\texttt{ction}) = null$$
$$F_e(p_1) = F_e(\texttt{reference}) = 2\ 2\ 3$$
$$F_e(p_3) = F_e(\texttt{hereplace}) = 2\ 5$$

With the above stride length sequences of p_2, p_5, p_1, p_3, the matching procedures using Skip-Stride FA are as follows:

$$F_e(p_2) : \xrightarrow[counter=1]{5 \text{ by SST}} 6$$

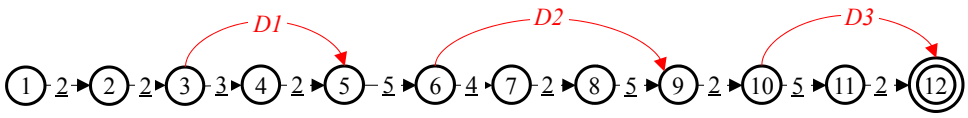


Figure 5.4: *Skip-Stride Finite Automaton of $\mathcal{F}$ with window $w = 5$.*

$$F_e(p_1) : \xrightarrow[counter=1]{\underline{2} \text{ by SST}} 2 \xrightarrow[counter=2]{\underline{2}} 3 \xrightarrow[counter=3]{\underline{3}} 4$$

$$F_e(p_3) : \xrightarrow[counter=1]{\underline{2} \text{ by SST}} 2 \xrightarrow[counter=1]{D1} 5 \xrightarrow[counter=2]{\underline{5}} 6$$

It can be seen that $F_e(p_2), F_e(p_1)$ and $F_e(p_3)$ can be matched by the Skip-Stride FA. As no tag can be found in p_5, its SL is set to null. The packet as p_5 (without tag inside) will be left out. If the rest of the packets (p_4, p_2, p_1, p_3) can be matched by the Skip-Stride FA, a possible match is alerted to the network administrator.

5.5.2 Problem of False Positive

Since the SL stream is a highly compressed form of an input stream, part of the information is left out before being sent to StriDFA. So it is only a potential match if StriDFA reports a match, because there may be false positives. For example, if given two strings T="efe" and T'="ere", we have $F_e(T') = \underline{2} = F_e(T)$. If the stride length $\underline{2}$ is matched, we are still not sure if it is T or T' that is matched.

In Figure 5.4, packet p_4 should be matched by state 11 and 12 instead of state 9 and 10. In order to reduce the probability of false positives, we propose a hybrid finite automaton in the next section.

Table 5.2: *Starting State Table of Skip-Stride Finite Automaton.*

Stride Length	***Starting State***
$\underline{2}$	2
$\underline{3}$	4
$\underline{4}$	7
$\underline{5}$	6

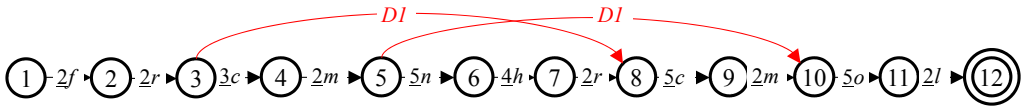


Figure 5.5: *Skip-Stride-Neighbor finite automaton with tag='e' and window w=5.*

5.6 Building Skip-Stride-Neighbor Finite Automaton

5.6.1 Skip-Stride-Neighbor Finite Automaton

Figure 5.5 depicts the combined skip-stride-neighbor finite automaton (S²N-FA). The pseudocode for building S²N-FA is given in Algorithm 5. Similarly we use an example to illustrate how to match with S²N-FA. Stride-neighbor length stream is $\underline{5}o\ \underline{2}l$ which is extracted from packet p_4 ("mentcolle"). When matching with S²N-FA, the first stride-neighbor length $\underline{5}o$ is sent to the SST table (Table 5.3) to get the next state number 11. The next stride-neighbor length $\underline{2}l$ can also be matched by final state 12, so p_4 is matched by S²N-FA.

$$N_e(p_4) : \xrightarrow[counter=1]{\underline{5}o \text{ by SST}} 11 \xrightarrow[counter=2]{\underline{2}l} 12$$

Definition 8. *Let $N_x(S)$ denote the stride-neighbor length sequences of S when using x as the tag.*

According to the above definition, the stride-neighbor length sequences of p_2, p_1 and p_3 are as follows:

$$N_e(p_2) = F_e(\texttt{mentionst}) = \underline{5}\text{n}$$
$$N_e(p_1) = F_e(\texttt{reference}) = \underline{2}\text{f}\ \underline{2}\text{r}\ \underline{3}\text{c}$$
$$N_e(p_3) = F_e(\texttt{hereplace}) = \underline{2}\text{r}\ \underline{5}\text{c}$$

Then the corresponding matching procedure with S²N-FA is as follows:

Table 5.3: *Starting State Table of Skip-Stride-Neighbor finite automaton.*

Stride-neighbor Length	$\underline{2}f$	$\underline{2}r$	$\underline{2}m$	$\underline{2}l$	$\underline{3}c$	$\underline{4}h$	$\underline{5}c$	$\underline{5}n$	$\underline{5}o$
Starting State	2	3	5	12	4	7	9	6	11

Algorithm 5: Building S²N-FA of signature $\mathcal{F} = c_i, \ldots, c_n$ with window = w

Procedure Building Stride-Neighbor DFA
$pos \leftarrow 1$; $counter \leftarrow 1$; $i \leftarrow 1$; /* pos: current position at the input */
while $i \leq n$ **do**
  if c_i = *tag and* $counter < w$ **then**
    $l_k \leftarrow i - pos + 1$ /* l: stride length, pos: current position at the input */
    $SN_k \leftarrow (l_k, c_{i-1})$; /* get the stride length l_k and current neighbor symbol c_{i-1} */
    $pos \leftarrow i$
  else if $counter = w$ **then**
    $SN_k \leftarrow (w, c_{i-1})$; /* get the stride length w and current neighbor symbol c_{i-1} */
    $pos \leftarrow i$
  $i \leftarrow i + 1$
  $k \leftarrow k + 1$
end

Procedure Building S²N-FA
$c \leftarrow 1$; $j \leftarrow |SN|$ /* j: the length of stride-neighbor length sequences */
while $j \geq 1$ **do**
  $k \leftarrow 1$
  while $k < j$ **do**
    $\alpha \leftarrow SN_k \ldots SN_{j-k+1}$
    if *Contains*(SN,α)> 2 *and Contains*(Ω,α) = 0 **then**
      /* Function Contains(A,B) means the occurrences of B in A */
      $D \cup D_{j-k+1}$; /* Add a new D_{j-k+1}-trans to D-trans set */
      $\Omega = \Omega \cup \alpha$ /* make sure every new stride-neighbor length sequences is not the sub-set of any elements in Ω */
    $k \leftarrow k + 1$
  end
  $j \leftarrow j - 1$
end
return D /* get D_i-trans and try to get corresponding state numbers then construct DFA according to the traditional methods */

$$p_2 : \xrightarrow[counter=1]{\underline{5}n \text{ by SST}} 6$$

$$p_1 : \xrightarrow[counter=1]{\underline{2}f \text{ by SST}} 2 \xrightarrow[counter=2]{\underline{2}r} 3 \xrightarrow[counter=3]{\underline{3}c} 4$$

$$p_3 : \xrightarrow[counter=1]{\underline{2}r \text{ by SST}} 3 \xrightarrow[counter=1]{D1} 8 \xrightarrow[counter=2]{\underline{5}c} 9$$

Hybrid S²N-FA has the same number of states and fewer Di-trans compared with Skip-FA. Moreover, it has the advantages of StriDFA (analyzed in Section 3.4) and the reduced probability of false positives of neighbor-based DFA.

5.6.2 Analysis of Stride-Neighbor DFA

The concept of Stride-Neighbor DFA is illustrated in subsection 4.5.1. Memory usage can often be reduced at the cost of increased false positive rate. Therefore we define the Efficiency Factor by

$$e = \frac{\Delta w}{\Delta fp} = \frac{percentage\ of\ reduced\ memory\ usage}{increased\ false\ positive\ rate}$$

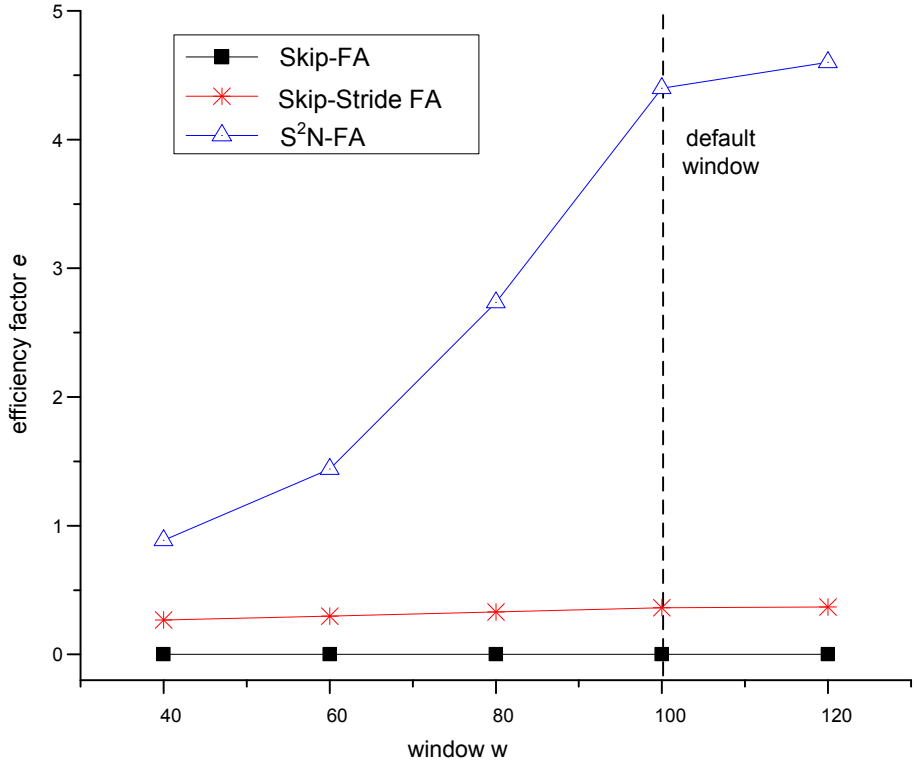


Figure 5.6: *Efficiency factor of Skip-FA, Skip-Stride FA of different window sizes.*

That is, it measures the amount of memory saved per increased false positive rate. A simple combination of Skip-FA and StriDFA cannot be made since the efficiency factor is not good ($e < 0.5$). As shown in Figure 5.6, after employing Neighbor DFA, the efficiency factor of S^{2}N-FA is much better than Skip-Stride FA ($e > 4$ when w=100). A clear turning point can be found at w=100 after which the increase of efficiency factor decelerates with respect to the window size. Consequently the **default window** size is set to the window size at the turning point (here w=100).

5.7 Experimental Results

In this section, the experimental results of S^{2}N-FA are presented. The characteristics of the experimental traces and details of the selected rules from different files are examined in Subsection 5.7.1. In Subsection 5.7.2, the rule sets are used to examine the memory consumption of S^{2}N-FA. Finally the matching speed is analyzed in Subsection 5.7.2 and the LO percentage is analyzed in Subsection 5.7.4.

5.7.1 Experiment Setup and Test Sets

The same real-life network traffic traces used in the previous chapter are used to evaluate S^{2}N-FA method. Defcon is from the Shmoo Group DefCon 17.0 Capture the Flag Contest [73]. Darpa is from the DARPA intrusion detection data sets collected by MIT Lincoln Laboratory [74]. Tsinghua trace is collected in the gateway of Tsinghua University campus network.

To evaluate the efficiency of the proposed algorithm, a full-featured prototype of the content scanning engine, S^{2}N-FA is implemented with real datasets. The S^{2}N-FA prototype is implemented in Java and Perl. The experiments are carried out on two desktop PCs, each of which has eight 3.8GHz CPU and 12 GB memory. Four test sets are chosen by the following file formats which stand for the four most typical file types in P2P file distribution: PDF: document files(.pdf), MP3: music files(.mp3), EXE: application files(.exe), AVI: movie files(.avi). The files

Table 5.4: *Basic characteristics of test sets*

File Type	Ave. Size (KB)	Default Tag	Ave. Stride	Default window size	Ave. Size *S^2N-FA*(KB)
PDF	2426	101	63	60	382.16
MP3	5370	100	112	200	268.89
EXE	7281	115	221	800	9027.33
AVI	1207281	74	286	1000	15872.61

are selected from the top 100 popular file list of thepiratebay [95], grouped by the above four file types. They are representative of some basic characteristics (shown in Table 5.4) that impact the performance of S^{2}N-FA.

5.7.2 Memory Usage

As shown in Figure 5.7, the number of Di-trans ($1 \leq i \leq 20$) and the number of transitions in SST with different threshold v are different for four file types: PDF, MP3, EXE and AVI.

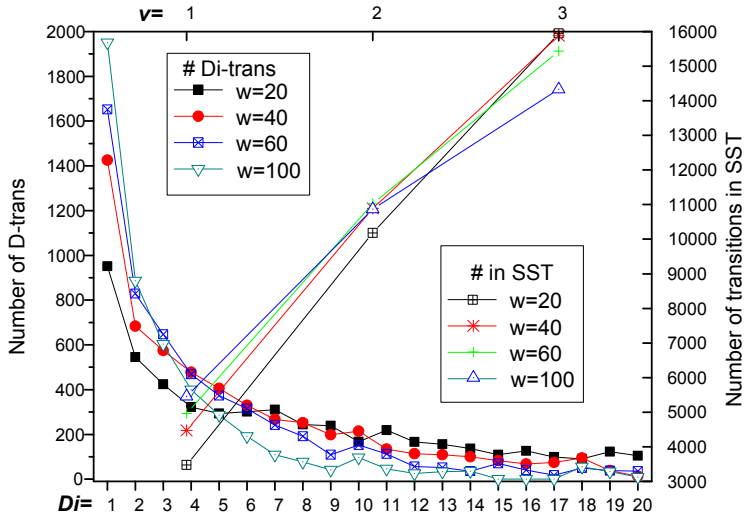
The total memory usage (M) can be estimated using the memory requirements for the basic transitions (M_b), SST (M_{sst}) and Di-trans (M_{di}). So:

$$M = M_b + M_{sst} + M_{di} = M_b + M_{di} + w|\Sigma| \approx M_b + M_{di}$$

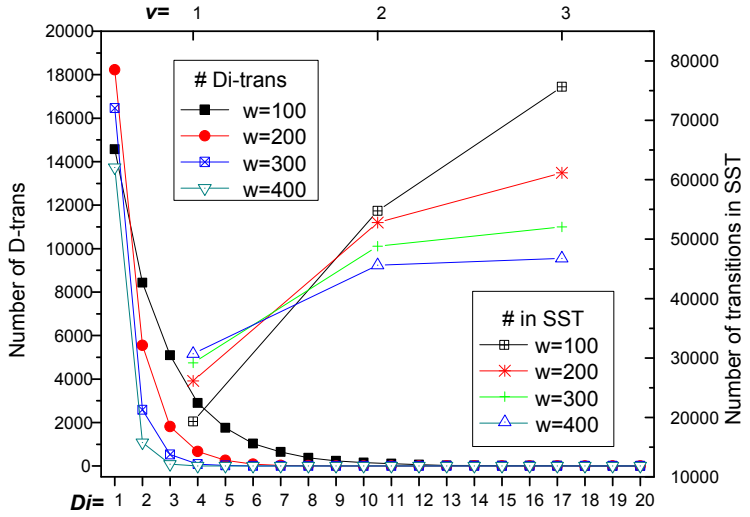
Since $w|\Sigma| = 256w \ll M_b + M_{di}$, the total memory $M \approx M_b + M_{di} = \frac{1}{w}W'$. W' is the memory usage of the original structure Skip-FA. So the total memory usage is reduced to $\frac{1}{w}$ of the original size.

5.7.3 Matching Speed

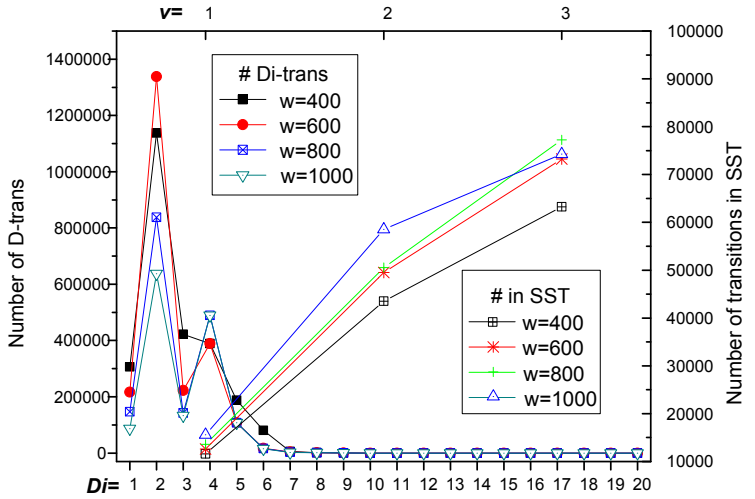
During the matching phase, the payloads of packets are first converted to a stride-neighbor stream, and then sent to S^{2}N-FA. These two stages are executed in parallel. The symbol τ is defined as the average number of bytes processed in each memory access (τ_1 for S^{2}N-FA and τ_2 for Skip-FA). We also assume that each transition costs one memory access and use $\underline{\ell}$ to represent the average stride of the incoming trace. Hence, the speedup θ is shown in the following equation:



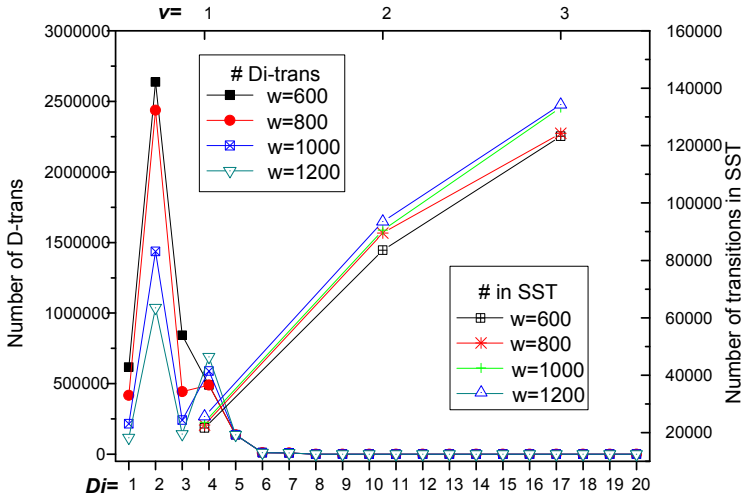
(a) PDF



(b) MP3



(c) EXE



(d) AVI

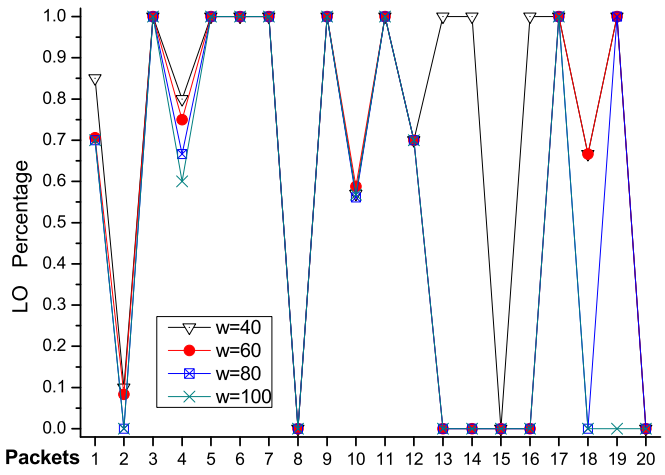
Figure 5.7: *Number of Di-trans with different window sizes and Start State Table (SST) numbers with various threshold v.*

$$\theta = \frac{\tau_1}{\tau_2} = \frac{\frac{\#\text{ of bytes} \times \underline{\ell}}{B+D}}{\frac{\#\text{ of bytes}}{B}} = \frac{\underline{\ell}}{1+\frac{D}{B}} \approx \frac{\underline{\ell}}{2}$$

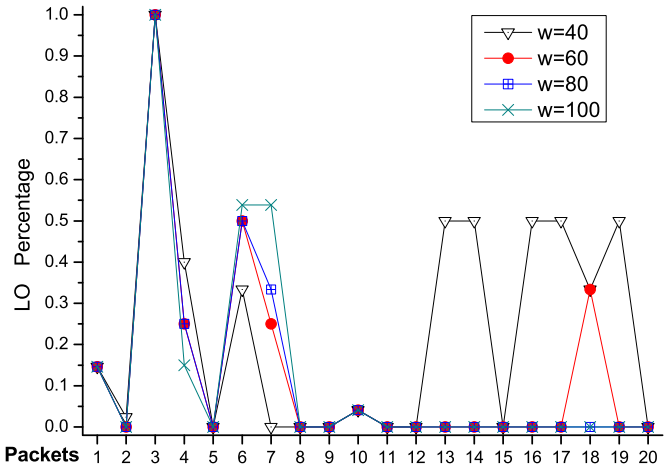
where B and D refer to the average numbers of basic-trans and the average number of Di-trans to be used, respectively. The increased speed is clearly influenced by $\underline{\ell}$. Since the the average stride of the incoming trace $\underline{\ell} \approx \frac{w}{2}$, the speedup is nearly $\frac{w}{4}$.

5.7.4 Longest Overlap Percentage

According to the match definition, if the LO part of a packet is over the threshold 0.9, then the packet is considered to be matched by the fingerprint signature $\mathcal{F}$. As shown in Figure 5.8, 20 consecutive packets are selected randomly to test the LO percentage of each packet. Two conclusions can be drawn from the figure. Firstly, the bigger the window size, the lower the LO percentage. Secondly, by introducing neighbor DFA, the LO percentage is significantly reduced when matching the same packet with S²N-FA as opposed to Skip-Stride FA for the same. For example, considering the 7^{th} packet, we find the LO percentage is 1.0 in Skip-Stride FA



(a) Skip-Stride FA



(b) S^{2}N-FA

Figure 5.8: *The LO percentage of Skip-Stride FA and S^{2}N-FA in 20 consecutive packets.*

while the LO percentage reduced to 0 in S^{2}N-FA when w=40. By adding neighbor DFA, the LO percentage can be reduced by 40% on average which also means a relatively low probability of false positives.

5.8 Conclusion

Given the increasing speed of the Internet and the importance of content scanning in DLPS and CIDS, it is imperative that high-throughput, memory-efficient file detection methods continue to be improved. It is a challenge to match the out-of-order data with the selected fingerprint signatures.

In this chapter a high speed and memory-efficient structure, the *Skip-Stride-Neighbor Finite Automaton*(*S^{2}N-FA*) is proposed. The hybrid finite automaton converts the original byte stream to stride-neighbor streams and matches the longest overlap part with the signatures. S^{2}N-FA is easy to implement in both software and hardware. On average, S^{2}N-FA consumes about 60% less memory and achieves about 20-fold speedup.

CHAPTER 6

Conclusions and Future Work

This chapter concludes the thesis by summarizing the major contributions of my work and suggesting some directions for future work.

6.1 Summary of thesis work

In this thesis, key network security issues have been addressed. It has been indicated that NIDS rulesets are becoming larger and larger including increasingly more complex payload descriptions. Continuously faster network processing requirements are posing significant performance and implementation challenges in intrusion detection systems. Deep Packet Inspection (DPI) functionality has been developed and widely deployed in NIDS. The processing time of DFA is proportional to the length of the input string, but the storage cost of Deterministic Finite Automaton (DFA) is quite high. The link bandwidth and Internet traffic are rapidly increasing. The size of the attack signature database is also growing larger and larger due to the diversification of the attacks. So high performance and low storage cost NIDS is required. Traditional software-based or hardware-based pattern matching algorithms are unable to satisfy the processing speed requirement. High performance network payload inspection methods are needed to enable deep packet inspection at line rate. DFA-based DPI matching methods

are powerful in packet inspection for NIDS. However, traditional DFA-based DPI solutions cannot be used for inspection of file distribution in P2P networks due to the potential out-of-order delivery of data. While technologies such as instant messaging software (Skype, MSN) or BitTorrent file sharing methods are becoming more and more important for today's Internet because of the basic need of increasing information transfer and huge file sharing.

6.1.1 StriFA

This thesis introduces a novel finite automata family, StriNFA and StriDFA to accelerate multi-string and multi-regular expression matching. Compared with other algorithms that also examine multiple characters at a time, StriFA is immune to the memory blow-up and byte alignment problems, and therefore requires much less memory. An example of StriDFA for multi-string matching is given in Chapter 3. One of the challenges of StriDFA for multi-string matching is it cannot extract stride length sequences because of the wildcards. In Chapter 4, two methods are proposed to solve this challenge.

6.1.2 S^2N-FA

The Skip-FA is used to solve the out-of-order data scanning problem for file detection in P2P networks. However, the memory usage of Skip-FA is large. In order to reduce the memory cost of Skip-FA, a hybrid finite automaton named Skip-stride FA. Neighbor-DFA is used to achieve a low false positive rate at the additional cost of a small increase in memory consumption. Finally S^2N-FA is generated which converts the original byte stream to stride-neighbor streams and matches the longest overlap part with the signatures. S^2N-FA is easy to implement in both software and hardware.

6.2 Future work

There are some interesting directions for future work that one could pursue based on the work presented in this thesis.

6.2.1 Tag selection

In future, a more scalable tag selection method needs to be designed. For a given ruleset, data mining or other methods may select tags automatically. With the changing of the characteristics of the incoming trace, the matching engine may change the tag according to the historical statistics.

Sometimes a pattern cannot be covered by a chosen tag. For example, for a given tag 'e', "`reference`" (P_1) can be covered by the tag 'e'. However "`fregg`" cannot be covered by the tag 'e' because there is only one character equals to the tag. So a stride length cannot be even extracted from it. In this situation, the tag 'e' (8-bits character) is not suitable for using as a tag. In fact, a tag can be some contiguous characters or some bits of a character. Here we give the formal definition of n-bit tag.

Definition 9. *A n-bit tag is n bits fingerprints of a character if $n \leq 8$; A n-bit tag is n bits fingerprints of more than one continuous characters if $n > 8$.*

Definition 10. *In each window, select n-bit contiguous fingerprints of each character. If these n-bit fingerprints can be found in the tag set, mark the corresponding $\lceil \frac{n}{8} \rceil$ characters as the tag and keep on processing from $\lceil \frac{n}{8} \rceil + 1$. If there is no tag inside a window, select the rightmost $\lceil \frac{n}{8} \rceil$ character as the tag of the window (Ceiling function $\lceil x \rceil$ returns the smallest integer not less than x).*

In subsection 3.2.2, the character which has the most frequency is chose as the tag (8-bit tag 'e' is chose in that example). In practice, we can choose different hash functions to get appropriate tags according to different requirements. A tag can be 8-bit character, it can also be 16-bit characters, or 6-bit character. In future, the readers could study and compare different tag selection algorithms so as to achieve a better performance.

6.2.2 Hardware implementation

StriFA-based matching method can be easily implemented on existing hardware or software, since the StriFA has exactly the same logic structure as a traditional NFA/DFA.

There are four different combinations in StriFA-based matching system (StriFA and NFA/DFA verification module): StriNFA + DFA, StriDFA + DFA, StriNFA + NFA and StriDFA + DFA. StriDFA is faster than StriNFA and DFA is faster than NFA. So the fastest combination of StriFA-based matching system is StriDFA + DFA. Similarly, StriNFA consumes smaller memory cost than StriDFA and NFA consumes smaller memory cost than DFA too. Obviously, the most memory-efficient choice is StriNFA + NFA. This is an interesting topic to test the performance of different combinations in StriFA-based hardware implementation matching system for future researches.

Appendix

I Convert Regex to NFA

To convert a regex to a NFA, two functions are defined as follows:

- The $\varepsilon - closure$ function takes a state and returns the set of states reachable from it based on (one or more) $\varepsilon - closure$. Note that this will always include the state itself.
- The function **move()** takes a state and a character, and returns the set of states reachable by one transition on this character.

Both of these functions are generalized to apply to sets of states by taking the union of the application to individual states. E.g., If A, B and C are states, move({A,B,C},'a') = $move(A, `a') \cup move(B, `a') \cup move(C, `a')$.

Suppose the regex rule is `.*abba.{2}caca`. Figure A-1 is the NFA structure for this regex.

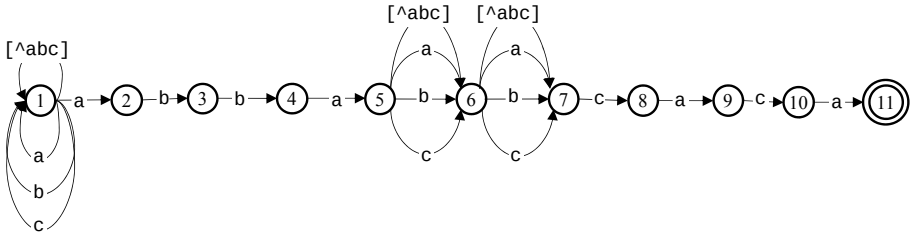


Figure A-1: *NFA of* `.*abba.{2}caca`

According to the above definitions, the transition relationship is listed as follows step by step:

$$\varepsilon - closure(move(\{1\}, a)) = \{1, 2\} \quad \text{(Step-1)}$$

$$\varepsilon - closure(move(\{1\}, b)) = \{1\} \quad \text{(Step-2)}$$

$$\varepsilon - closure(move(\{1\}, c)) = \{1\} \quad \text{(Step-3)}$$

$$\varepsilon - closure(move(\{1\}, [\hat{}abc])) = \{1\} \quad \text{(Step-4)}$$

$$\varepsilon - closure(move(\{1, 2\}, a)) = \varepsilon - closure(\{1, 2\}) = \{1, 2\} \quad \text{(Step-5)}$$

$$\varepsilon - closure(move(\{1, 2\}, b)) = \varepsilon - closure(\{1, 3\}) = \{1, 3\} \quad \text{(Step-6)}$$

$$\varepsilon - closure(move(\{1, 2\}, c)) = \varepsilon - closure(\{1\}) = \{1\} \quad \text{(Step-7)}$$

$$\varepsilon - closure(move(\{1, 3\}, a)) = \varepsilon - closure(\{1, 2\}) = \{1, 2\} \quad \text{(Step-8)}$$

$$\varepsilon - closure(move(\{1, 3\}, b)) = \varepsilon - closure(\{1, 4\}) = \{1, 4\} \quad \text{(Step-9)}$$

$$\varepsilon - closure(move(\{1, 4\}, a)) = \{1, 2, 5\} \quad \text{(Step-10)}$$

$$\varepsilon - closure(move(\{1, 2, 5\}, a)) = \{1, 2, 6\} \quad \text{(Step-11)}$$

$$\varepsilon - closure(move(\{1, 2, 5\}, b)) = \{1, 3, 6\} \quad \text{(Step-12)}$$

$$\varepsilon - closure(move(\{1, 2, 5\}, c)) = \{1, 6\} \quad \text{(Step-13)}$$

$$\varepsilon - closure(move(\{1, 2, 5\}, [\hat{}abc])) = \{1, 6\} \quad \text{(Step-14)}$$

$$\varepsilon - closure(move(\{1, 2, 6\}, a)) = \{1, 2, 7\} \quad \text{(Step-15)}$$

$$\varepsilon - closure(move(\{1, 2, 6\}, b)) = \{1, 3, 7\} \quad \text{(Step-16)}$$

$$\varepsilon - closure(move(\{1, 2, 6\}, c)) = \{1, 7\} \quad \text{(Step-17)}$$

$$\varepsilon - closure(move(\{1, 2, 6\}, [abc])) = \{1, 7\} \quad \text{(Step-18)}$$

$$\varepsilon - closure(move(\{1, 3, 6\}, a)) = \{1, 2, 7\} \quad \text{(Step-19)}$$

$$\varepsilon - closure(move(\{1, 3, 6\}, b)) = \{1, 4, 7\} \quad \text{(Step-20)}$$

$$\varepsilon - closure(move(\{1,3,6\},c)) = \{1,7\} \quad \text{(Step-21)}$$

$$\varepsilon - closure(move(\{1,3,6\},[\hat{}abc])) = \{1,7\} \quad \text{(Step-22)}$$

$$\varepsilon - closure(move(\{1,6\},a)) = \{1,7\} \quad \text{(Step-23)}$$

$$\varepsilon - closure(move(\{1,6\},b)) = \{1,7\} \quad \text{(Step-24)}$$

$$\varepsilon - closure(move(\{1,6\},c)) = \{1,7\} \quad \text{(Step-25)}$$

$$\varepsilon - closure(move(\{1,6\},[\hat{}abc])) = \{1,7\} \quad \text{(Step-26)}$$

$$\varepsilon - closure(move(\{1,7\},a)) = \{1,2\} \quad \text{(Step-27)}$$

$$\varepsilon - closure(move(\{1,7\},c)) = \{1,8\} \quad \text{(Step-28)}$$

$$\varepsilon - closure(move(\{1,2,7\},a)) = \{1,2\} \quad \text{(Step-29)}$$

$$\varepsilon - closure(move(\{1,2,7\},b)) = \{1,3\} \quad \text{(Step-30)}$$

$$\varepsilon - closure(move(\{1,2,7\},c)) = \{1,8\} \quad \text{(Step-31)}$$

$$\varepsilon - closure(move(\{1,3,7\},b)) = \{1,4\} \quad \text{(Step-32)}$$

$$\varepsilon - closure(move(\{1,3,7\},c)) = \{1,8\} \quad \text{(Step-33)}$$

$$\varepsilon - closure(move(\{1,4,7\},a)) = \{1,2,5\} \quad \text{(Step-34)}$$

$$\varepsilon - closure(move(\{1,4,7\},c)) = \{1,8\} \quad \text{(Step-35)}$$

$$\varepsilon - closure(move(\{1,8\},a)) = \{1,2,9\} \quad \text{(Step-36)}$$

$$\varepsilon - closure(move(\{1,2,9\},b)) = \{1,3\} \quad \text{(Step-37)}$$

$$\varepsilon - closure(move(\{1,2,9\},c)) = \{1,10\} \quad \text{(Step-38)}$$

$$\varepsilon - closure(move(\{1,10\},a)) = \{1,2,11\} \quad \text{(Step-39)}$$

$$\varepsilon - closure(move(\{1,2,11\},b)) = \{1,3\} \quad \text{(Step-40)}$$

With all above transitions, the corresponding original DFA is shown in Figure A-2. For better reading, as shown in Figure A-3, the original DFA can be converted to a new DFA by renumbering the states.

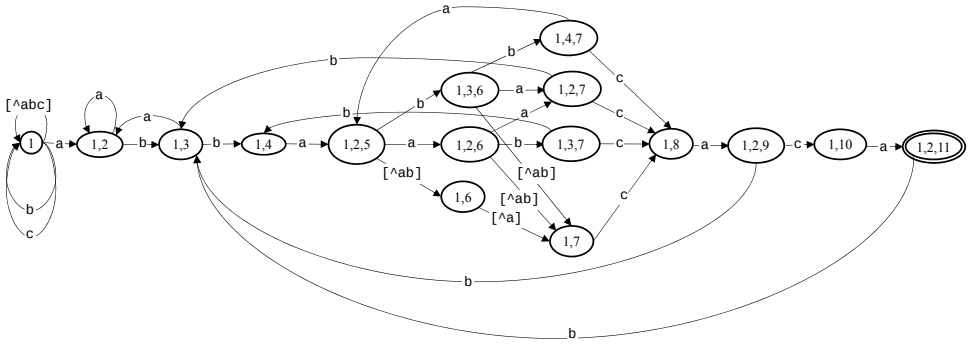


Figure A-2: *Original DFA of* `.*abba.{2}caca`.

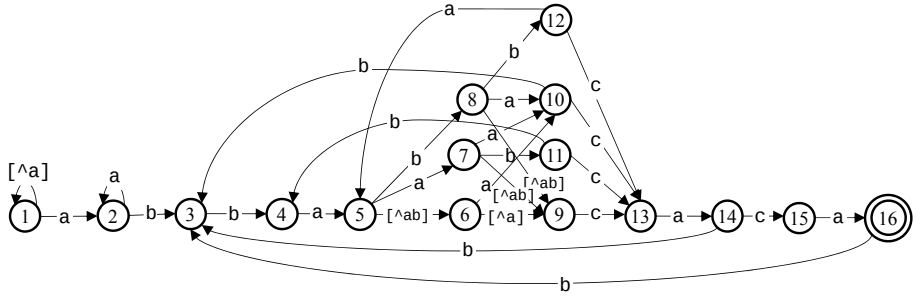


Figure A-3: *DFA of* `.*abba.{2}caca` *after renumbering.*

II Restructure Traditional DFA to StriDFA

Figure A-4 shows the Tag decision FA which is explained in subsection 4.3.1. Transforming Tag decision FA to StriNFA with the algorithm described in step 4 of the subsection 4.3.1, we can get StriNFA (shown in Figure A-5) after state renumbering.

The corresponding StriDFA can be constructed from StriNFA in Figure A-5 by the traditional construction method which is called "structural induction" in

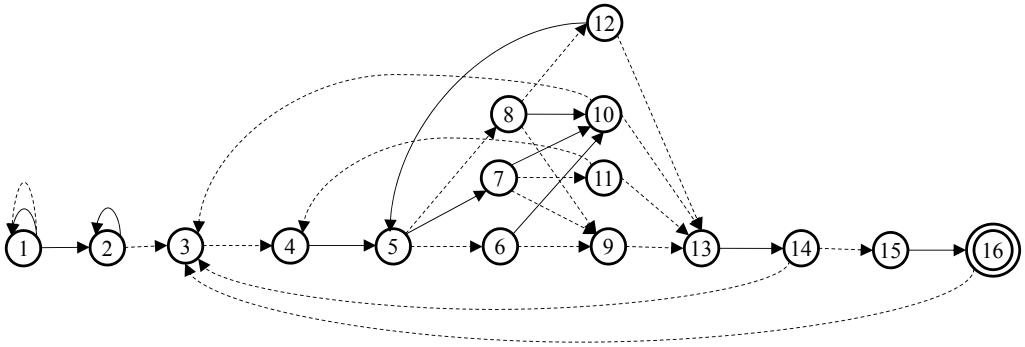


Figure A-4: *Restructure DFA of* `.*abba.{2}caca` *by classifying transitions.*

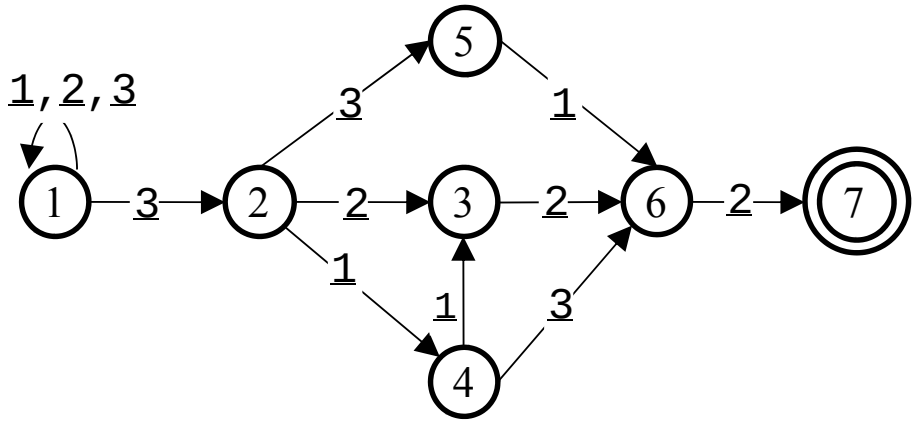


Figure A-5: *StriNFA of* `.*abba.{2}caca`.

textbooks [20]. The detailed steps can be found from Step-41 to Step-70. As shown in Figure A-6, the StriDFA can be generated according to the above steps. Finally, the StriDFA after renumbering is shown in Figure A-7.

$$\varepsilon - closure(move(\{1\}, \underline{1})) = \{1\} \quad \text{(Step-41)}$$

$$\varepsilon - closure(move(\{1\}, \underline{2})) = \{1\} \quad \text{(Step-42)}$$

$$\varepsilon - closure(move(\{1\}, \underline{3})) = \{1, 2\} \quad \text{(Step-43)}$$

$$\varepsilon - closure(move(\{1, 2\}, \underline{1})) = \{1, 4\} \quad \text{(Step-44)}$$

$$\varepsilon - closure(move(\{1, 2\}, \underline{2})) = \{1, 3\} \quad \text{(Step-45)}$$

$$\varepsilon - closure(move(\{1, 2\}, \underline{3})) = \{1, 2, 5\} \quad \text{(Step-46)}$$

$$\varepsilon - closure(move(\{1, 4\}, \underline{1})) = \{1, 3\} \quad \text{(Step-47)}$$

$$\varepsilon - closure(move(\{1, 4\}, \underline{2})) = \{1\} \quad \text{(Step-48)}$$

$$\varepsilon - closure(move(\{1, 4\}, \underline{3})) = \{1, 2, 6\} \quad \text{(Step-49)}$$

$$\varepsilon - closure(move(\{1, 3\}, \underline{1})) = \{1\} \quad \text{(Step-50)}$$

$$\varepsilon - closure(move(\{1, 3\}, \underline{2})) = \{1, 6\} \quad \text{(Step-51)}$$

$$\varepsilon - closure(move(\{1, 3\}, \underline{3})) = \{1, 2\} \quad \text{(Step-52)}$$

$$\varepsilon - closure(move(\{1, 2, 3\}, \underline{1})) = \{1, 4, 6\} \quad \text{(Step-53)}$$

$$\varepsilon - closure(move(\{1,2,3\},\underline{2})) = \{1,3\} \quad \text{(Step-54)}$$

$$\varepsilon - closure(move(\{1,2,3\},\underline{3})) = \{1,2,5\} \quad \text{(Step-55)}$$

$$\varepsilon - closure(move(\{1,2,6\},\underline{1})) = \{1,4\} \quad \text{(Step-56)}$$

$$\varepsilon - closure(move(\{1,2,6\},\underline{2})) = \{1,3,7\} \quad \text{(Step-57)}$$

$$\varepsilon - closure(move(\{1,2,6\},\underline{3})) = \{1,2,5\} \quad \text{(Step-58)}$$

$$\varepsilon - closure(move(\{1,6\},\underline{1})) = \{1\} \quad \text{(Step-59)}$$

$$\varepsilon - closure(move(\{1,6\},\underline{2})) = \{1,7\} \quad \text{(Step-60)}$$

$$\varepsilon - closure(move(\{1,6\},\underline{3})) = \{1,2\} \quad \text{(Step-61)}$$

$$\varepsilon - closure(move(\{1,4,6\},\underline{1})) = \{1,3\} \quad \text{(Step-62)}$$

$$\varepsilon - closure(move(\{1,4,6\},\underline{2})) = \{1,7\} \quad \text{(Step-63)}$$

$$\varepsilon - closure(move(\{1,4,6\},\underline{3})) = \{1,2,6\} \quad \text{(Step-64)}$$

$$\varepsilon - closure(move(\{1,3,7\},\underline{1})) = \{1\} \quad \text{(Step-65)}$$

$$\varepsilon - closure(move(\{1,3,7\},\underline{2})) = \{1,6\} \quad \text{(Step-66)}$$

$$\varepsilon - closure(move(\{1,7\},\underline{3})) = \{1,2\} \quad \text{(Step-67)}$$

$$\varepsilon - closure(move(\{1,7\},\underline{1})) = \{1\} \quad \text{(Step-68)}$$

$$\varepsilon - closure(move(\{1,7\},\underline{2})) = \{1\} \quad \text{(Step-69)}$$

$$\varepsilon - closure(move(\{1,7\},\underline{3})) = \{1,2\} \quad \text{(Step-70)}$$

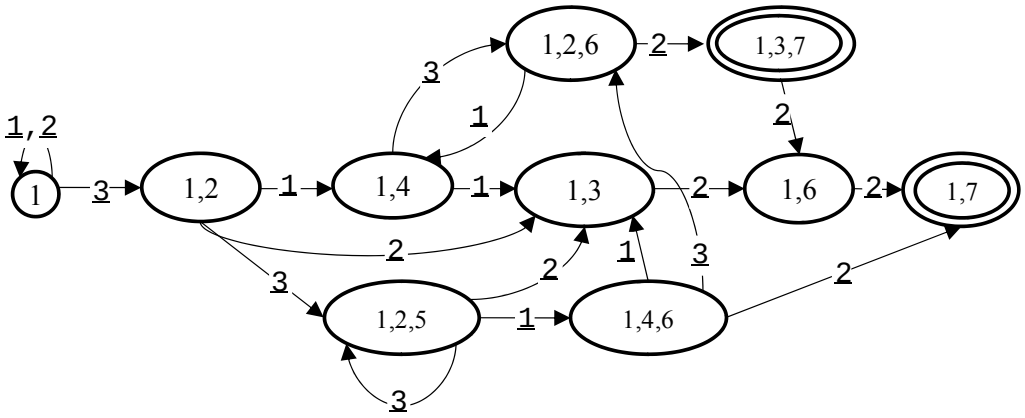


Figure A-6: *StriDFA of* `.*abba.{2}caca` *after determination.*

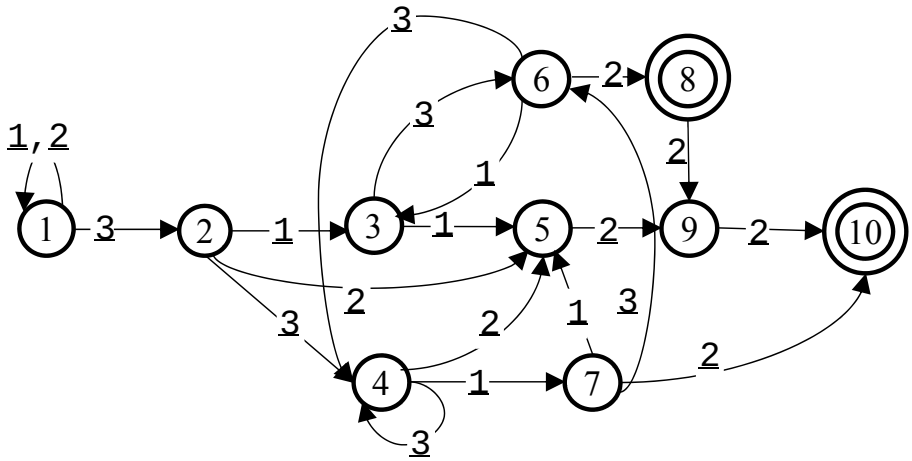


Figure A-7: *StriDFA of* `.*abba.{2}caca` *after renumbering and minimization.*

III Regular Expression Syntax

The detailed regular expression syntax is as follows:

Table A.1: *Detailed regular expression syntax.*

Symbol	Description	Example
.	Matches any single character except newline	a. matches aa, ab, ac
*	Repeats the previous item zero or more times.	a* matches a, aa, aaa, . . .
^	Matches beginning of line.	^a matches ab, ac or aa
$	Matches end of line.	a$ matches ba, ca or aa
+	Repeats the previous item once or more.	a+ matches a, aa, aaa
?	Matches **zero** or more instances of previous item.	abc? matches ab or abc
|	Causes the regex engine to match either the part on the left side, or the part on the right side.	a|b|c matches a, b or c
[ ]	Matches a single character that is contained within the brackets.	[abc] matches a, b, or c
[^]	Matches a single character that is not contained within the brackets.	[^abc] matches any character other than a, b, or c
{n}	Repeats the previous item exactly n times.	a{3} matches aaa
{n,m}	Repeats the previous item between n and m times.	a{1,3} matches a, aa or aaa
\b	Matches the empty string, but only at the beginning or end of a word.	\bfoo\b matches “foo” but not “foobar”.
\d	matches any decimal digit, it is equivalent to the set [0-9].	\d{2} matches 25.
\w	matches any alphanumeric character and the underscore, it is equivalent to the set [a-zA-Z0-9_].	\w{2} matches ad, ac, or eZ.

REFERENCES

[1] B.M. Leiner, V.G. Cerf, D.D. Clark, R.E. Kahn, L. Kleinrock, D.C. Lynch, J. Postel, L.G. Roberts, and S. Wolff. A brief history of the Internet. *ACM SIGCOMM Computer Communication Review*, 39(5):22–31, 2009.

[2] Minnesota Internet Traffic Studies (MINTS). `http://www.dtc.umn.edu/mints/home.php`, 2012.

[3] Cisco Visual Networking Index. `http://www.cisco.com/en/US/netsol/ns827/networking_solutions_sub_solution.html`, 2012.

[4] K. Cho, K. Fukuda, H. Esaki, and A. Kato. Observing slow crustal movement in residential user traffic. In *Proceedings of the 2008 ACM CoNEXT Conference*, pages 1–12. ACM, 2008.

[5] C. Labovitz, S. Iekel-Johnson, D. McPherson, FJJ Oberheide, and M. Karir. ATLAS Internet observatory 2009 annual report. *NANOG47, http://tinyurl.com/yz7xwvv*, 2009.

[6] Internet Bandwidth Growth: Dealing with Reality. `http://www.isoc.org/isoc/conferences/bwpanel/`, 2012.

[7] Trend Micro Annual Report. `http://us.trendmicro.com/us/trendwatch/research-and-analysis/threat-reports/`, 2012.

[8] L. Grossman. Attack of the Love Bug'. *Time Atlantic*, 155(19):17–23, 2009.

[9] MessageLabs Intelligence: 2010 Annual Security Report. http://www.messagelabs.com/resources/mlireports, 2012.

[10] Blaster (computer worm). http://en.wikipedia.org/wiki/Blaster_(computer_worm).

[11] R. Oliver and T. Mavens. Countering SYN flood denial-of-service attacks. In *Invited Talks of USENIX Security Symposium*, 2001.

[12] Denial of Service (DDoS) Attacks/tools. http://staff.washington.edu/dittrich/misc/ddos/, 2012.

[13] Win32/Blaster: A Case Study From Microsoft's Perspective. http://download.microsoft.com/download/b/3/b/b3ba58e0-2b3b-4aa5-a7b0-c53c42b270c6/Blaster_Case_Study_White_Paper.pdf, 2012.

[14] UN's website breached by hackers. http://news.bbc.co.uk/2/hi/6943385.stm, 2012.

[15] FBI Uncovers Worldwide $9M ATM Card Scam. http://www.foxnews.com/story/0,2933,487184,00.html, 2012.

[16] I. Sourdis and S.H. Katamaneni. Longest prefix match and updates in range tries. In *Application-Specific Systems, Architectures and Processors (ASAP)*, pages 51–58. IEEE, 2011.

[17] I. Sourdis, J. Bispo, J.M.P. Cardoso, and S. Vassiliadis. Regular expression matching in reconfigurable hardware. *Journal of Signal Processing Systems*, 51(1):99–121, 2008.

[18] J.B.D. Cabrera, J. Gosar, W. Lee, and R.K. Mehra. On the statistical distribution of processing times in network intrusion detection. In *Decision and*

Control, 2004. CDC. 43rd IEEE Conference on, volume 1, pages 75–80. IEEE, 2005.

[19] G. Navarro and M. Raffinot. *Flexible pattern matching in strings*. Cambridge University Press Cambridge, 2007.

[20] J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, second edition, 2001.

[21] B. W. Watson. A Taxonomy of Finite Automata Construction Algorithms. Computing Science Note 93/43, Eindhoven University of Technology, The Netherlands.

[22] K. Thompson. Regular Expression Search Algorithm. *Comm. ACM 11(6), June 1968, pp. 419-422*.

[23] Regex Syntex. `http://www.regular-expressions.info/reference.html`, 2012.

[24] G. Berry and R. Sethi. From regular expressions to deterministic automata. *Theoretical computer science*, 48:117–126, 1986.

[25] J.A. Brzozowski. Derivatives of regular expressions. *Journal of the ACM (JACM)*, 11(4):481–494, 1964.

[26] A. Bruggemann-Klein. Regular expressions into finite automata. *Theoretical Computer Science*, 120(2):197–213, 1993.

[27] C.H. Chang and R. Paige. From regular expressions to DFA's using compressed NFA's. In *Combinatorial Pattern Matching*, pages 90–110. Springer, 1992.

[28] A.V. Aho, M.S. Lam, R. Sethi, and J.D. Ullman. *Compilers: principles, techniques, and tools*. Pearson/Addison Wesley, 2007.

[29] B. W. Watson. A Taxonomy of Finite Automata Minimization Algorithms. Computing Science Note 93/44, Eindhoven University of Technology, The Netherlands.

[30] J. E. Hopcroft. An $nlogn$ Algorithm for Minimizing the States in a Finite Automaton. *Z. Kohavi (ed.) The Theory of Machines and Computations, Academic Press, New York, pp. 189-196*.

[31] ipoque Internet Studies. `http://www.ipoque.com`, 2012.

[32] Movie provider: Voltage pictures. `http://voltagepictures.com`, 2012.

[33] K. Bigelow. Voltage pictures, 2008.

[34] Seventh annual bsa and idc global software piracy study. `http://portal.bsa.org/globalpiracy2009/index.html`, 2012.

[35] J.H. Wang, C. Wang, J. Yang, and C. An. A study on key strategies in P2P file sharing systems and ISPs P2P traffic management. *Peer-to-Peer Networking and Applications*, pages 1–10.

[36] SNORT : Network Intrusion Detection System. http://www.snort.org/.

[37] L7-filter:Application Layer Packet Classifier for Linux. http://l7-filter.sourceforge.net/.

[38] R. Smith, C. Estan, S. Jha, and S. Kong. Deflating the big bang: fast and scalable deep packet inspection with extended finite automata. In *Proc. of ACM SIGCOMM*, 2008.

[39] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In *Proc. of ACM SIGCOMM*, 2007.

[40] S. Kumar, B. Chandrasekaran, J. Turner, and G. Varghese. Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia. In *Proc. of ACM/IEEE Symposium on Architectures for Networking and Communcations Systems (ANCS)*, 2007.

[41] M. Becchi and P. Crowley. An improved algorithm to accelerate regular expression evaluation. In *Proc. of ACM/IEEE Symposium on Architectures for Networking and Communcations Systems (ANCS)*, 2007.

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

[43] R.S. Boyer and J.S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

[44] B. Commentz-Walter. A string matching algorithm fast on the average. *Automata, Languages and Programming*, pages 118–132, 1979.

[45] C.J. Coit, S. Staniford, and J. McAlerney. Towards faster string matching for intrusion detection or exceeding the speed of snort. In *DARPA Information Survivability Conference and Exposition II, (DISCEX'01)*, volume 1, pages 367–373. IEEE, 2001.

[46] M. Fisk. Fast content-based packet handling for intrusion detection. Technical report, DTIC Document, 2001.

[47] L. Tan and T. Sherwood. A high throughput string matching architecture for intrusion detection and prevention. In *Proc. of ISCA*, 2005.

[48] H. Lu, K. Zheng, B. Liu, X. Zhang, and Y. Liu. A memory-efficient parallel string matching architecture for high-speed intrusion detection. *IEEE JSAC*, 24(10), 2006.

[49] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood. Deep packet inspection using parallel bloom filters. In *IEEE Micro, 24(1): 52-61.*, 2004.

[50] S. Dharmapurikar and J. Lockwood. Fast and scalable pattern matching for content filtering. In *Architecture for networking and communications systems, 2005. ANCS 2005. Symposium on*, pages 183–192. IEEE, 2005.

[51] N. Hua, H. Song, and T.V. Lakshman. Variable-stride multi-pattern matching for scalable deep packet inspection. In *Proc. of INFOCOM*, 2009.

[52] S. Schleimer, D.S. Wilkerson, and A. Aiken. Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 76–85. ACM, 2003.

[53] Z. Li, G. Xia, H. Gao, Y. Tang, Y. Chen, B. Liu, J. Jiang, and Y. Lv. Netshield: Matching with a large vulnerability signature ruleset for high performance network defense. In *Proc. of ACM SIGCOMM*, 2010.

[54] M. Becchi and P. Crowley. Efficient regular expression evaluation: theory to practice. In *ANCS*, pages 50–59, 2008.

[55] L. Vespa, N. Weng, and R. Ramaswamy. Ms-dfa: Multiple-stride pattern matching for scalable deep packet inspection. *The Computer Journal*, 54(2):285, 2011.

[56] D. Ficara, G. Antichi, A.D. Pietro, S. Giordano, G. Procissi, and F. Vitucci. Sampling techniques to accelerate pattern matching in network intrusion detection systems. In *Proc. of IEEE ICC*, 2010.

[57] P.C. Wu, F.J. Wang, and K.R. Young. Scanning regular languages by dual finite automata. *ACM Sigplan Notices*, 27(4):12–16, 1992.

[58] M. Kobayashi, T. Murase, and A. Kuriyama. A longest prefix match search engine for multi-gigabit ip processing. In *Communications, 2000. ICC 2000. 2000 IEEE International Conference on*, volume 3, pages 1360–1364, 2000.

[59] R.S. Boyer and J.S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

[60] H.Lu, K.Zheng, B.Liu, X.Zhang, and Y.Liu. A memory-efficient parallel string matching architecture for high-speed intrusion detection. *IEEE Journal on Selected Areas in Communications*, 24(10), 2006.

[61] S. Dharmapurikar and J. W. Lockwood. Fast and scalable pattern matching for network intrusion detection engines. *IEEE JSAC*, 24(10), 2006.

[62] R. Smith, C. Estan, and S. Jha. Xfas: Faster signature matching with extended automata. In *IEEE Symposium on Security and Privacy (Oakland)*, 2008.

[63] M. R. Garey and D. S.n Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. 1979.

[64] Qutoes from the book of Harry Potter. `http://en.wikiquote.org/wiki/Harry_Potter_and_the_Philosopher's_Stone_(film)`, 2012.

[65] Clam antivirus, 2009. `http://www.clamav.net/`.

[66] Cisco ios ips deployment guide. `www.cisco.com`.

[67] T. Song, W. Zhang, D. Wang, and Y. Xue. A memory efficient multiple pattern matching architecture for network security. In *Proc. of IEEE Conference on Computer Communications(INFOCOM)*, 2008.

[68] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz. Fast and memory-efficient regular expression matching for deep packet inspection. In *Proc.*

of ACM/IEEE Symposium on Architectures for Networking and Communcations Systems (ANCS), 2006.

[69] M. Becchi and P. Crowley. A hybrid finite automaton for practical deep packet inspection. In *Proc. of ACM International Conference on emerging Networking EXperiments and Technologies(CoNEXT)*, 2007.

[70] S. Kumar, J. Turner, and J. Williams. Advanced algorithms for fast and scalable deep packet inspection. In *Proc. of ACM/IEEE Symposium on Architectures for Networking and Communcations Systems (ANCS)*, 2006.

[71] J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to Antomata Theory, Languages and Computation*. Addison Wesley.

[72] S.J. Horng, P. Fan, Y.P. Chou, Y.C. Chang, and Y. Pan. A feasible intrusion detector for recognizing iis attacks based on neural networks. *Computers & Security*, 27(3):84–100, 2008.

[73] Def con 17 archive. `https://www.defcon.org/html/links/dc-archives/dc-17-archive.html`, 2011.

[74] Mit darpa intrusion detection data sets. `http://www.ll.mit.edu/mission/communications/ist/corpora/ideval/data/index.html`, 2012.

[75] Cable news network. `http://www.cnn.com/`.

[76] Multi-purpose web crawler. `http://larbin.sourceforge.net/index-eng.html/`, 2012.

[77] S. Webb, J. Caverlee, and C. Pu. Introducing the webb spam corpus: Using email spam to identify web spam automatically. In *Proceedings of the 3rd Conference on Email and Anti-Spam (CEAS)*.

[78] Phishing corpus. http://monkey.org/~jose/wiki/doku.php?id=phishingcorpus, 2012.

[79] M. Becchi and P. Crowley. A hybrid finite automaton for practical deep packet inspection. In *Proceedings of the 2007 ACM CoNEXT conference*, pages 1–12. ACM, 2007.

[80] F. Yu, Z. Chen, Y. Diao, TV Lakshman, and R.H. Katz. Fast and memory-efficient regular expression matching for deep packet inspection. In *Architecture for Networking and Communications Systems. ANCS 2006. ACM/IEEE Symposium on*, pages 93–102. IEEE, 2008.

[81] R. Ramaswamy, L. Kencl, and G. Iannaccone. Approximate fingerprinting to accelerate pattern matching. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 301–306. ACM, 2006.

[82] M.O. Rabin. *Fingerprinting by random polynomials*. Center for Research in Computing Techn., Aiken Computation Laboratory, Univ., 1981.

[83] U. Manber et al. Finding similar files in a large file system. In *Proceedings of the USENIX winter 1994 technical conference*, pages 1–10. Citeseer, 1994.

[84] H. Pucha, D.G. Andersen, and M. Kaminsky. Exploiting similarity for multi-source downloads using file handprints. In *Proc. 4th USENIX NSDI*, Cambridge, MA, April 2007.

[85] K. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim. A survey and comparison of peer-to-peer overlay network schemes. In *Communications Surveys & Tutorials, IEEE*, 2005.

[86] K. P. Gummadi, R. J. Dunn, S. Saroiu, S. D. Gribble, H. M. Levy, and J. Zahorjan. Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In *In Proc. ACM SOSP, Oct.*, 2003.

[87] F. Zhao, T. Kalker, M. Medard, and K. Han. Signatures for content distribution with network coding. In *in Proc. of IEEE ISIT07, (Nice, France)*, July 2007.

[88] S. Shin, J. Jung, and H. Balakrishnan. Malware prevalence in the kazaa file-sharing network. In *In ACM SIGCOMM Internet Measurement Conference (IMC)*, 2006.

[89] ironport software. `http://www.ironport.com/pdf/ironport_dlp_booklet.pdf`, 2012.

[90] Mcafee host data loss prevention. `http://www.mcafee.com/us/local_content/datasheets/ds_host_dlp.pdf`, 2011.

[91] Cisco services. `http://www.cisco.com/en/US/services/ps2961/ps2952/lw_cisco_dlp_ds.pdf`, 2011.

[92] Antivirus, anti-spam and internet security software - trend micro. `http://us.trendmicro.com/us/home`.

[93] ipoque : Bandwidth management with deep packet inspection. `http://www.ipoque.com/`.

[94] J. Jiang, Y. Tang, B. Liu, Y. Xu, and X. Wang. Skip finite automaton: A content scanning engine to secure enterprise networks. In *GLOBECOM 2010, 2010 IEEE Global Telecommunications Conference*, dec. 2010.

[95] The pirate bay : The world's most resilient bittorrent site. `http://thepiratebay.org/top/`, 2011.