# Hardware Acceleration of Network Intrusion Detection and Prevention

by

Brendan Cronin, B.E., M.Eng.

A thesis submitted in partial fulfilment of the requirements

for the Degree of Doctor of Philosophy

(Electronic Engineering)

Supervised by Dr. Xiaojun Wang

Dublin City University

School of Electronic Engineering

January 2014

# Declaration

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, 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:

Date:

# Acknowledgements

I would like to thank my supervisor Dr. Xiaojun Wang for his excellent guidance, advice and encouragement throughout my time in Dublin City University. I am also grateful to all my colleagues in the Network Processing Group for their support and help, in particular Yachao and Xiaofei. Thanks also to Dr. Olga Ormond, Research Officer with the Network Innovations Centre, for her assistance with my work.

I delivered the taught postgraduate module *HDL and High Level Logic Synthesis* in 2010 and 2011 under the supervision of the module coordinator, Dr. Wang. I would like to thank Dr. Wang, and Dr. Noel Murphy, head of the School of Electronic Engineering at the time, for giving me this opportunity. Teaching this module gave me a more in depth knowledge of VHDL, Verilog and FPGA design which helped me greatly in the development of prototype designs for evaluation of my research.

I would like to dedicate this thesis to my late parents, Teresa and Patrick, who actively encouraged my learning and education from an early age. I must include special mention for my godfather and uncle, Denis, who has always been a great support throughout my time in school and university.

Finally, these last four years of study would have been very difficult without the love, patience and support of Ilaria.

# Table of Contents

# List of Figures

# List of Tables

# List of Acronyms and Abbreviations

| | | |
|---|---|---|
| AMBA | – | Advanced Microcontroller Bus Architecture |
| API | – | Application Programming Interface |
| ART | – | Allotment Routing Table |
| ASIC | – | Application Specific Integrated Circuit |
| AXI | – | Advanced eXtensible Interface |
| BDD | – | Binary Decision Diagram |
| BP | – | Bit Parallel |
| BRAM | – | Block RAM |
| BSD | – | Berkeley Software Distribution |
| BV | – | Bit Vector |
| CAM | – | Content Addressable Memory |
| CCP | – | Champarnaud - Coulon – Paranthoën |
| CCR | – | Character class with Constraint Repetition |
| $CD^2FA$ | – | Content Addressed Delayed Input DFA |
| CES | – | CCR regExp Scanner |
| CPU | – | Central Processing Unit |
| CX-NFA | – | CAM-based eXtended NFA |
| DCE/RPC | – | Distributed Computing Environment / Remote Procedure Calls |
| DDR | – | Double Data Rate |
| DF | – | Don't Fragment |
| DFA | – | Deterministic Finite Automaton |
| DMA | – | Direct Memory Access |
| DMZ | – | DeMilitarised Zone |

| | | |
|---|---|---|
| DoS | – | Denial of Service |
| DPI | – | Deep Packet Inspection |
| DPICO | – | DPI COmpact |
| DRAM | – | Dynamic Random Access Memory |
| $D^2FA$ | – | Delayed Input DFA |
| EGT-PC | – | Extended Grid-of-Tries with Path Compression |
| ET | – | Emerging Threats |
| FA | – | Finite Automaton |
| FIFO | – | First In First Out |
| FPGA | – | Field Programmable Gate Array |
| FSBV | – | Field-Split parallel Bit Vector |
| FSM | – | Finite State Machine |
| Gb/s | – | Gigabits per second |
| GB/s | – | Gigabytes per second |
| GPU | – | Graphics Processing Unit |
| GW | – | Gigawatt |
| G-NFA | – | Glushkov NFA |
| HIDS | – | Host-based Intrusion Detection System |
| HIPS | – | Host-based Intrusion Prevention System |
| HTTP | – | HyperText Transfer Protocol |
| H-FA | – | History-based FA |
| ICMP | – | Internet Control Message Protocol |
| IDS | – | Intrusion Detection System |
| IDPS | – | Intrusion Detection and Prevention System |
| IET | – | Institution of Engineering and Technology |

| | | |
|---|---|---|
| ioctl | – | input/output control |
| IP | – | Internet Protocol, or, Intellectual Property |
| IPS | – | Intrusion Prevention System |
| ISN | – | Initial Sequence Number |
| ITU | – | International Telecommunication Union |
| IXP | – | Internet eXchange Processor |
| I/O | – | Input/Output |
| KMP | – | Knuth-Morris-Pratt (algorithm) |
| LAN | – | Local Area Network |
| LSB | – | Least Significant Bit |
| MAC | – | Media Access Control |
| Mb/s | – | Megabits per second |
| MF | – | More Fragments |
| MSB | – | Most Significant Bit |
| MSS | – | Maximum Segment Size |
| MTU | – | Maximum Transmission Unit |
| NAT | – | Network Address Translation |
| NBA | – | Network Behavioural Analysis |
| NGFW | – | Next Generation FireWall |
| NGIPS | – | Next Generation Intrusion Prevention System |
| NIC | – | Network Interface Card |
| NIDS | – | Network Intrusion Detection System |
| NIPS | – | Network Intrusion Prevention System |
| NFA | – | Nondeterministic Finite Automaton |
| NFP | – | Network Flow Processor |

| | | |
|---|---|---|
| NPU | – | Network Processing Unit (Network Processor) |
| OBDD | – | Ordered Binary Decision Diagrams |
| OISF | – | Open Information Security Foundation |
| OS | – | Operating System |
| PAF | – | Protocol Aware Flushing |
| PAWS | – | Protection Against Wrapped Sequence numbers |
| PCI | – | Peripheral Component Interconnect |
| PCRE | – | Perl Compatible Regular Expression |
| PDN | – | Packet Data Network |
| PDU | – | Protocol Data Unit |
| PE | – | Processing Engine |
| PHP | – | PHP: Hypertext Preprocessor |
| PL | – | Programmable Logic |
| PLPMTUD | – | Packetization Layer Path MTU Discovery |
| PMTUD | – | Path MTU Discovery |
| P2P | – | Peer to Peer |
| RAM | – | Random Access Memory |
| RE | – | Regular Expression |
| regex | – | Regular Expression |
| RFC | – | Request For Comments |
| RX | – | Receive |
| SDRAM | – | Synchronous Dynamic Random-Access Memory |
| SMB | – | Server Message Block |
| SPAN | – | Switched Port ANalyzer |
| SRAM | – | Static Random Access Memory |

| | | |
|---|---|---|
| TCAM | – | Ternary CAM |
| TCP | – | Transmission Control Protocol |
| TCP/IP | – | Transmission Control Protocol/Internet Protocol |
| TFO | – | TCP Fast Open |
| TOE | – | TCP/IP Offload Engine |
| TW | – | Terawatt |
| TX | – | Transmit |
| UDP | – | User Datagram Protocol |
| VHDL | – | VHSIC Hardware Description Language |
| VHSIC | – | Very High Speed Integrated Circuit |
| VLAN | – | Virtual Local Area Network |
| VRT | – | Vulnerability Research Team$^{TM}$ (Sourefire) |
| XFA | – | Extended FA |
| XST | – | Xilinx Synthesis Technology |

# Abstract

Network Intrusion Detection and Prevention Systems (NIDPS) are important elements of network security. Their role is to monitor internet traffic for malicious content and, on detection, generate an alert message and/or block the offending traffic. Potential attacks are described in a database of rules known as the rule set, where each rule consists of an IP header part and a payload signature part. The payload signature can be in the form of a fixed string and/or regular expression. This thesis studies the three main stages of these systems, namely TCP/IP reassembly, multi-match header matching and Deep Packet Inspection (DPI).

TCP/IP reassembly is a necessary prerequisite to DPI as attack patterns may span more than one IP fragment or TCP segment. Either target-based reassembly or traffic normalisation is required in order to overcome insertion/evasion attacks. This thesis builds upon existing research by outlining an FPGA-based architecture that handles the common case of reassembling in-sequence data streams in hardware and the much rarer out-of-sequence data streams in software.

Multi-match header matching involves the matching of each packet header against the header section of all rules. This differs from the single-match classification used in routers where there is a single highest priority match per packet. The strategy adopted in this thesis was to adapt a number of single match algorithms to perform multi-matching and to compare their performance with existing solutions. Existing solutions typically involve the use of Ternary Content Addressable Memory (TCAM) and therefore suffer from disadvantages such as high cost, high energy consumption, and low storage efficiency. Algorithmic solutions, which use SRAM instead of TCAM, can therefore have an advantage. The adapted algorithms were implemented in C code and evaluated in terms of speed and energy efficiency on an ARM processor.

DPI is particularly challenging due to the number and complexity of regular expressions. This thesis builds on existing research into Bit-Parallel hardware architectures. The main contribution is an extension for the efficient handling of the constrained {min,max} repetition syntax, including a solution to the issue of counter overlap. This allows for the handling of many additional regular expressions that would otherwise be unsuitable. The design was implemented in VHDL and evaluated using the Xilinx tool set. A comprehensive review of the most significant research works in the DPI field is also provided.

# List of Publications

**Journal Papers**

Cronin, B. and Wang, X. Hardware Acceleration of Regular Expression Repetitions in Deep Packet Inspection, *IET Information Security*, Vol.7, No.4, 2013, pp.327-335, doi:10.1049/iet-ifs.2012.0340.

Cronin, B. and Wang, X. Pattern Overlap in Bit-Parallel Implementation of Regular Expression Repetition Quantifiers, *Inderscience International Journal of Security and Networks*, Vol. 8, No. 4, 2013, pp.231-238, doi:10.1504/IJSN.2013.058154.

**Conference Paper**

Cronin, B. and Wang, X. Algorithmic Multi-match Packet Classification in Network Intrusion Detection Systems, *Proceedings of 2010 China-Ireland International Conference on Information and Communications Technologies (CIICT2010)*, Wuhan, China, 10-11 Oct., 2010. pp.150-156.

# Chapter 1 - Introduction

## 1.1. Background

### 1.1.1. Growth of the Internet

Recent years have witnessed rapid growth in both internet penetration and bandwidth due to huge improvements in telecommunication infrastructure, the proliferation of competitively priced computers and internet-capable mobile devices, and the reduced cost of internet access resulting from increased competition. The number of individuals using the internet has increased from 1 billion in 2005 to over 2.7 billion in 2013 (ITU, 2013). Cisco (2013) estimates that global internet traffic has increased from 2,000 GB/s in 2007 to 12,000 GB/s in 2012 and forecasts that this will increase to 35,000 GB/s by 2017, equivalent to 1 zettabyte per year, mainly driven by increased video traffic. Business IP traffic is expected to triple between 2012 and 2017, mainly due to the increased use of high quality video communications.

### 1.1.2. Energy Consumption

The increase in energy consumption associated with expanding internet use has become a concern because of the associated economic and environmental costs. Raghavan and Ma (2011) estimated that the power consumed globally by the internet in 2011 was between 170 and 307 GW, in other words between 1.1 and 1.9% of the total 16TW used by the world population. Although this may seem like a small fraction, it is equivalent to the power output of over 350 typical nuclear reactors. Raghavan and Ma argue that we should apply the internet to reducing other forms of energy consumption (e.g. video conferencing versus travel) in addition to making the internet itself more efficient. According to Lanzisera et al. (2012), network equipment consumed about 1% of buildings electricity in the USA in 2008 and was increasing at a rate of approximately 6% per annum, with most of this consumption occurring in offices and residences rather than data centres. They found that office building networking equipment is one of the largest energy consumers, accounting for 40% of the total in 2008.

### 1.1.3. Internet Security

Approximately 7.6 million new unique pieces of malware were detected by the AV-Test Institute (2013) for the month of June 2013. In other words, a new malware was created every 0.35 seconds. The ever increasing penetration and speed of the internet means that these viruses can spread even faster. The MyDoom worm was one of the fastest spreading email worms ever. Within a few hours of its first appearance in January 2004, it had slowed the internet by 10% and average web page load times by 50% (Jones 2006). The worm spread as an email attachment, and spammed itself to addresses listed in computer's address books when the attachment was clicked on. It's estimated that 10% of email messages, sent in the hours immediately after its first appearance, contained the worm. Consultancy firm mi2g (2004) estimated the economic losses caused by MyDoom at $38.5 billion, although this figure has been disputed by others. Another famous worm, Sasser, appeared in April 2004. Unlike MyDoom, it was not transmitted via email. It instead exploited a buffer-overrun flaw in unpatched versions of Microsoft Windows 2000 and XP which allowed it to take control of the infected computer (Vamosi 2004). It then scanned local networks and the internet for other computers to infect. It caused French satellite communications to be shut down, the cancellation of several Delta flights and the shutdown of many computer systems worldwide. The economic damage is estimated at between $14.8 and $18.1 billion (ThinkQuest 2004). The virus was created by a German student who released it on his 18th birthday.

Mobile internet traffic is currently growing rapidly due to the recent surge in smartphone take-up and the rollout of 4G networks. Smartphones are particularly attractive to cyber criminals as owners regularly use them for personal tasks such as online purchases, email and social media – all involving the use of sensitive personal information such as usernames, passwords and credit card details (Ruggiero and Foote 2011). They also pose an easier target than PCs as many users do not recognise the need to install or enable security software on their smartphones. Many naively believe that surfing the internet on their phone is safer than on their PC. NQ Mobile (2013) found that mobile malware attacks increased 163% in 2012, with 95% of all attacks targeting the Android OS.

The conventional way of defending against malware attacks is to use end-host based solutions such as patches to vulnerable operating systems and applications, anti-virus software and firewalls. The main issue with these approaches is that there is a time lag between the appearance of a virus, the availability of a software patch and virus database update, and finally the actual update of the end-hosts. Given the speed with which some viruses can spread, this time lag can be more than sufficient for many systems to be infected. Moreover, the repeated updating of end-host software is an added maintenance cost for businesses, which also disrupts the normal work of computer users.

## 1.2. Motivation

Given the issues with end-host security software, a more attractive approach is to block the malware in the network before it arrives at the end-hosts. This is known as intrusion prevention. In the case of office networks, this is typically performed at the edge of the network, just inside the firewall. It can also be performed internally to protect a particularly important segment of the network. In the case of the mobile internet, next generation security gateways would block attacks at the Gi/SGi interface between the 3G/4G network and the external PDN.

In addition to matching against the TCP/IP header, this type of Network Intrusion Prevention System (NIPS) needs Deep Packet Inspection (DPI) in order to analyse packet payloads for the presence of malicious content. Existing hardware systems commonly use energy inefficient TCAM to perform pattern matching. The ever increasing number and complexity of attack signatures and traffic speeds will lead to such systems becoming a significant consumer of power in the enterprise network. Due to customer demand, there is a growing requirement to design more efficient systems that miminise the use of energy inefficient technologies such as TCAM. The challenge is therefore to find hardware solutions which can accelerate, in an energy efficient manner, the analysis of network traffic for particular header values and the presence of complex attack signatures.

## 1.3. Intrusion Detection and Prevention

The NIPS is one member of a larger family of what are known as Intrusion Detection and Prevention Systems (IDPS).

### 1.3.1. Classification

IDPS can be classified in the following categories:

- Host-based – this system is a software agent installed on an individual computer. In addition to monitoring all incoming and outgoing traffic for attacks such as a virus, a worm or hacking activity, it also monitors applications running on the computer for suspicious behaviour. Although host-based agents provide additional security features compared to network-based systems, they can be more difficult to administer because of their distributed nature

- Network-based – this system is a standalone system which monitors all traffic into and out of a network. It can also be used to monitor internal network traffic. It can be either a dedicated hardware system from a networking equipment vendor or a software program running on an off-the-shelf server. One of the most well-known network-based software solutions is the open-source Snort (Roesch et al. 2012).
- Wireless – this system monitors wireless network traffic and the associated wireless networking protocols for suspicious activity.

### 1.3.2. Detection Methods

IDPS use one or a combination of the following techniques to detect attacks:

- Signature-based – attacks are described in a large database of attack signatures known as the rule set
- Anomaly-based – attacks are detected by comparing the current activity with pre-defined "normal" activity. Such systems have the advantage that they can detect attacks hidden within encrypted traffic, but often suffer from a high number of false positives, i.e. incorrectly generating an attack alert notification. Note that a network-based system that uses anomaly-based detection is also known as a Network Behavioural Analysis (NBA) System
- Stateful protocol analysis – the state of network, transport and application protocols are tracked and the activity compared with correct protocol behaviour in order to detect attacks. Some signature-based systems provide the ability to specify stateful signature-based rules, e.g. *flowbits* keyword in Snort allows a number of rules to be linked together in order to track state across multiple datagrams in a single transport layer session; *flow:established* keywords restrict application of the rule to established sessions only.

### 1.3.3. Modes

IDPS can be split into two types based on their mode of operation:

- Passive – An Intrusion Detection System (IDS) is passive in that it only monitors traffic for attacks and generates an alert and logs an event on detection

- Reactive – An Intrusion Prevention System (IPS) is reactive in that it can be configured to perform an action on detection of a particular attack.

  In the case of an NIPS, such an action could be to block the connection carrying the malicious traffic. Snort can function as an NIPS by running it in *inline* mode. Although an NIPS is a very powerful solution, it suffers from a couple of issues. Firstly, false positives can result in valid, and perhaps critical, connections being dropped. Secondly, processing overload or DoS attacks can result in valid traffic being dropped or attacks left through.

  The action perform by a Host-based IPS (HIPS) depends on the exact detection technique used – e.g. it could prevent code being executed, block a network connection, stop inappropriate file access.

### 1.3.4. NIDS Sensor Location

The most common location for an NIDS system is inside an enterprise's firewall so as to reduce its incoming traffic workload and exposure to DoS attacks. The firewall is the first line of defence which is configured to block all incoming connections on ports which have not been opened. The NIDS will monitor traffic passed by the firewall for attack patterns, e.g. a virus inside HTTP connection traffic.

On detection of potential malicious traffic by the NIDS, the event is typically logged on the management server and an alert sent to the console.

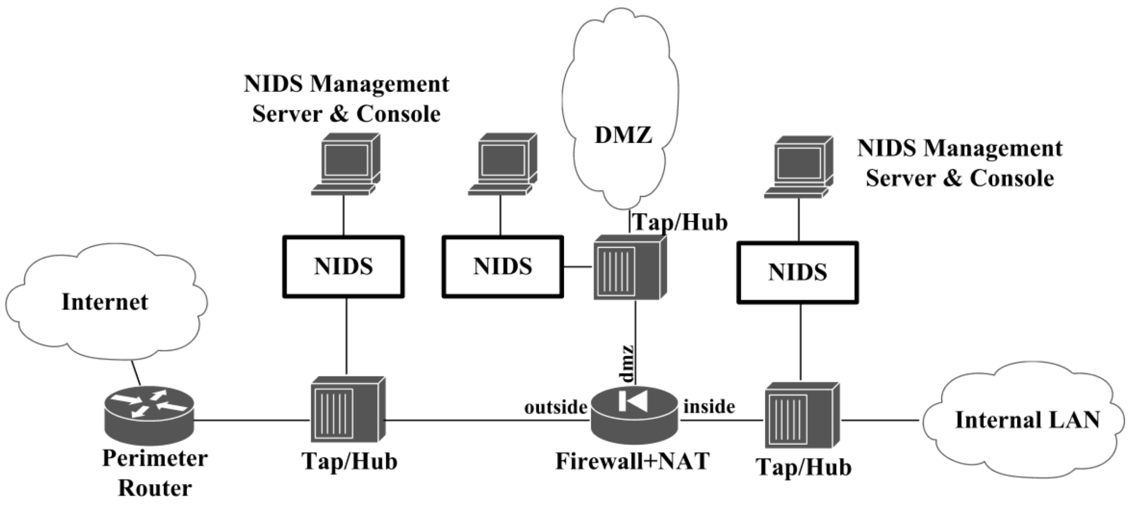


**Figure 1. Possible locations for NIDS in enterprise network**

Figure 1 shows an example enterprise network with NIDS systems placed in a number of locations:

- NIDS outside the firewall in order to detect attacks against the firewall
- NIDS in DMZ (demilitarised zone) to detect attacks against web/mail servers, etc. Each server should also run a HIDS agent for increased security
- NIDS in the internal network to detect internal attacks and external attacks that firewall left through.

An NIDS is a passive system that sniffs packets from the network. It can be connected to the network using a hub, ethernet tap or via the SPAN port of a switch. In the case of a switch, it may be possible to mirror a number of ports to the SPAN port using a VLAN. The disadvantages of the SPAN port are that the total VLAN traffic may exceed the bandwidth of the port and, the performance of the switch may be degraded. Bandwidth is typically more of an issue when the NIDS is used to monitor internal network traffic since the traffic throughput is likely to be much higher than that found at the gateway to the external internet. Finally, some networking equipment vendors have switches and firewalls with built-in NIDS functionality.

### 1.3.5. NIPS Sensor Location

An NIPS system is an inline sensor which the monitored traffic must flow through. It is typically deployed on secure side of the firewall in order to reduce its workload. As time goes on, the line between firewall and IPS is becoming blurred as more and more firewall vendors provide IPS functionality as part of next generation firewall systems. The NIPS can be configured to carry out various actions on detection of a particular attack or undesirable traffic, e.g.:

- drop packets containing an attack pattern
- block the corresponding transport layer connection. This could be done inline or by automatically reconfiguring the firewall
- reset the transport layer connection
- reconfigure router to redirect offending connection traffic to a honey pot

- throttle the bandwidth used by undesirable traffic (e.g. P2P file sharing, suspected DoS attack, etc.)
- run a script written by the NIPS administrator – script gives a lot of flexibility to automatically reconfigure third-party networking equipment.

Figure 2 shows an example enterprise network with two NIPS systems. One is positioned just inside the firewall to detect any attacks that manage to get through it. The second is used to protect a particularly important segment of the network against internal intrusions, e.g. finance department, labelled segment 1.

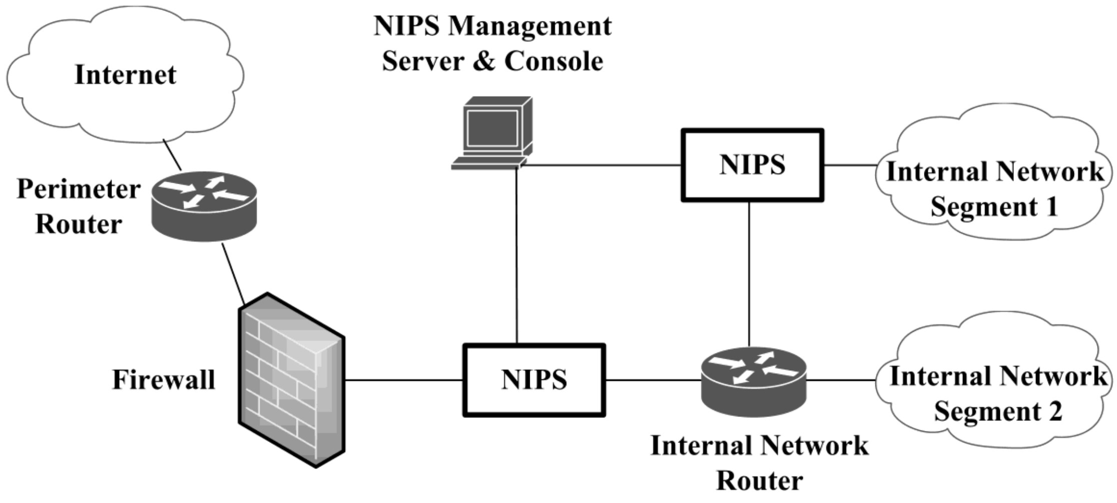

Internet
NIPS Management Server & Console
Perimeter Router
NIPS
Internal Network Segment 1
Firewall
NIPS
Internal Network Segment 2
Internal Network Router

**Figure 2. Example of NIPS placement in enterprise network**

## 1.4. Research Goals

This thesis focuses on the three primary parts of an NIDPS system, namely TCP/IP reassembly, multi-match header classification and regular expression (regex) DPI, as highlighted in grey in Figure 3. Depending on the requirements of the DPI stage implementation, the multi-match header classification stage may run either in series or in parallel with the DPI stage. When placed in front of the DPI, the header classification stage acts as a pre-filter which reduces the number of rules that need to be processed at the DPI stage for a particular connection flow. However, some DPI algorithms cannot take advantage of this as they always examine every rule for every packet, and, in this case, it makes more sense to run the header classification in parallel. In the parallel architecture, a negative header match will result in a fast overall negative match decision which will cut short the processing in the DPI block for that particular connection flow.

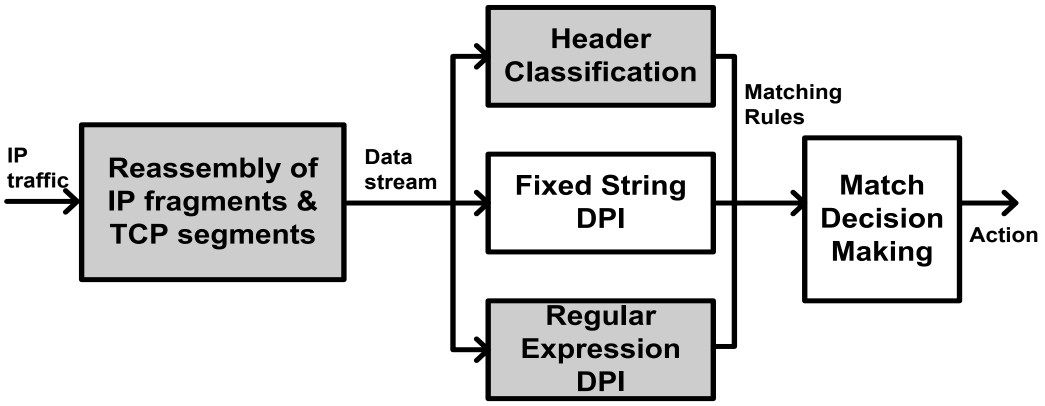


**Figure 3. Example NIDPS Architecture**

The overall goal of this thesis is to propose new, or extend existing, algorithms and architectures that lead to systems that can handle higher traffic throughputs, greater numbers and complexity of attack signatures, while keeping power consumption to a minimum. The specific goals in each area are as follows:

- *Improve on existing hardware acceleration techniques for the acceleration of TCP/IP reassembly in the context of DPI*:

  Existing hardware-based designs typically drop out-of-order TCP segments in order to force the originating host to resend. Dropping packets in this way is not ideal as network performance is adversely affected. This leads to the thesis goal of outlining a hardware-based architecture that avoids unnecessary packet dropping.
- *Develop and evaluate algorithmic solutions to the problem of multi-match header matching*:

  Hardware-based NIDPS typically use TCAM-based technology to perform TCP/IP header matching. The strategy adopted in this thesis is to adapt a number of single match algorithms to perform multi-matching and to compare their performance with existing solutions. Such algorithms can use SRAM instead of TCAM and should therefore be less expensive and more energy efficient.
- *Survey existing research work on DPI, with a particular focus on regular expression matching*:

  A review of the most significant research in the area of DPI will be of use to other researchers looking to improve the state of the art.
- *Extend Bit-Parallel (BP) hardware architecture from existing research to include improved handling of constrained {min,max} repetition syntax*:

Constrained repetition syntax is commonly used in DPI regular expressions. Existing BP architectures handle such repetitions by unrolling of the repetition with the result that the regex is often unsuitable for processing because of its excessive length. The goal is to modify the BP architecture based on the Glushkov NFA so that it can handle these repetitions without unrolling, thereby greatly increasing the number of DPI signatures that can be handled.

## 1.5. Contributions

The main contributions of this thesis are summarised as follows:

**TCP Segment Reassembly**

The importance of IP fragment and TCP segment reassembly in DPI systems is examined and the reassembly functionality of open source software NIDPS is analysed. Existing research solutions to hardware acceleration of TCP/IP reassembly do not fully handle all cases of out-of-sequence packets. This thesis outlines an FPGA-based architecture that handles the common case of reassembling in-sequence data streams in hardware and the much rarer out-of-sequence data streams in software.

**Multi-match Packet Classification**

A number of algorithmic approaches to multi-match classification which use SRAM instead of TCAM are evaluated and compared in terms of throughput performance and energy efficiency. These algorithms are mainly for single-match classification and so have to be adapted for multi-match. The adapted algorithms were implemented in C code and evaluated on an ARM simulation platform. The EGT-PC and ART algorithms were found to be a suitable alternative to TCAM. Although these algorithms do not currently match the performance of existing bit vector based algorithms, such as FSBV and StrideBV, due to the commonality of field values in recent rule sets, this may change in the future.

**Deep Packet Inspection**

While extensive research has been conducted into algorithms for performing fixed string and general regex matching, the majority has ignored some of the more complicated regex syntax such as constrained repetition quantifiers and back references. This thesis describes a hardware architecture for handling regexes containing constrained repetitions. The issue of pattern overlap affecting the handling of these repetitions is then examined and, a First-In-First-Out (FIFO) queue based solution is described for susceptible regexes. The algorithms were implemented in VHDL and evaluated using the Xilinx tool set and the open-source NetFPGA (Naous et al., 2008) research platform as the target.

The impact of this work is that the handling of many regexes that would otherwise be unfeasible due to their unrolled length can now be efficiently and correctly handled by the BP architecture based on the Glushkov NFA. This enables the hardware acceleration of over half the regexes found in recent Snort rule sets. The remainder could be handled by extracting suitable sub-expressions and using the BP system as an imprecise pre-filter followed by full verification of any positive matches in software.

The design evaluated in this thesis matches against all regexes in parallel. An alternative approach would be to use multi-match header and fixed string matching as pre-filters so as to greatly reduce the number of regexes to match against per packet. The counting block algorithm outlined in this thesis could equally be used in such a design. Such an approach would allow for regex data to be stored in external SRAM, thereby allowing for the storage of a much larger number of regexes. Such a design would give high throughput through the use of pipelining and parallel processing of packets.

Hardware acceleration of regex matching for DPI is a very challenging task. It is hoped that the contributions of this thesis will be useful to other researchers looking to further advance the state of the art.

## 1.6. Thesis Organisation

The remainder of this thesis is structured as follows:

- *Chapter 2: Background*

  This gives background information useful for a better understanding of the thesis. Operation of the open-source NIDPS, Snort, is looked at and the rule syntax examined. Some of the mathematical concepts related to the finite automaton representation of regexes are described.

- *Chapter 3: TCP/IP Reassembly*

  Most research articles on signature-based NIDPS do not mention TCP/IP reassembly. This chapter looks at what is an essential element of any NIDPS as attack patterns may be split over multiple IP fragments or TCP segments. Moreover, target OS–based reassembly is necessary in order to avoid attack evasion. An FPGA-based design is outlined for the acceleration of this reassembly.

- *Chapter 4: Multi-match Header Classification*

  This chapter describes the adaption and evaluation of a number of single-match packet classification algorithms for multi-match classification. Multi-match header classification is needed in NIDPS because a number of rules may match the header of the incoming IP packet.

- *Chapter 5: Pattern Matching Methods*

  This chapter looks at the general theory of both fixed string and regex matching in DPI and related research.

- *Chapter 6: Constrained Repetition Handling Algorithm*

  A counter-based algorithm and a corresponding Bit-Parallel (BP) hardware architecture are presented for the more efficient processing of regexes which include constrained repetitions.

- *Chapter 7: Dealing with Pattern Overlap in the case of Constrained Repetitions*

  This chapter describes how certain regexes which include constrained repetitions are not suitable for the counter-based algorithm as they are susceptible to a pattern overlap issue. A FIFO-based mechanism to deal with the issue is outlined and evaluated.

- *Chapter 8: Conclusion and Future Work*

  A summary is presented of the results achieved in the preceding chapters and possible directions for future research are discussed.

# Chapter 2 - Background

This chapter provides a brief introduction to those aspects of automata theory that are helpful for a better understanding of the thesis. An overview is also provided of some of the main open source and commercial NIDPS, including a discussion of the recent trend for IPS functionality to be included in next generation firewall products. The choice of platform for an NIDPS product has a significant impact on achievable performance and price. All the various commodity and custom hardware platforms, suitable for an NIDPS implementation, are examined.

## 2.1. Automata Theory

### 2.1.1. Formal Languages

**Alphabet**

An *alphabet*, denoted by the symbol Σ, is a finite, nonempty ordered set of symbols. e.g.:

- Σ = {a,b,...z} is the set of all lower-case letters
- Σ = {`00,01,02,...FF`} is the set of 256 symbols that can be represented by 8-bit values (using hexadecimal representation).

**Strings**

A *string* is a finite sequence of symbols from a particular alphabet. e.g. bxsf is a string from the alphabet Σ = {a,b,...z}. An *empty string*, denoted by ε, has zero occurrences of symbols from Σ.

Exponential notation is used to express the set of all strings of a particular length from an alphabet, e.g.

If $\Sigma = \{0,1\}$, then $\Sigma^2 = \{00, 01, 10, 11\}$.

$\Sigma^0 = \{ \varepsilon \}$, regardless of the alphabet.

$\Sigma^* = \{ \varepsilon , 0, 1, 00, 01, 10, 11, 000, 001, ....\}$, the set of all strings over Σ.

**Languages**

If $\Sigma$ is an alphabet, and $L \subseteq \Sigma^*$ , then $L$ is a language over $\Sigma$. In other words, $L$ is a set of strings chosen from $\Sigma^*$. Formal languages are treated in the same way as mathematical sets and so set theory operations such as union and intersection can be applied. It can be defined using an automaton or formal grammar system.

Formal languages are often used to define computer programming languages.

**Grammars**

A formal grammar consists of

- a finite set of non-terminal variable symbols that can be rewritten as a sequence of symbols
- a finite set of terminal symbols, $\Sigma$, the alphabet of the language, that cannot be rewritten – hence "terminal"
- a finite set of rewrite/derivation rules $X \rightarrow Y$, (i.e. X directly derives Y), where X and Y consist of non-terminals and/or terminals
- a start variable, S, which is an element of the set of non-terminals

Chomsky (1956) categorised formal grammars into four classes, as shown in Table 1, by restricting the forms of X and Y.

**Table 1. Chomsky Hierarchy**

| | *Language* | *Grammar* | *Automaton* | *Rewrite rule restriction*[*] |
|---|---|---|---|---|
| 3 | Regular | Regular (Right-linear or Left-linear)[†] | NFA or DFA | $A \to a$ and $A \to aB$ (or $A \to Ba$)[†] and $A \to \varepsilon$ |
| 2 | Context-free | Context-free | Push-Down Automaton | $A \to \alpha$ |
| 1 | Context-sensitive | Context-sensitive | Linear-Bounded Automaton | $\alpha A \beta \to \alpha \mu \beta$ |
| 0 | Unrestricted/Free | Recursively enumerable | Turing Machine | $\alpha \to \beta$ |

* *A* and *B* represent single non-terminal variables, *a* represents a single terminal symbol and, greek letters represent strings of terminals and non-terminals. *α* and *β* can be empty.

† See section 2.1.2 for explanation of left-linear and right-linear grammars

## 2.1.2. Regular Languages

### Regular Grammars

Strictly regular grammars generate regular languages and can be represented by finite state automata. The rewrite rules are restricted to having a left-hand side consisting of a single non-terminal and a right-hand side consisting of a single terminal possibly followed by a single non-terminal in the case of a right-linear grammar, or it can alternatively be preceded by a single non-terminal in the case of a left-linear grammar. Left and right-linear rules cannot be mixed in the same regular grammar. The rule $S \to \varepsilon$ is allowed, provided the start variable, *S*, does not appear on the right-hand side of any rule. Left and right-linear grammars are discussed further at the end of this section.

An Extended Regular grammar is similar to a regular grammar except that in the rule $A \to aB$ (or $A \to Ba$), *a* can be a string of terminals. It can be shown that any extended regular grammar can be also expressed as an equivalent strictly regular grammar.

A regular grammar is said to be *non-deterministic* if it includes two rules $A \rightarrow aB$ and $A \rightarrow a$ or two rules $A \rightarrow aB$ and $A \rightarrow aC$. Otherwise it is said to be *deterministic*.

**Finite State Automata**

A finite automaton is a 5-tuple ( $Q, \Sigma, q_0, F, \delta$ ) where

- $Q$ is a finite set of states (circles in state diagram).
- $\Sigma$ is a finite set of symbols called the alphabet.
- $q_0 \in Q$ is the start state (state with incoming arrow not connected to any other state in the state diagram).
- $F \subseteq Q$ is the finite set of accept or final states (double circles in state diagram).
- $\delta$ is the transition relation, indicating where to go for a given state and input symbol. In the same way as for regular grammars, a finite state automaton is said to be non-deterministic if there exist states for which the same input symbol results in more than one transition. The transition function can therefore be defined:
  - for a Non-deterministic Finite Automaton (NFA)

    $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow P(Q)$

    where $P(Q)$ is the power set (set of all subsets) of $Q$.

    $\times$ denotes Cartesian product, the set of all ordered pairs from two sets.

    This is a multi-valued transition function, i.e. for a given state and input symbol, there can be more than one transition.
  - for a Deterministic Finite Automaton (DFA)

    $\delta : Q \times \Sigma \rightarrow Q$

    This is a single valued transition function.

It can be shown that the languages accepted by finite automata are regular languages. Therefore any language represented by a regular grammar can also be represented by an equivalent finite automaton.

Consider the following regular grammar in which {a,b} is the alphabet of the language and $q_0$ ,$q_1$ are non-terminal variables

$$q_0 \rightarrow aq_0 \qquad q_0 \rightarrow bq_1 \qquad q_1 \rightarrow a \qquad q_1 \rightarrow aq_1$$

Now if $q_2$ is added as a non-terminal variable corresponding to accept or final state of the equivalent NFA, the regular grammar can be rewritten as

$$q_0 \rightarrow aq_0 \qquad q_0 \rightarrow bq_1 \qquad q_1 \rightarrow aq_2 \qquad q_1 \rightarrow aq_1 \qquad q_2 \rightarrow \varepsilon$$

The state transition table and the state diagram of the equivalent NFA are shown in Table 2 and Figure 4, respectively. $q_0$ is the start state and $q_2$ is the accept state. Note that state $q_1$ has two outgoing edges with the same symbol, i.e. this is an NFA.

**Table 2. State Transition Table for example regular grammar**

| *Input Symbol* / *State* | *a* | *b* |
|---|---|---|
| $q_0$ | $\{q_0\}$ | $\{q_1\}$ |
| $q_1$ | $\{q_1, q_2\}$ | ∅ (null) |

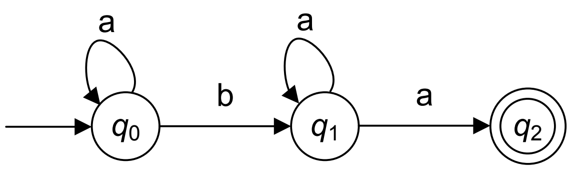


**Figure 4. NFA equivalent to example regular grammar**

**NFA vs. DFA**

In the case of DFA, for each state, $q$, and input symbol, $\alpha$, there is exactly one transition leaving state $q$. This includes transitions to the *null* state, which are typically omitted from state diagrams. Therefore a DFA has at most one edge leaving state $q$ labelled with the symbol $\alpha$.

In the case of an NFA, there may be multiple transitions for each combination of state and input symbol. It can also include transitions for the empty string, ε, i.e. it can transition from one state to another without consuming any input symbol.

It can be shown that any regular language $L$ is accepted by a DFA if and only if it is also accepted by an NFA. In other words, DFA and NFA are equivalent in what they express and it is always possible to convert between them. It should also be noted that a DFA is in fact a special case of an NFA.

**Regular Expressions**

A regex, *r*, is an algebraic formula which represents the language *L*(*r*) of the regex, i.e. a set of strings in $\Sigma^*$. The fundamental operators used in regexes are:

- Union/Alternation: If $r_1$ and $r_2$ are regexes then $r_1|r_2$ is also a regex.
- Concatenation: If $r_1$ and $r_2$ are regexes, then $r_1r_2$ is also a regex.
- Kleene Closure: If *r* is a regex, then *r** is also a regex.

Two regexes over the same alphabet are equivalent if, and only if, their respective languages are equal sets. It can be shown that every language defined by a finite state automaton can also be expressed as an equivalent regex, e.g. FA in Figure 4 can alternatively be expressed as the equivalent regex *a*ba*a* .

Some example regexes:

- *a|b** denotes { ε, *a*, *b*, *bb*, *bbb*, ....}.
- (*b*|*c*)* denotes all strings made up of only the symbols *b* and *c*, plus the empty string.
- (*ab*|*c*)*d* denotes { ε, *abd*, *cd* }.

Regex operator precedence, as outlined in Table 3, is relatively simple. The most important point is that concatenation has higher priority than alternation.

**Table 3. Regex operator precedence**

| *Precedence* | *Operator* | *Description* |
|---|---|---|
| Highest | () | Parentheses and other grouping operators |
| | *, +, ?, {min,max}, etc. | Repetition |
| | ^xyz | Concatenation |
| Lowest | \| | Alternation |

**Right versus Left Linear (Regular) Grammars**

A right-linear grammar generates the strings of the language (i.e. the words) from left to right, whereas a left-linear grammar generates the words from right to left. Any left-linear grammar can be converted to an equivalent right-linear grammar and vice versa.

Figure 5 and Figure 6 show the right-linear and left-grammars, respectively, corresponding to the regex *x*yz*,* along with their equivalent automata.

Right-linear grammar:

$S \rightarrow xS \mid yA$

$A \rightarrow zA \mid \varepsilon$

Regex: $x^*yz^*$

Finite automaton:

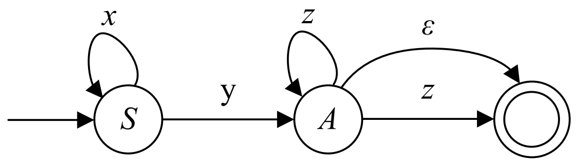


**Figure 5. Example of right-linear grammar and corresponding automaton**

Left-linear grammar:

$S \rightarrow Sz \mid Ay$

$A \rightarrow Ax \mid \varepsilon$

Regex: $x^*yz^*$

Finite automaton:

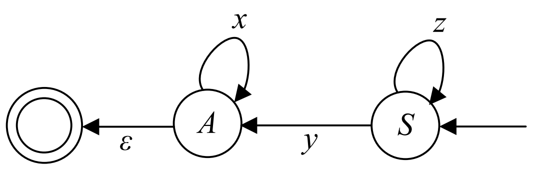


**Figure 6. Equivalent left-linear grammar and corresponding automaton**

### 2.1.3. Perl Compatible Regular Expressions

PCRE is a regex library written in C which implements pattern matching based on the syntax and semantics used in Perl 5.The library is used by a number of open source programs, including Apache HTTP server, PHP and Snort.

**Snort PCRE syntax**

The *pcre* keyword in Snort allows PCRE regexes to be written in the following format:

```
pcre:[!]"(/<regex>/|m<delim><regex><delim>)[ismxAEGRUBPHMCOIDKYS]"
;
```

- A regex is usually delimited using "/". However, it is possible to use almost any other special character, provided it is preceded by the letter "*m*" (meaning match).
- Prefixing the regex with an exclamation mark negates its meaning. e.g. the regex */foo/* matches any string that contains "*foo*", whereas the regex !*/foo/* matches any string that does not.
- The regex may be followed by a list of modifiers, `ismxAEGRUBPHMCOIDKYS`, some of which are Snort specific extensions. Three of the most common modifiers are:

- i – case insensitive matching
- s – single line mode, i.e. the dot wildcard metacharacter ,“.”, is to match everything including new line (otherwise new line is excluded)
- m – multi-line mode. This affects the operation of the start and end anchors, “^” and “*$*”, respectively. By default the input string is treated as a single line and the anchors apply to the start and end of the string. However, in multi-line mode, the “^” and “*$*” anchors additionally apply immediately after and before, respectively, any newline in the input stream.

- The PCRE syntax and semantics most commonly used in Snort is described in Table 4.

**Regex anchors**

An anchor is a type of zero-width assertion that specifies a position in the input string where a match must occur. Assertions do not actually consume any characters. The most important anchors used in Snort rules are described in Table 5.

For example, the multi-line start-anchored regex */^hello/m* would find a match in the strings “*helloworld*” and “*world\nhelloworld*”, but not in “*worldhelloworld*”. Similarly, the multi-line end-anchored regex */world$/m* would find a match in “*goodbyeworld*” and “*world\ngoodbye*”, but not in “*goodworldgoodbye*”.

**Table 4. Important PCRE Syntax and Semantics**

| *Operator Type* | *Example* | *Meaning* |
|---|---|---|
| Literals | a 4 % | Letters, digits, other characters |
| | \^ \? | Special characters must be preceded by `\` to cancel their special meaning |
| | \n \t \r | New line, tab, carriage return |
| | \xa3 | Hex code |
| Anchors and assertions | ^ | Regex must match at start of string, or after a new line in multi-line mode |
| | $ | Regex must match at end of string, or before a new line in multi-line mode |
| | \b | Word boundary – matches before and after an alphanumeric sequence (matched by `\w` character class) |
| Character Classes | [acEoi] | Any character in the list will match |
| | [^acEoi] | Any character apart from those in list will match |
| | [a-fA-F0-9] | Any hex character (dash indicates a range of characters) |
| | . | Dot means any character except new line.<br>If single line mode modifier is specified, then new line is also allowed. |
| | \s | Any space character `[ \t\r\n]` |
| | \w | Any word character `[A-Za-z0-9_]` |
| | \d | Any digit `[0-9]` |
| | \h | Any horizontal whitespace character `[ \t]` |
| | \S \W \D \H | Inverse of above four |
| Repetition (applied to preceeding regex element) | + | 1 or more |
| | * | 0 or more |
| | ? | 0 or 1 |
| | {10} | Exactly 10 |
| | {10,} | 10 or more |
| | {,10} | Up to 10 |
| | {5,10} | Between 5 and 10 |
| | Counting is "greedy" by default. i.e. System tries to find the longest match before backtracking if necessary. It can be made "lazy" by appending "*?*" after the count – i.e. system first tries to complete a match using the shortest number of repetitions, before then trying longer ones. | |
| Alternation | \| | either,or, e.g. a\|b means a or b |
| Grouping | () | Parentheses allow an operator to be applied to a part of a regex, rather than a single element.<br>This also creates a back-reference. Each group is numbered from left to right from 1. |
| Back-references | \n where n is a number | \2 is a back-reference to the 2nd matched group. Note that it signifies the matched fixed string that was matched and not the regex group. |
| Lookahead Assertion | (?=regex) | Zero-width positive lookahead. (Note: Lookahead assertions do not consume characters – i.e. matching position is not moved)<br>e.g. */foo(?=bar)/* will match *foo* if it is followed by *bar*. |
| | (?!regex) | Zero-width negative lookahead.<br>e.g. */foo(?!bar)/* will match if *foo* is found and is not followed by *bar*. |
| Lookbehind Assertion | (?<=regex) | Zero-width positive lookbehind.<br>e.g. */(?<=foo)bar/* will match *bar* if it is preceded by *foo*. |
| | (?<!regex) | Zero-width negative lookbehind.<br>e.g. */(?<!foo)bar/* will match if *bar* is found and it is not preceded by *foo*. |

**Table 5. Snort regex anchors**

| *Anchor* | *Multi-line mode* | *Description* |
|---|---|---|
| ^ | Disabled | The match must occur at the beginning of the string. |
| ^ | Enabled | The match must occur at the beginning of the string or line, i.e. at beginning or immediately following any newline character. |
| $ | Disabled | The match must occur at the end of the string. |
| $ | Enabled | The match must occur at the end of the string or line, i.e. at end or immediately before any newline character. |
| \b | - | The match must occur on a word boundary, i.e. between a word and a non-word character. Word characters consist of all alphanumeric characters and underscores. |

### 2.1.4. Construction of NFA from regex

Several algorithms have been proposed for the construction of a finite automaton from a regex. The algorithms differ in their level of complexity, in whether or not the result is deterministic, and in whether or not there are ε-transitions. The two best known classic methods are the Thompson (1968) construction algorithm and the Glushkov (1961) construction algorithm (equivalent to McNaughton-Yamada (1960) method). The Thompson method is simpler and produces an NFA with at most $2m$ states and at most $4m$ transitions, where $m$ is the number of characters (from alphabet) in the regex – i.e. linear relationship. It does, however, have ε-transitions. The Glushkov method produces an NFA with exactly $m$+1 states but up to O($m^2$) transitions. It has the advantage of not generating ε-transitions, but the construction takes longer compared to the Thompson method. It also has the important property that all transitions into a particular state are for the same character.

## Thompson NFA

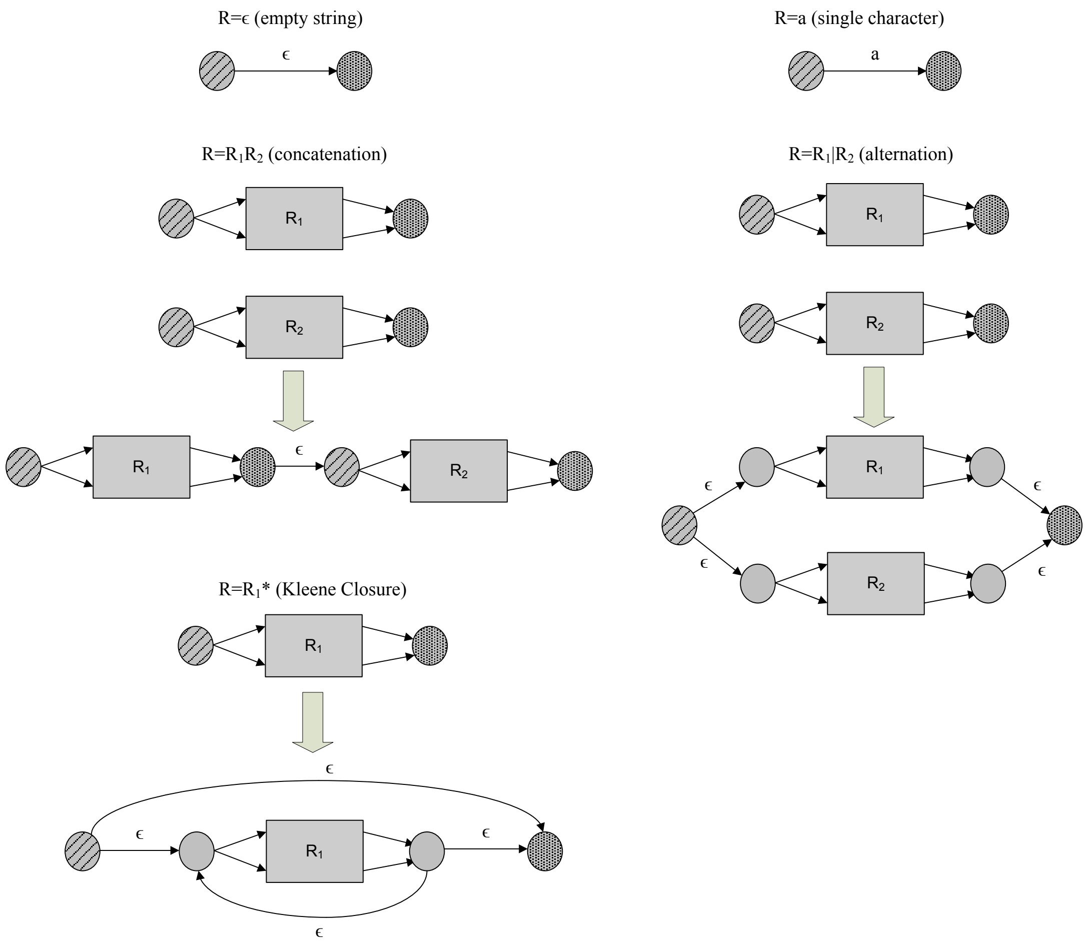


**Figure 7. Thompson Construction – glueing together automata**

The Thompson method first constructs a tree representation of the regex before computing, at each node of the tree, an automaton that recognises the language represented by the subtree at that node. ε-transitions are used to “glue” these automata together to eventually produce the overall NFA. Figure 7 illustrates how the sub-automata are glued together and Figure 8 shows the NFA constructed from regex *(a|b)*ca*.

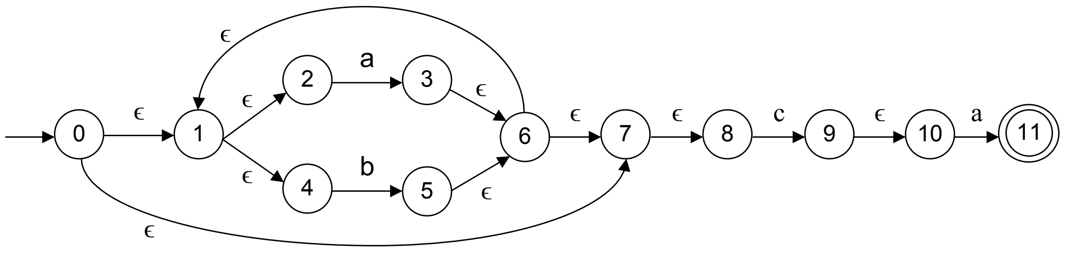


**Figure 8. Thompson NFA for regex (a|b)*ca**

**Glushkov NFA**

The Glushkov construction method was first explicitly described by Berry and Sethi (1986) and is explained in detail by Navarro and Raffinot (2002). Consider the example regex $RE=(a|b)^*ca$. The positions of the characters in the regex are marked with a number to give the marked expression $\overline{RE}=(a_1|b_2)^*c_3a_4$. Note that the bar over $RE$ signifies that it is the marked form of the regex.

Using the following notation:

- $L(\overline{RE})$ represents the language of $\overline{RE}$, i.e. all the strings accepted by $\overline{RE}$. In the case of the example, $L(\overline{RE}) = \{c_3a_4, a_1c_3a_4, b_2c_3a_4, a_1a_1c_3c_4, b_2b_2c_3a_4, a_1b_2c_3a_4, b_2a_1c_3a_4, \ldots\}$
- $\overline{\Sigma}$ represents the marked character alphabet
- $*$ is the regex operator meaning that the preceding symbol or sub-pattern is repeated zero or more times
- $\overline{\Sigma}^*$ represents all possible combinations of characters in the alphabet (can be null)
- $Pos(\overline{RE})=\{1 \ldots m\}$ represents the set of positions in $\overline{RE}$
- $\alpha_y$ is the indexed character at position $y$

the following definitions are made:

- $First(\overline{RE}) = \{x | x \in Pos(\overline{RE}), \exists u \in \overline{\Sigma}^*, \alpha_x u \in L(\overline{RE})\}$

  i.e. the set of initial positions of $L(\overline{RE})$.

  In the case of the example, $First(\overline{RE}) = \{1,2\}$
- $Last(\overline{RE}) = \{x | x \in Pos(\overline{RE}), \exists u \in \overline{\Sigma}^*, u\alpha_x \in L(\overline{RE})\}$

  i.e. the set of positions in $\overline{RE}$ with index $x$ whose corresponding character $\alpha_x$ forms a string from the language of $\overline{RE}$ when prefixed by some combination of characters from the alphabet. In other words this is the set of final, or accept, states of the automaton which, when reached, indicate a match has been found.

  In the case of the example, $Last(\overline{RE}) = \{4\}$
- $Follow(\overline{RE}, x) = \{y | y \in Pos(\overline{RE}), \exists u,v \in \overline{\Sigma}^*, u\alpha_x\alpha_y v \in L(\overline{RE})\}$

  i.e. for a given position $x$ in $\overline{RE}$, the set of positions in $\overline{RE}$ with index $y$ for which the combination of the two characters $\alpha_x$ followed by $\alpha_y$ form a substring of some string from the language of $\overline{RE}$. In other words, for each position *x*, *Follow*($\overline{RE}$,*x*) is the set of positions reachable from *x*.

In the case of the example

- *Follow*($\overline{RE}$,1)={1,2,3}
- *Follow*($\overline{RE}$,2)={1,2,3}
- *Follow*($\overline{RE}$,3)={4}

- $Empty_{RE}$ has value {ε} if ε belongs to *L*(RE) and ∅ otherwise.

There is a transition from state *x* to *y* in the automaton if *y* ∈ *Follow*($\overline{RE}$,*x*). The resulting marked Glushkov automaton is shown in Figure 9. The Glushkov automaton is then simply obtained by removing the position indices from the marked automaton.

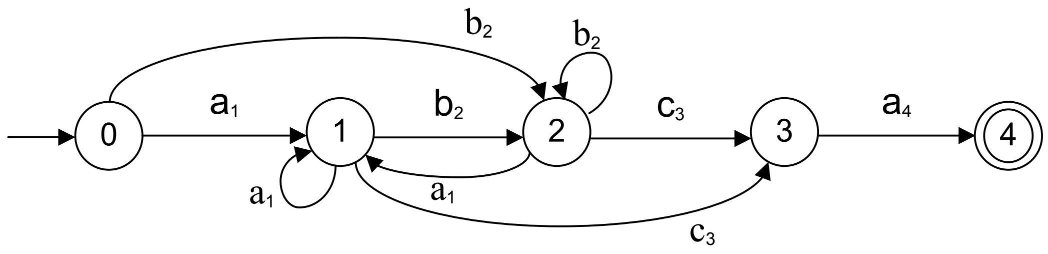


**Figure 9. Marked Glushkov NFA for** $\overline{RE}$**=($a_1$|$b_2$)*$c_3a_4$**

The Glushkov construction algorithm, in the same way as the Thompson algorithm, makes use of a tree representation of the regex, where each node *v* of the tree represents a sub-expression $RE_v$ of the overall regex *RE*. *First(v)*, *Last(v)* and $Empty_v$ are calculated for each node *v*, starting at the leaves and working back towards the root of the tree. A global variable *Follow(x)* is maintained for each position in RE and this is updated at each node. Full details of the recursive algorithm can be found in the textbook by Navarro and Raffinot (2002).

## 2.1.5. NFA to DFA conversion

Every language that can be described by an NFA can also be described by an equivalent DFA. In practice, a DFA usually has around the same number of states as an NFA but with more transitions. However, in the worst case, the smallest equivalent DFA can have $2^n$ states compared to the *n* states of the smallest NFA.

The classic method to convert an NFA to a DFA is known as the *subset construction* or *powerset construction* (Rabin and Scott, 1959). Each single state in the equivalent

DFA corresponds to a set of states in the NFA. The algorithm, as illustrated in Figure 10 for the Thompson NFA of Figure 8, is as follows:

- The DFA start state is the set of NFA states reachable by an ε-transition.
- Starting with the DFA start state, repeat the following for every new DFA state created until no more new states can be found:
    - For each character from the alphabet of the language, compute the set of states reachable from the DFA state – this set of states constitutes a new state.
- The final or accept states in the DFA are those whose set of NFA states contains at least one final state from the NFA.

Transition Table

| DFA State, $q$ | $\delta(q,a)$ | $\delta(q,b)$ | $\delta(q,c)$ |
|---|---|---|---|
| {0,1,2,4,7,8} | {3,6,7,8} | {5,6,7,8} | {9,10} |
| {3,6,7,8} | {3,6,7,8} | {5,6,7,8} | {9,10} |
| {5,6,7,8} | {3,6,7,8} | {5,6,7,8} | {9,10} |
| {9,10} | {11} | ∅ | ∅ |
| {11} | ∅ | ∅ | ∅ |

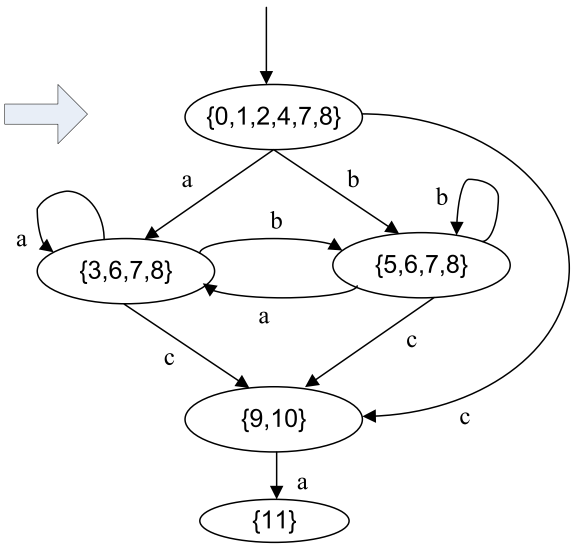


**Figure 10. NFA to DFA conversion of *(a|b)*ca* using Subset Construction**

### 2.1.6. Trie

A trie (from re*trie*val) is a multi-way ordered data tree structure which can be used for storing strings. All strings that branch from the same node share the same prefix. Figure 11 shows the trie for the set of strings *P*={bale, ball, bark}.

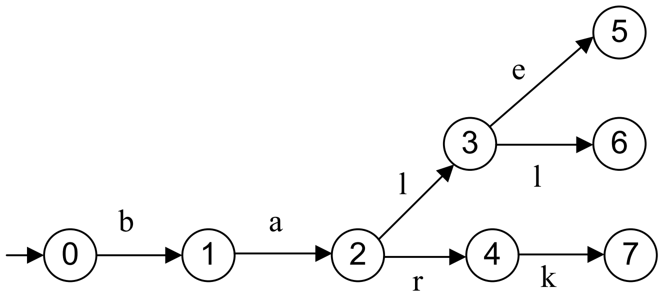


**Figure 11. Trie for set of strings *P*={bale, ball, bark}**

A trie is actually a deterministic acyclic automaton which recognises the corresponding set of strings. Tries are important in string matching.

## 2.2. Network Intrusion Detection and Prevention Systems

### 2.2.1. Snort

Snort is probably the best known network-based NIDPS software and is used as the reference point for the architectures proposed in this thesis. It can be run in three different modes:

- NIDS
- NIPS (inline mode)
- Sniffer mode – like tcpdump

### 2.2.2. Snort Rules

Snort is a signature-based NIDPS. Sourcefire Vulnerability Research Team$^{TM}$ (VRT) rules are the official rules for Snort. Updates to the VRT database are made immediately available to users who have availed of the paid subscription service and are released free of charge to all registered users after 30 days. (Note: Sourcefire is currently in the process of being acquired by Cisco). An alternative rule set provider company to Sourcefire is Emerging Threats Pro LLC. It maintains two rule sets, ETopen which is free, and ETpro which is a paid subscription service. The ET and VRT rules have the same format and it is possible to load both rule sets on the same Snort installation.

Users can also write their own rules. Snort rules have two parts, the rule header and the rule options, as shown in Figure 12.

**Rule Header**

The rule header defines the criteria for matching against packet headers and the action to be taken on finding a match. The criteria consists of the protocol type, the source and destination IP addresses and port numbers and, whether the rule is unidirectional (->) or bidirectional (<>).

Rule Header:
```
[alert, log, pass, activate, dynamic, drop, reject, sdrop]
[ip, icmp, tcp, udp]
[any, <Source IP address subnet>]
[any, <port>]
[->, <>]
[any, <Destination IP address subnet>]
[any, <port>]
```

Rule Options:
```
([content:, msg:, flags: pcre:, byte_test:, flowbits:])
```

**Figure 12. Snort rule syntax (only some rule options are shown)**

*alert* is the most commonly used action and results in the generation of an alert message and the logging of the packet. The other actions are described by Roesch & Green (2012) in the Snort manual along with a description of how to create custom user-defined actions.

**Rule Options**

The rule *options* follow the rule header and are enclosed in a pair of parentheses. There may be one or more options separated by a semicolon. A rule matches only when its header and all of its options match, i.e. logical AND. The following are some of the more important rule options, full details of which can be found in the Snort manual:

- *content* – the *content* keyword defines a fixed string to be searched for in the packet payload. This fixed string can be text and/or bytecode. A single rule can contain multiple *content* keywords, e.g.
  ```
  alert tcp any any -> any 90 (content: "Some string";)
  alert tcp any any <> 10.1.1.0/24 88 (content:"|3c
  ff|G|01|H";)
  ```

- *msg* – this defines the message to use when signalling an alert or when logging a packet, e.g.
  ```
  alert tcp any any -> 10.1.1.0/24 53 (msg:"Some string attack
  attempt"; content:"Some string";)
  ```
- *flags* – this checks if the specified TCP flags are present in the packet.
- *pcre* – this allows patterns to be specified as PCRE regexes.
- *byte_test* – this checks the value of a byte, at a specified offset in the packet payload, using less than, greater than, equals, bitwise AND or bitwise OR operator.
- *flowbits* – the *flowbits* keyword allows rules to track TCP state, effectively linking rules together, e.g.
  ```
  alert tcp any 143 -> any any (content:"IMAP login";
  flowbits:set,logged_in, flowbits:noalert)
  alert tcp any any -> any 143 (msg:"IMAP lsub";
  content:"LSUB"; flowbits:isset,logged_in;)
  ```
  A positive match in the first rule sets the user defined `logged_in` state name but generates no alert because the *noalert* keyword is specified. The second rule checks if the `logged_in` state name is set.

### 2.2.3. Suricata

The open-source software Suricata was developed as a multi-threaded alternative to Snort (Open Information Security Foundation, n.d.). Although the code is original, many architectural concepts were borrowed from Snort, and it can use the same rule sets. A performance comparison of Snort and Suricata is not straightforward as it very much depends on the rule sets, the test traffic and any optimisation settings used. White et al. (2013) found that Suricata performs better than Snort, even for a single core. Albin and Rowe (2012) observed no significant speed advantage of Suricata over Snort except on processors with a large number of cores.

### 2.2.4. Bro

Bro (Paxon, 1999; Sommer, 2011) is an open-source UNIX-based NIDS and network traffic analysis system. Its detection mechanism is activity-based with some support for anomaly detection. The Bro system consists of two parts, the event engine and the policy scripts. The C++ engine analyses the traffic and generates neutral events which do not necessarily indicate an attack. These events are then processed by the policy scripts written in the Bro scripting language which may result in logging to a file, email notification, execution of another script, etc. This scripting language gives a lot of flexibility but the scripts that are provided with Bro by default typically need significant customisation.

### 2.2.5. Market Trends

Gartner recently produced a report (Young and Pescatore, 2012) on how the IPS market is evolving. It found that IPS functionality is increasingly being absorbed into Next Generation Firewall (NGFW) products, although standalone next generation IPS (NGIPS) products are still available. Commercial stand-alone IPS systems include:

- The McAfee (an Intel subsidiary) Network Security Platform includes a range of models capable of high speed performance via a load-balanced cluster. It supports both string and regex DPI and can read Snort rules.
- HP Tipping Point IPS is available as a standalone hardware IPS, as IPS blades for use in HP switches and, as a software version. It supports both string and regex matching and provides tools for importing Snort rules.
- Sourcefire Inc. (in the process of being acquired by Cisco), the commercial manager of the open-source Snort software, has an NGIPS product which runs on its FirePOWER hardware platform.
- Cisco stand-alone IPS products include the 4300 and 4500 series appliances, plus blades for adding IPS capability to Cisco routers. IPS software is also available for the Cisco IOS platform. Cisco IPS systems cannot read Snort rules.

NGFW systems which include next generation IPS capability include:

- Cisco Adaptive Security Appliances.
- Sourcefire has a NGFW solution that can run on the same FirePOWER platform as its NGIPS.
- Checkpoint’s Software Blade Architecture allows customers to select and combine firewall, VPN, IPS, anti-spam, etc., as part of a single NGFW system.
- Palo Alto Networks provides IPS functionality, including hardware acceleration, as part of its NGFW platforms.
- Fortinet’s FortiGate Network Security Platform incorporates a wide range of security technologies including firewall, VPN, IPS, etc. FortiGate has its own rule format which is quite similar to that of Snort.

## 2.3. Platforms

### 2.3.1. Commodity Hardware

Commodity hardware has the advantage of comparatively low cost and its performance may be adequate for many applications. Companies entering into IDS product development may prefer to use commodity rather than specialised hardware in order to reduce time-to-market, keep development costs down and deliver a more maintainable product. Such companies might see the use of specialised hardware as a second step reserved for the development of higher performance products once the commodity-based products have gained a market foothold.

#### Commercial Off-The-Shelf (COTS) Server

Most deployments of software-based NIDPS, such as Snort and Suricata, are on standard servers. Multi-core servers can be used to take advantage of the multi-threaded architecture of Suricata.

#### Graphics Processing Unit (GPU)

The highly parallel architecture of GPUs makes them effective for many complex algorithms. Their relatively low cost has prompted much research into their use for offloading of regex matching from the CPU (Antonello et al., 2012). Vasiliadis et al.'s (2009) GPU-based DPI system uses fixed string pre-filtering software running on the CPU in order to reduce the amount of regex matching that needs to be performed by the GPU. Payloads that match in the pre-filter are forwarded along with a regex identifier to the GPU for regex matching. Zu et al. (2012) evaluated an NFA design on an NVIDIA GTX-460 GPU.

### 2.3.2. Custom Hardware

#### ASIC

Application Specific Integrated Circuits (ASIC) have numerous advantages when it comes to implementing complicated network processing algorithms as they give the great design flexibility, thereby allowing very high traffic speeds, consuming relatively little power and having a small footprint. Unfortunately, these advantages come at a cost. ASIC development is slow and very expensive. The high costs include library and design software licences, manufacturing and engineering design. The extensive testing required and slow manufacturing process result in a long project timescale. ASICs are inflexible in that they cannot be updated with bug fixes or improved algorithms.

**FPGA**

Field Programmable Gate Arrays (FPGA) provide a flexible design platform without the high cost of ASIC fabrication. Time-to-market is also reduced as the development timescale is much shorter. In terms of performance, an FPGA is typically much slower than an ASIC, but much faster than a CPU. The highly parallel nature of an FPGA makes it suitable for the multi-pattern matching involved in intrusion detection.

An FPGA consists of a mix of configurable embedded SRAM (Block RAM), Clock Managers, high speed transceivers and I/Os, and logic blocks, all of which can be wired together via a configurable interconnect fabric. A memory interface is also provided for interfacing the FPGA with external DRAM or SRAM. Some FPGA's include a hard core processor, e.g. Xilinx Virtex 5 includes a PowerPC 440 processor. Alternatively, a soft core processor can typically be generated using the vendor's design tools.

Besides reduced cost and development time, FPGA's also have the important advantage of being reconfigurable, thereby allowing systems to be updated with new improved designs. Designs can also be ported to newer improved FPGAs when they become available.

**TCAM**

A Content Addressable Memory (CAM) is a special type of memory which returns the address of the first memory location that contains a supplied piece of data. The entire CAM is searched in just one clock cycle. A standard CAM is binary in that it can only search for ones and zeros. A Ternary CAM (TCAM) additionally allows bits to be masked as Don't Care. One of the most common uses of TCAMs is in IP routers, where the Don't Care bits are used to mask out some of the address bits in order to represent a subnet. The main disadvantages of TCAM are its high cost and high power consumption due to the extensive circuitry required for the parallel search.

Another disadvantage of TCAM from an NIDPS design perspective is that it only returns the first match rather than all matches. Yu and Katz (2004) describe how this issue can be overcome for IP header multi-matching. Yu et al. (2004) propose how TCAM can be used for multi-pattern fixed string matching. Long patterns are split into multiple parts and so occupy several TCAM locations. One TCAM search is performed for each byte in the incoming packet payload. A full match is declared if all sub-patterns match in the correct order. Meiners et al. (2010) describe how TCAM can also be used for regex matching. They use a number of techniques to reduce the TCAM space required by the minimised DFA and to maximise the matching speed.

**Network Processor**

A network processor (NPU) is a software programmable device with multiple cores or engines. It differs from a standard multi-core processor in that it includes a number of optimised network processing features such as pattern matching and queue management. NPU vendors typically supply an API (Application Programming Interface) as a software library to simplify the development of application software to control the engines and hardware acceleration features, e.g. Netronome supplies an IPS/IDS Application Kit (Netronome, 2012) for the NFP and IXP processors.

## 2.4. Summary

The functionality of the architectures proposed in this thesis is designed to emulate that of Snort, the well-known open-source NIDPS. Snort uses a database of rules, known as the rule set, to list all the attacks it must search for in the network traffic. Each rule consists of a header and an options part. The header part lists the header values to be searched for in the IP packets' IP and TCP/UDP headers, while the options part includes fixed string and/or regex patterns to be searched for in the packet payload. This thesis looks in detail at both multi-match header and regex matching.

Regexes can be equivalently expressed as finite automata (FA) or regular grammar, and represent what are known as regular languages. Finite automata are widely used in regex-matching implementations, and can be categorised as either deterministic (DFA) or non-deterministic (NFA). Both forms are equivalent and a DFA is actually just a special case of an NFA. Several different algorithms exist for the construction of an FA from a regex and for the conversion between NFA and DFA forms. The essential difference between the behaviour of a DFA and an NFA is that an NFA can have multiple concurrently active states whereas a DFA only has one state active at a time. Intuitively, a DFA typically needs many more states that the equivalent NFA, and therefore it occupies significantly more memory. An NFA implementation is more suited to platforms, such as an FPGA or ASIC, that can efficiently handle its parallel nature. A DFA, on the other hand, is typically used in processor-based software implementations that have access to a large amount of memory.

Besides open-source NIDPS such as Snort, there are several commercial products available. Although a number of these products are standalone NIDPS, there is a growing trend to include this functionality in next generation firewall systems.

# Chapter 3 - TCP/IP Reassembly

In order to correctly analyse network traffic, NIDPS systems must reassemble any IP fragments or TCP segments before examining the reconstructed data flow for attack patterns. This reassembly must exactly match that on the target destination host, as otherwise an attacker may evade detection. Different operating systems have subtle differences in their implementations of IP fragment and TCP segment reassembly and so an NIDPS must select the reassembly procedure to use based on the OS of the target host, i.e. target-based reassembly. An alternative approach is for the NIDPS to normalise the traffic, by removing any ambiguities, before forwarding to the destination.

Performing TCP/IP tracking and reassembly in software at high traffic speeds places a very heavy work load on the processor due to the amount of memory copying and the potentially huge number of flows that need to be tracked. TCP Offload Engine (TOE) technology is available to reduce the load on server CPUs by shifting TCP layer processing to the Network Interface Card (NIC). Several commercial hardware IP cores are available as building blocks for ASIC and FPGA designs, e.g. from Intilop Corp. (2012) and PLDA (2012). However, these solutions are aimed at end host systems and are not suitable for performing connection tracking and reassembly in DPI solutions on intermediate hosts.

Most existing research proposals on TCP/IP reassembly for NIDPS are either fully software based, such as the work of Novak and Sturges (2007), or fully hardware based, such as the works of Necker et al. (2002) and Schuehler and Lockwood (2004). In this thesis, a hybrid architecture is described which splits the processing between a slow path and a fast path, as shown in Figure 13. The fast path handles the most frequent tasks that can take advantage of the parallelism of hardware logic, while the slow path handles the less frequent but more involved tasks that are more suitable for software implementation. The outlined architecture is based on the Xilinx Zynq-7100 System on Chip (SoC) with built-in hard dual-core ARM processor, but is equally applicable to any suitable FPGA or ASIC device with an internal or external CPU and sufficient internal memory.

This chapter first covers the main points of the theory of IP fragmentation and TCP segmentation, including the dependency on OS type, followed by an overview of the reassembly functionality included in the open source software NIDPS systems, Snort and Suricata. Finally, the hardware acceleration system is presented. This system is designed to emulate as much as possible the reassembly functionality of Snort.

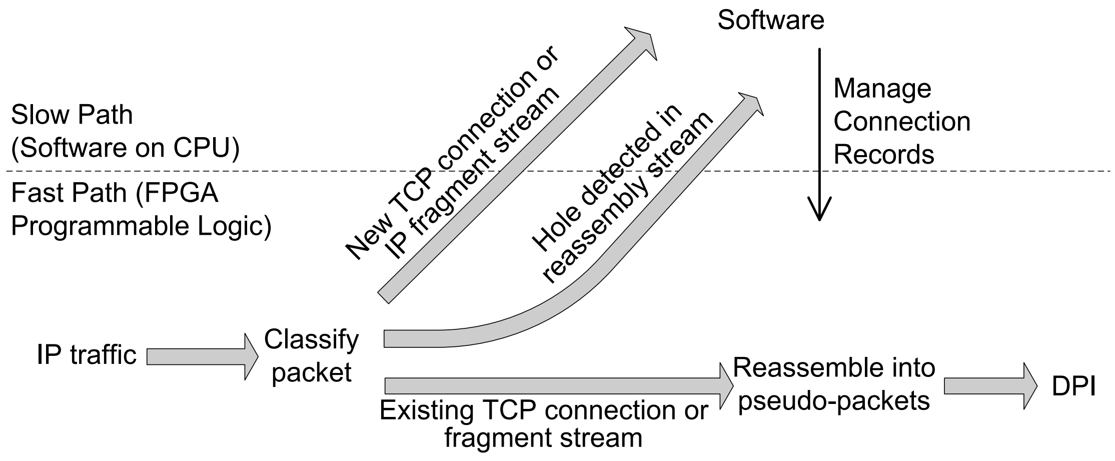

Software
Manage Connection Records
Slow Path (Software on CPU)
Fast Path (FPGA Programmable Logic)
New TCP connection or IP fragment stream
Hole detected in reassembly stream
IP traffic
Classify packet
Existing TCP connection or fragment stream
Reassemble into pseudo-packets
DPI

**Figure 13. Hybrid software-hardware processing**

## 3.1. Theory of IP Fragmentation and TCP Segmentation

### 3.1.1. TCP Connections

#### Three-way handshake

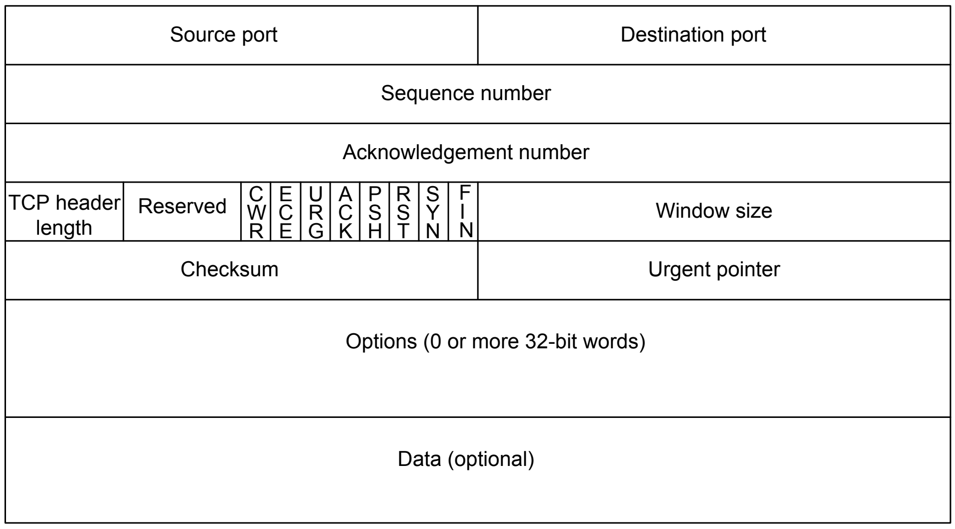


**Figure 14. TCP Header**

The establishment of a TCP connection between a client and a server is performed using the three-way handshake, as illustrated in Figure 15. First the client's TCP layer sends a TCP segment to the server with the SYN flag set, the sequence number set to a randomly generated Initial Sequence Number (ISN) and, optionally, the Maximum

Segment Size (MSS) set to the largest segment size it can handle. The server receives the request and responds with a TCP segment with both the SYN and ACK flags set, the sequence number set to its own randomly generated ISN, the acknowledge number set to the sequence number it expects to receive next from the client, i.e. the received number plus one, and, optionally, the MSS it can handle. The connection is completed by the client responding to the server's reply with a segment with the ACK flag set, its sequence number incremented by one and the acknowledge number set to the sequence number is expects next from the server, i.e. the sequence number just received plus one. If no MSS is included in a SYN segment, then the default value of 536 bytes is assumed. Once the connection is established, both client and server send TCP segments at the lesser MSS of the two nodes.

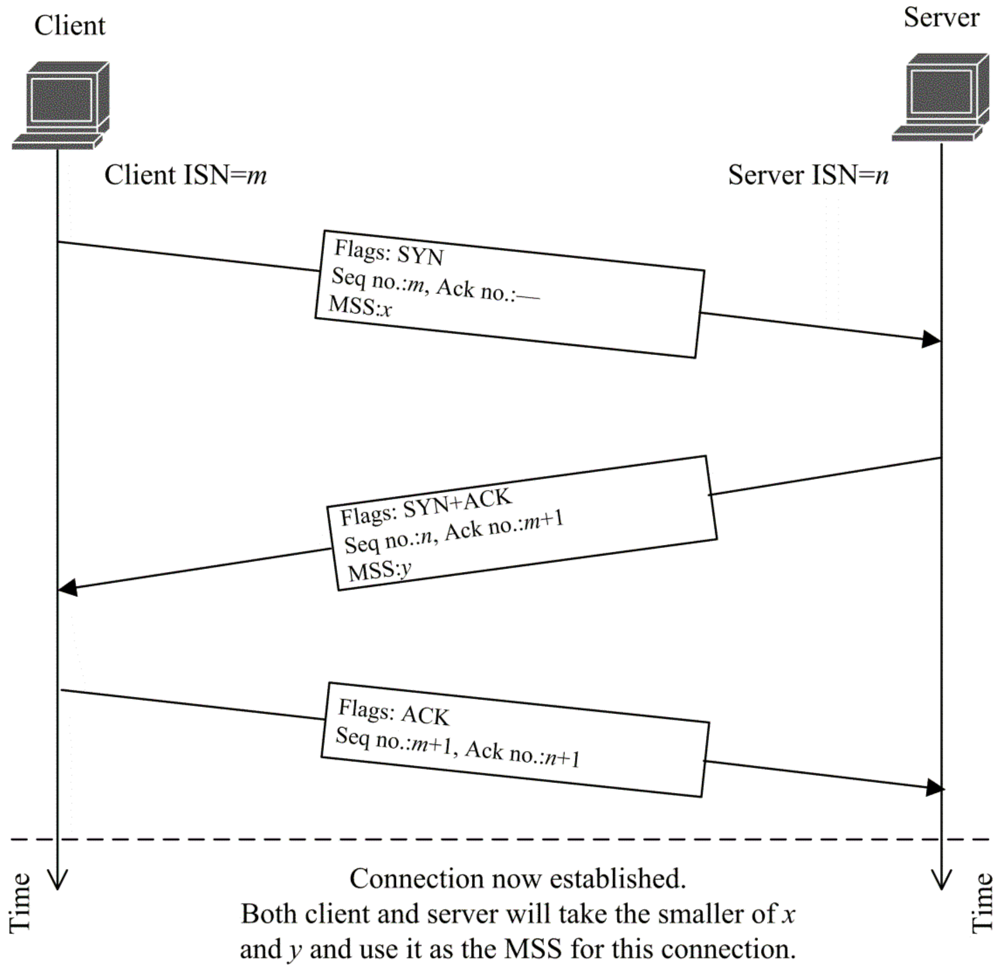


**Figure 15. TCP 3-way handshake**

**Sequence and Acknowledgment numbers**

Each byte of data in TCP is assigned a sequence number. During the 3-way handshake, the sequence number is set to the ISN when the SYN flag is, and the first byte of data is numbered ISN + 1. The Acknowledgement number is valid if the ACK flag is set, and it contains the value of the next sequence number that the sender of the acknowledgement expects to receive. Note that once a connection is established and data is being transmitted, the ACK flag is always set. Figure 16 illustrates how the values of sequence and acknowledgement numbers are set during data transmission. Note that an ACK segment without any data does not result in the next sequence number being increased. Things to observe from Figure 16 are that multiple segments can be acknowledged with one ACK segment, that the ACK is cumulative, and that data can be included in the same segment as an acknowledgement.

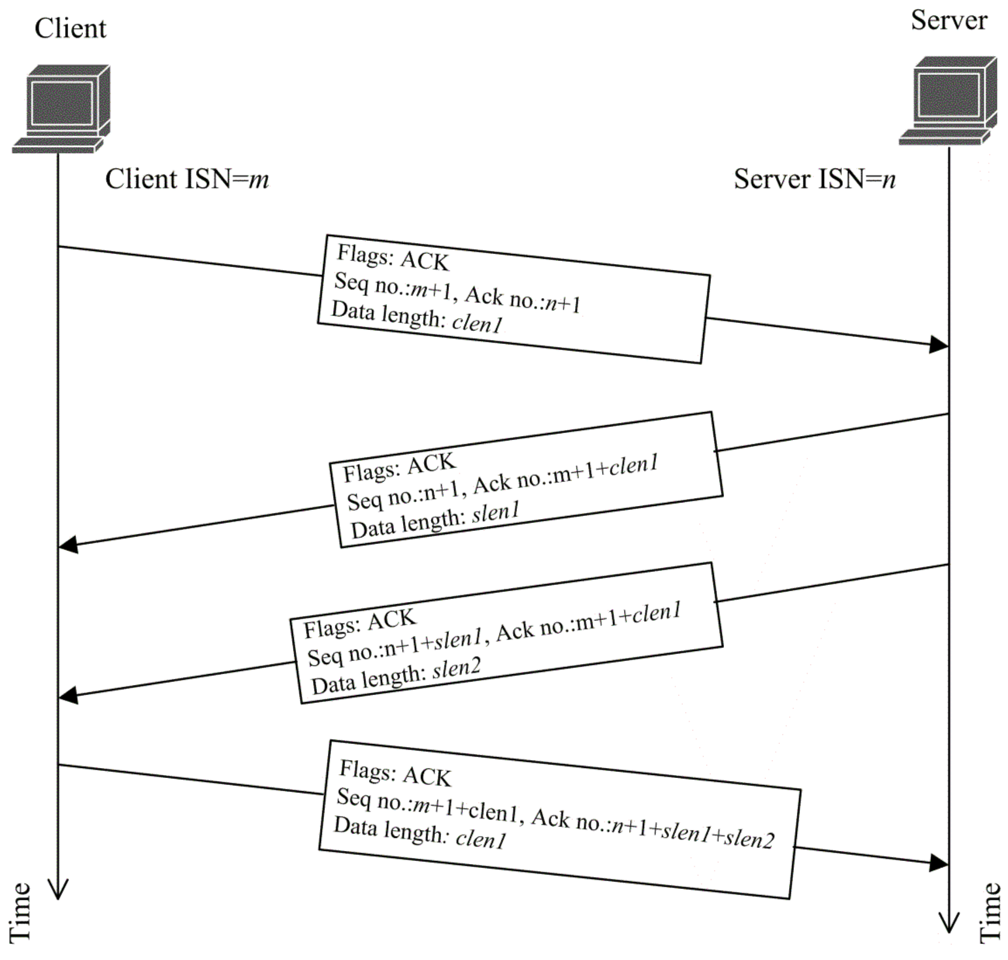


**Figure 16. TCP data segment transmission**

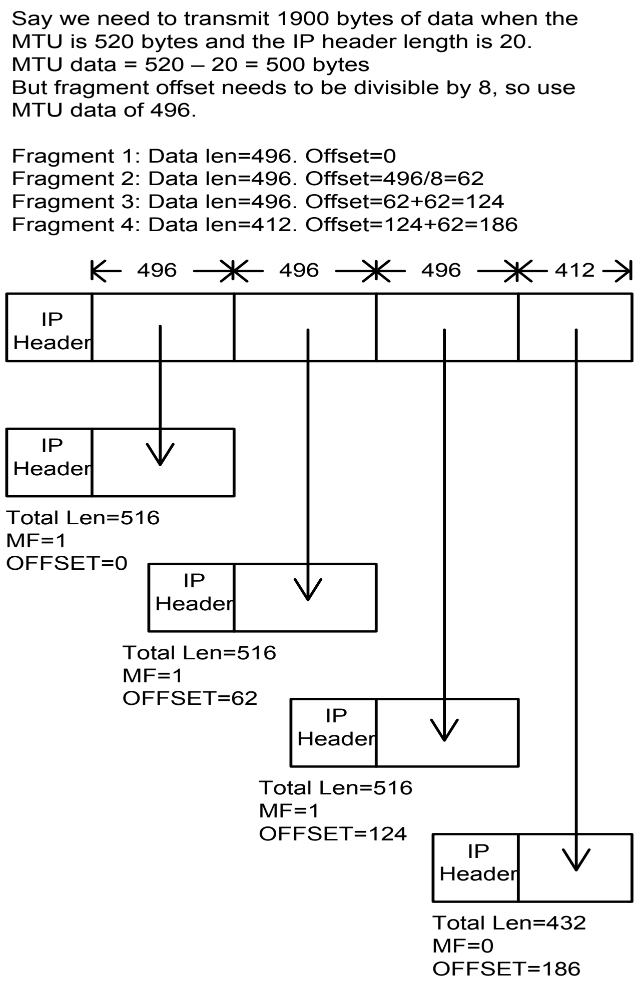
TCP uses a *sliding window* flow control algorithm for the fast and efficient transfer of data. The *window* is the maximum amount of data that can be sent without waiting for an ACK. The size of the window is based on the size of the receiver's available buffer space for the connection, and it can be dynamically adjusted in an ACK segment. If transmitted data is not acknowledged within a certain time, it is retransmitted. TCP sequence numbers are used in the receiver's TCP layer to distinguish retransmitted data and to reassemble out-of-order and overlapping segments.

### 3.1.2. The need for IP fragmentation

IP fragmentation occurs in a network node if the size of the packets exceeds the Maximum Transmission Unit (MTU) of the outgoing interface. IP Fragmentation at intermediate nodes is usually avoided by using the technique of Path MTU Discovery (PMTUD) that determines the minimum MTU for the entire path from source to destination node. As the UDP protocol does not perform payload fragmentation, the most common occurrence of IP fragmentation is in the case of UDP traffic on source nodes. The TCP protocol, however, does perform fragmentation, which is known as TCP segmentation, if the data size is greater than the MSS. The MSS value usually ensures that fragmentation is avoided at the IP layer in the case of TCP traffic.

The following IP header fields are used in fragmentation

- *Identification* field: this value is the same in each fragment of a particular datagram
- *Flags* field: The MF (More Fragments) bit is set in all fragments apart from the last; the DF (Don't Fragment) bit is used to prevent fragmentation of an IP packet
- *Fragment Offset* field: contains the offset, in 8-byte units, of this fragment in the original datagram, and, as a result, the data portion of each datagram, apart from the last, must be a multiple of 8 bytes in length
- *Total Length* field: contains the size of the fragment, including the header (same as any IP packet)


Say we need to transmit 1900 bytes of data when the MTU is 520 bytes and the IP header length is 20.
MTU data = 520 – 20 = 500 bytes
But fragment offset needs to be divisible by 8, so use MTU data of 496.

Fragment 1: Data len=496. Offset=0
Fragment 2: Data len=496. Offset=496/8=62
Fragment 3: Data len=496. Offset=62+62=124
Fragment 4: Data len=412. Offset=124+62=186

**Figure 17: Example of IP packet fragmentation**

### 3.1.3. Path MTU Discovery

Path MTU Discovery (PMTUD) (Mogul and Mooring, 1990) is a technique commonly used to avoid unnecessary IP fragmentation. It allows each node to determine the minimum MTU on the path to each destination in its routing table. PMTUD is a continuous process as dynamic routing may result in a decrease in the PMTU due to a different path being used. PMTUD works by setting the Don't Fragment (DF) bit in the header of outgoing IP packets. When the packet reaches a node along the path that is unable to forward the packet because it is larger than the MTU of its outgoing interface, then that node will drop the packet and notify the sender using an ICMP Destination Unreachable – Datagram too big message. This message includes the next-hop MTU which allows the sender to try again with a smaller packet. This process is repeated until the packet successfully travels the full path to the destination.

Unfortunately, PMTUD frequently does not work because the ICMP messages are often blocked by firewall configuration. Packetization Layer Path MTU Discovery (PLPMTUD) (Mathis and Heffner) is a more robust alternative that does not require the use of ICMP messages. It uses a transport layer protocol, such as TCP, to probe the path with progressively larger packets. Another method is a "hack" known as MSS clamping. This involves configuring a router or firewall to lower the MSS of all TCP connections passing through. MSS clamping should only be used as a last resort as it can cause problems for some protocols.

The TCP layer of a host calculates the MSS by subtracting the fixed lengths of the IP and TCP headers from the PMTU, i.e. PMTU minus 40. The length of IP or TCP options is not considered in the calculation. It is up to the sender of a TCP segment to ensure that it reduces a segment's data length to compensate for any IP or TCP options fields. The MSS used for a particular connection is negotiated during the 3-way handshake. With functioning PMTUD or PLPMTUD, IP fragmentation is only necessary on the sending host in the case of non-TCP datagrams, typically UDP, that are too large to fit in a single IP packet.

### 3.1.4. IP Reassembly

Reassembly is a complex task due to the fact that fragments may arrive out of order having followed different routes to the destination. Packet reordering in routers is now rare as most use connection-level parallelism instead of packet-level-parallelism (Dharmapurikar and Paxon, 2005). Fragments may also be retransmitted and their payloads may even overlap. Unfortunately, the standards do not specify what to do in the case of overlapping fragments, duplicate fragments, and duplicate fragments received after the packet has already been reassembled and consequently, different OS implementations differ in which copy of the data they give precedence to when reassembling the packet.

### 3.1.5. TCP Segmentation

The value of a TCP's connection MSS usually ensures that fragmentation is avoided at the IP layer when carrying TCP traffic. The TCP layer on the source node is responsible for segmenting the data stream received from the application layer and adding a TCP header to create a TCP segment. This process is known as TCP segmentation or packetisation. The TCP layer on the destination node then reassembles the data as it streams it to the application layer. This reassembly includes reordering and validating of segments as well as handling of duplicates and overlaps.

A short note on terminology: all TCP traffic is in the form of TCP *segments* (from the segmented data stream); this is somewhat different to the IP layer where an IP *fragment* refers to a chunk of the data portion (or the corresponding IP packet containing that chunk) of a larger IP *packet* that was split up.

TCP header fields relevant to segmentation/reassembly

- *Sequence number* field: identifies the position of the first byte of data in this segment in the overall data stream
- *Acknowledgment number* field: is only valid if the *ACK* flag is set. It identifies the position of the byte in the overall data stream that the sender of the ACK is expecting to receive next
- *Flags*:
    - *ACK:* indicates that the Acknowledgment number is significant
    - *SYN:* this flag is set in the first packet from both client and server
    - *FIN:* no more data, close connection
    - plus a few other less significant flags

## 3.2. Handling of Reassembly in different Operating Systems

### 3.2.1. Simple Insertion and Evasion Attacks

These attacks involve either sending an IP fragment or a TCP segment that is accepted by the NIDPS but dropped by the target host, or vice versa. One example would be where the attacker sends a FIN TCP segment with an invalid header which is dropped by the target host, but is accepted by the NIDPS, resulting in the NIDPS removing its record of the connection. The attacker can then send the malicious content in subsequent segments which would typically be ignored by the NIDPS since it no longer has a record of the connection, but the target host may accept the malicious content and thus be affected. Another example would be where the malicious content is spread over several segments, with the invalid segment in the middle. If the invalid segment is accepted by the NIDPS, it may not detect the malicious content, as the data contents of the invalid segment disguise the attack pattern. In the example shown in Figure 18, the attacker spreads the attack over six segments of a TCP connection. The attacker, however, sends two versions of segment 4, the first containing part of the attack and the second carrying a harmless pattern. If the NIDPS uses the second version, then its pattern matching engine will not detect the attack. Whether or not the destination host is successfully attacked depends on its operating system. A Linux host, for example, would discard the old version of segment 4 and use the new harmless segment, and so would not be affected. A Windows host, on the other hand would discard the new duplicate segment and use the old segment, and so the attack would be successful.

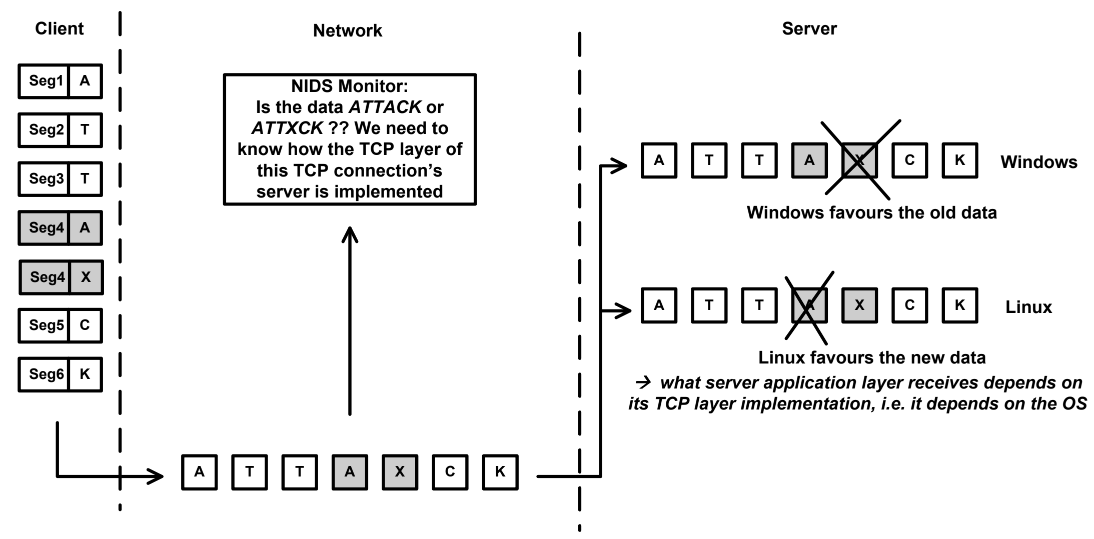

Client
Network
Server
Seg1 A
Seg2 T
Seg3 T
Seg4 A
Seg4 X
Seg5 C
Seg6 K
NIDS Monitor: Is the data ATTACK or ATTXCK ?? We need to know how the TCP layer of this TCP connection's server is implemented
A T T A X C K
A T T A X C K Windows
Windows favours the old data
A T T A X C K Linux
Linux favours the new data
→ what server application layer receives depends on its TCP layer implementation, i.e. it depends on the OS

**Figure 18: Example of insertion attack**

To avoid these attacks the NIDPS must take account of the following points:

- How duplicate fragments and segments are dealt with depends on the implementation, i.e., on the destination OS.
- Check for invalid combinations of code bit flags (SYN, ACK, etc.).
- Check for valid checksum.
- Many implementations (apart from Linux) only accept data if a code bit flag is set.
- Some implementations (apart from MacOS) do not handle data in a SYN segment.
- TCP options are handled differently by different operating systems, e.g. handling of PAWS (Protection Against Wrapped Sequence numbers), as described in RFC 1323.

### 3.2.2. Creation of connection session

An NIDPS has numerous options on how to implement connection session creation, the two main ones being:

- Require 3-way handshake.
- Synchronize on data segments (this has the advantage that existing connections can be detected by the NIDPS after it starts up).

Both options need to be implemented carefully in order to avoid potential attacks (Ptacek and Newsham, 1998).

### 3.2.3. TCP Stream Reassembly – Connection Window

The NIDPS must handle the connection "window" in the same way as the target host. The window is the maximum number of bytes of data the receiver will accept from the sender without generating an ACK. The receiver discards any data received past the window. The receiver informs the sender of the new window size in an ACK segment. According to RFC 793, the receiver should not *shrink* the sender's window, i.e. move the right window edge to the left. It can, of course, reduce the window size to a minimum equal to the existing window size minus the amount of data acknowledged by this ACK. However, RFC 793 also states that senders must be robust against window shrinking.

Window shrinking presents a difficult problem because the instant in time at which the NIDPS detects changes in the window size is delayed with respect to the change on the target, e.g. there is a short interval of time between generation of an ACK segment on the target and its receipt by the NIDPS and during this interval the NIDPS is still using the old window size and is therefore vulnerable to an insertion attack. Fortunately, TCP window shrinking has been avoided in most TCP/IP stack implementations.

## 3.2.4. Overlapping Fragments or Segments

**Table 6. Reassembly Policies – segment data favoured when overlap occurs**

| *Start of new segment compared to start of old* | *End of new segment compared to end of old* | *Segment Labels from* Figure 19 | | *Data selected by target OS*[*] | |
|---|---|---|---|---|---|
| | | *New* | *Old* | *New* | *Old* |
| New starts before old | New ends before old | J | B | Linux 2.4+<br>Linux 2.2<br>Windows/BSD | Vista<br>Solaris |
| | New ends same as old | M | E | Linux 2.4+<br>Linux 2.2<br>Windows/BSD<br>Solaris | Vista |
| | New ends after old | L | D | Linux 2.4+<br>Linux 2.2<br>Windows/BSD<br>Solaris | Vista |
| New starts same as old | New ends before old | P | H | Windows/BSD | Vista<br>Linux 2.4+<br>Linux 2.2<br>Solaris |
| | New ends same as old | K | C | Linux 2.2<br>Windows/BSD<br>Solaris | Vista<br>Linux 2.4+ |
| | New ends after old | Q | I | Linux 2.4+<br>Linux 2.2<br>Windows/BSD<br>Solaris | Vista |
| New starts after old | New ends before old | N | F | | Vista<br>Linux 2.4+<br>Linux 2.2<br>Windows/BSD<br>Solaris |
| | New ends same as old | O | G | | Vista<br>Linux 2.4+<br>Linux 2.2<br>Windows/BSD<br>Solaris |
| | New ends after old | J | A | Solaris | Vista<br>Linux 2.4+<br>Linux 2.2<br>Windows/BSD |

**Windows* signifies all Microsoft Windows versions released before Vista. More recent versions of Microsoft Windows probably exhibit the same reassembly behaviour as Vista, but this has not yet been verified. Others that behave like *Windows* include Win2003 Server, BSD, MacOS, HPUX10 & IRIX. HPUX11 has the same behaviour as *Solaris*. *Linux 2.4+* signifies Linux 2.4 and newer.

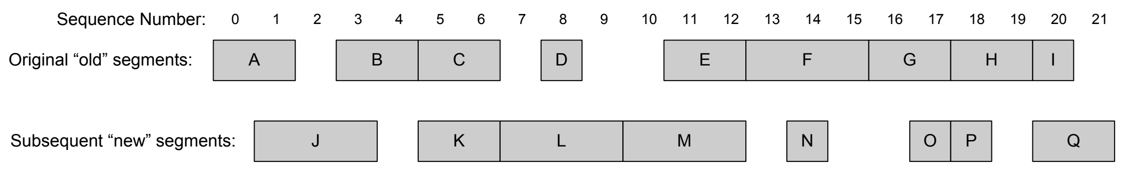


**Figure 19. TCP Segment overlap**

It is possible for an attacker to send several IP fragments with the same identification and offset but different data, or several TCP segments with the same sequence number but different data. Which fragment or segment is accepted by the destination host is OS dependent. Similarly, it's possible for fragments/segments to partially overlap, and again, which data is used by the destination is OS dependent. The NIDPS system must match the target host behaviour in order to correctly detect any attacks (Novak and Sturges, 2007). The different ways in which TCP segments can overlap are illustrated in Figure 19 and the corresponding reassembly policies, based on target OS, are outlined in Table 6.

### 3.2.5. TCP Stream – RST Validity Check

RFC 793 states that, in all states except SYN-SENT, all RST segments are validated by checking their sequence numbers. A reset is valid if the sequence number is in the window. However, different operating systems actually differ in how they validate a RST. Table 7 shows how Snort and Suricata validate RST segments when not in the SYN-SENT state.

**Table 7: Target-based checking of RST segment**

| *Target OS Policy* | *RST validity check* |
|---|---|
| **HPUX11** | RST sequence no. >= expected sequence no. |
| **Linux, Solaris** | RST sequence no. + payload len. >= last acknowledgement no. *AND* RST sequence no. < expected sequence no. + window size |
| **BSD, HPUX11, IRIX, MacOS, Windows, Win2003, Vista** | RST sequence no == expected sequence no. |

### 3.2.6. TCP Stream – Timestamp Validity Check

The TCP timestamp is used by the PAWS algorithm in order to reject old duplicate segments. Handling of the timestamp depends on the target OS:

- HPUX11 ignores timestamps for out of order segments.
- Old Linux (2.2 and earlier), Windows & Vista allow the 3 way handshake to use a zero timestamp whereas Linux and Win2003 do not.
- Linux accepts timestamps that are off by one.
- Old Linux, all Windows OS and Solaris allow a 0 timestamp value, others do not.
- Solaris stops using timestamps if it receives a segment without a timestamp on a stream where timestamps were in use.

### 3.2.7. TCP Stream – Handling of repeated SYN segment

- All Windows operating systems reset the connection if the sequence number of the repeated SYN segment is the next expected sequence number, otherwise the repeated SYN segment is dropped.
- MacOS always ignores a repeated SYN segment.
- Every other OS resets the connection if the SYN segment is not a retransmission of the original, i.e. if the sequence number of this SYN segment does not match the initial sequence number of the connection. Otherwise the repeated SYN segment is dropped.

## 3.3. Target-based Reassembly and Normalisation

In order to avoid the possibility of evasion or insertion attacks, an NIDPS must either perform target-based reassembly (Novak and Sturges, 2007) or traffic normalisation/scrubbing (Malan et al., 2000).

Target-based reassembly requires the NIDPS to be aware of the OS type of every destination node in the network it is protecting. In the case of the well-known open-source software NIDPS systems Snort and Suricata, the OS must be configured manually for each destination IP address or subnet. Unfortunately, this approach is not ideal as reconfiguration of the NIDPS may be overlooked when nodes are added or removed and, it does not scale well for large networks. Dynamic Host Configuration Protocol (DHCP) servers also need to be carefully configured so that subnets are split by OS type. Active mapping (Shanker and Paxon, 2003) and passive OS fingerprinting (Taleck, 2003) are two techniques used for the automatic discovery of the target OS. The active mapping method involves sending specially built probe packets to target hosts and inferring how the OS performs reassembly from the format of the response packets. The downsides of this method are that the probe packets may be dropped by the destination nodes or blocked by a firewall and, it requires integration with any DHCP servers in the protected network. The passive OS fingerprinting method infers how the OS performs reassembly by monitoring the destinations' responses to fragmented traffic from the source. Problems with this technique are the added workload of monitoring the traffic and that the correct destination reassembly policy is only discovered after some fragmented traffic has already passed through. All of the above methods become more complicated if Network Address Resolution (NAT) occurs between the NIDPS and any destination node in the protected network.

Malan et al. (2000) proposed a protocol scrubber aimed at converting ambiguous traffic flows into well-behaved flows. The simplistic approach to normalization is to buffer all unacknowledged data for every connection and to compare retransmitted data against that in storage. A more efficient scheme is to store hashes of the unacknowledged data. Vutukuru et al. (2008) proposed such a hash-based system which is able to correctly handle retransmissions which are not aligned with the original segments boundaries.

## 3.4. TCP/IP Reassembly in Software

The following is an analysis of how the open source Snort and Suricata software NIDPS systems perform IP packet and TCP fragment reassembly. An abstract illustration of the Snort software stack, operating in passive NIDS mode, is given in Figure 20.

### 3.4.1. Snort

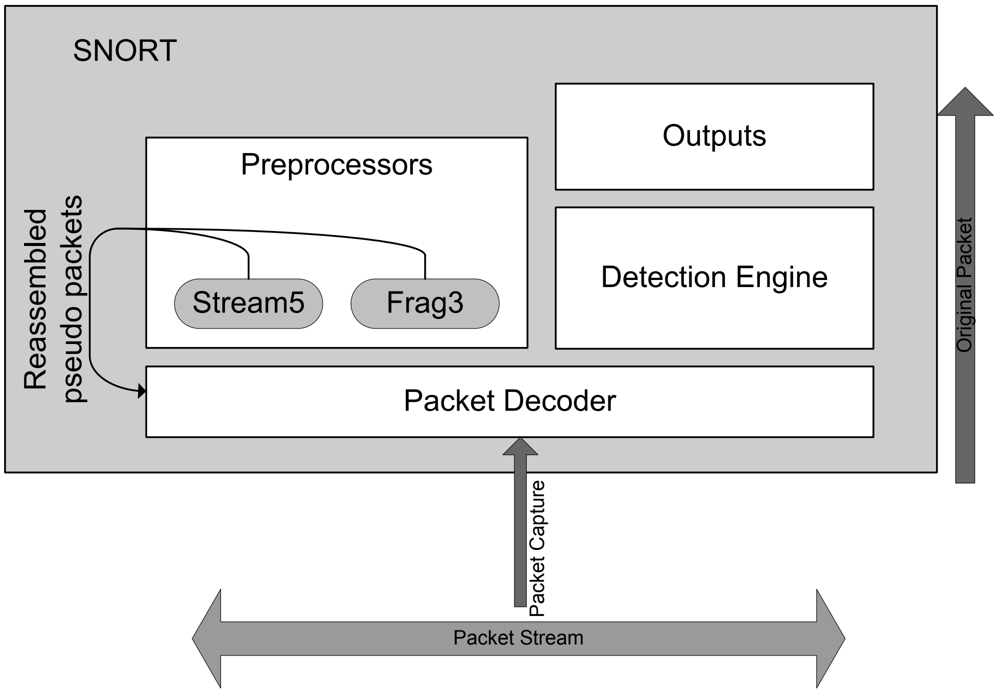


**Figure 20: SNORT Architecture**

**Packet Decoder**

The Packet Decoder receives captured frames and adds pointers to critical data locations – the ethernet header, IP header, TCP header and payload. It also carries out some simple validity checks.

**Frag3 & Stream5 pre-processor**

The Snort preprocessor modules, Frag3 and Stream5, are responsible for IP defragmentation and TCP segment reassembly, respectively. The numbers at the end of the pre-processor names are used to differentiate these new versions from previous major versions which are now deprecated. The reassembled pseudo-packets generated by these modules are injected back into the Packet Decoder so that they can be processed by other preprocessors as appropriate. Both the pseudo-packets and original packets are analyzed by the DPI engine. The Frag3 module reassembles fragments into a single IP pseudo-packet with a maximum size of 65535 bytes. The Stream5 module, by default, operates footprint-based flushing which results in the generation of a pseudo-packet once its footprint reaches the flush point limit. The footprint is the amount of data in the connection flow's list of segments that has been acknowledged by the destination. Each TCP connection stream's flush point is, by default, a pseudo-random number in order to make it more difficult for an attacker to avoid detection by having the attack data span the boundary between two pseudo-packets.

Stream5 also performs session tracking for TCP, UDP and ICMP. Both preprocessors generate alerts for certain fragment/segment-based attacks.

Different reassembly policies can be configured on a per IP subnet basis, i.e. one particular target subnet can be configured to have a Linux reassembly policy, another to have a Windows Vista policy, etc.

**Pseudo-packets**

Each TCP session has two lists, one for the direction towards the client, and one for the direction towards the server, in which it queues received segments. When an ACK arrives, Stream5 checks if the flushing condition has been satisfied and, if so, constructs the pseudo-packet.

The flush points used are contained in an array of 64 elements and can be

- static values between 128 and 256 bytes
- all elements 192 bytes
- random values between 128 and 256 bytes (default)

Each newly created session takes a flush-point value from this array, and the next session will take the following value, etc.

The acknowledgment number of the ACK is compared to the sequence number of the first queued segment, and if the difference is greater than or equal to the flush point, a pseudo-packet is generated. Queued segments (that have been ACKed) are then copied into the pseudo-packet. If there are missing segments, then the pseudo-packet will only contain the segments up to the first gap, and subsequent segments will be dropped. The pseudo-packet structure has an IP packet buffer size of 65535 bytes. Pseudo-packets are injected into the detection engine (in the same way as the original packets). Note that the stream is also flushed when a connection is about to close, and any remaining ACKed segments are merged into a pseudo-packet.

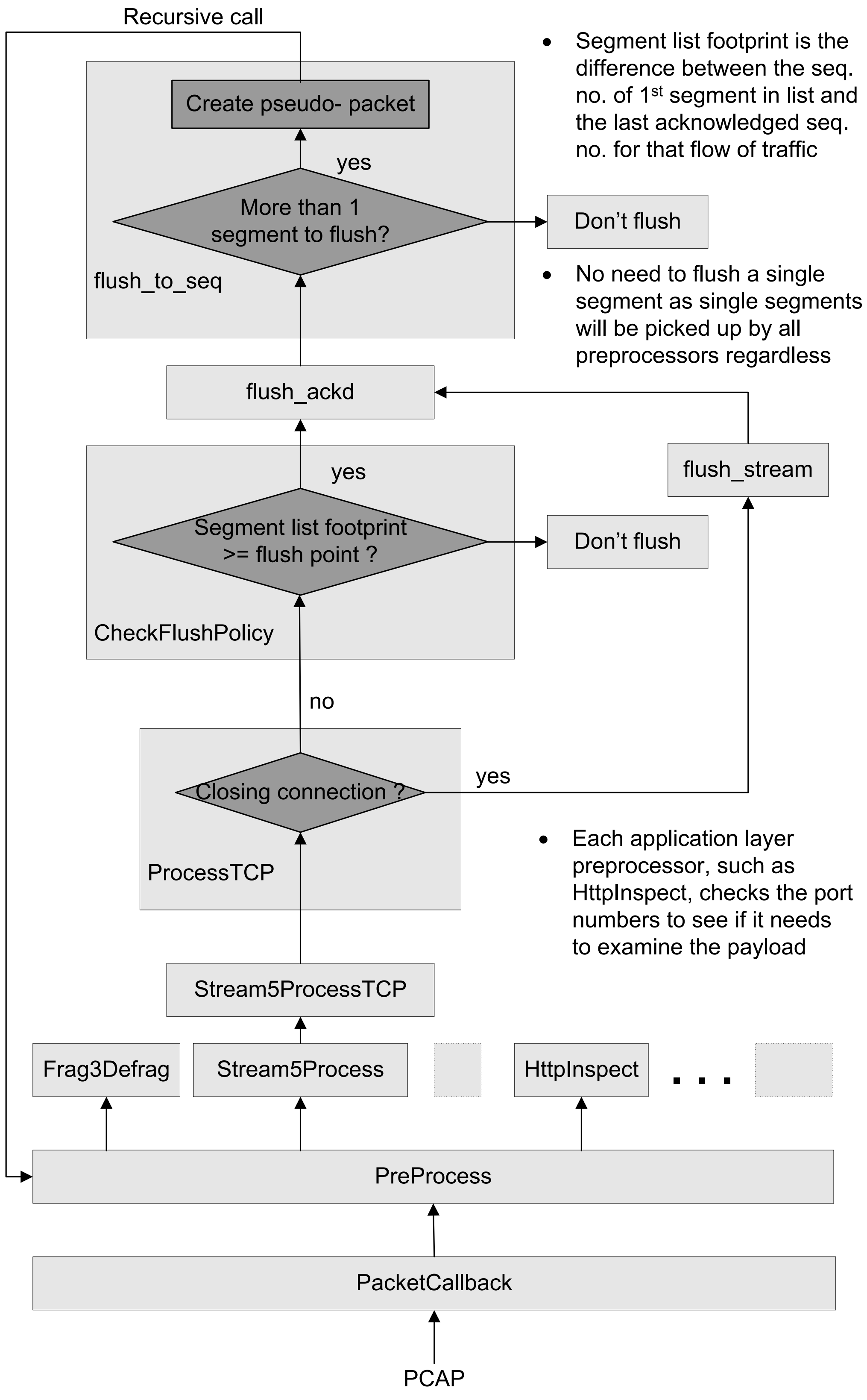

Recursive call
Create pseudo- packet
yes
More than 1 segment to flush?
flush_to_seq
Don't flush
Segment list footprint is the difference between the seq. no. of 1st segment in list and the last acknowledged seq. no. for that flow of traffic
No need to flush a single segment as single segments will be picked up by all preprocessors regardless
flush_ackd
flush_stream
yes
Segment list footprint >= flush point ?
Don't flush
CheckFlushPolicy
no
Closing connection ?
yes
ProcessTCP
Each application layer preprocessor, such as HttpInspect, checks the port numbers to see if it needs to examine the payload
Stream5ProcessTCP
Frag3Defrag
Stream5Process
HttpInspect
. . .
PreProcess
PacketCallback
PCAP

**Figure 21: Snort TCP reassembly flowchart**

Footprint-based flushing is the default in Snort. Protocol Aware Flushing (PAF) was added as a configurable alternative in Snort version 2.9.1 in order to allow reassembly of complete PDUs for HTTP, SMB and DCE/RPC protocols up to a configured maximum length. PAF support for FTP was added in version 2.9.2. For example, PAF guarantees a single HTTP request per pseudo-packet in the case of a HTTP request that spans several TCP segments and also in the case of a TCP segment that contains data from more than one HTTP request. PAF involves stateful analysis of the TCP data stream in order to pinpoint the start and end of each protocol's PDU. The user can configure a maximum PDU length between zero (PAF disabled) and 63780 which Snort then adds to a value from the flush point array to give the actual flush point value for a particular flow. If the length of the reassembled PDU is less than the flush point, then the pseudo-packet will consist of a single PDU, otherwise it will be split. The higher the configured maximum PDU length, the better the detection accuracy. This comes at the cost of increased packet latency and the default value of 16k was found to be a good compromise.

### 3.4.2. OISF Suricata

Suricata, also includes a target-based TCP reassembly engine which has many similarities to that of Snort.

One difference from Snort is that all analysis is conducted on pseudo-packets when operating in IDS non-inline mode, i.e. reassembled TCP segments that have been acknowledged.

**Important structures in software (non-inline mode)**

- Flow is associated with each 4-tuple of the source & destination IP addresses and the source & destination TCP port numbers.
- TcpSession is created for each new TCP connection and a reference is made to it in the Flow structure, i.e. there are 2 Flows for each TcpSession, one for each direction.
- For every packet received, the Flow is looked up in a hash table, and, if none is found, a new Flow is created.
- For each TCP segment received, the TCP stream code checks if the Flow structure refers to a TcpSession. If not, a new TcpSession is allocated.

- Application analysis layer checks each Packet's Flow structure to see if it contains the application layer protocol (i.e. was set on analysis of an earlier packet on the same flow). If not found, it determines the protocol type (e.g. HTTP) from analysing the application layer header and saves this protocol type in the Flow structure. So the application protocol type only needs to be worked out for the first packet in the flow.

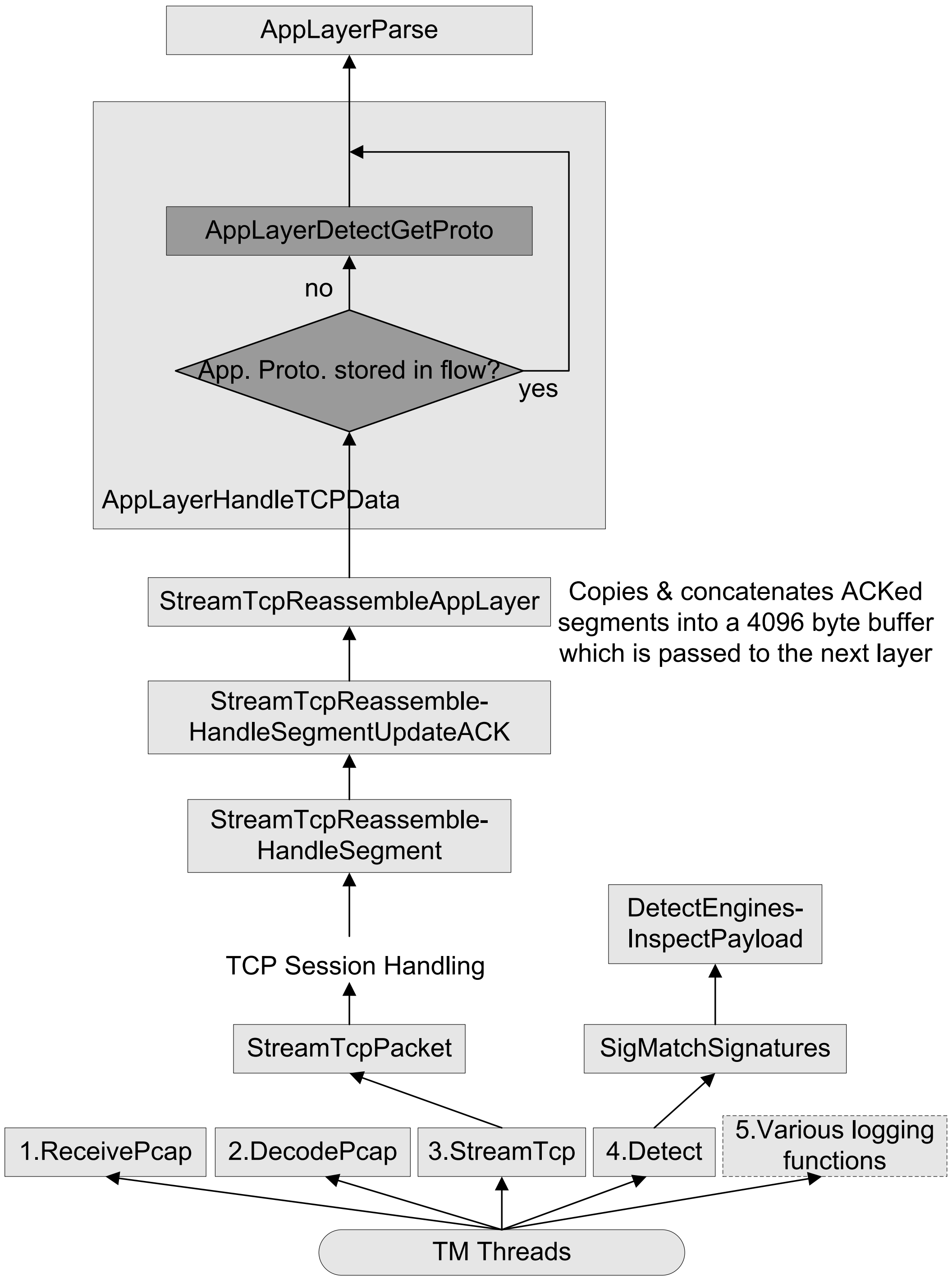

AppLayerParse
AppLayerDetectGetProto
no
App. Proto. stored in flow?
yes
AppLayerHandleTCPData
StreamTcpReassembleAppLayer
Copies & concatenates ACKed segments into a 4096 byte buffer which is passed to the next layer
StreamTcpReassemble-HandleSegmentUpdateACK
StreamTcpReassemble-HandleSegment
TCP Session Handling
DetectEngines-InspectPayload
StreamTcpPacket
SigMatchSignatures
1.ReceivePcap
2.DecodePcap
3.StreamTcp
4.Detect
5.Various logging functions
TM Threads

**Figure 22: Suricata TCP reassembly flowchart (non-inline mode)**

**TCP Stream Inline Mode**

Suricata can be run in IPS mode by combining it with iptables and NetFilter. In this mode, you can enable TCP inline streaming. In inline mode, the TCP reassembly engine does not wait for ACKs before reassembling segments. Instead it effectively normalises the TCP traffic, and payload inspection is performed on a sliding window basis.

## 3.5. TCP/IP Reassembly in Hardware

The overall hardware architecture of the proposed NIDPS system based on a Xilinx FPGA, such as the Zynq-7100 All Programmable SoC, is shown in Figure 23. The grey modules are Xilinx IP (Intellectual Property) cores while the white modules represent custom logic. All cores are connected using the Advanced eXtensible Interface 4 (AXI4) interconnect fabric which is the fourth generation of the Advanced Microcontroller Bus Architecture (AMBA) from ARM. Use of AXI4 effectively enables plug-and-play of IP cores on Xilinx FPGAs, simplifying the integration of cores from various sources (Sundaramoorthy et al, 2010).

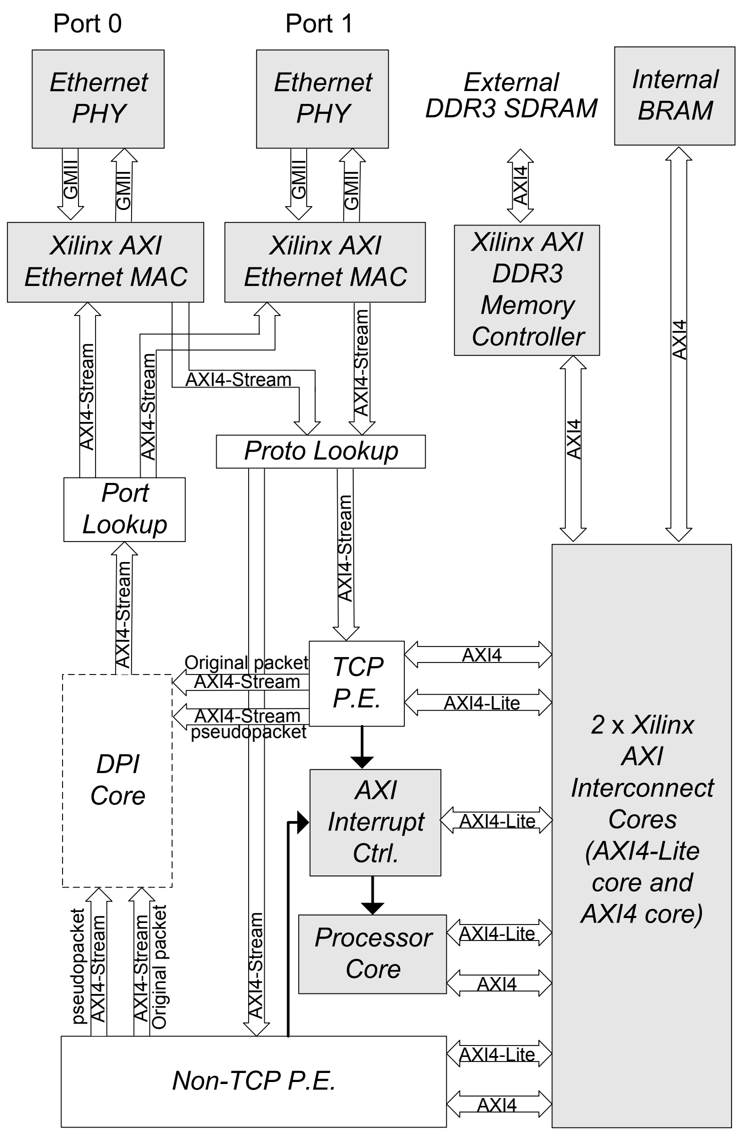


**Figure 23. TCP/IP reassembly and DPI architecture on Xilinx FPGA**

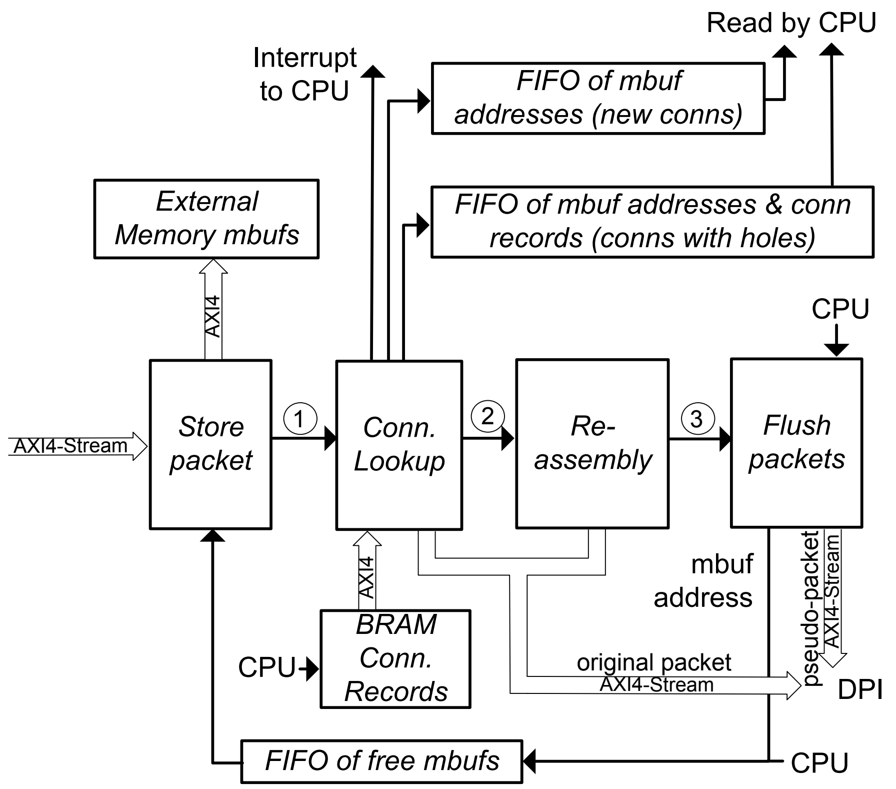
IP header and TCP/UDP checksums are validated by the Xilinx AXI Ethernet core and invalid packets are dropped. The Proto Lookup module is responsible for differentiating between TCP and non-TCP traffic which are streamed to two separate cores for processing. Each packet is prepended with an Ethernet port number field which is later used by the Port Lookup core to route the packet to the correct output Ethernet port. The TCP Processing Engine (PE) core carries out combined reassembly of all IP fragments and TCP segments into pseudo-packets which are then streamed to the DPI core. The Non-TCP PE handles all non-TCP traffic, reassembling any IP fragments into pseudo-packets. All the original packets, provided they are not dropped by the reassembly system, are also streamed to the DPI core.

In Snort, if the pseudo-packet generated by the Frag3 module contains TCP segments, it will be injected into the Stream5 module for TCP segment reassembly. It is therefore possible that the same payload data may be present in two pseudo-packets, one generated by Frag3 and one generated by Stream5, as well as in the original packet. In the hardware architecture proposed here, however, only one pseudo-packet is generated because TCP segment and IP fragment reassembly are carried out in combination in the TCP PE.

### 3.5.1. TCP Processing Engine

The operation of the TCP PE is illustrated in Figure 24. The scheme for storing packets in external memory, which is derived from the mbuf system used in the BSD Unix operating system, is shown in Figure 25. All mbufs are of the same fixed size for simplicity. Packets that are too large to fit in a single mbuf are spread over several mbufs in an mbuf *chain* using the *next* pointer. All packets for a particular connection flow are stored in an mbuf *queue* which is a list of mbuf *chains* linked using the *next_pkt* pointer. Each mbuf header also includes a *len* field which specifies the amount of data stored in that particular mbuf. The addresses of all free mbufs are stored in an FPGA FIFO primitive which is initialized by software running on the CPU. Buffers are allocated by the TCP PE on receipt of a packet and freed once the packet is flushed to the DPI core for inspection.

The first operation of the TCP PE is to store each received IP packet in an mbuf chain in external memory. The packet’s IP addresses and TCP port numbers are then used to look up the matching connection record in Block RAM which is then updated. The Reassembly module passes the mbuf queue to the Flush module once the queue’s footprint has exceeded the configured flush point. Once an mbuf has been flushed to the AXI4 Stream, its address can be returned to the FIFO of free mbufs. Note that all the original, non-reassembled, packets are streamed to the DPI core in order that the original TCP header can be analyzed. This matches the behavior of the Snort software system. The CPU can, however, drop out-of-sequence packets under certain circumstances, which are outlined later.



1. mbuf address + IP addresses + port numbers
2. conn. record
3. mbuf address (chain of mbufs to flush)

**Figure 24. TCP Processing Engine**

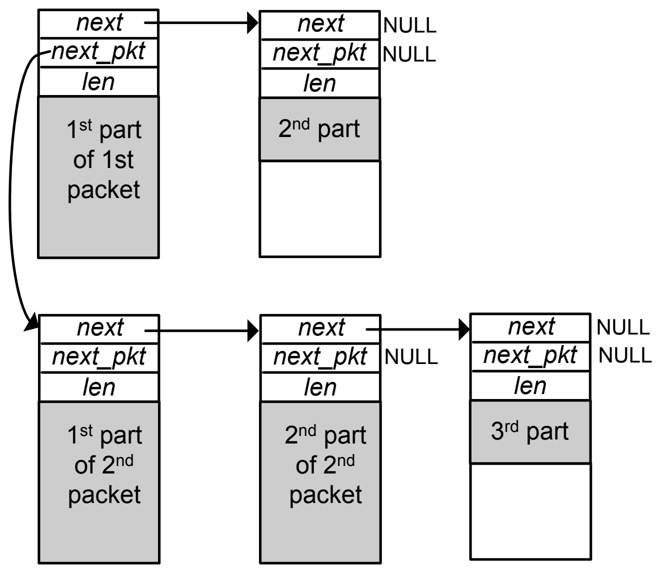


**Figure 25. External memory packet buffers**

**Connection Records**

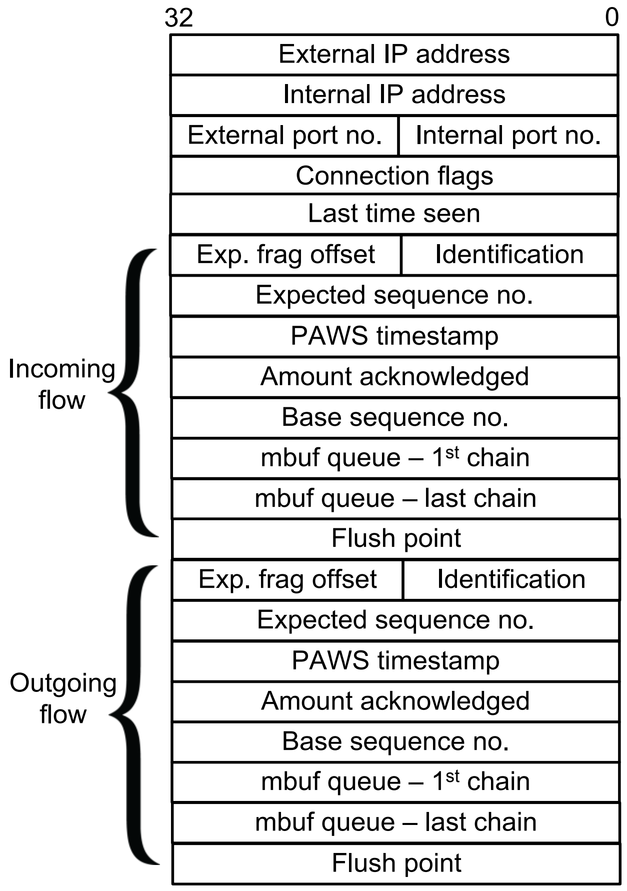


**Figure 26. TCP Connection Record**

TCP connections records, as shown in Figure 26, are stored in internal Block RAM (BRAM), which is accessible via the AXI4 interconnect. These records are created and deleted by software running on the CPU. There is a single connection record for each unique 4-tuple of the IP addresses and port numbers. As each record corresponds to a bidirectional traffic flow, there are separate sections for each direction of flow. Each section contains the IP *expected fragment offset* and TCP *expected sequence number* for the next packet received in a particular flow, the amount of data acknowledged by the receiver, and the addresses of the first and last mbuf chain in that flow's mbuf queue in external memory. Each flow's *flush point* is configured by software when the connection record is created. The *expected fragment offset* and/or *sequence number* fields are updated, as appropriate, following the receipt of each new packet on the flow. The new packet's mbuf chain is linked to the last mbuf chain in the queue, if any, and the mbuf fields in the connection record are updated appropriately. If there is an existing queue, this simply involves reading the *mbuf chain – last chain* address from the record, incrementing it to obtain the *next_pkt* location, and updating this with the address of the new packet's mbuf chain address. The *mbuf chain – last chain* field can then be updated to the same value. The *Amount acknowledged* field is updated on receipt of packets on the opposing flow. The *Base sequence number* field, which contains the TCP sequence number of the first TCP segment in the mbuf queue, allows quick recalculation of the segment queue footprint when the amount of acknowledged data increases. The *identification* field stores the last IP header's Identification value. It is used to validate that the second and subsequent fragments in a flow belong to the same reassembled packet. If a packet is received with a non-zero fragment offset and a mismatching identification, it is sent to the CPU for processing as there is a hole in the reassembled data stream.

The *Last time seen* is a hardware generated timestamp which is updated every time a packet is received on the connection. The same current time can also be read by software in order to initialize the field on creation of the connection record. The timestamp is used by software to remove connections which have been inactive for a pre-configured amount of time. The *Connection flags* field includes the following single bit flags

- `DIVERT_TO_CPU` indicates that any traffic on this connection is to be dealt with by software on the CPU
- `WAITING_FOR_FRAG_IN` and `WAITING_FOR_FRAG_OUT` indicate that the mbuf queue of the corresponding flow is waiting for an IP fragment in order to allow completion of IP defragmentation
- `CHECK_PAWS_IN` and `CHECK_PAWS_OUT` indicate whether or not PAWS (Protection Against Wrapped Sequence numbers) timestamp checking is to be performed

**Connection Lookup**

The most obvious way to perform connection lookup using the 4-tuple of the two IP addresses and two TCP port numbers is to use a Xilinx Content Addressable Memory (CAM) core. The CAM index can then be mapped to the connection record address using a simple array type table. As this 4-tuple lookup is simply a single match, fixed string lookup, it may be more efficient to use a design based on the Aho-Corasick (1975) algorithm which is suitable for FPGA implementation (Kennedy et al., 2010).

The operation of the Connection Lookup module is summarized in Figure 27. If no matching connection is found for a packet originating from the internal network then handling is handed over to the CPU in order to create a connection record. Note that in order to be robust against Denial of Service attacks such as a SYN flood, packets originating from the external network do not trigger the creation of a new record. Such packets are passed directly to the DPI core without undergoing reassembly. If, on the other hand, a matching connection is found and the packet is not a TCP SYN, SYN-ACK, FIN or RST, and it has a valid PAWS timestamp, then the connection record is passed to the Reassembly block to determine if the opposing data stream is ready to be flushed. Otherwise, or if the DIVERT_TO_CPU override flag is set, handling of the connection is passed over to software running on the CPU.

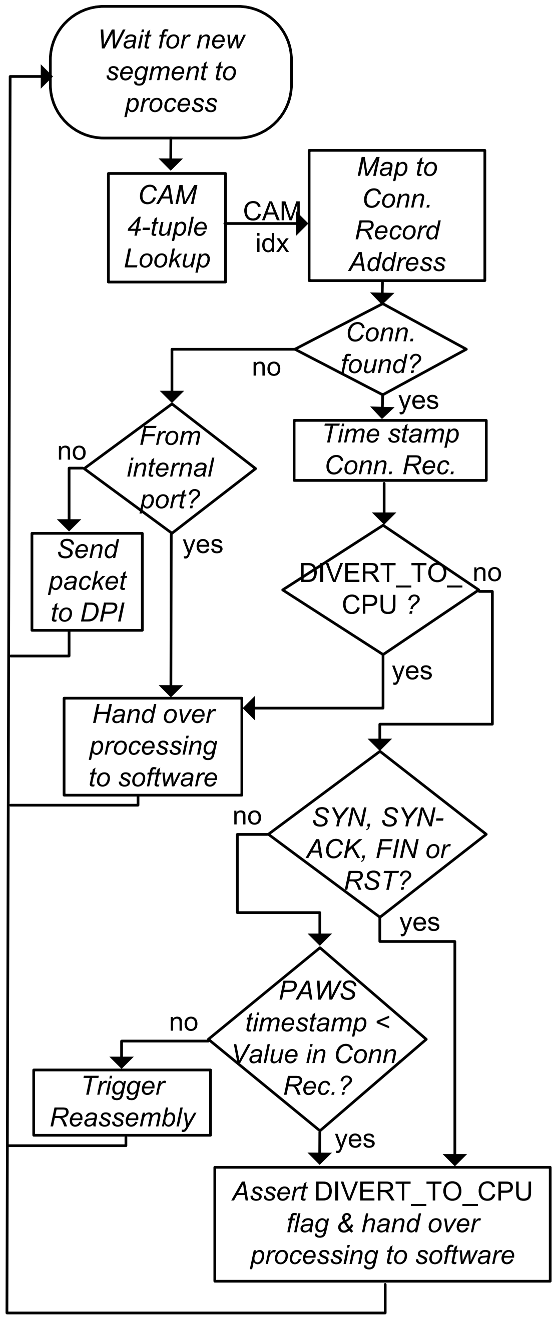

Wait for new segment to process
CAM 4-tuple Lookup
CAM idx
Map to Conn. Record Address
Conn. found?
no
yes
From internal port?
no
yes
Send packet to DPI
Time stamp Conn. Rec.
DIVERT_TO_CPU ?
no
yes
Hand over processing to software
SYN, SYN-ACK, FIN or RST?
no
yes
PAWS timestamp < Value in Conn Rec.?
no
yes
Trigger Reassembly
Assert DIVERT_TO_CPU flag & hand over processing to software

**Figure 27. Connection Lookup flow chart**

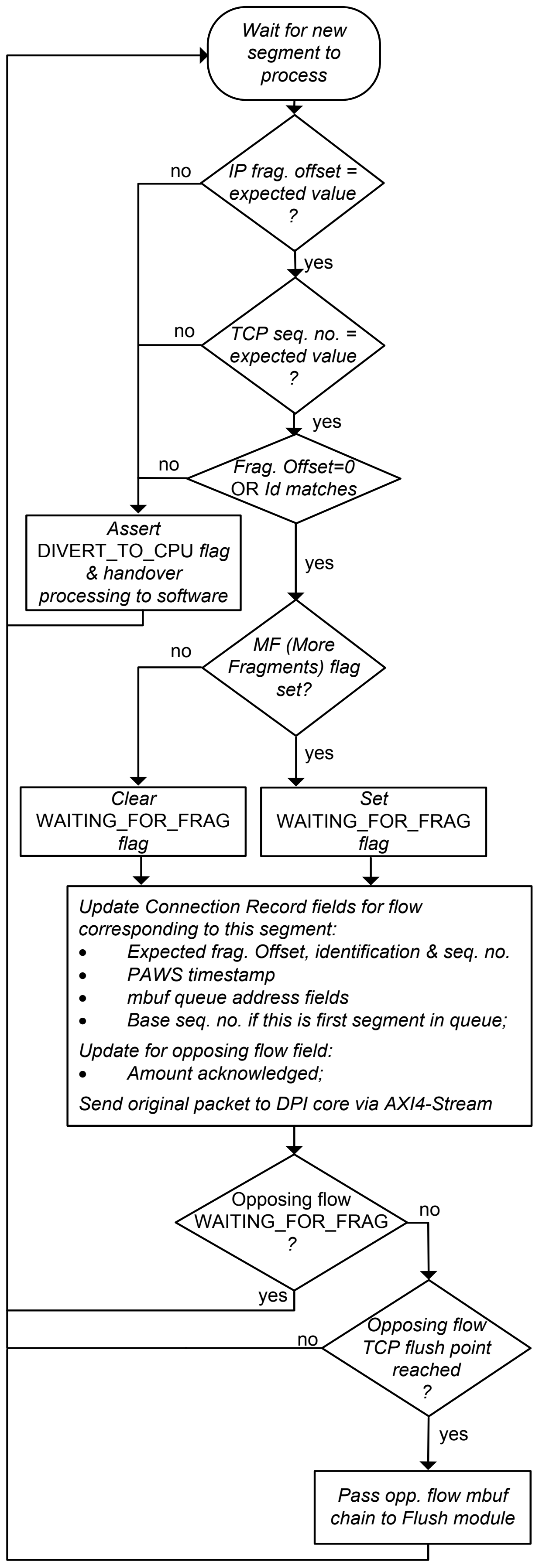

Wait for new segment to process
IP frag. offset = expected value ?
no
yes
TCP seq. no. = expected value ?
no
yes
Frag. Offset=0 OR Id matches
no
yes
Assert DIVERT_TO_CPU flag & handover processing to software
MF (More Fragments) flag set?
no
yes
Clear WAITING_FOR_FRAG flag
Set WAITING_FOR_FRAG flag
Update Connection Record fields for flow corresponding to this segment:
• Expected frag. Offset, identification & seq. no.
• PAWS timestamp
• mbuf queue address fields
• Base seq. no. if this is first segment in queue;
Update for opposing flow field:
• Amount acknowledged;
Send original packet to DPI core via AXI4-Stream
Opposing flow WAITING_FOR_FRAG ?
no
yes
Opposing flow TCP flush point reached ?
no
yes
Pass opp. flow mbuf chain to Flush module

**Figure 28. Reassembly flow chart**

**Reassembly**

The Reassembly module, the functionality of which is summarized in Figure 28, is responsible for handling the reassembly of both IP fragments and TCP segments. It first checks that the received TCP segment's IP fragment offset and sequence number match the expected value of the corresponding flow in the connection record. A mismatch indicates a special condition, such as a duplicate, overlapping fragment/segment or a hole in the data stream. When a mismatch is detected, handling of the connection is handed over to software running on the CPU. If the offset and sequence numbers are as expected, then the mbuf is inserted into the mbuf queue linked list and the connection record information is updated.

The Acknowledge number in the header of the received TCP segment may result in the increase of the *Amount acknowledged* field for the opposing flow. The opposing flow is therefore checked to see if its mbuf queue is ready to be flushed as a pseudo-packet. If no more IP fragments are required and the queue's footprint has reached the flush point, then flushing can proceed. The footprint is simply the *Amount acknowledged* minus the *Base sequence number*.

**Flushing of Packets**

The Flush module converts an mbuf queue into an IP pseudo-packet with a single TCP header which is streamed to the DPI core via the AXI4-Stream interface. It can be triggered by both the Reassembly module and the CPU.

### 3.5.2. Non-TCP Processing Engine

The Non-TCP PE is simpler than its TCP equivalent. It monitors all non-TCP traffic for IP fragments by checking the header's MF flag and fragment offset. On detection of a fragment, it uses the 4-tuple of the source & destination addresses, protocol type and identification field to look up the corresponding fragment tracker record, as illustrated in Figure 29. If no record is found, then the mbuf address is sent via a FIFO to the CPU for processing. Software running on the CPU creates the fragment tracker record and updates the TCAM or search algorithm data to enable reassembly of the fragment stream in hardware.

Fragment tracking for a particular fragment flow is handed back to the CPU on detection of a hole in the reassembled data stream.

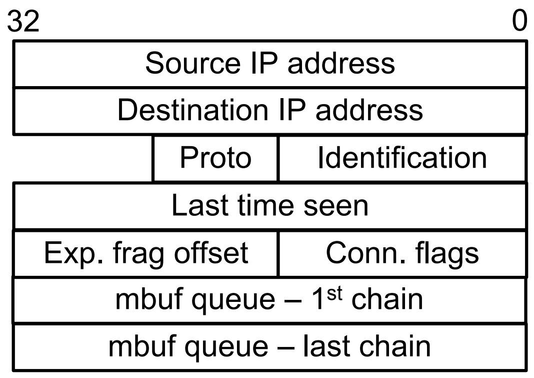


**Figure 29. Non-TCP Fragment Tracker Record**

### 3.5.3. Software on CPU

The functionality of the software running on the CPU is mainly based on that of Snort. The tasks it performs include

- TCP connection record creation based on receipt of TCP segments from the internal network. Connections can be created on receipt of SYN, SYN-ACK and data segments. The latter is known as midstream pickup in Snort Stream5. Each connection record is configured with a pseudo-random flush point.
- Non-TCP fragment tracker record creation based on receipt of an IP fragment carrying a non-TCP protocol.
- Update of the CAM or search algorithm data.
- Handles TCP RST and FIN segments, closure of connection and deletion of connection record.
- Target-based handling of segments/fragments that are out of order, duplicates or overlapping, or have unexpected PAWS timestamps.
- Timing out of inactive TCP connection & IP fragment records and, freeing up of associated memory.

### 3.5.4. Race condition

A race condition can occur if software on the CPU creates a new connection or fragment tracker record and disables the DIVERT_TO_CPU flag while there are other packets from the same connection flow in the mbuf FIFO awaiting processing by the CPU. If a new packet were then received by the hardware PE, it would incorrectly detect a hole in the data stream and therefore enable the DIVERT_TO_CPU flag. The likelihood of this condition can be minimized by having the software only disable the flags of all the newly created records once it has emptied the FIFO.

### 3.5.5. Handling high processor load

If the load on the CPU becomes too high, then out-of-order TCP segments will be dropped, thereby forcing the source to resend. It will also drop out-of-order segments from a connection if the corresponding mbuf queue becomes excessively long.

The system is robust to a TCP SYN flood DoS attack because connection records are only created based on traffic from the internal network. However, the creation of IP fragment tracker records based on the receipt of fragments from the external network makes the system vulnerable to IP fragment DoS attacks. Such a scenario is unlikely if the incoming traffic has already been firewalled. Nevertheless, a limit must be imposed on the number of mbufs used for reassembly of IP fragments received from the external network.

### 3.5.6. Evaluation and comparison with related work

Given that all packets are stored in external memory, the memory architecture is critical to achieving high traffic throughput. The Zynq-7100's integrated DDR memory controller can be configured to provide 16-bit or 32-bit wide access at up to 1333Mb/s per pin in the case of DDR3 (Xilinx, 2013), giving a total maximum memory throughput of 42.66Gb/s if 32-bit wide access is used. The controller is multi-ported, allowing shared access to the memory from the CPU and the Programmable Logic (PL). It has four 64-bit AXI slave ports – one dedicated to the CPU, two dedicated to the PL, and one accessed via the AXI4 Interconnect. These 64-bit ports can operate at up to half the maximum DDR3 frequency, i.e. 666MHz. Assuming all traffic is written to, and read from, SDRAM once, the maximum theoretical traffic throughput is 21.33 Gb/s. If the system has two Ethernet ports, then the maximum port throughput is 10.5 Gb/s, making it feasible to use 10G ports. This theoretical maximum throughput is dependent on the PL design achieving timing closure through the use of pipelining, and maximum use being made of the available memory bandwidth. The Zynq-7100 device has approximately 3MB of internal memory, sufficient space for almost 36,000 TCP connection records. However, in practice some of this memory will be required for other purposes such as FIFOs.

Related work includes the research done by Necker et al. (2002), who describe how TCP/IP reassembly and tracking can be performed on an FPGA as part of an NIDS. Their design, which uses the Xilinx Virtex 2000E, can process a single TCP flow at 3.2Gb/s and is capable of being extended to handle up to 30 flows. Schuehler and Lockwood (2002,2004) describe a high performance TCP flow monitor called TCP-Processor which is capable of handling 8 million bidirectional TCP flows at 2.5Gb/s. Rather than perform packet reordering, their design simply drops out of order packets, thereby forcing the source to resend. This simplifies the design and makes it more robust against attacks, but increases the amount of network traffic. An alternative design involves buffering the packets of a particular connection until a missing packet arrives and the "hole" is filled. Only then can the payload data be streamed to the DPI engine for inspection. However this buffering makes the system vulnerable to an attack in which an adversary tries to overflow the buffer by injecting a high number of packets on a particular connection following a hole and/or by opening a very large number of connections containing holes. Dharmapurikar and Paxon (2005) describe an architecture which performs reordering but is robust in the face of these attacks. They observed that the most common case of out-of-order packets is a single hole in one direction of the connection. Their design limits each connection to a single hole and applies a limit to the buffer usage of each connection. In order to resist an adversary who opens multiple connections with holes, it limits the number of connections with holes to just one per client. The design also includes a randomised eviction policy to be used when the buffer reaches capacity. The issue of insertion and evasion attacks is avoided by normalising the traffic in order to remove any ambiguities.

The architecture described in this thesis handles the most common case, of TCP connections without any holes in the reassembled stream, directly in hardware without any software intervention. In normal cases, only TCP connection setup and tear-down are handled in the CPU. Once any special case occurs, such as an out-of-order or duplicate segment, then handling is passed to the CPU. Because both the external SDRAM and internal BRAM are shared between the PL (Programmable Logic) and CPU, only a reference to the TCP connection record and to the latest packet buffer needs to be passed to the CPU to instigate the handover. If the CPU becomes overloaded due to an excessive number of TCP connections with holes, then the hardware can be configured to directly drop any out-of-order segments, i.e. revert to the Schuehler & Lockwood proposal. The architecture, unlike previous proposals, also handles IP fragment reassembly in hardware.

Existing research confirms that out-of-order packets are relatively rare. Based on the analysis of traffic traces from the Sprint backbone network in 2002, Jaiswal et al. (2007) found that about 4% of packets on TCP connections were out-of-sequence. Studies by Murray & Koziniec (2012) and by Zhao et al. (2012) found that the proportion of out-of-sequence packets was well under 1%. These figures back up the design decision to handle streams with out-of-sequence packets in the slow path.

## 3.6. Conclusion and Future Work

This chapter explains why TCP/IP reassembly is a necessary prerequisite to accurate DPI in NIDPS systems. A description is given of the theory of the reassembly of IP fragments and TCP segments and of how this is implemented in the open source NIDPS Snort and Suricata. Passive NIDS systems need to perform target-based reassembly in order to detect insertion and evasion attacks. Inline NIPS systems can, on the other hand, normalise the traffic in order to remove any ambiguities. Implementing TCP/IP reassembly in hardware is a challenging task due to decision-making required to reorder packets, avoid buffer overflow in the face of DoS attacks, and track the states of a potentially huge number of connections.

The proposed hardware-based reassembly system takes advantage of the fact that out-of-sequence packets are rare under normal circumstances by carrying out target-based reassembly of the affected streams in software while dealing with the normal in-sequence streams directly in FPGA programmable logic. The proposed system is capable of handling the traffic from two 10G Ethernet ports and up to 36,000 concurrent TCP connections. The latter number could be increased by using an FPGA device with a larger amount of internal memory. The main contribution of this research work is that it improves on existing schemes by dealing with out-of-order and overlapping IP fragments and TCP segments, avoiding the need to drop packets in order to force the originating host to resend.

The proposed system uses footprint based flushing, the default setting in Snort. Further research is required to determine if the PAF feature is suitable for hardware implementation. The recently proposed TCP Fast Open (TFO) extension (Cheng et al., 2013) also needs to be looked at. The current TCP specification allows clients to include data in the SYN segment but in practice only the Mac OS has accepted data in a SYN segment (Novak and Sturges, 2007). TFP, however, allows SYN segments to contain data in addition to allowing servers to send data to the client before completion of the 3-way handshake. The design proposed in this thesis sends such SYN+data segments, received from the external network, to the DPI core without performing reassembly as the connection record is only created on receiving the first segment from the internal network. Full reassembly of TFO data streams is for future consideration.

Other future work includes investigating the possibility of handling most out-of-sequence packets in hardware, with only the ambiguities caused by duplicate and overlapping packets being dealt with in software. The design also needs to be extended to support IPv6 in addition to IPv4.

# Chapter 4 - Multi-match Header Classification

The most commonly proposed solutions to multi-match classification are TCAM-based and, as a result, suffer from several disadvantages such as higher cost, high energy consumption, and low storage efficiency. This thesis examines alternative algorithmic solutions that can use SRAM in place of TCAM. A number of well-known single-match packet classification algorithms were adapted and their multi-match classification performance compared in terms of memory requirements, energy consumption and packet processing speed. These were then compared with two existing multi-match solutions.

NIDPS designs are generally split into a header-based multi-match classification stage and a payload-based Deep Packet Inspection (DPI) stage. These stages can be in parallel or in series. In the multi-match classification stage, the header of the incoming IP packet is compared against the 5-tuple header section of all rules. In a series architecture the payload sections of all the resulting matching rules are then compared against the packet payload in the follow-up DPI stage.

Multi-match classification differs from the single-match classification used, for example, in routers, in that it must return all matching rules as opposed to just the single highest priority match.

## 4.1. Characteristics of NIDS Rule Sets

**Table 8: Example Snort rule**

| | |
|---|---|
| *Header* | alert tcp $EXTERNAL_NET 6666:7000 -> $HOME_NET any |
| *Options* | content:"\|3A\|"; offset:0; content:" 302 "; content:"=+"; metadata:policy security-ips drop; classtype:policy-violation; sid:1790; rev:5; |

Snort rules are made up of two parts, the rule *header* and the rule *options*, as shown in the example rule in Table 8. Only the rule header is relevant to multi-match header classification. The two most important Snort variables are *$EXTERNAL_NET* and *$HOME_NET*. *$HOME_NET* has a default value of *any* and can optionally be set to the network or networks being protected. In this analysis it is assumed to be a single subnet. *$EXTERNAL_NET* is typically left at its default value of *any* or can be set to *!$HOME_NET* if *$HOME_NET* is not set to its default value of *any*. In this analysis it is assumed that *$EXTERNAL_NET* is set to its default value. The example rule header of Table 8 specifies a protocol type of "*any*" (i.e. wildcard), source IP address of

*$EXTERNAL_NET*, a source port in the range 6666 to 7000, a destination IP address of *$HOME_NET* and a destination port of "*any*" (i.e. wildcard).

As shown in Table 9, the Snort rule set downloaded from www.snort.org, in January 2010, contains a total of 8454 rules. A combination of Linux shell scripts and software implemented in C code was used to analyse the rule set in order to extract the information given in this table. It was found through this analysis that there are only 743 unique rule *headers*. Snort rules can specify port fields in a number of ways, including lists of individual ports, negations, etc. The classification algorithms evaluated require the fields to be specified as either single ranges or single prefixes and therefore the Snort rules need to be converted into this format. This conversion results in an increase in the number of unique rule headers to 797 in the case of port ranges and 1273 in the case of prefixes.

**Table 9: Statistical information for Snort 2.8 rule set (January 2010)**

| | Unprocessed rule set | After expansion of port fields to single ranges | After expansion of port ranges to prefixes |
|---|---|---|---|
| Total number of rules | 8454 | – | – |
| No. unique header 5-tuple combinations | 743 | 797 | 1273 |
| No. unique Source addresses | 8 | 8 | 8 |
| No. unique Destination addresses | 12 | 12 | 12 |
| No. unique Source ports | 216 | 225 | 285 |
| No. unique Destination ports | 420 | 412 | 508 |
| No. unique Protocol types | 4 | 4 | 4 |
| No. unique Address pair combinations | – | 21 | 21 |
| No. unique Port pair combinations | – | 621 | 1028 |

On examination of the compressed set of 1273 rules (with ports expressed as prefixes), it is observed that there are only 8 unique source address prefixes, 12 unique destination address prefixes and 21 unique combinations of these two fields. The statistics given in Table 9 have a significant impact on the performance of the algorithms evaluated in this chapter. The relatively low numbers of unique IP addresses, for example, means that only a very small TCAM would suffice for IP address matching. The low numbers of unique ports relative to the overall number of rules means that a compressed bit vector can be used to represent the rules for each field, thereby reducing the amount of memory required. The relative number of unique ports versus addresses also influences the assignment of these fields to dimensions in multi-dimensional algorithms such as EGT-PC discussed in section 4.3.3.

## 4.2. Proposed Architecture

### 4.2.1. Pre-processing

The Snort rule set first needs to be compressed into a set which only contains unique 5-tuple header combinations. If the original rule set contains rules numbered *i*=1..*N*, then each of the rules in the compressed set will have an associated list of one or more original rule numbers with values in the range 1 to *N*. The mapping from compressed rule to original rules and the retrieval of the associated rule option are performed in the follow-up stage of the proposed scheme as shown in Figure 30.

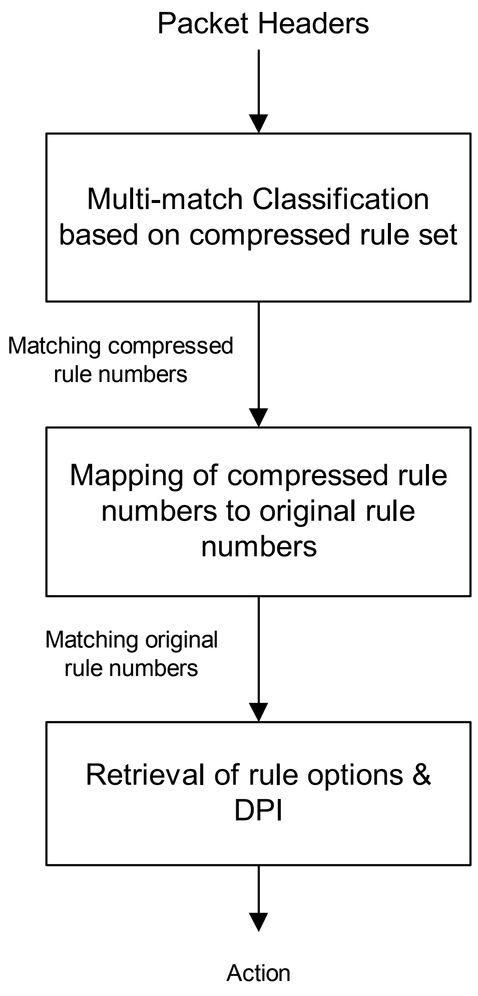

Packet Headers
Multi-match Classification based on compressed rule set
Matching compressed rule numbers
Mapping of compressed rule numbers to original rule numbers
Matching original rule numbers
Retrieval of rule options & DPI
Action

**Figure 30: Overall proposed NIDS Scheme**

### 4.2.2. Top-level Architecture

This thesis looks into possible solutions for the multi-match classifier, i.e. the first block of the scheme shown in Figure 30. This block takes the IP packet header as input and outputs a bit vector of length equal to the number of compressed rules, i.e. 1273 bits in the case of the Snort rule set used.

## 4.3. Algorithms

### 4.3.1. Introduction

Ternary Content Addressable Memory (TCAM) provides the most straightforward hardware solution to single-match packet classification. However, its high cost and high power consumption led to extensive research into several alternative algorithmic solutions (Taylor, 2005), including the decision tree–based algorithms Hypercuts (Singh et al., 2003), Extended Grid of Tries with Path Compression (EGT-PC) (Baboescu et al., 2003; Srinivasan et al., 1998), and Allotment Routing Table (ART) (Hariguchi, 2002), each of which is analysed in this thesis. Each algorithm was implemented in C code and adapted for multi-match classification. Existing open source single-match software was reused where possible.

Evaluation of software implementations was carried out using sample packet header sets generated using ClassBench (Hoffman and Strooper, 1997) based on the compressed Snort rule set.

Estimates of energy consumption per packet, as presented later in Table 16, were obtained by simulating the software implementation on a StrongARM SA-1100 using Sim-Panalyzer (Mudge et al., 2004). The following typical configuration parameters were used:

- clock frequency 200 MHz
- 5% clock skew
- voltage 1.8 V
- 16 Kbyte instruction cache
- 8 Kbyte data cache
- 180 nm process technology

The software was built with compiler optimization enabled.

In order to fairly compare the energy efficiency of the software and hardware implementations, the power figures were normalised according to the approach outlined by Kinane (2006). The power $P$ in a process $L$ with voltage $V$ can be normalized to a reference process $L'$ with voltage $V'$ using the following formula:

$$P' = P \times \left( \frac{L'}{L} \right)^2 \times \frac{V'}{V}$$

All power figures given in this chapter are for a device using 65nm technology with a core voltage of 1V, i.e. $L' = 65$nm and $V' = 1$.

### 4.3.2. Hypercuts

Hypercuts, a well-known single-match algorithm, was the first to be evaluated using existing C and VHDL single-match implementations from the author's MEng research work as the starting point. It has the advantage that it needs little change for use in multi-match classification other than modifying it to return all matches from the linear search of a leaf node's rule list. Unfortunately, Hypercuts was quickly ruled out as a possible multi-match solution because of two serious disadvantages that quickly became apparent:

- The high degree of overlap in the rule set results in a memory explosion.
- Leaf nodes generally contain a large number of rules resulting in a lengthy linear search.

### 4.3.3. EGT-PC

The EGT-PC (Baboescu et al., 2003; Srinivasan, 1998) algorithm is based on a structure called the *grid-of-tries*.

#### Basic Grid-of-Tries

A trie is basically a binary search tree where each branch leaving a node is labelled with 0 or 1. The prefix corresponding to a particular leaf node is the concatenation of all of the bits encountered on travelling from the root to that leaf node. A grid-of-tries is used for two dimensional (i.e. two field) matching. For each two dimensional rule, nodes in the first dimension trie have a pointer to the root of a corresponding second dimension trie. Nodes in the 2nd dimension trie contain a list of rules corresponding to the node.

**Table 10: Example Rule Set**

| *1st Dimension* | *2nd Dimension* | *Rule* |
|---|---|---|
| 0111* | 000* | R1 |
| 0111* | 00* | R2 |
| 0111* | 1* | R3 |
| 0* | 001* | R4 |
| 0* | 00* | R5 |
| 0* | * | R6 |
| * | * | R7 |

An example 2-dimensional rule set and the corresponding basic grid-of-tries are shown in Table 10 and Figure 31, respectively. This rule set results in three second dimension tries in addition to the usual single first dimension trie. Each successive second dimension trie corresponds to a shorter first dimension prefix. The 1st second dimension trie corresponds to the first dimension prefix 0111, the 2nd to 0* and the third to *. The search algorithm involves finding the longest prefix match in the 1st dimension and then using a pointer stored at that node to jump to the 2nd dimension trie where the matching rules are found at each node traversed until the longest matching prefix is found. e.g. (0000, 0001) would match "0" in the 1st dimension, the matching node of which is linked to the second 2nd dimension trie. "00" would match in that trie, giving R5, R6, R7 as the matching rules. In single match packet classification, each rule has a cost, and the single matching rule is that which has the lowest cost.

The basic grid-of-tries suffers from a memory blowup problem because each 2nd dimension trie must include the 2nd dimension prefixes which correspond to shorter 1st dimension prefixes, e.g., the branch and rule corresponding to rule R4 in the second 2nd dimension trie must be replicated in the first 2nd dimension trie.

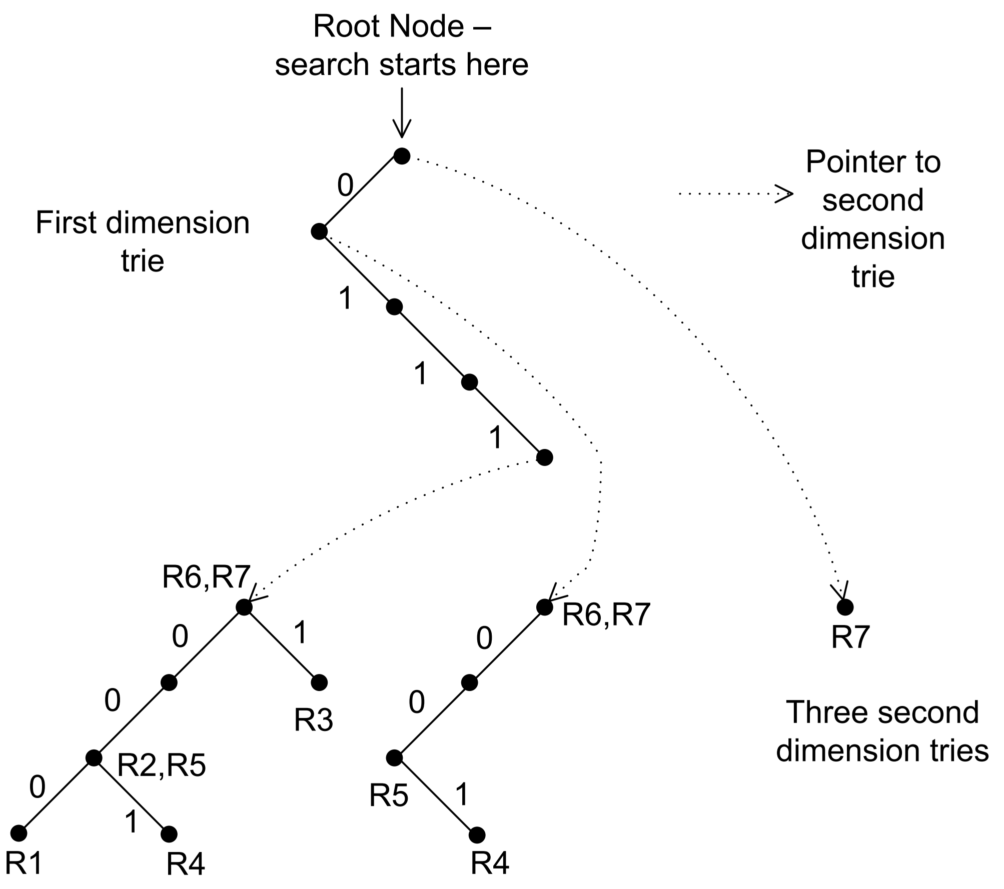


**Figure 31. Example basic Grid-of-Tries**

**Backtracking**

The memory blowup issue can be avoided if each 1st dimension node points to a 2nd dimension node which only includes rules with a 1st dimension field which exactly matches the prefix of the 1st dimension node. The grid-of-tries for the example rule set in Table 10 is shown in Figure 32. The search algorithm is modified to search all 2nd dimension tries associated with the matching 1st dimension node and all its ancestor nodes, i.e. backtracking is used. e.g. (0111, 0010) matches “0111” in the 1st dimension trie and “00” in the corresponding 2nd dimension trie, giving R2 as a match; but “0” in the 1st dimension is also a match, with the corresponding 2nd dimension trie giving R6, R5 and R4 as matches; similarly, R7 is a match in the third trie. Therefore R2, R4, R5, R6, R7 are all matches. Although the backtracking system saves storage, it requires more time to search multiple second dimension tries.

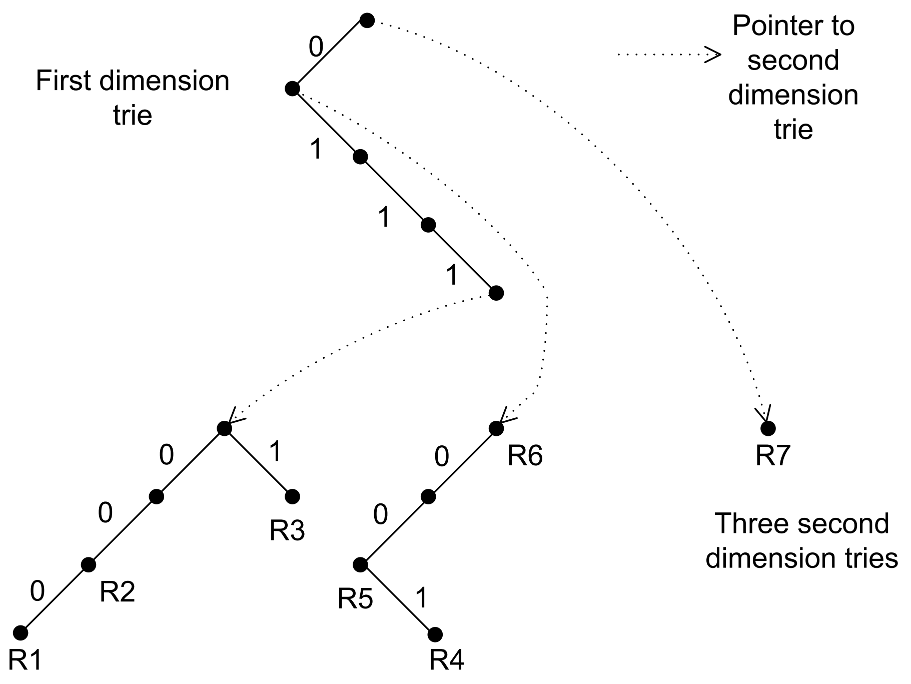

First dimension trie
Pointer to second dimension trie
0
1
1
1
0
1
0
0
R2
R1
R3
0
R6
0
R5
1
R4
R7
Three second dimension tries

**Figure 32. Example basic Grid-of-Tries with backtracking**

**Switch Pointers**

Srinivasan et al. (1998) describe how the need for backtracking can be eliminated, and thereby the search time improved, through the use of *switch pointers.* If the input is ( 0111, 0010) in the example shown in Figure 33, then matching will fail at node *x* and the switch pointer will transfer the search directly to node *y*, avoiding the need to search the entire second 2nd dimension trie from its root. The search will therefore return R2 and R4 as the matching rules. R5, R6 and R7, which could potentially be lower cost matches, are missed. Srinivasan et al. (1998) solve this problem by maintaining a variable, *storedFilter*, in each node of the 2nd dimension trie. Each node *v* with 1st dimension prefix $P_1$ and 2nd dimension prefix $P_2$ has a variable *storedFilter*(*v*) that stores the lowest cost rule having a 1st dimension field which is a prefix of $P_1$ and a 2nd dimension field which is a prefix of $P_2$.

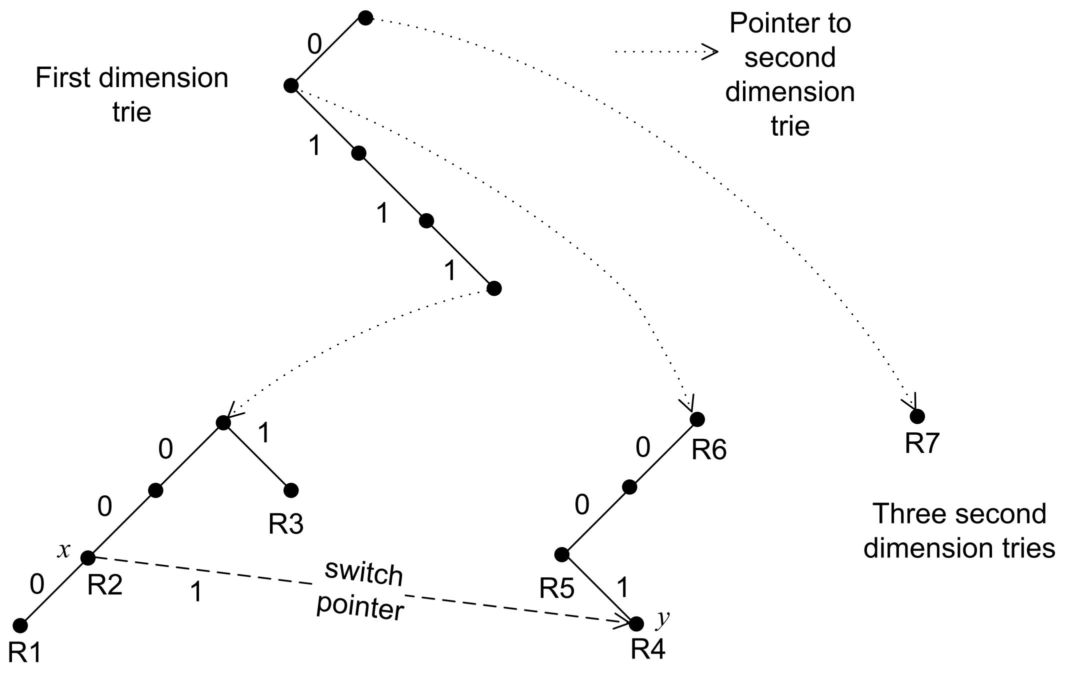


**Figure 33. Example basic Grid-of-Tries with Switch pointers**

**Extended Grid-of-Tries (EGT)**

Baboescu et al. (2003) describe how 2D search schemes can be extended to handle 5-tuple packet classification by simply modifying the 2D scheme to return all rules (and not just the lowest cost rule) which match the IP source and destination address fields and then using a linear search to find which of those rules match the protocol type and port number fields. Their Extended Grid of Tries (EGT) algorithm uses *jump pointers* instead of *switch pointers*, as illustrated in Figure 34. Jump pointers are designed so that all matching rules are found and not just the lowest cost rule. If the node containing the jump pointer is associated with prefix $P_1$, then the jump is to a node, in another second dimension trie, which has an associated prefix that is the longest matching prefix of $P_1$ and contains at least one rule. Each node, with an associated rule, that be reached directly by a jump pointer, also includes a link to its closest ancestor node that has associated rules. This is to ensure that all matching rules can be efficiently retrieved when a matching node is arrived at via a jump pointer.

If the input is (0111, 0010) in the example shown, then the matching will fail at node L2 (rule R2 matches) and a jump is made to L5 (rule R5 matches, link to L6 gives R6 as a match). This trie is then traversed to node L4 (rule R4 matches), followed by jump to L7 (R7 matches). The final list of all matching rules is then R2, R4, R5, R6 and R7.

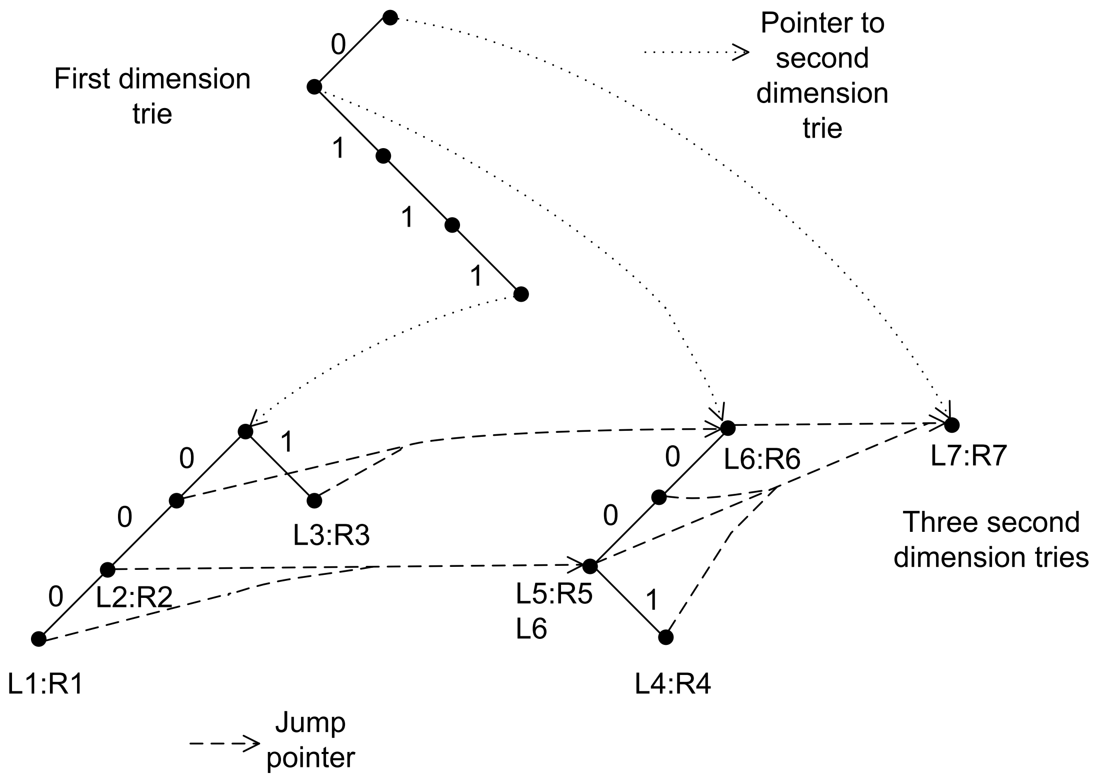


**Figure 34. Example EGT using jump pointers**

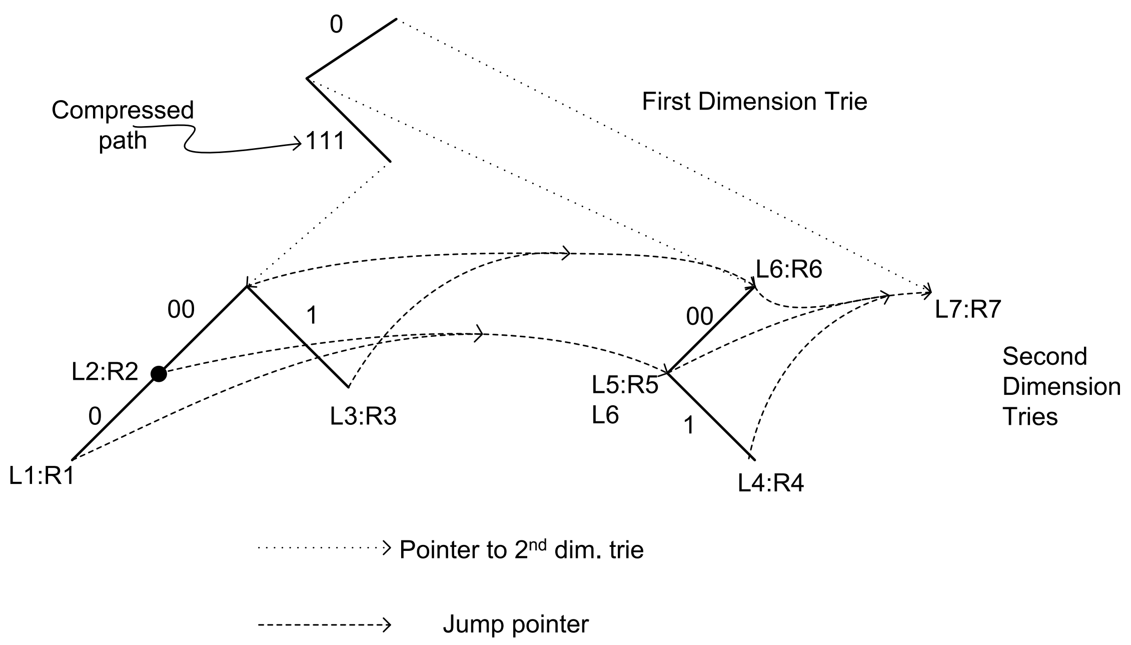


**Figure 35: Example EGT with path compression**

### EGT with Path Compression (EGT-PC)

A further improvement in efficiency is obtained using path compression, first proposed by Morrison (1968) in the Patricia tree structure. This eliminates nodes with single branches and no associated rule list. The use of jump pointers combined with path compression is illustrated in Figure 35.

### EGT-PC – multi-match

A grid-of-tries is suitable for performing multi-match classification because the single-match algorithm, in its basic format, finds all matches before performing a linear search to find the highest priority one. EGT-PC jump pointers could however, depending on the implementation, result in some matching rules being missed. This issue can be dealt with as shown in Figure 35 by having each node, with an associated rule, that be reached directly by a jump pointer, also include a link to its closest ancestor node that has associated rules. This is recursive in that each ancestor node may also, along with its own list of rules, include a link to an ancestor.

A simpler and usually faster grid implementation is not to use links to ancestor nodes, but for each node that can be reached by jump pointers to include the rules that apply to its ancestors in its own list directly. For example, node L5 in includes Figure 36 both rules R5 and R6.

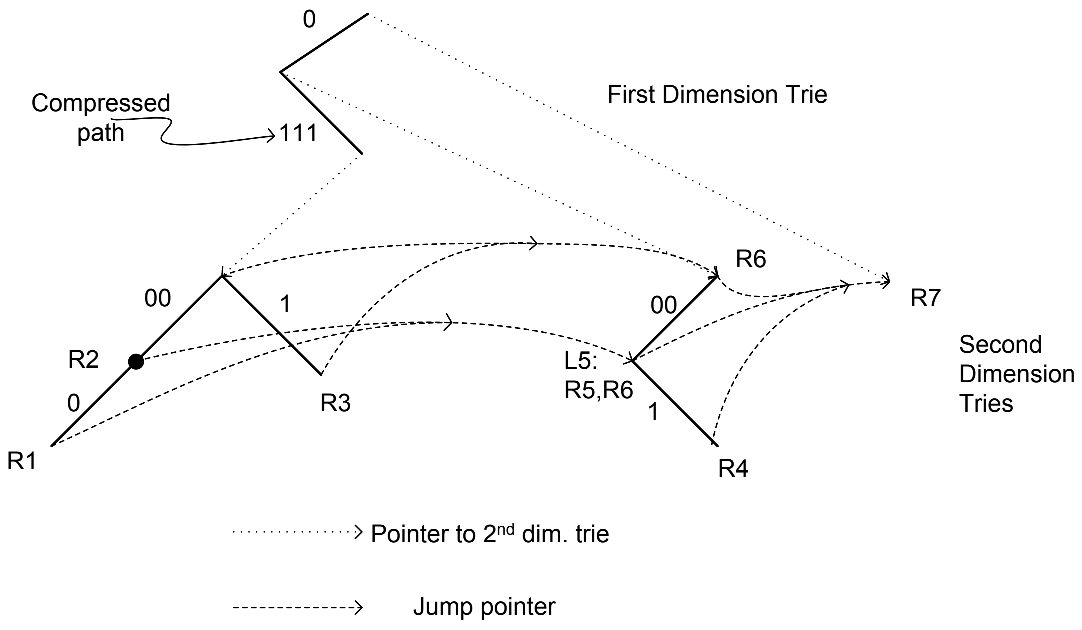


**Figure 36. Alternative EGT-PC implementation**

The architecture used for evaluating the EGT-PC algorithm for multi-match header classification consisted of three blocks running in parallel, as illustrated in Figure 37.

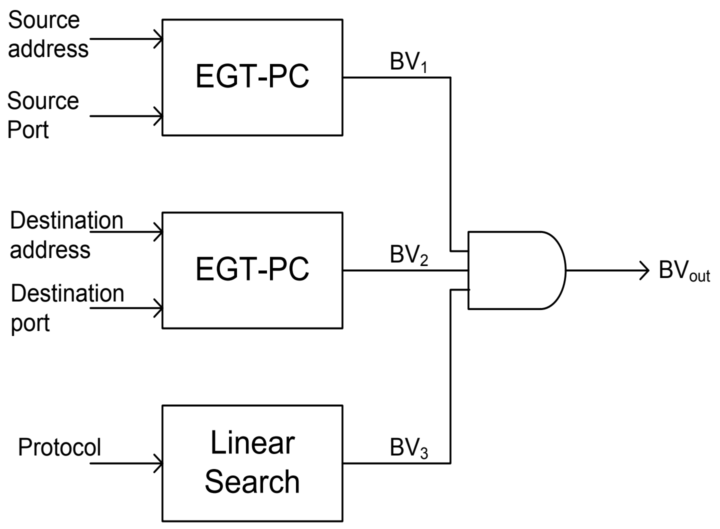


**Figure 37: EGT-PC Multi-match Architecture**

Two instances of the EGT-PC algorithm are used to classify the source address and port pair, and the destination address and port pair, respectively. Each EGT-PC block's IP address is mapped to the 1st dimension trie and the port number to the 2nd dimension trie. Note that all other pairings of address and port number were tested but those shown in Figure 37 were found to perform best in terms of both speed and required memory. This is due to the relatively large number of unique port numbers compared to the number of unique IP addresses. A linear lookup is performed on the very small number of protocol types. Each block produces a bit vector with length equal to the number of compressed rules (1273 bits in the case analysed in Table 9). The three bit vectors are then ANDed together to give the overall output bit vector.

**Table 11. Example rule set with port ranges**

| *Source IP address* | *Source port number* | *Destination IP address* | *Destination port number* | *Protocol* | *Rule No.* |
|---|---|---|---|---|---|
| 10.0.0.0/24 | 3904:3919 | 52.1.1.0/24 | 992 | TCP | 1 |
| any | 3904 | any | 992:1007 | TCP | 2 |
| 10.0.0.0/16 | any | 52.1.1.4 | any | TCP | 3 |
| 10.0.0.0/16 | 3904 | any | any | TCP | 4 |

Consider the example rule set shown in Table 11. Note that each rule would also have an associated payload signature (fixed string and/or regex) which has been omitted for clarity. This rule set cannot be compressed any further as each header part is unique. The first step in generating the EGT-PC is to convert any port number ranges to prefixes, which can sometimes lead to rules having to be split into several rules, each corresponding to a particular sub-range. In this simple example, there is a one-to-one mapping from range to prefix, resulting in the rule set shown in Table 12.

**Table 12. Example rule set with port prefixes**

| *Source IP address* | *Source port number* | *Destination IP address* | *Destination port number* | *Protocol* | *Rule No.* |
|---|---|---|---|---|---|
| 0x0A0000* | 0x0F4* | 0x340101* | 0x03E0 | TCP | 1 |
| * | 0x0F40 | * | 0x03E* | TCP | 2 |
| 0x0A00* | * | 0x34010104 | * | TCP | 3 |
| 0x0A00* | 0x0F40 | * | * | TCP | 4 |

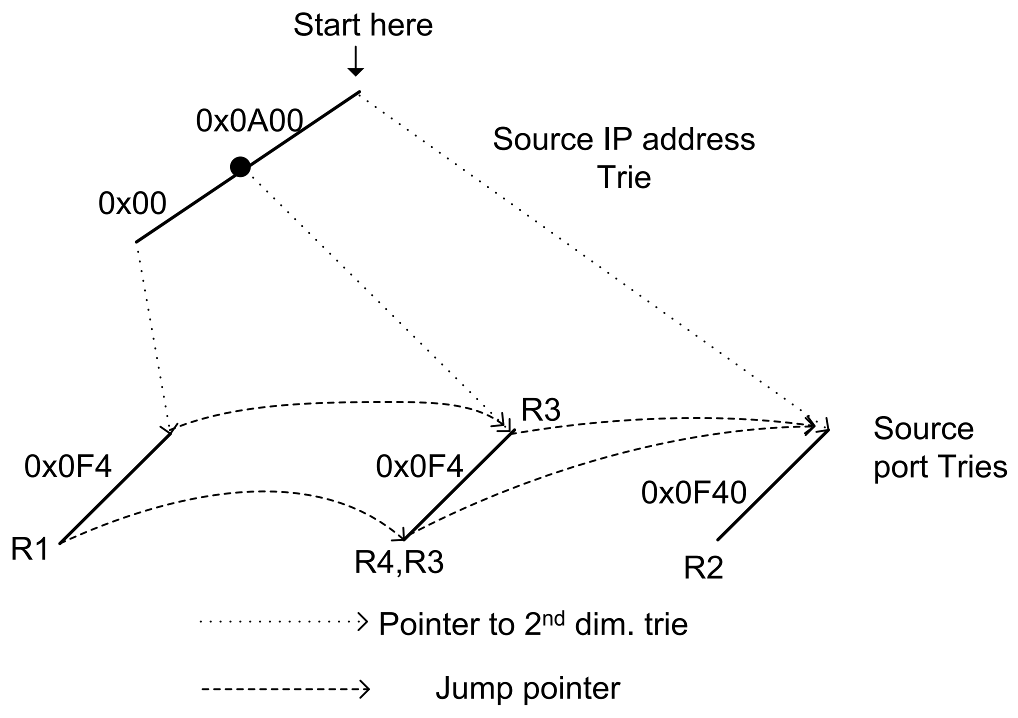


**Figure 38. Source IP address & port number trie for example rule set**

The EGT-PC for the combination of source IP address and port number is shown in Figure 38. A similar EGT-PC would be constructed for the destination IP address and port number. Each EGT-PC implementation indicates the matching rules by asserting the corresponding bits in a rule number bit vector – there are 4 rules in this example, so the bit vector would consist of 4 bits. The bit vectors from the two EGT-PC and the linear protocol lookup are then combined as shown in Figure 37 to produce the overall bit vector.

The evaluation was performed by adapting a single-match implementation originally available from the Packet Classification Repository (Singh and Baboescu, 2002). This implementation consists of single-bit first and second dimension tries with path compression. It does not use multi-bit tries which would give improved performance.

### 4.3.4. ART (Allotment Routing Table)

ART is a multi-bit trie–based routing table invented by Donald Knuth (Hariguchi, 2002). The free single-match software implemented by Y. Hariguchi and D. Knuth was adapted to perform multi-match classification.

**Description of single match ART algorithm**

The ART algorithm is based on the use of a table which can match against a configurable number of bits known as the address length. This table is shown in the form of a binary tree in Figure 39 for an address length of three. The number in brackets is known as the base index and is simply the table array index.

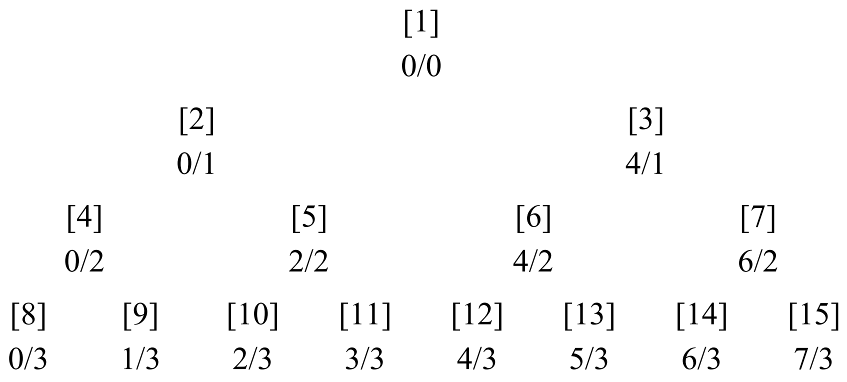


**Figure 39. All 3-bit prefixes mapped into complete binary tree**

The bottom row of the table contains what are known as the fringe nodes. These are the nodes which are accessed by the search algorithm using the simple formula (*input* + $2^{len}$) or equivalently ( 1<<*len* + *input* ), where *input* is the 3-bit input to look up and *len* is the ART address length. Non-fringe nodes are used during rule deletion, as the rule being deleted from node must be replaced by the rule specified in the parent node.

Large address lengths results in excessively large tables and so the address is typically split into multiple short addresses called strides. In multi-level ART, each fringe node contains a pointer to a table in the next level. This pointer will be NULL if there is no rule with a longer prefix than the one corresponding to the fringe node. The address length of each level of the multi-level ART algorithm is chosen to minimise the number of tables created from the particular rule set.

**Extending ART to perform multi-matching**

In order to perform multi-match lookups, each fringe node may correspond to multiple rules and it must also include all rules that apply to its ancestor nodes. Non-fringe nodes are therefore superfluous and may be omitted. An example multi-level ART is shown in Figure 40 for the simple IP address field rule list of Table 13.

**Table 13. Simple example rule list**

| *Address* | *Rule* |
|---|---|
| 9.0.0.0/16 | R1 |
| 9.0.1.0/24 | R2 |
| 9.0.2.0/24 | R3 |
| 9.0.2.4/31 | R4 |
| 9.0.2.4/32 | R5 |

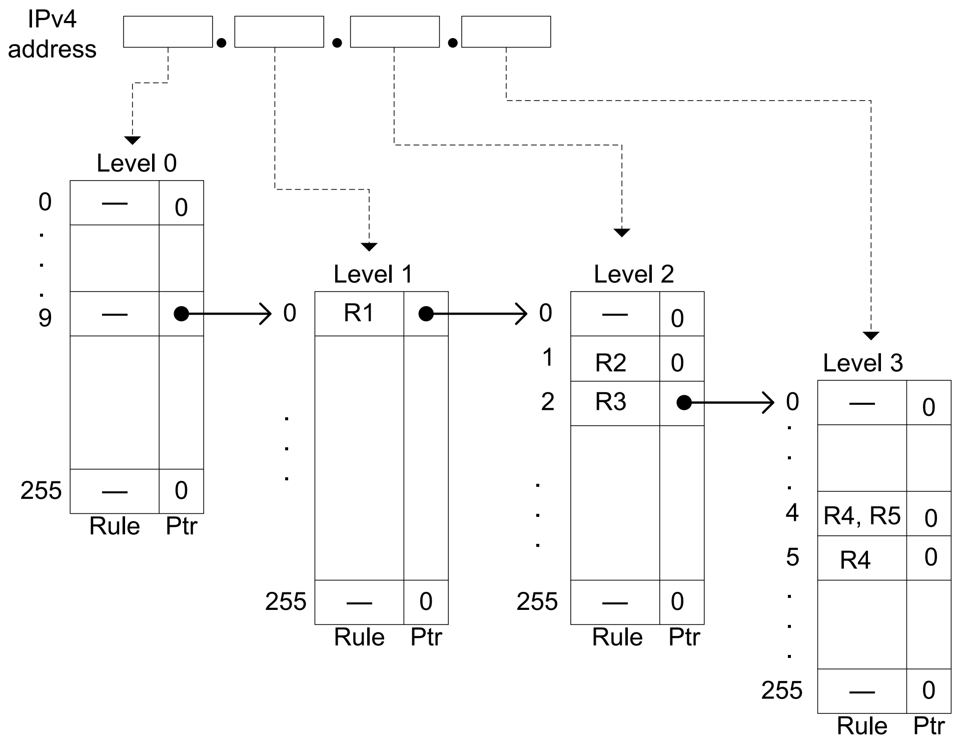


**Figure 40. Multi-level ART for example rule set**

If the IP header address being matched against is 9.0.1.5, then the first byte of the address, 9, is used to index the level 1 array, giving a pointer to a level 1 array. The second byte of the address, 0, is used to index this level 1 array, giving R1 as a match and also a pointer to a level 2 array. The third byte of the address, 1, is used to index this level 2 array, giving R2 as a match and no connecting pointer. The search then terminates with R1 and R2 as the two matching rules.

The ART algorithm is used to perform matching separately on each IP address and port number field and the matching rules are found by ANDing together the output bit vectors as shown in Figure 41.

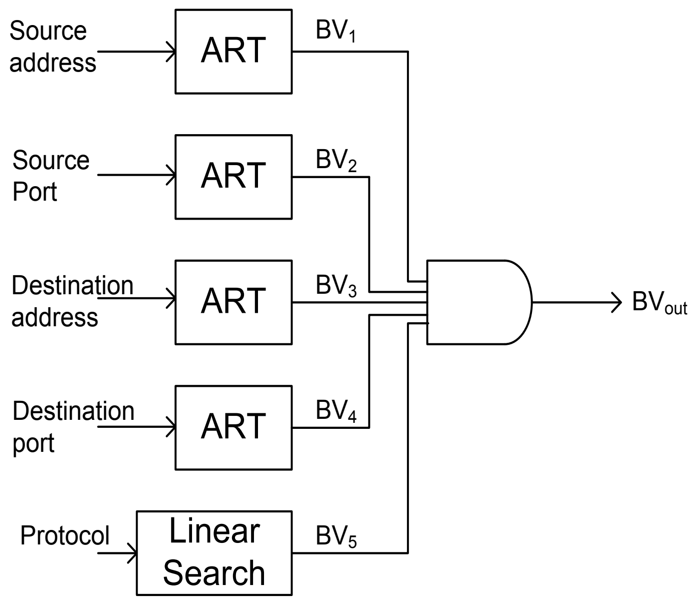


**Figure 41. Multi-match architecture using ART**

## 4.4. Related Work

Research into multi-match packet classification has to date mainly focused on TCAM-based solutions as single-match algorithms have been regarded as unsuitable for multi-match classification due to the extensive intersections between rules in NIDS rule databases (Yu and Katz, 2004). TCAM-based solutions require additional logic to return all matches as TCAMs typically only return the single highest priority match (Yu et al., 2005). Song and Lockwood (2005) and Jiang and Prasanna (2009) researched algorithmic non-TCAM approaches, but both papers propose architectures which incorporate TCAM, albeit on a much smaller scale compared to solutions based entirely on TCAM. Both use bit vectors where each bit corresponds to a rule index.

### 4.4.1. Bit Vector – TCAM architecture

Song and Lockwood's (2005) BV-TCAM (Bit Vector – TCAM) architecture, as illustrated in Figure 42, classifies each of the port numbers in parallel using a multi-bit trie to produce two bit vectors. The multi-bit trie that is used is based on the Tree Bitmap specified by Eatherton (1998), which is subject to patent. A small TCAM implementation, that can handle multiple matches, classifies the concatenation of source, destination address and protocol type to produce a third bit vector. A reduction in the number of TCAM entries is achieved by mapping the concatenation of IP addresses and protocol type in the rule set to a substantially shorter rule set containing only the unique triples. Consequently, the bit vector output from the TCAM has to be decompressed, i.e. mapped to a wider bit vector corresponding to the original rule set. The three bit vectors are finally ANDed together to produce the final result.

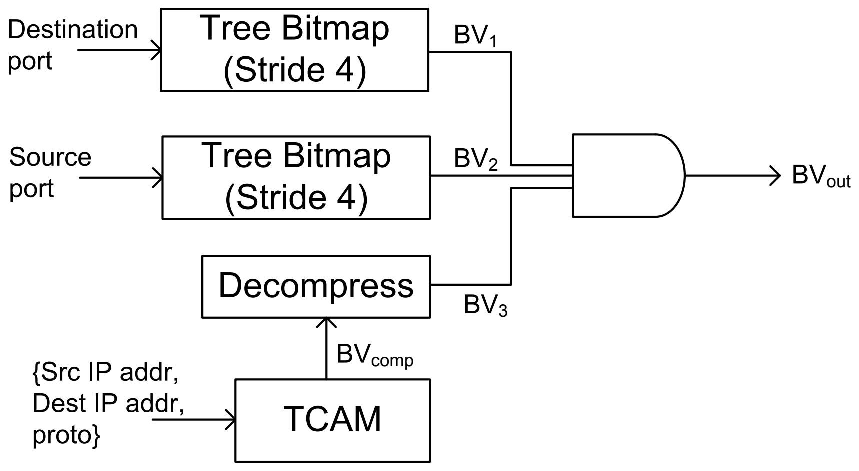


**Figure 42. Overall BV-TCAM architecture**

Many trie-based algorithms make use of prefix expansion and leaf pushing to improve performance. Prefix expansion (Srinivasan and Varghese, 1998) transforms a set of prefixes into an equivalent set with fewer prefix lengths in order to allow multi-bit matching – as also used in ART algorithm. Leaf pushing involves pushing as much node information as possible out to the leaf (or fringe) nodes – as is the case in the multi-match extension of ART.  Eatherton's (1998) Tree Bitmap avoids the need for prefix expansion and leaf pushing by using an indexing scheme that significantly reduces memory consumption. Each node in the multi-bit trie has two associated bit maps, an *Internal Prefix Bitmap* representing the prefixes associated with the node, and an *Extending Paths Bitmap* representing the child nodes that are present.

**Table 14. Example rule set**

| *Prefix* | *Rule* |
|---|---|
| * | 1 |
| 11* | 2 |
| 00* | 3 |
| 0000 01* | 4 |
| 0000 1* | 5 |
| 0001 * | 6 |

Table 14 shows an example rule set, and the corresponding Tree Bitmap based on a 4-bit stride length is shown in Figure 43. Each black dot represents a prefix from the rule set. The three valid prefixes in the root multi-bit node are indicated by the asserted bits in the *Internal Prefix Bitmap* and the two valid branches to child nodes are indicated by the asserted bits in the *Extending Paths Bitmap*. In addition to these two bit maps, each multi-bit node also has two pointers, one to the first valid prefix's rule bit vector and one to the first child node. The addresses of other rule bit vectors and child nodes are obtained by offsetting from these first addresses.

The search algorithm is quite simple. Say the input port number is 0x10FF. The decimal value, *P*, of the first four bits, 1, is used to index the *Extending Paths Bitmap* – bit number 1 (the second from the left) is asserted, indicating that there is a valid child node. The number of 1s to the left of, and including, position *P* is then counted – there are two. The pointer to the child node is then obtained as *stored_child_node_pointer* + (2 × sizeof(node)). Before moving on to the child node, the *Internal Prefix Bitmap* is checked to see if there are any matching rules for the first four bits. In theory, this is done by successively removing bits from the right of *P* and indexing into the corresponding position in the internal bit map. In practice this can be done in parallel in hardware. In the example *P*=0001 (in binary). The rightmost bit is first removed, resulting in prefix 000*, which corresponds to position 8 in the internal bit map. Bit 8 is not asserted and so the search continues by removing the next bit from *P*, giving 00*, which corresponds to position 4 in the bitmap. Bit 4 is set and so a corresponding rule bit vector exists. The address of the bit vector is determined, in a similar fashion to the child node address, by offsetting from the address of the first rule bit vector.

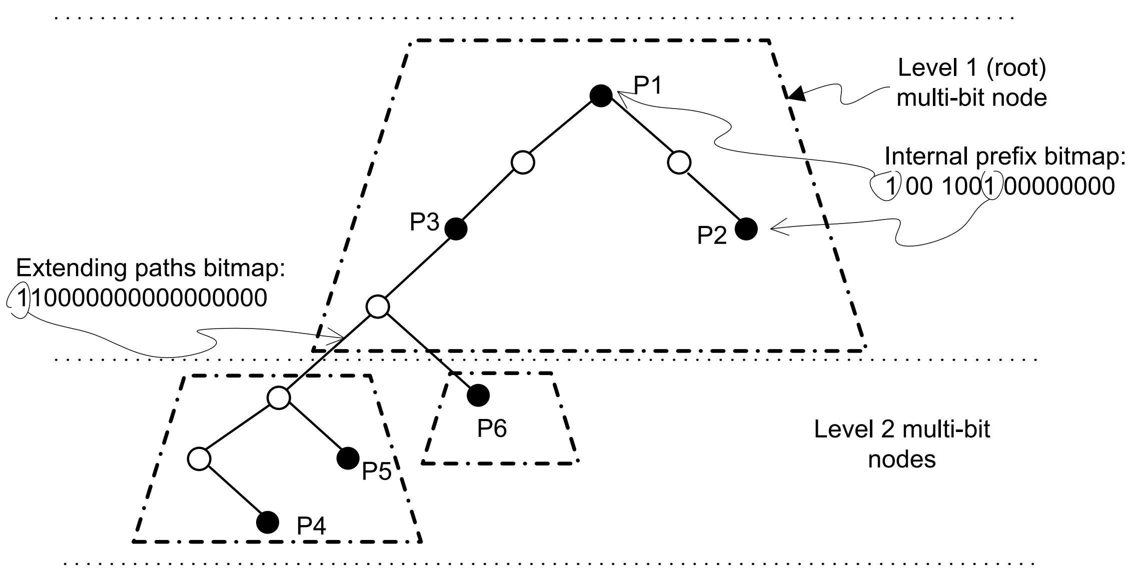

Level 1 (root) multi-bit node
P1
Internal prefix bitmap: 100 1001 00000000
P3
P2
Extending paths bitmap: 1100000000000000
P6
P5
P4
Level 2 multi-bit nodes

**Figure 43. Tree bitmap for example rule set**

## 4.4.2. Field-Split parallel Bit Vector architecture

Jiang and Prasanna's (2009) "Field-Split Parallel Bit Vector" (FSBV) splits the port number fields into single bits which are classified individually to produce a bit vector per bit. The two IP address fields and the protocol type field are classified using two TCAMs and a CAM, respectively, in a manner similar to that of the BV-TCAM architecture. All bit vectors are then ANDed together to give the overall result, as illustrated in Figure 44.

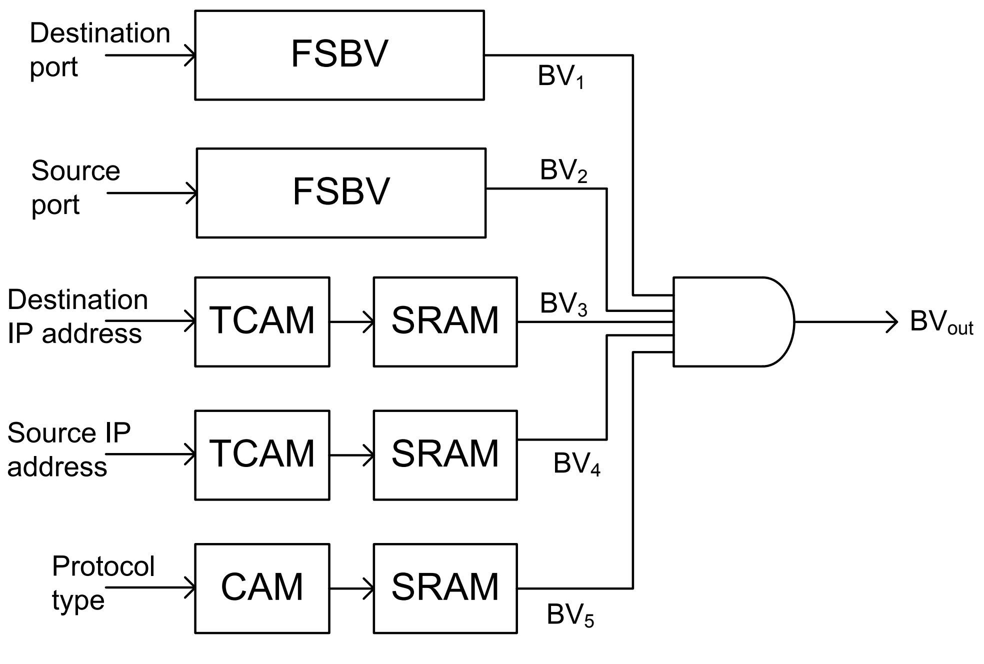


**Figure 44. Multi-match using FSBV algorithm**

**Table 15. Example rule set for 4-bit field**

| 4-bit field match patterns (Bits numbered 3210) | Rule |
|---|---|
| 01** | R1 |
| 0101 | R2 |
| 111* | R3 |

Table 15 and Figure 45 illustrate the FSBV scheme for a 4-bit field and a rule set consisting of three rules.

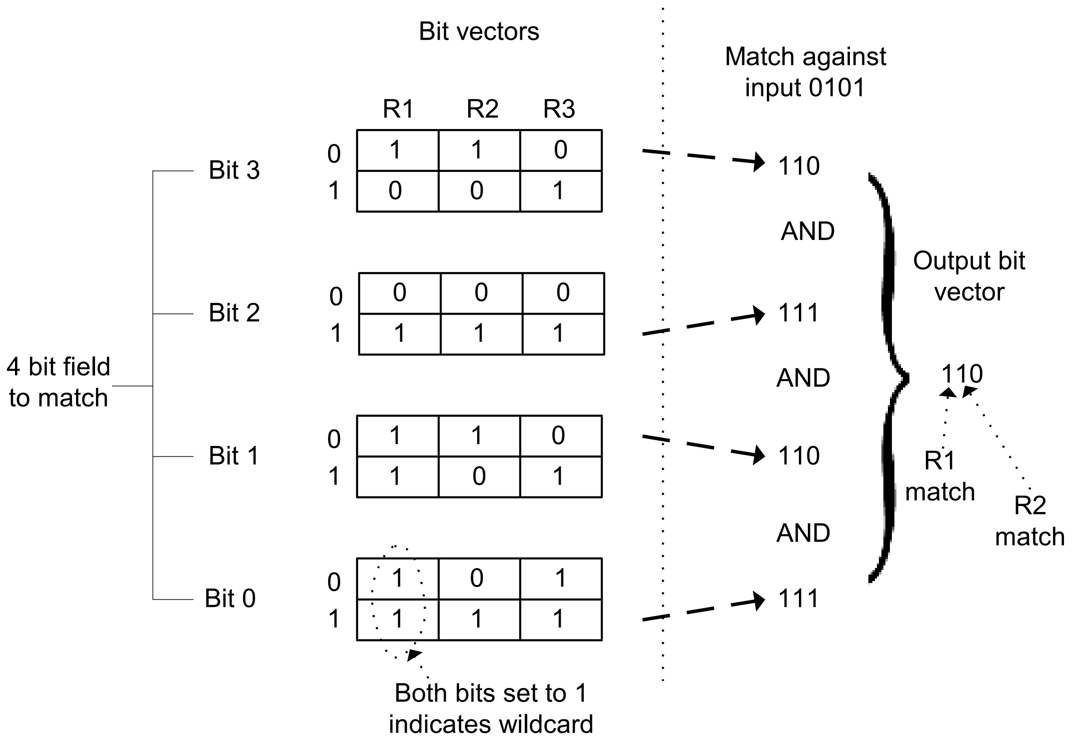


**Figure 45. FSBV scheme for example rule set**

In the overall architecture shown in Figure 44, the FSBV scheme is used to match both port number fields, each consisting of 16 bits. A hardware implementation therefore requires 32 memories, each of depth 2, to store the rule bit vectors. Using FSBV for matching the port numbers is efficient because the numbers of unique source and destination port numbers present in the Snort rule set are much less than the total number of rules, as evident from Table 9. This enables significant compression of the rule bit vectors. Jiang and Prasanna chose TCAM and CAM to match the IP addresses and protocol type, respectively, because they can efficiently map the extremely small number of unique values of each. The bit vectors for each are stored in SRAM and indexed by the TCAM/CAM output.

## 4.5. Comparison

Table 16 summarizes and compares the results for the three algorithms examined with estimated performance figures given by Song and Lockwood (2005) and Jiang and Prasanna (2009) for the BV-TCAM and FSBV systems. Based on these figures, the FSBV architecture uses relatively little memory, and its simplicity should result in an efficient FPGA solution.

**Table 16: Comparison of Algorithms**

| | *Single-match algorithms adapted for multi-matching and evaluated on ARM platform* | | | *Existing multi-match algorithms* | |
|---|---|---|---|---|---|
| | ***Hypercuts*** | ***EGT-PC*** | ***ART*** | ***BV-TCAM*** | ***FSBV*** |
| **Memory (bytes per rule)** | 465693 | 120 | 142 | 74* | 17* |
| **Cycles per packet** | 109961 | 3824 | 658 | 13** | 0.5* |
| **Normalised Energy (µJ/packet)** | 6.85 | 0.55 | 0.15 | 0.01* | 0.01* |

* Based on figures given by Jiang & Prasanna (2009). Note that both static (Virtex 5 XC5VFX200T device) and dynamic consumption are factored into the energy figures given here.

** Based on figures given by Song and Lockwood (2005).

The BV-TCAM and FSBV schemes use only a minimal amount of TCAM which accounts for a negligible proportion of the total energy consumption. As a result they are much more energy efficient than purely TCAM-based schemes and this is borne out by performance comparison results provided by Jiang and Prasanna (2009).

FSBV also performs significantly better than the EGT-PC and ART solutions. This is because FSBV makes the most of the characteristics of the header sections of the Snort rule set by compressing the source and destination IP addresses into very small TCAMs and by splitting the port fields using the efficient FSBV technique.

One possible issue with the FSBV scheme is its reliance on the characteristic that past Snort rule sets had a very small number of unique IP addresses. Ganegedara and Prasanna (2012) proposed the StrideBV algorithm that extends FSBV by applying bit splitting to the entire 5-tuple header, thereby avoiding the use of TCAM. However, instead of single-bit inspection, StrideBV uses multi-bit strides. Ganegedara and Prasanna explain that while StrideBV requires more memory per rule than FSBV, it is capable of handling higher traffic throughput with improved energy efficiency. Sanny, Ganegedara and Prasanna (2013) performed a detailed comparison between StrideBV and TCAM based approaches by evaluating both on a Xilinx Virtex 7 device. They found that the StrideBV implementation performed much better in terms of throughput and power efficiency, at the cost of increased memory consumption.

## 4.6. Conclusion

A number of single match packet classification algorithms were adapted to perform multi-matching and their performance evaluated in terms of speed and energy consumption. It was found that some single match algorithms, such as Hypercuts, do not adapt well to multi-match classification because the high degree of overlap between rules results in an excessive storage requirement. The EGT-PC and ART algorithms adapted reasonably well to multi-matching. Although more efficient than a purely TCAM-based solution, their performance and efficiency does not match that of the FSBV algorithm. The memory efficiency of FSBV is due to the characteristics of recent Snort rule sets. The relatively low quantities of unique values of each header field in the rule set allows the bit vectors representing the rule set to be significantly compressed for each field. The resulting relatively short bit vectors makes implementation of FSBV feasible on FPGA devices. Unlike the BV-TCAM algorithm, FSBV is not subject to patent. Its relative simplicity lends itself to an FPGA implementation where the bit vectors can be stored in block RAM in order to maximise performance.

The recent StrideBV extension to FSBV splits the entire 5-tuple header into multi-bit strides and does not use any TCAM. The main advantage of both algorithms is that they can be easily mapped onto high performance FPGA/ASIC architectures. Evaluation of StrideBV by Ganegedara and Prasanna (2012) found that, although it consumes additional memory, it can handle higher traffic speeds than the original FSBV scheme.

FSBV and its multi-bit extension, StrideBV, appear to be the best approaches to multi-match packet classification in the case of hardware implementations. Further independent evaluation of the two approaches is required in addition to comparison with both TCAM and algorithm based implementations. It should be borne in mind that the performance of these algorithms is mainly due to the relatively low numbers of unique field values in recent rule sets. Should this change in future rule sets, it may render the implementation of these algorithms more challenging and give an advantage to algorithms such as EGT-PC and ART.

# Chapter 5 - Pattern Matching Methods

DPI is the examination of packet payloads for the presence of patterns, known as signatures, listed in a database of rules called the rule set. Signatures are typically in the form of fixed strings or regexes, or a combination of both. The use of regexes has become more common in recent years in order to describe increasingly complex attacks.

## 5.1. Fixed String Matching

### 5.1.1. Precise Matching

The subject of fixed string matching has been well researched due to its importance in many applications such as internet search engines, parsers, word processors and digital libraries. It is important in signature-based NIDPS, as most rules contain at least one fixed string pattern to be matched. Although fixed string matching is not a focus of this thesis, the following is a brief overview in order to give a complete picture of the functionality of an NIDPS.

The string matching problem can be simply stated – Given two strings $T$ and $P$, of length $m$ and $n$, respectively, determine if $P$ occurs in $T$. A naive or brute-force search involves trying to match the pattern using a window size of length $n$ and iterating through each position in $T$ from left to right, resulting in a worst-case complexity of O($mn$). Two classic single-string matching algorithms are Boyer-Moore (1977) and KMP (Knuth-Morris-Pratt, 1977). Both these algorithms also use a window of size $n$, but they use a skip or shift table to determine where to search next after each mismatch. The shifts used by the Boyer-Moore algorithm are based on two rules known as the bad character shift rule and the good suffix shift rule. The first rule avoids the need to repeat unsuccessful comparisons against a target character and the latter ensures that the matching only aligns against target characters already successfully matched. The KMP algorithm similarly uses information learnt from partial matches to skip over alignments that are guaranteed not to result in match. The Boyer-Moore algorithm was later simplified by Horspool (1980) resulting in an algorithm that is easier to implement. The Boyer-Moore algorithm has a worst-case search time of O($m$+$n$) if the pattern does not appear in the text and of O($mn$) if it does. Its average search time is sub linear and improves with increasing pattern length. KMP is O($m$+$n$) in both the average and worst case. Baeza-Yates and Gonnet (1992) found that the average performance of Boyer-Moore-Horspool improves with increasing pattern length and is better than KMP for $n > 3$. These algorithms are unsuitable for multi-pattern matching as the search time increases linearly with the number of patterns.

Two well-known multi-pattern matching algorithms are Aho-Corasick (1975) and Commenzt-Walter (1979). The Aho-Corasick algorithm is an extension of the KMP algorithm for a set of patterns. The algorithm, as illustrated by the example in Figure 46, consists of three functions:

- Goto function: a trie of the set of patterns. Let $L(\nu)$ denote the string produced by traversing the tree from the root to state ν. If ν is a node state $L(\nu)$ represents the prefix of one or more patterns and, if ν is a leaf state then $L(\nu)$ represents a search pattern.

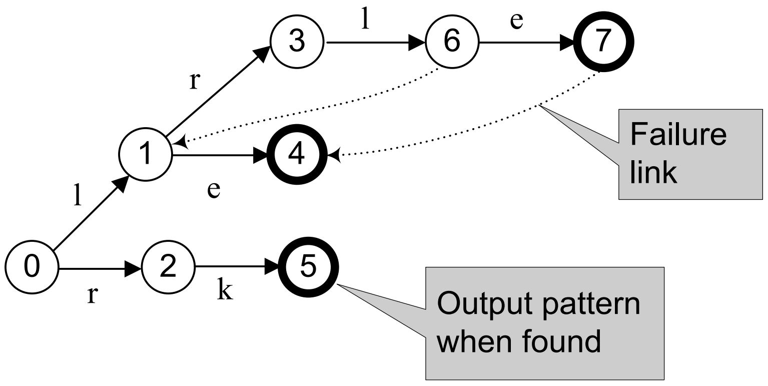


**Figure 46. Aho-Corasick – automaton for set of patterns *P*={lrle, le, rk}**

- Failure function: maps a state to another state when the Goto function reports a failure to match or a terminal state has been reached. The failure function for node ν is the state which is reached by the longest suffix of $L(\nu)$. This is basically a generalisation of the KMP algorithm.
- Output function: The output function for state ν is
  - $L(\nu)$ if ν is a leaf node (i.e. represents a pattern)
  - Output($\nu_{fail}$) where $\nu_{fail}$ is the state reached by the failure link from ν.

If $n$ is the number of states in the automaton and *nocc* is the number of occurrences of a pattern in the string, then the search time complexity of the algorithm is $O(n + nocc)$ when automaton transitions are stored in a transition table and, $O(n\log|\Sigma| +$

*nocc*) when the transitions are stored in a balanced tree (Navarro and Raffinot, 2002, p.50).

The Commentz-Walter algorithm combines ideas from both the Boyer-Moore and Aho-Corasick algorithms. For a string of length $m$ and maximum pattern length $l_{max}$, its worst-case time complexity is O($ml_{max}$). In practice, it is only faster than Aho-Corasick for small numbers of search patterns.

Both the Aho-Corasick and Commentz-Walter algorithms suffer from the fact that the memory requirement can increase exponentially as the number of patterns increases. This degrades software performance as the entire automaton cannot be stored in cache. A number of solutions have been proposed for this memory explosion problem, most of which involve the use of hash tables (Wu and Manber, 1994). The Wu-Manber algorithm is a multi-pattern variant of the Boyer-Moore algorithm which looks at the text in blocks of size B instead of single characters, i.e. it is a multi-stride algorithm. It makes use of three tables, the SHIFT, the PREFIX and the HASH tables. The SHIFT table stores the shift, or skip, values for each of the block characters, indexed by hashing their value. When a potential match is found, then the HASH and PREFIX tables are accessed to check for an actual match. Navarro and Raffinot (2002, pp.59-62) provide a detailed description accompanied by examples. The algorithm requires only O($k$) memory space, where $k$ is the number of patterns and is very fast on average. It was previously used in Snort but has been removed because its worst case performance makes it vulnerable to DoS attacks. Snort now uses the standard Aho-Corasick algorithm by default, but it can be configured to use other versions of the algorithm (Norton, 2004) which trade off memory versus speed. It also includes a binary trie–based algorithm, known as SFK Search, for systems with very low memory.

Much research has been conducted into finding improved variants of the Aho-Corasick algorithm, in particular for hardware implementation. The algorithms proposed by Tuck et al. (2004) reduce memory consumption through the use of bitmap nodes and path compression. Bitmaps reduce the number of transition pointers at states and path compression combines a series of successive states. Tan and Sherwood (2005) uses bit-splitting to split the Aho-Corasick automaton into eight separate automata, each operating on one bit from each input character, thereby reducing the maximum number of transitions from each state from 256 to just 2.

Kennedy et al. (2010) proposed an FPGA architecture, based on the Aho-Corasick algorithm, which uses multiple string matching engines operating in parallel.

A well-known FPGA approach to string matching is to consider the string as a simple regex which can be represented by an NFA which is translated into FPGA logic (Sidhu and Prasanna, 2001). The main disadvantage of this method is the need to reprogram the FPGA whenever the set of strings changes. Moreover, it does not scale well as recent rule sets generate too much logic.

TCAMs can perform parallel searches at high speed but pose two problems for multi-pattern matching: (i) TCAM entries have a fixed length, unlike the string patterns found in NIDS rule sets, and, (ii) TCAMs return the first matching entry and not all matches. Yu et al. (2004) propose a solution which overcomes these two difficulties. The number of TCAM lookups is of the order $O(n)$ where $n$ is the number of input characters. Sung et al. (2005) present a jumping window scheme which reduces the number of TCAM lookups to $O(n/m)$ where $m$ is the window size. Although it gives very good matching performance, TCAM suffers from the problems of relatively high cost and energy inefficiency.

### 5.1.2. Imprecise Matching (with false positives)

Dharmapurikar et al. (2004) describe a hardware-based technique using Bloom filters (Bloom, 1970) for the detection of fixed strings in streaming data. A Bloom filter is a randomised data structure which is "programmed" with strings using multiple hash functions and is "queried" for a string's presence based on multiple bits. A query can result in a false positive but never a false negative. (A false positive is where the matching result incorrectly indicates a match exists, whereas a false negative is where the matching result incorrectly indicates a match does not exist). The main advantage of this technique is that it is likely to only require a relatively small amount of memory even for a very large set of patterns. The disadvantages are that multiple bloom filters are required, one for each pattern length found in the rule set, and that all possible matches must be fully checked for false positives. Song and Lockwood (2005) propose a more efficient data structure called the Extended Bloom Filter in an architecture that makes the most of an FPGA's block RAM. Zhou and Wang (2010) propose an FPGA implementation of multi-pattern string matching using parallel engines based on the Counting Bloom Filter.

Markatos et al. (2002) propose an algorithm based on the use of exclusion-based matching. It basically breaks the patterns into several fixed-size bit strings and searches for these without checking if they are in the correct sequence. If any of sub-patterns does not match, then the entire pattern does not match. On finding a matching sub-pattern, the system falls back on a standard algorithm, such as Boyer-Moore, in order to check the full pattern.

## 5.2. Regular Expression Matching

Regexes are now a common form of signature as they allow the expression of complex attacks which would be very difficult with plain fixed strings. It is well known that a regex can be represented by a non-deterministic finite automaton (NFA) or an equivalent deterministic finite automaton (DFA) (Hopcroft et al., 2006). DFAs have the advantage of fast matching but can consume very large amounts of memory in the case of certain forms of regexes or when some DFAs are combined. NFAs, on the other hand, are memory efficient but can be very slow when a large number of states are concurrently active. This time/space trade-off has led to much research into improving the memory efficiency of DFA-based schemes, and, to a lesser extent, the speed of NFAs.

### 5.2.1. DFA-based solutions

#### Regular Expression Rewriting

Yu et al. (2006) analysed the Snort rule set to identify the typical patterns that result in large DFAs. They propose a number of rewrite rules for these types of patterns in order to reduce the DFA size. Unfortunately, these rules can only be applied to a relatively small number of regexes.

#### DFA Grouping

A simplistic DFA implementation is to amalgamate all of the DFAs, each corresponding to a regex, into a single DFA. This, of course, typically results in a memory explosion because of the interaction between the individual DFAs. Yu et al. (2006) found that it is more efficient to group particular DFAs together and process these groups in parallel, and they devised algorithms to perform this grouping. These algorithms partition the regexes into groups such that the patterns in each group do not adversely interact with each other. Interaction is defined as being present when the number of states of the composite DFA exceeds the sum of the number of states of the individual DFAs. The grouping approach is particularly suited for implementation on multi-core processors, where each composite DFA is assigned to a particular core. Ideally the composite DFA needs to fit in the core's local memory in order to avoid contention between cores that would adversely affect performance. In the case of single core general processors, the composite DFA would be assigned to particular software process or thread. In this case the memory saving resulting from the grouping approach is at the cost of an increase in the memory bandwidth requirement.

**Delayed Input DFA ($D^2$FA)**

Kumar et al. (2006a) observed that many DFA states have similar sets of outgoing transitions and so they propose the $D^2$FA scheme which reduces the memory requirement of a DFA by replacing redundant transitions common to a pair of states with a single default transition. The disadvantage of $D^2$FA is that multiple states can be traversed when processing a single input symbol which degrades performance due to the resulting increase in the number of memory accesses. In order to prevent excessive memory bandwidth, a heuristic construction algorithm is used to limit the length of default transition chains. This algorithm has a number of disadvantages including the requirement for the user to provide an input parameter value which depends on the particular rule set as well as relatively slow construction. Becchi and Crowley (2007) propose a modified version of this scheme which improves its worst case performance and simplifies the construction.

**Content Addressed Delayed Input DFA ($CD^2$FA)**

Kumar et al. (2006b) propose the Content Addressed Delayed Input DFA ($CD^2$FA) which is equivalent to a $D^2$FA in which the state numbers are replaced with content labels. These content labels contain enough information to avoid unnecessary default traversals, thereby resulting in improved performance. Kumar et al.'s experimental evaluation shows that $CD^2$FA uses 10% of the memory space required by a table compressed DFA and can achieve twice the performance of an uncompressed DFA in the case of systems with a small data cache.

### DFA Splitting

Kumar et al. (2007) observed that normal data streams typically only match the first few symbols of a regex. They propose a cutting algorithm which splits a regex into two parts based on the probability that the corresponding NFA state is visited during the matching process. The most frequently visited states constitute the fast path automaton and the remaining tail is processed as a slow path automaton. These tails portions can be "put to sleep" by default and only woken up when the fast path automaton detects a match, e.g. they can be stored in external memory and retrieved when required. This system is vulnerable to a DoS attack in that an attacker can inject large amounts of traffic that match the fast path signatures. Kumar et al. describe a lightweight algorithm to counter these attacks. It uses anomaly counters to track the number of fast path matches for each flow. The slow path then prioritises flows having the lowest anomaly count.

### History-based FA (H-FA)

Kumar et al. (2007) propose a history-based FA (H-FA). They observed that a state explosion occurs in a DFA because it is very inefficient in following multiple partially matching patterns. Their history-based approach consists of an automaton similar to a DFA plus a history buffer. Unlike a standard DFA, multiple transitions for a particular symbol can leave from a state, and which transition is taken is determined by examining the contents of the history information stored in memory. This scheme reduces the total amount of memory required, but can increase the worst case time complexity.

### Hybrid-FA

Becchi and Crowley (2007a) propose a Hybrid-FA consisting of a head-DFA and multiple tail-NFAs. During construction of the Hybrid-FA, any nodes that would result in a state explosion are retained as NFA nodes, while the remaining nodes are transformed into DFA nodes. The tail-NFAs can be transformed into tail-DFAs for improved performance in certain cases, e.g. dot-star terms.

### Extended FA (XFA)

Smith et al. (2008) propose an extended FA (XFA) scheme which avoids the problem of state explosion by using auxiliary variables. They present a model for augmenting a

DFA with these variables and instructions to manipulate them. These are attached to the nodes and edges of the DFA. Experimental results show very good performance and efficient use of memory. However, XFA construction can be quite complicated and may require manual intervention.

**Delta Finite Automaton (δFA)**

Ficara et al. (2008) also observed that many adjacent DFA states share several common transitions. They propose a compressed representation for DFA, called Delta Finite Automata, which only stores the differences between these adjacent states. Unlike $D^2$FA, δFA only requires one transition per single input symbol. However, the difference between the current and next state must be computed on each state traversal, resulting in a time complexity of $O(|\Sigma|)$.

### 5.2.2. NFA-based solutions

The PCRE regex software library, used by both Snort and Suricata, performs NFA style pattern matching. Nonetheless, less research has been conducted into NFAs than DFAs because of the difficulty of handling a potentially large number of concurrently active states and state traversals. However this issue can be circumvented by utilizing the inherent parallelism of hardware.

**PCRE software library (Snort and Suricata)**

The standard matching algorithm used by Snort and Suricata performs a depth-first search of an NFA-based pattern tree. In other words, it follows a single path through the NFA until a mismatch occurs or an accept state is reached. In the case of a mismatch, it checks all other transition branches at the current state before back tracking to an earlier state with multiple transitions and tries the alternative transitions at that point. Back tracking usually involves back tracking of both the NFA and the current position in the input symbol stream. The PCRE library's match function is called recursively at each branch point in order to remember the state in case back tracking to that state is required. Snort and Suricata provide configuration options that place a maximum limit on the number of recursive calls and amount of backtracking. Performance, however, can still be severely degraded in the face of a backtracking attack that injects packets crafted to maximise the amount of backtracking that occurs.

**FPGA logic–based NFA**

The most common method of implementing NFAs in hardware is to convert it into FPGA logic gates and registers (Sidhu and Prasanna, 2001). The disadvantages of this approach are that the logic needs to be re-synthesized whenever the rule set is updated and large rule sets may result in more logic than is available on even the most high-end FPGAs.

**NFA-OBDD**

Yang et al. (2010) propose a scheme in which Ordered Binary Decision Diagrams (OBDD) are used to operate NFAs. Their evaluation was performed using a software implementation that made use of the Cudd C++ based OBDD library. A Binary Decision Diagram (BDD) is a data structure used to represent a boolean function, as illustrated in the example shown in Figure 47. A BDD is said to be "ordered" if the different variables (*x*1, *x*2, *x*3 in the example) appear in the same order in all paths from the root. The OBDD is effectively a maximally reduced version of a standard binary tree.

| *x*1 | *x*2 | *x*3 | f(*x*1,*x*2,*x*3) |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

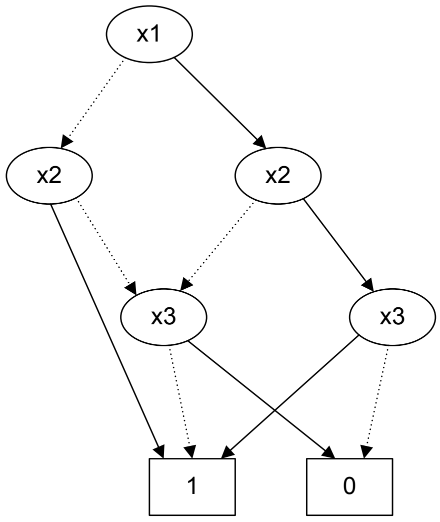


**Figure 47. Example boolean function and corresponding OBDD**

OBDDs can be used to represent a set of elements, such as a subset of the states of an NFA. This could be regarded as a compressed version of the bit-parallel representation discussed in Chapter 6. OBDDs can therefore be used to represent the set of currently active states, the set of accept states, input symbols and transitions between states. A transition is a triple (*s*, *i*, *t*) such that there is a transition labelled *i* from state *s* to state *t*. These OBDDs can then be manipulated to determine the OBDD representing the next set of currently active states following receipt of each input symbol and to check if an accept state has been reached. The operation to find the next set of active states is effectively a boolean AND combination of the transition, active and input symbol OBDDs. The authors found that the NFA-OBDD scheme outperformed standard NFA implementations in the case of HTTP and FTP signatures. NFA-OBDDs performed best when the set of currently active states was large. Sinnappan and Hazelhurst (2001) describe how BDDs can be implemented as a logic circuit on an FPGA and Yang et al. suggest that this method could also be used for NFA-OBDDs.

**FPGA/ASIC memory–based NFA schemes**

The limitations of the logic-based approach have led to research into memory-based hardware architectures which can be easily and quickly reconfigured. Examples of such architectures include the bitmap-based approach for the Glushkov NFA (Lee, 2009) and the dynamic reconfigurable bit-parallel NFA architecture (BP-NFA) (Kaneta et al., 2010). The first is based on the Bit Parallel Glushkov algorithm and the second on the extended SHIFT-AND algorithm, both of which are detailed in a textbook by Navarro and Raffinot (2002). Both algorithms utilize bit-parallelism (Wu and Manber, 1992; Baeza-Yates and Gonnet, 1992). Bit-parallelism is a technique to code multiple elements of information into a single bitmask which can then be operated on simultaneously. In the case of regex matching, the bitmask stores the active and inactive states.

Several methods exist for constructing an NFA from a regex of $m$ characters (excluding special symbols). The most common method is the Thompson (1968) construction, which produces an NFA with between $m+1$ and $2m$ states. Its most important property is that, apart from the ε-transitions, all transitions go from states $i$ to states $i+1$. This is exploited by Wu and Manber (1994) in their bit-parallel scheme. Note that the BP Thompson algorithm is equivalent to the extended SHIFT-AND algorithm. An alternative method is the Glushkov construction which has the important advantage that it has only $m+1$ states, although this comes at the price of not having the simple forward transitions of the Thompson NFA. The Glushkov NFA also has the property that all incoming transitions arriving at a particular state node have the same symbol label. This property, along with the minimal number of states, gives the Glushkov construction the edge in bit parallel implementations (Navarro and Raffinot, 2002).

The bit-parallel representation of an NFA can be considered either as a bit-parallel implementation of an NFA or as a DFA. Each possible value of the bitmask of currently active and inactive NFA states, that these BP algorithms use, is effectively the identifier of a DFA state. These BP schemes use the bitmask to index a table and the resulting value, along with the current input symbol, gives the new bitmask value.

### 5.2.3. Imprecise Matching Finite Automata

**StriFA**

Wang et al. (2013) propose Stride FA (StriFA) for the acceleration of both fixed string and regex matching. The algorithm can be implemented as either an NFA or a DFA depending on which is most suitable for the platform used. Some of the matches found by StriFA may be false positives, but the algorithm is designed to keep these to a reasonable level. StriFA can therefore be used as a fast and efficient pre-filter to greatly reduce the number of regexes against which to match in a follow-up FA which does not produce false positives. In the case of a fixed string pattern, the first step in the construction of the StriFA involves the selection of an appropriate tag character from the string. The distances, known as stride lengths, between successive occurrences of the tag character in the pattern are then used to construct the StriFA. The construction from a regex is more involved, but the basic concept of tag characters and stride lengths remains essentially the same. During matching, the incoming data stream is converted to the corresponding tag length stream before being fed into the StriFA matching engine.

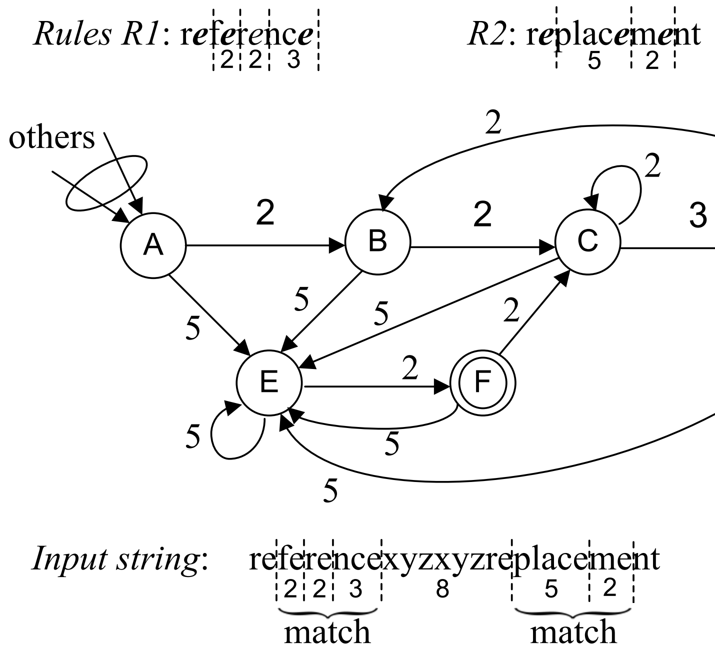


**Figure 48. Example StriDFA for patterns "reference" & "replacement" with tag 'e'**

Consider the fixed string patterns R1="reference" and R2="replacement". If character tag 'e' is chosen then the corresponding stride lengths for R1 and R2 are 2-2-3 and 5-2, respectively. The stride lengths are then used to construct a standard DFA as illustrated in Figure 48, in which the transitions are labelled with the stride length. Note that the states are labelled with letters rather than numbers in order to avoid any confusion. During matching, the example input string *referencexyzxyzreplacement* is converted to a stream of stride lengths 2-2-3-8-5-2 with 2-2-3 and 5-2 matching the StriDFA.

Selection of an appropriate tag character is fundamental to the performance of StriFA. Considering each pattern individually, the best tag is the character which occurs most frequently. However, the incoming data stream needs to be converted into a separate stride length stream for each different tag character. The strategy is therefore to select a tag character that covers as many patterns in the rule set as possible, with a minimum of three occurrences per pattern, and with the highest average number of occurrences per pattern. Wang et al. found that StriFA gives a 10-fold increase in speed and much lower memory consumption compared to traditional NFA/DFA.

**DFA Abstraction**

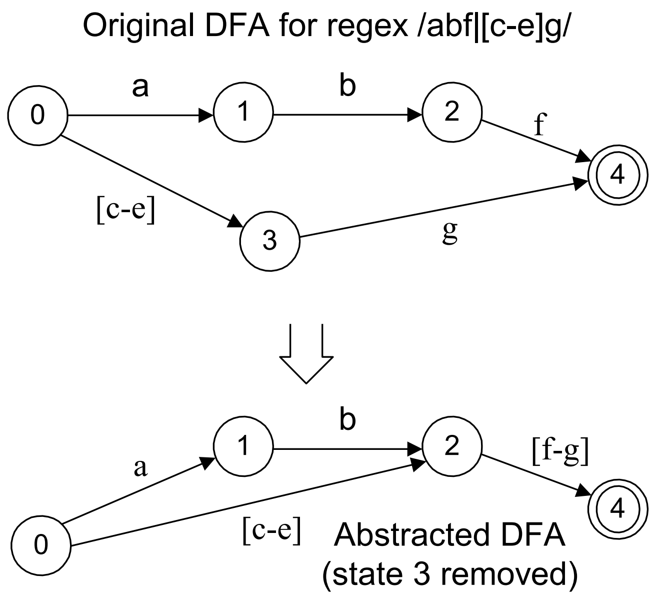


**Figure 49. Example of construction of abstracted DFA by state removal**

Cadambi et al. (2007) describe a method, which they term DFA abstraction, by which a DFA can be reduced in size by removing some states, leading to a smaller DFA which can produce false positives but no false negatives. Packets that match the compacted DFA are checked by pattern matching against a follow-up FA, typically a space efficient NFA implementation. Construction of the abstracted DFA is not straightforward as the algorithm must ensure that removing a state does not result in false negatives.

Figure 49 illustrates the concept of DFA abstraction in the case of regex /abf|[c-e]g/. State 3 is removed from the original DFA and all its transitions changed to state 2. The resulting DFA has no false negatives but has a number of patterns that will result in false positives, namely abg, cf, df and ef.

### 5.2.4. Alphabet Reduction

The FA corresponding to regexes over an alphabet Σ can potentially have |Σ| outgoing transitions per state. However, the alphabet can be reduced by translating it into a set of equivalence classes. Two symbols are members of a particular class if the target of their transition from a particular state is the same, and this is the case for all states in the FA. Alphabet reduction can result in significant savings since regexes usually only use a small subset of all possible symbols.

### 5.2.5. Multi-stride Automata

Making an automaton multi-stride is a technique used to increase throughput by reducing the memory bandwidth requirement, i.e. the automaton processes multiple input symbols at the same time. Implementing multi-stride in isolation leads to an unacceptable increase in the number of states. Brodie et al. (2006) explain how a multi-stride DFA can be made feasible by applying optimisations to the memory structures used. Becchi and Crowley (2008b) propose improved optimisations in the form of alphabet reduction and default transition compression. Alphabet reduction works due to the fact that increasing the stride results in only a small subset of the entire alphabet being used. Similarly, as the stride is increased, the number of transitions increases at a higher rate than the number of states, and so the fraction of distinct transitions falls, allowing a greater degree of compression. Alphabet reduction can be applied to both NFA- and DFA-based schemes, while default transition compression is a DFA optimisation technique.

### 5.2.6. Commodity versus Speciality Hardware

Considerable research has been carried out into improving the performance of solutions based on commodity hardware such as off-the-shelf servers and Graphics Processing Units (GPU). Commodity hardware has the advantage of comparatively low cost and its performance may be adequate for many applications. Companies entering into IDS product development may prefer to use commodity rather than specialised hardware in order to reduce time-to-market, keep development costs down and deliver a more maintainable product. Such companies might see the use of specialised hardware as a second step reserved for the development of higher performance products once the commodity-based products have gained a market foothold.

Yu (2006) explains how CPU-based software approaches such as Snort and Linux L7-filter cannot cope with high traffic rates. Becchi et al. (2009) measured the performance of DFA, NFA and Hybrid-FA implementations on network and general purpose processors. They found that the 4-way AMD Opteron performed much better than the Intel Xeon and IXP2800, with throughput of between 15 and 70 Mb/s in the case of the NFA design, and between 151 and 534 Mb/s in the case of the Hybrid-FA. Day & Burns (2011) compare the performance of the single threaded Snort software and the multi-threaded Suricata software on multi core systems. They recommend using multiple instances of Snort on multiple cores rather than a single multi-threaded instance of Suricata. Running multiple instances of Snort would require the use of flow pinning in order to split the traffic between the Snort instances. Wun et al. (2009) explain why running Snort on a multi core system may not provide the expected improvement in worst case performance due to the fact that certain exceptional packets may cause a bottleneck in the regex matching in one of the Snort instances. Albin & Rowe (2012) evaluated Snort on a Dell Poweredge R710 dual quad-core server where each CPI was an Intel Xenon E5630 operating at 2.4GHz. They found that Snort was limited in its ability to scale beyond 200-300Mb/s throughput per instance.

The highly parallel architecture of GPUs makes them effective for many complex algorithms. Their relatively low cost has prompted much research into their use for offloading of regex matching from the CPU (Antonello et al., 2012). Vasiliadis et al.'s (2009) GPU-based DPI system uses fixed string pre-filtering software running on the CPU in order to reduce the amount of regex matching that needs to be performed by the GPU. Payloads that match in the pre-filter are forwarded along with a regex identifier to the GPU for regex matching. The GPU-based system performs well provided each payload only needs to be matched against a small number of regexes. However, the performance drops significantly if a large number of regexes need to be matched against each payload. Vasiliadis et al. measured a worst case throughput of 700Mb/s on their NVIDIA GeForce 9–based system. It must be noted that their DFA-based design cannot handle certain complex PCRE syntax such as constrained repetitions, and that regexes containing such syntax must be handled by Snort software running on the CPU. Zu et al. (2012) evaluated an NFA design on an NVIDIA GTX-460 GPU. They give throughput figures of in excess of 10 Gb/s, but it's unclear what type and length of regex it can handle.

## 5.3. Conclusion

DPI typically involves a combination of fixed string and regex matching. Regexes are now commonly used in NIDS rule sets and can be modelled as NFA or DFA automata. In general, NFA-based schemes typically require much less memory but are slower, whereas DFA-based schemes require much more memory but are faster. Much research has been conducted into improving the memory efficiency of DFA models using a variety of techniques and into improving the speed of NFA models by exploiting hardware parallelism.

Software-based systems, such as Snort, are unable to perform DPI at the high traffic speeds found in today's networks, particularly where the NIDPS system is to be used to monitor an enterprise network's internal traffic. There is therefore a need to use other technology such as GPUs or FPGA-based hardware to accelerate the DPI functionality. A classic FPGA-based scheme is to synthesise an NFA representation of the regexes as FPGA logic. The problem with this approach is that the FPGA configuration needs to be resynthesised whenever the rule set changes, something which can take a considerable amount of time and effort. The FPGA device also needs to be reconfigured and, unless partial reconfiguration is supported, the system will be offline during this procedure. A more suitable design is one in which the regex data is stored in memory and so can be easily and quickly updated while the system is operating. Bit-Parallel architectures based on NFA are an example of a memory-centric approach. Although a significant amount of research (Lee, 2010; Kaneta et al., 2010) has been conducted into such architectures, considerable opportunities remain to improve on the handling of some of the more complicated regex syntax such as constrained {min,max} repetitions and back references. New algorithms for the handling of constrained repetitions are the subject of Chapters 6 and 7.

# Chapter 6 - Constrained Repetitions in Regular Expressions

The continuous emergence of new attacks means an ever increasing number of rules for NIDPS, and the increasing complexity of attacks has resulted in increased usage of regex-based signatures. The need to match against very large numbers of complex regexes at multi-gigabit traffic rates is too demanding for a software solution. Moreover, the increasing use of more complex regex features such as constrained repetition quantifiers and back references places even more intensive processing demands on NIDPS systems. As a result, there is an increasing need to find efficient hardware solutions that can handle large numbers of rules, including all the features of Perl Compatible Regular Expressions (PCRE), at high bit rates.

The most common approach to implementing a standard NFA in hardware is to convert it into FPGA logic gates and registers (Sidhu and Prasanna, 2001). The disadvantage of this approach is that the logic needs to be re-synthesised whenever the rule set is updated. This has led to research into memory-based FPGA and ASIC architectures which can be easily reconfigured. One such system is the Bit-Parallel (BP) architecture based on the Glushkov NFA. This chapter proposes a modification to this architecture in order to more efficiently handle constrained {min, max} repetitions. To enable handling by the standard BP system, these repetitions first need to be unrolled, often resulting in an excessive memory requirement. The solution presented here can deal with the repetition directly without unrolling, thereby making it possible to handle regexes that would not be suitable for the standard system.

## 6.1. Constrained Repetitions in Snort Rule Set

Table 17 illustrates the ever increasing number of rules in the Snort rule set, approximately half of which include at least one regex. In the September 2012 snapshot, roughly 19% of the unique regexes contain at least one constrained repetition quantifier, the syntax of which is explained in Table 18. Table 19 shows the statistics for the constrained repetitions present in the September 2012 Snort rule set. Most of the repeated sub-expressions are single rather than multi symbol, and the quantifier values can be quite high, with a maximum of 4 017.

**Table 17: Snort Rule Set Statistics**

| Snort version | 2.8 | 2.9.0.0 | 2.9.3.1 |
|---|---|---|---|
| Snort VRT rule set snapshot date | 17.02.2010 | 07.01.2011 | 18.09.2012 |
| Number of rules | 8 454 | 9 852 | 23 170 |
| Number of rules with string signature | 8 273 | 9 686 | 22 762 |
| Number of rules with regex signature | 4 386 | 4 577 | 12 460 |
| Number of unique regexes | 3 697 | 3 892 | 5 555 |
| Number of regexes with constrained repetitions | 497 | 598 | 1 043 |

**Table 18: Constrained repetition quantifier syntax**

| Syntax | Meaning |
|---|---|
| *R{num}* | Match *R* exactly *num* times. |
| *R{min, max}* | Match *R* at least *min* times and at most *max* times. |
| *R{min,}* | Match *R* at least *min* times. *R{min,}* can be rewritten as *R{min}R** |
| *R{,max}* | Match *R* at most max times. Equivalent to *R{0,max}*. |

**Table 19: Snort Constrained Repetition Statistics (v2.9.3.1, 18.09.2012 snapshot)**

| | |
|---|---|
| Number of unique regexes which contain constrained repetitions | 1 043 |
| Total number of constrained repetitions | 2 030 |
| Single symbol repeated sub-expr. (e.g. literal, meta-character, character class) | 1887 |
| Multi-symbol repeated sub-expression | 143 |
| Maximum quantifier value (i.e. value of *{min, max}*) for single symbol repetitions | 4017 |
| Maximum quantifier value for multi-symbol repetitions | 499 |

## 6.2. Bit-Parallel (BP) Architectures

Existing bit-parallel algorithms handle constrained repetition quantifiers by unrolling which is not efficient considering that recent SNORT rule sets contain a very high number of such quantifiers, many of which have high min/max count values. Examples of memory-based architectures include the bitmap-based approach for the Glushkov NFA (Lee, 2009) and the dynamic reconfigurable BP-NFA (Kaneta et al., 2010). The former is based on the BPGlushkov algorithm and the latter on the extended SHIFT-AND algorithm, both of which are detailed by Navarro and Raffinot (2002) and utilise bit-parallelism (Wu and Manber, 1992; Baeza-Yates and Gonnet, 1992). Although these algorithms can be considered to be NFA-based, the representation is in fact a DFA because the bitmask of active NFA states, that these algorithms use, represents the current equivalent DFA state.

## 6.3. Glushkov NFA

The fundamentals of the Glushkov construction method were presented in section 2.1.4. The following describes how state traversal functions.

Consider the example G-NFA shown in Figure 50 for the example regex RE=*((ABA|C)B*)A*.

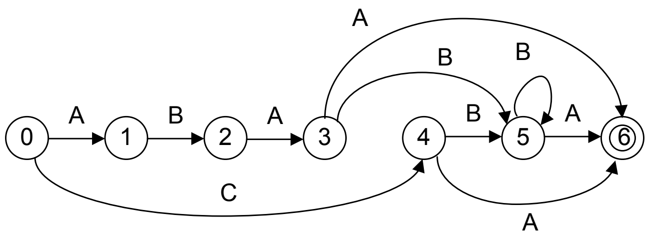


**Figure 50: The G-NFA for *RE = ((ABA|C)B*)A***

Two important properties of the G-NFA are that:

- all the transitions into a particular state y are labelled with the same character $\alpha_y$
- it's free of ε-transitions

The first property allows the construction of a table, *Enter[σ]*, which gives the set of states reachable by each character $\sigma \in \Sigma$. In the given example *Enter[A]*={1,3,6}, and so on for each character. If *Active* is the set of currently active states, then the first set of active states prior to reading any character is *Active={0}*, and all subsequent sets of active states, for each input character, *σ*, are given by:

$Active \leftarrow ( \bigcup_{x \in Active} Follow(\overline{RE}, x)) \bigcap Enter[\sigma]$ in the case of a regex with a start anchor

$Active \leftarrow (( \bigcup_{x \in Active} Follow(\overline{RE}, x)) \bigcap Enter[\sigma]) \bigcup \{0\}$ in the case of regex without a start anchor

where *x* is the index for each position in the marked regex, $\overline{RE}$. (See section 2.1.3 for a detailed description of anchors and multi-line mode).

i.e., the new *Active* set is the union of sets of states reachable from all states currently in the *Active* set intersected with states reachable by the current input symbol, *σ*. In other words, the new *Active* is the set of states reachable from all currently active states, subject to each element being reachable by the current input symbol, $\sigma$. In the case of an unanchored regex, state 0 is held active. Note that in the case of a regex with a start anchor and multi-line mode enabled, state 0 must also be held active when a newline character is consumed.

## 6.4. Counting Glushkov NFA

The standard G-NFA construction algorithm handles constrained repetition quantifiers, *RE{min,max}*, by unrolling. This results in a total of $max \times m$ states (where $m$ is the number of symbol positions in the sub pattern, *RE*), which is clearly inefficient if *max* is large. Smith et al. (2008) and Kumar et al. (2007) propose how a counter variable can be used to improve the storage efficiency of their DFA-based algorithms. This thesis describes how this idea can be used in the case of a G-NFA by constructing a modified counting form of it in which repeated sub-patterns do not need to be unrolled.

Consider the example regex looked at earlier, but with the addition of a constrained repetition, $RE = ((ABA|C)B^*)\{min,max\}A$, where *min* and *max* are non-negative integers. In the example NFA, the repeated sub-pattern, $\overline{RE_{CNTi}} = (A_1B_2A_3|C_4)B_5^*$, is part of what will be referred to as a counting block, with associated language $L(\overline{RE_{CNT_i}})$ $= \{C_4, A_1B_2A_3, C_4B_5, A_1B_2A_3B_5, C_4B_5B_5, A_1B_2A_3B_5B_5, \ldots\}$. There may be multiple constrained repetition quantifiers in a single regex resulting in $c$ counting blocks, $CNT_i$, indexed $i=\{1..c\}$. The first step in the NFA construction is as before, except that transitions out of each counting block are omitted. The example counting block contains marked positions 1 to 5 as shown in Figure 51.

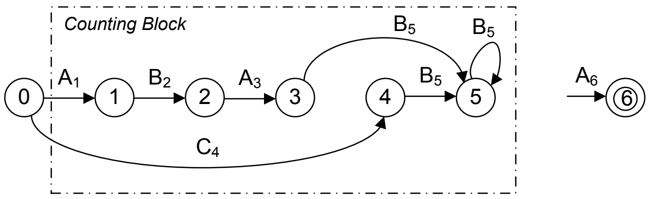


**Figure 51: G-NFA for marked sub-pattern**

Definitions of $Last(\overline{RE})$ and $Enter[\sigma]$ are as before, but $Follow(\overline{RE},x)$ must be redefined so as to omit transitions out of the counting block for each position, $x$, within the block. In the example in Figure 51, $Follow(\overline{RE},4) = \{5\}$. Note that the G-NFA state numbers correspond directly with the positions in the marked regex, $\overline{RE}$, and so the index, *x*, can be regarded equally as a position in the regex or as a state in the G-NFA.

$Follow'(\overline{RE},x)$ is the set of positions in $\overline{RE}$ which are reachable from *x*, excluding any transition from within a counting block to outside of the block. This can be described mathematically as follows:

$Follow'(\overline{RE},x) =$

$$\begin{cases} \{y \mid y \in Pos(\overline{RE}), \exists u,v \in \overline{\Sigma}^*, u\alpha_x\alpha_y v \in L(\overline{RE})\}, if \quad x \notin \bigcup_i Pos(\overline{RE_{CNT_i}}), \\ \qquad \{y \mid y \in Pos(\overline{RE_{CNT_i}}), x \in Pos(\overline{RE_{CNT_i}}), \\ \qquad \exists u,v \in \overline{\Sigma}^*, u\alpha_x\alpha_y v \in L(\overline{RE_{CNT_i}})\}, if \quad x \in \bigcup_i Pos(\overline{RE_{CNT_i}}) \end{cases}$$

This definition is the same as the original $Follow(\overline{RE},x)$ for all positions with index *x* outside of a counting block. For positions with index *x* within a counting block, $Follow'(\overline{RE},x)$ is the set of positions within that counting block for which the combination of the two characters $\alpha_x$ followed by $\alpha_y$ form a substring of some string from the language of $\overline{RE_{CNT_i}}$, where y is also the index of some position within the same counting block.

Note: $\overline{\Sigma}^*$ is the set of all strings over the alphabet, $\overline{\Sigma}$, of the language expressed by the marked regex (see section 2.1.1 for details). In other words, $\overline{\Sigma}^*$ is the set of all strings that can be formed from the characters accepted by the marked regex.

Table 20 lists the values of $Follow'(\overline{RE},x)$ for each state *x*.

**Table 20. Values of *Follow'* for each state *x* of example Counting G-NFA**

| $x$ | $Follow'(\overline{RE},x)$ |
|---|---|
| 0 | {1,4} |
| 1 | {2} |
| 2 | {3} |
| 3 | {5} |
| 4 | {5} |
| 5 | {5} |
| 6 | Ø |

The following definitions specific to each counting block, $CNT_i$, can now be added:

- *FirstBlk* is the set of states within the counting block that can be reached by states outside of the block. It is also the set of states that can be reached by the first character of each repetition cycle. Mathematically, this can be expressed as:

  $FirstBlk(\overline{RE_{CNT_i}}) = \{x | x \in Pos(\overline{RE_{CNT_i}}), \exists u \in \overline{\Sigma}^*, \alpha_x u \in L(\overline{RE_{CNT_i}})\}$

  i.e. the set of positions within the counting block with index $x$ for which the corresponding character $\alpha_x$ forms a string from the language of the counting block's sub-regex when appended with some combination of characters.

  In the example shown in Figure 52, *FirstBlk* = {1,4}

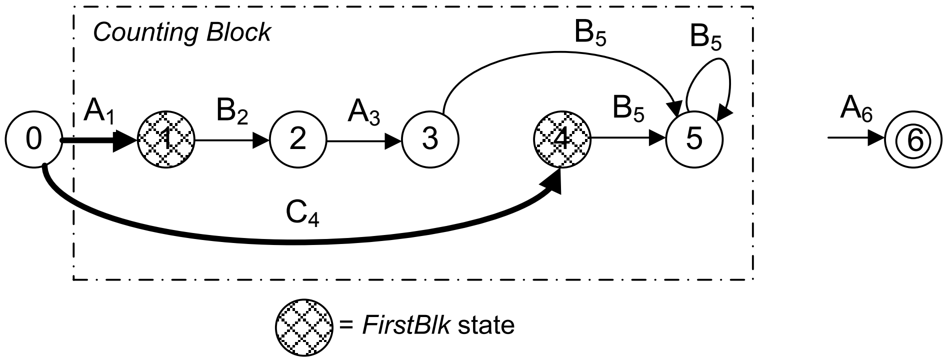


**Figure 52. FirstBlk states for example Counting G-NFA**

- *FinalBlk* is the set of states within the counting block where a decision is made on repeating and/or transitioning out of the block. Mathematically, this can be expressed as:

  $FinalBlk(\overline{RE_{CNT_i}}) = \{x | x \in Pos(\overline{RE_{CNT_i}}), \exists u \in \overline{\Sigma}^*, u\alpha_x \in L(\overline{RE_{CNT_i}})\}$

i.e. the set of positions within the counting block with index x for which the corresponding character $\alpha_x$ forms a string from the language of the counting block's sub-regex when prefixed with some combination of characters.

- *IncrementBlk* is the set of states within the counting block where the counter is incremented. In many cases, this will be identical to the set *FinalBlk* set. However, in order to avoid incrementing the counter more than once per traversal of the counting block, any states in *FinalBlk* that are reached via traversals through other *FinalBlk* states must be excluded.

$$IncrementBlk(\overline{RE_{CNT_i}}) = \{y \mid y, x \in FinalBlk(\overline{RE_{CNT_i}}), \nexists$$

$$u, v \in \overline{\Sigma}^*,\ u\alpha_x v\alpha_y \in L(\overline{RE_{CNT_i}})\}$$

i.e. the set of positions within the *FinalBlk* set excluding those that are positions which are reachable from other positions within the same set via a single or multiple state traversal.

In the example, as shown in Figure 51, *FinalBlk* = {3,4,5}. State *x*=5 can only be reached from other *FinalBlk* states and so is excluded from the *IncrementBlk* set, giving *IncrementBlk* = {3,4}.

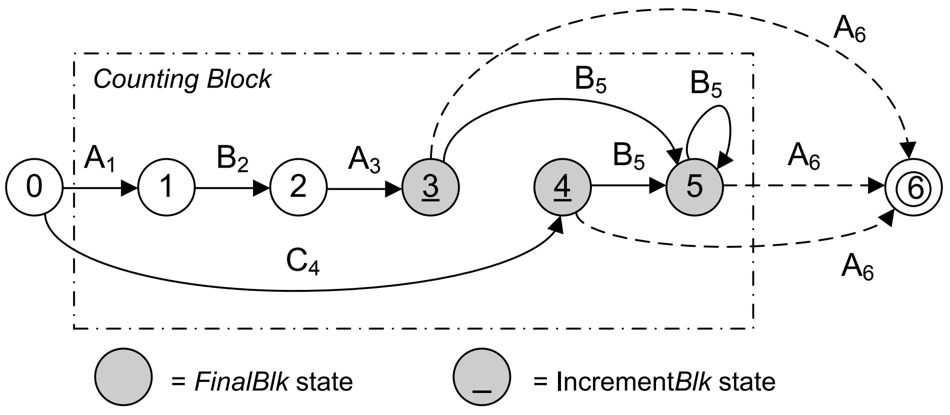
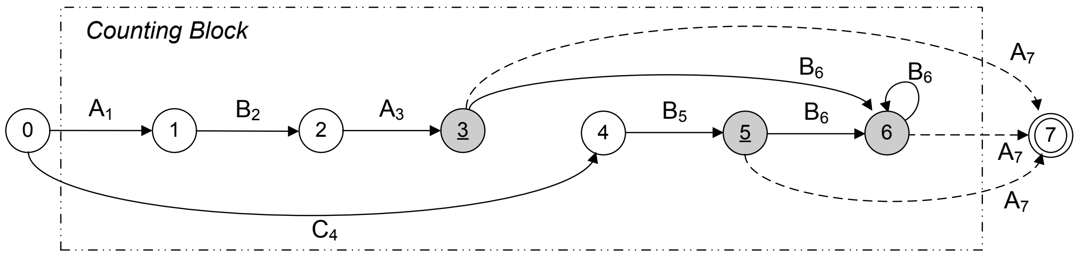


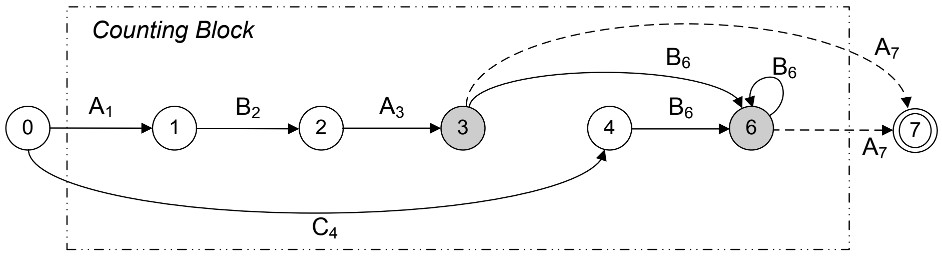
**Figure 53. *FinalBlk* and *IncrementBlk* states in example Counting G-NFA**

In some cases it is possible to compact the constructed G-NFA as illustrated in Figure 54. The resulting NFA is not however a Glushkov NFA and is not suitable for use in the Counting G-NFA algorithm as it may include *FinalBlk* states that can be reached both directly and via other *FinalBlk* states. This would make it impossible to create a valid *IncrementBlk* set.



(i) Counting G-NFA for regex /((A$_1$B$_2$A$_3$|C$_4$B$_5$)B$_6$*){min,max}A$_7$/

● = *FinalBlk* state (_) = *IncrementBlk* state



(ii) Invalid Counting G-NFA for regex /((A$_1$B$_2$A$_3$|C$_4$B$_5$)B$_6$*){min,max}A$_7$/

**Figure 54. Counting G-NFA for regex /((ABA|CB)B*){min,max}A/**

- *FollowBlk* is the set of states outside of the counting block that are reachable from states within the counting block. In the example in Figure 51, *FollowBlk* = {6}.

  $FollowBlk(\overline{RE_{CNTi}}) =$

  $\{y \mid y \in Pos(\overline{RE}), y \notin Pos(\overline{RE_{CNTi}}), x \in FinalBlk(\overline{RE_{CNTi}}), \exists u,v \in \overline{\Sigma}^*,\ u\alpha_x\alpha_y v \in L(\overline{RE})\}$

  i.e. the set of positions, with index *y*, outside of the counting block for which the corresponding character $\alpha_y$ preceded by a character $\alpha_x$ corresponding to a position *x* within the *FinalBlk* set, forms a substring of the language of the regex. In other words,.

The counting block has a counter variable, $cnt_i$, with the following operations carried out for every character read:

1) Reset the counter if no state within counting block is currently active:

   $Active \cap Pos(\overline{RE_{CNT_i}}) \equiv \varnothing \Rightarrow cnt_i \leftarrow 0$

2) Increment counter when one of the states within the *IncrementBlk* set is active:

   $Active \cap IncrementBlk(\overline{RE_{CNT_i}}) \neq \varnothing \Rightarrow cnt_i \leftarrow cnt_i + 1$

3) Initialise *Follow′* to be the set of states reachable from *x*, omitting any transitions from a position inside a counting block to outside:

$Follow'(\overline{RE},x) \leftarrow Follow(\overline{RE},x)$

4) Now if the counting block counter is ≥ *min* then the *Follow′* set is updated to include transitions from positions within the counting block to positions outside of the block:

   $((x \in FinalBlk(\overline{RE_{CNT_i}})) \wedge (cnt_i \geq \min_i)) \Rightarrow$

   $Follow'(\overline{RE},x) \leftarrow Follow'(\overline{RE},x) \bigcup FollowBlk(\overline{RE_{CNT_i}})$

5) Now if the counting block counter is less than *max* then the *Follow′* set is updated to include transitions from the *FinalBlk* positions to the counting block's entry positions (*FirstBlk*):

   $((x \in FinalBlk(\overline{RE_{CNT_i}})) \wedge (cnt_i < \max_i)) \Rightarrow$

   $Follow'(\overline{RE},x) \leftarrow Follow'(\overline{RE},x) \bigcup FirstBlk(\overline{RE_{CNT_i}})$

The definition of the next set of active states can now be redefined as

$Active \leftarrow ((\bigcup_{x \in Active} Follow'(\overline{RE},x)) \bigcap Enter[\sigma]) \bigcup \{0\}$ in case of regex without start anchor,

$Active \leftarrow (\bigcup_{x \in Active} Follow'(\overline{RE},x)) \bigcap Enter[\sigma]$ in case of regex with a start anchor.

As before, the new set of *Active* states is the union of the sets of positions reachable from currently active positions intersected with states reachable with the current input symbol, *σ*.

## 6.5. Bit Parallelism

### 6.5.1. Standard G-NFA

The G-NFA maps quite elegantly to a BP representation. The equivalent DFA states are stored in a bitmask of length *m*+1, where each bit corresponds to an NFA state and a particular bit has value 1 if that NFA state belongs to the DFA state. In other words, each DFA state corresponds to a set of NFA states which are indicated by the bits set to 1 in the DFA state bitmask. This is illustrated in Table 21 for *m*=2.

**Table 21. Mapping between DFA state bitmask and equivalent NFA states**

| *DFA state index* | *DFA state bitmask* | *Set of equivalent NFA states* |
|---|---|---|
| 0 | 001 | {0} |
| 1 | 010 | {1} |
| 2 | 011 | {0,1} |
| 3 | 100 | {2} |
| 4 | 101 | {0,2} |
| 5 | 110 | {1,2} |
| 6 | 111 | {0,1,2} |

The following is a brief description of the bit-parallel algorithm for the standard G-NFA, full details of which are explained by Navarro and Raffinot (2002) and Lee (2009).

The main decision in this algorithm is how best to represent the set, $Follow(\overline{RE},x)\forall x \in \{0..m\}$, where $x$ is the marked regex position index (or equally the NFA state number) and $m$ is the number of positions in the marked regex. The most compact representation, as illustrated in Table 22 for the example G-NFA from Figure 50, is a table of $m+1$ rows, where each row is a bitmask of length $m+1$ representing $Follow(\overline{RE},x)$ and $x$ is the row index. The problem with this table is that in the worst case scenario, when $m+1$ states are currently active, $m+1$ memory accesses are required per input character, i.e. slow speed.

**Table 22. Follow Table indexed by NFA state index for $\overline{RE} = ((A_1B_2A_3|C_4)B_5{}^*)A_6$**

| *NFA state index* | *$Follow(\overline{RE},x)$ set* | *$Follow(\overline{RE},x)$ bitmask* |
|---|---|---|
| | | 6543210 |
| 0 | {1,4} | 0010010 |
| 1 | {2} | 0000100 |
| 2 | {3} | 0001000 |
| 3 | {5,6} | 1100000 |
| 4 | {5,6} | 1100000 |
| 5 | {5,6} | 1100000 |
| 6 | {Ø} | 0000000 |

At the other extreme, a table, `FOLLOW_ACTIVE`, could be created with $2^{m+1}$ rows, where each row is a bitmask of length $m+1$ representing $Follow(\overline{RE}, Active)$ and *Active* is the row index. This table only needs one access per input character, but occupies a huge amount of memory for practical values of *m*. There is therefore the usual trade-off between memory and speed with the best solution being a hybrid of the above two approaches. The size of the `FOLLOW_ACTIVE` table can be reduced using a *horizontal partitioning* scheme (Wu and Manber, 1992; Navarro and Raffinot, 2002, p.119). Table 24 shows how the example table from Table 23 can be split into two much shorter tables. The overall FOLLOW_ACTIVE bitmask is found by ORing together the values obtained from the two separate tables.

**Table 23. `FOLLOW_ACTIVE` indexed by *Active* bitmask for $\overline{RE} = ((A_1B_2A_3|C_4)B_5{*})A_6$**

| *Active* bitmask<br>6543210 | `FOLLOW_ACTIVE`[*Active*]<br>6543210 |
|---|---|
| 0000000 | 0000000 |
| 0000001 | 0010010 |
| 0000010 | 0000100 |
| 0000011 | 0010110 |
| 0000100 | 0001000 |
| 0000101 | 0011010 |
| ... | |
| 1111111 (127 decimal) | 1111110 |

**Table 24. Horz. partitioning by 2 of `FOLLOW_ACTIVE` for $\overline{RE} = ((A_1B_2A_3|C_4)B_5{*})A_6$**

| *Active[3..0]* bitmask<br>3210 | `FOLLOW_ACTIVE`[*Active[3..0]*]<br>654 3210 | *Active[6..4]* bitmask<br>7654* | `FOLLOW_ACTIVE`[*Active[6..4]*]<br>654 3210 |
|---|---|---|---|
| 0000 | 000 0000 | 0000 | 000 0000 |
| 0001 | 001 0010 | 0001 | 110 0000 |
| 0010 | 000 0100 | 0010 | 110 0000 |
| 0011 | 001 0110 | 0011 | 110 0000 |
| ... | ... | ... | ... |
| 1111 (15 decimal) | 111 1110 | 1111 | 110 0000 |

*Extra bit added in order to have two equally sized partitions

The standard BPGlushkov algorithm consists of the following stored data for each regex. The bitmask values given are for an 8-bit BP implementation, with example RE=*((ABA|C)B*)A*.

- `ACTIVE` bitmask representing the set of currently active states, *Active*, and is updated after the processing of each symbol. The Least Significant Bit (LSB) of this bitmask is set to 1 prior to reading the first character, to represent initial state 0. In the case of a regex without a start anchor, this LSB will be held asserted throughout the processing of the packet in order to detect the pattern at any position in the payload.

- LAST bitmask representing the set, $Last(\overline{RE})$, i.e. the 'accept' or final states of the G-NFA. In the G-NFA for the example, RE=((ABA|C)B*)A, there is only one 'accept' state. LAST has the binary value 0100 0000. The ACTIVE and LAST bitmasks are ANDed together after processing each symbol, and a non-zero result indicates a match has been found.
- ENTER table, with a row for each symbol, σ, in the alphabet of the regex, where each row corresponds to the set Enter[σ]. The table has $2^8$ = 256 entries, one for each 8-bit input symbol, where each entry represents the states reachable on processing of a symbol with value equal to the entry's index. The ENTER table for the example, *RE*=((ABA|C)B*)A, is shown in Table 25.

**Table 25. ENTER table for** $\overline{RE} = ((A_1B_2A_3|C_4)B_5{}^*)A_6$

| *Symbol. σ* | *Symbol in hex (8 bits)* | *Enter[σ] set* | ENTER[σ] *bitmask* 7654 3210 |
|---|---|---|---|
| A | 41 | {1,3,6} | 0100 1010 |
| B | 42 | {2,5} | 0010 0100 |
| C | 43 | {4} | 0001 0000 |
| | all other hex values between 00 and FF | {Ø} | 0000 0000 |

On reading a new payload symbol, the algorithm looks up the corresponding entry in the ENTER table to find which states can be reached with that particular symbol. e.g. If the input symbol is B, then the ENTER bitmask is 0010 0100 has bits numbered 2 and 5 set to 1, which means that states 2 and 5 are reachable with the symbol B.

- FOLLOW_ACTIVE table, with a row for each possible value of the ACTIVE bitmask. Such a table occupies $(m+1)2^{m+1}$ bits, which is usually too large. This can be reduced by horizontally splitting it into k tables, giving a total occupied space of $k(m+1)2^{(m+1)/k}$ bits. If there are s bits of available memory then k can be calculated as follows

$$s = k(m+1)2^{(m+1)/k}$$

$$\Rightarrow \log_2 \frac{s}{k(m+1)} = \frac{m+1}{k}$$

⇨ $k = \frac{(m+1)}{\log_2 s - \log_2(k(m+1))}$

⇨ $k \approx \frac{(m+1)}{\log_2 s - \log_2(m+1)}$ if $k \ll m$

⇨ $k \approx \frac{(m+1)}{\log_2 s}$ if $m \ll s$

Horizontal partitioning for the example, RE=((ABA|C)B*)A, is shown in Table 24.

On reading a new payload symbol, the algorithm uses the current value of the ACTIVE bitmask to index the FOLLOW_ACTIVE table in order to find which states can be reached.

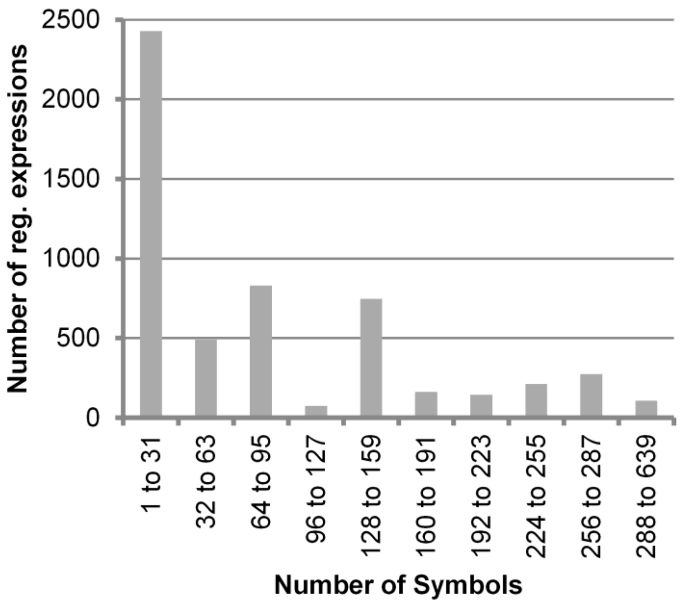
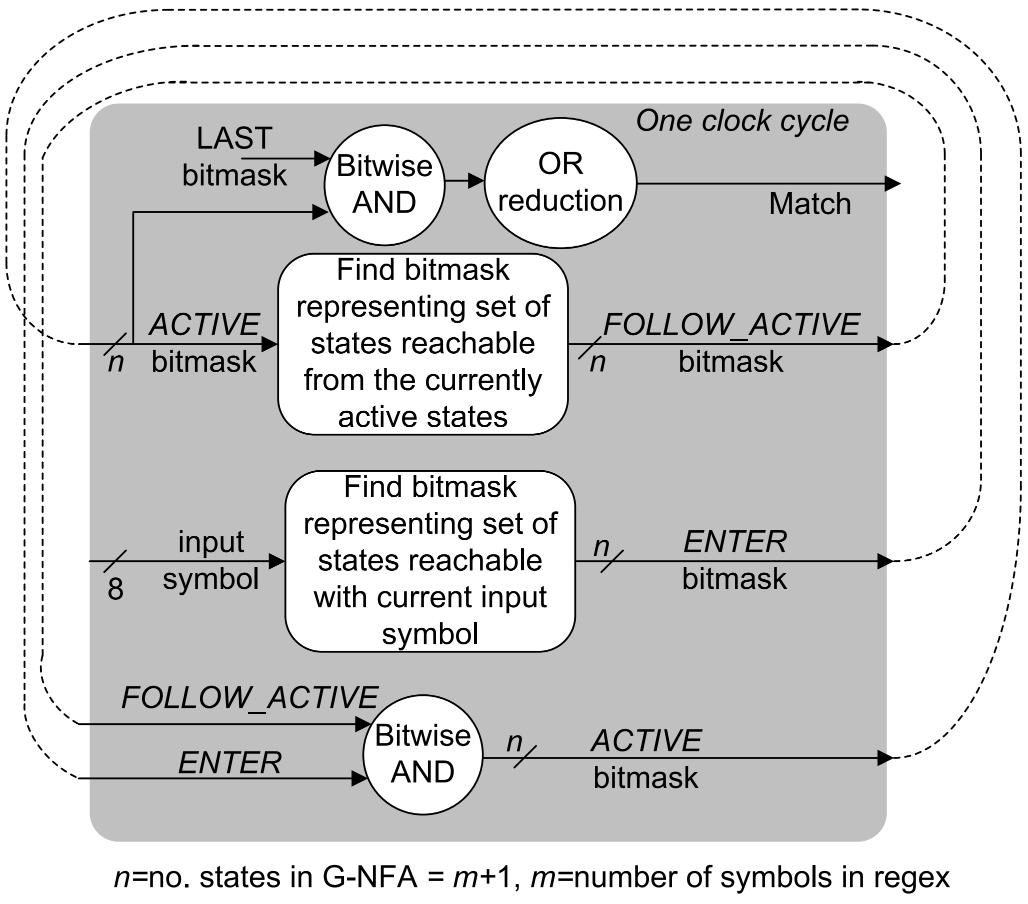


**Figure 55: SNORT regular expression symbol count distribution**
(Snort 2.9.3.1, rule snapshot from 18.09.2012. Back-references & subroutine sub-patterns were not considered)

Figure 56 shows a simple high level view of the functionality of the BP G-NFA algorithm. As shown in Figure 55**Error! Reference source not found.**, a very large roportion of SNORT regexes consist of less than 32 symbols. These can be represented by a 32-bit bitmask. The search algorithm for *m*=31 and *k*=4 is then as follows

```
ACTIVE ← 0x00000001 /*(0x denotes hex) Set LSB of 32-bit mask to
1*/
FOR σ ∊ Σ DO /* repeat for each input symbol (element of
alphabet)*/
  FOLLOW_SUM ←
     FOLLOW_ACTIVE[3][ACTIVE(31..24)] |
     FOLLOW_ACTIVE[2][ACTIVE(23..16)] |
     FOLLOW_ACTIVE[1][ACTIVE(15..8)] |
     FOLLOW_ACTIVE[0][ACTIVE(7..0)]
```

```
   ACTIVE ← FOLLOW_SUM & ENTER[σ]
   IF (ACTIVE & LAST)
         REPORT MATCH FOUND
         EXIT
   END IF
END FOR
REPORT NO MATCH FOUND
```



One clock cycle
LAST bitmask
Bitwise AND
OR reduction
Match
ACTIVE bitmask
n
Find bitmask representing set of states reachable from the currently active states
FOLLOW_ACTIVE bitmask
n
input symbol
8
Find bitmask representing set of states reachable with current input symbol
n
ENTER bitmask
FOLLOW_ACTIVE
ENTER
Bitwise AND
n
ACTIVE bitmask
n=no. states in G-NFA = m+1, m=number of symbols in regex

**Figure 56. High level view of BP G-NFA algorithm**

The bitmasks retrieved from the ENTER and FOLLOW_ACTIVE tables are ANDed together to find the new value of the ACTIVE bitmask. This can be seen from Table 26 for the example regex, RE=*((ABA|C)B*)A*. As the regex has no start anchor, then bit 0 is held asserted.

**Table 26. Input string *ABABBA* with RE=*((ABA|C)B*)A* (no anchor)**

| Inp. Sym. | ENTER 7654 3210 | FOLLOW_ ACTIVE [ACTIVE[7..4]] 7654 3210 | FOLLOW_ ACTIVE [ACTIVE[3..0]] 7654 3210 | FOLLOW_ ACTIVE 7654 3210 | ACTIVE 7654 3210 | ACTIVE *AND* LAST 7654 3210 |
|---|---|---|---|---|---|---|
| | | | | | 0000 0001 | 0000 0000 |
| A | 0100 1010 | 0000 0000 | 0001  0010 | 0001 0010 | 0000 0011 | 0000 0000 |
| B | 0010 0100 | 0000 0000 | 0001  0010 | 0111 0010 | 0000 0111 | 0000 0000 |
| A | 0100 1010 | 0000 0000 | 0001 1110 | 0001 1110 | 0000 1011 | 0000 0000 |
| B | 0010 0100 | 0000 0000 | 0111 0110 | 0111 0110 | 0010 0101 | 0000 0000 |
| B | 0010 0100 | 0110 0000 | 0001 1010 | 0111 1010 | 0010 0001 | 0000 0000 |
| A | 0100 1010 | 0110 0000 | 0001 0010 | 0111 0010 | 0100 0001 | 0100 0000 Match found! |

### 6.5.2. Counting G-NFA

This section explains how the standard bit parallel algorithm can be extended to implement the counting G-NFA by using a bitmask representation of the sets explained earlier. Each counting block has the following storage elements:

- `MIN`: integer value, min. number of repetitions
- `MAX`: integer value, max. number of repetitions
- `POS_CNT`: bitmask indicating states contained in the counting block
- `CNT`: integer value, number of times the pattern has been seen
- `INCREMENT_BLK`: bitmask indicating states where counter value can be incremented
- `FIRST_BLK`: bitmask indicating the entry states of the counting block
- `FINAL_BLK`: bitmask indicating the exit states of the counting block, assuming counter has reached the required value
- `FOLLOW_BLK`: bitmask indicating states outside the counting block which can be reached from within the block

The following is the BP counting G-NFA search algorithm for *m*=31, *k*=4

```
ACTIVE ← 0x00000001 /* Set LSB of 32-bit bitmask to 1 */
FOR σ ∊ Σ DO /* repeat for each input symbol */
  FOLLOW_SUM ← 0x00000000 /* Zero 32-bit bitmask */
  /* Consider each of the c counting blocks */
  FOR i ∊ 1...c /* where c=no. of counting blks */
  ACTIVE_IN_CNT_BLK ← ACTIVE & POS_CNT(i)
    IF (ACTIVE_IN_CNT_BLK) /* Counting blk state active? */
      /* Are we in one of the states where counter *
       * must be incremented                       */
      IF (ACTIVE_IN_CNT_BLK & INCREMENT_BLK(i))
        CNT(i) ← CNT(i) + 1
      END IF
      /* Are we in one of the states where we need to check  *
       * if we can repeat or transition out of counting blk? */
      IF (ACTIVE_IN_CNT_BLK & FINAL_BLK(i))
         IF (CNT(i) ≥ MIN(i)) /* Can we move onto next stage? */
            /* Allow transitions out of counting block */
            FOLLOW_SUM ← FOLLOW_SUM | FOLLOW_BLK(i)
         END IF
         IF  (CNT(i)  <  MAX(i))  /*Can  we  repeat  this  counting
      block?*/
            /* Allow transitions back to counting *
             * block's initial state(s)           */
            FOLLOW_SUM ← FOLLOW_SUM | FIRST_BLK(i)
         END IF
      ENDIF
   ELSE
      /* No state active in this counting block,
       * so reset the counter
       */
      CNT(i) ← 0
   END IF
END FOR
/* Combine the set of follow states calculated in the counting
*
 * block section above with the standard horz.
*
 * partitioned FOLLOW_ACTIVE values
*/
FOLLOW_SUM ← FOLLOW_SUM |
   FOLLOW_ACTIVE[3][ACTIVE(31..24)] |
   FOLLOW_ACTIVE[2][ACTIVE(23..16)] |
   FOLLOW_ACTIVE[1][ACTIVE(15..8)] |
   FOLLOW_ACTIVE[0][ACTIVE(7..0)]
/* Calculate new set of active states by ANDing together   *
 * the bitmask of states reachable from the currently      *
 * active states with the bitmask of states reachable      *
 * with the current input symbol                           */
ACTIVE ← FOLLOW_SUM & ENTER[σ]
IF (ACTIVE & LAST) /* Is one of the Accept states active? */
   REPORT MATCH FOUND
   EXIT
END IF
END FOR
REPORT NO MATCH FOUND
```

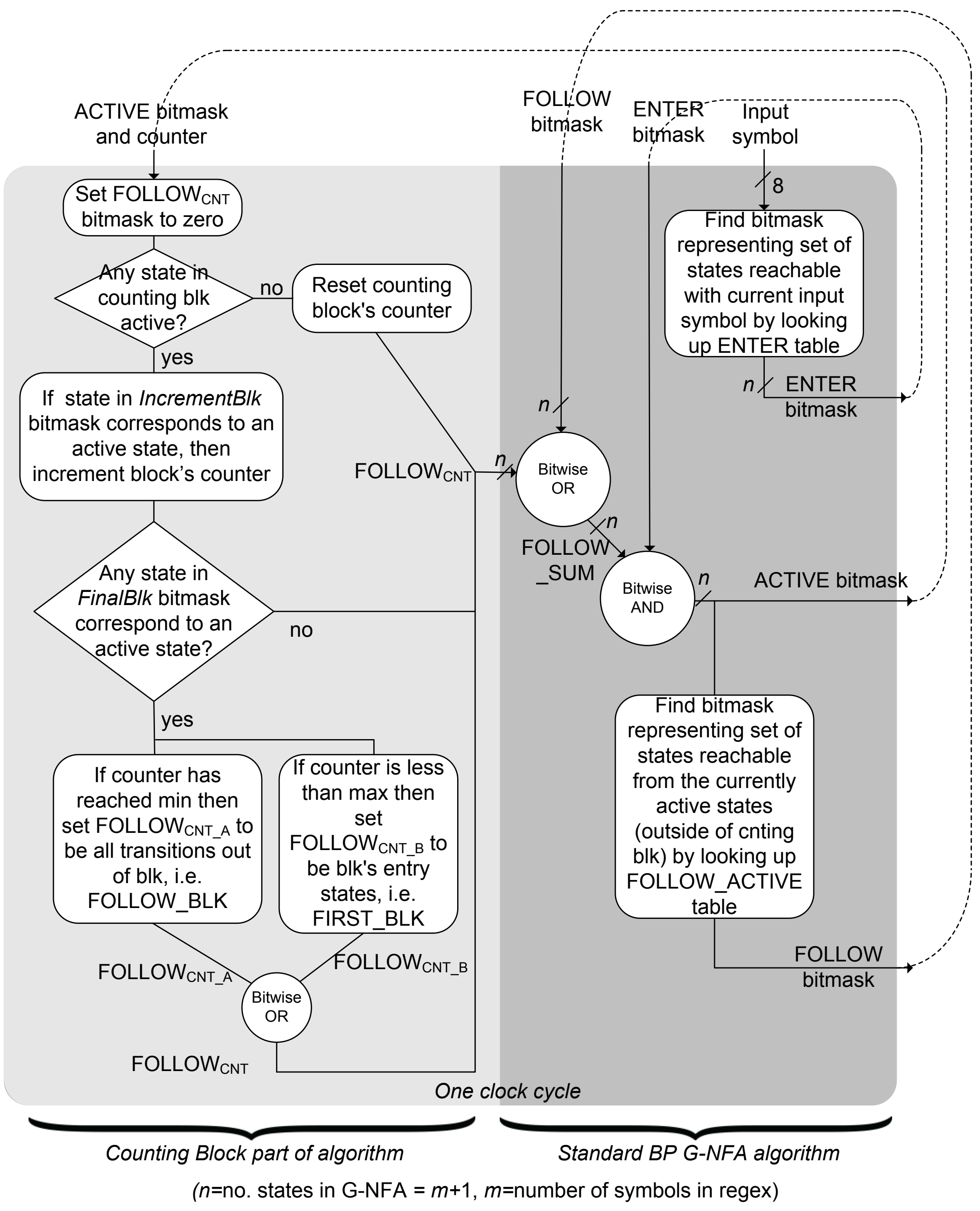

ACTIVE bitmask and counter
FOLLOW bitmask
ENTER bitmask
Input symbol
8
Set FOLLOW$_{CNT}$ bitmask to zero
Any state in counting blk active?
no
Reset counting block's counter
yes
If state in *IncrementBlk* bitmask corresponds to an active state, then increment block's counter
Find bitmask representing set of states reachable with current input symbol by looking up ENTER table
n
ENTER bitmask
n
FOLLOW$_{CNT}$
n
Bitwise OR
n
FOLLOW_SUM
Bitwise AND
n
ACTIVE bitmask
Any state in *FinalBlk* bitmask correspond to an active state?
no
yes
Find bitmask representing set of states reachable from the currently active states (outside of cnting blk) by looking up FOLLOW_ACTIVE table
If counter has reached min then set FOLLOW$_{CNT\_A}$ to be all transitions out of blk, i.e. FOLLOW_BLK
If counter is less than max then set FOLLOW$_{CNT\_B}$ to be blk's entry states, i.e. FIRST_BLK
FOLLOW bitmask
FOLLOW$_{CNT\_A}$
FOLLOW$_{CNT\_B}$
Bitwise OR
FOLLOW$_{CNT}$
One clock cycle
Counting Block part of algorithm
Standard BP G-NFA algorithm
($n$=no. states in G-NFA = $m$+1, $m$=number of symbols in regex)

**Figure 57. High level view of BP Counting G-NFA algorithm**

Figure 57 illustrates how the counting scheme is combined with the standard BP G-NFA to produce the overall BP counting G-NFA algorithm. The large grey box represents the functionality that executes in one clock cycle of operation. Although in practice there can be multiple counting blocks, this simplified diagram just shows one. Comparison of the `ACTIVE` and `LAST` bitmasks in order to find a match has been omitted for clarity.

### 6.5.3. Counting G-NFA for single symbol elements

The above algorithm can be simplified if the repeated element is just a single symbol. The `INCREMENT_BLK`, `FIRST_BLK` and `FINAL_BLK` bitmasks are not required, as `POS_CNT` indicates the single state contained in the counting block. Figure 58 illustrates this simplification of the algorithm.

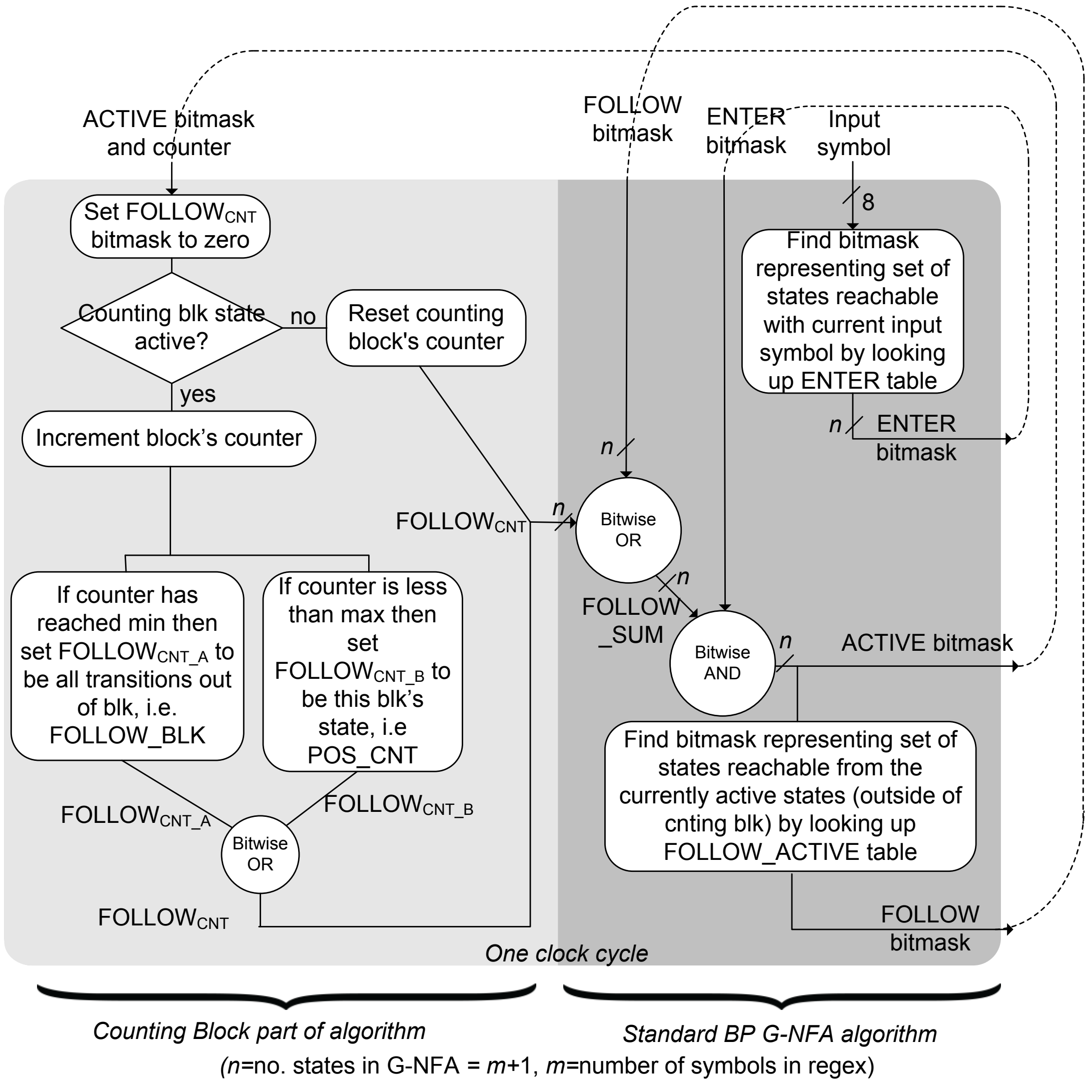

ACTIVE bitmask and counter
Set FOLLOW$_{CNT}$ bitmask to zero
Counting blk state active?
no
Reset counting block's counter
yes
Increment block's counter
If counter has reached min then set FOLLOW$_{CNT\_A}$ to be all transitions out of blk, i.e. FOLLOW_BLK
If counter is less than max then set FOLLOW$_{CNT\_B}$ to be this blk's state, i.e POS_CNT
FOLLOW$_{CNT\_A}$
FOLLOW$_{CNT\_B}$
Bitwise OR
FOLLOW$_{CNT}$
FOLLOW$_{CNT}$
FOLLOW bitmask
ENTER bitmask
Input symbol
8
Find bitmask representing set of states reachable with current input symbol by looking up ENTER table
n
ENTER bitmask
n
n
Bitwise OR
n
FOLLOW _SUM
Bitwise AND
n
ACTIVE bitmask
Find bitmask representing set of states reachable from the currently active states (outside of cnting blk) by looking up FOLLOW_ACTIVE table
FOLLOW bitmask
One clock cycle
Counting Block part of algorithm
Standard BP G-NFA algorithm
(n=no. states in G-NFA = m+1, m=number of symbols in regex)

**Figure 58. High level view of Counting G-NFA for single symbol repetition**

## 6.6. Implementation

### 6.6.1. Hardware Architecture

The target platform is the open-source NetFPGA-10G development PCI board based on the Xilinx Virtex-5 TX240T FPGA. This FPGA can operate at up to 550MHz and has significant internal memory in the form of 648 block RAMs of size 18Kb, giving a total of 11,664Kb, and 2,400Kb of distributed RAM. The aim is to make efficient use of this internal memory so as to maximize performance.

The architecture, as illustrated in Figure 59, reuses several components of the NetFPGA reference pipeline (Naous et al., 2008). This pipeline has two buses, a 64-bit wide packet bus and a register bus. Software running on the host can access the registers using *ioctl* calls. The register bus allows indirect modification of the contents of the block RAM used by the DPI implementation. The NetFPGA ethernet MAC queues, input arbiter, output port lookup and memory interface controller Verilog modules, are reused and are summarised as follows:

- Ethernet MAC Queues – each ethernet MAC has a corresponding block RAM FIFO
- Input Arbiter – selects which RX queue to service next
- Output port lookup – longest prefix matching on destination IP address to determine the output ethernet port
- Memory Interface Controller – NetFPGA project includes modules for interfacing to SRAM and DRAM

The following new modules are added:

- Fixed String DPI – carries out fixed string matching on the packet payload
- regex DPI – carries out regex matching on the packet payload
- Multi-match header classification – classifies packet headers against IDS ruleset
- Match Decision – combines results from the three NIDS classification modules. If any of the three matching modules reports no match found for a particular packet, then the matching can be terminated in the other two modules and processing of the next packet initiated
- Packet Buffer Control – each incoming packet is buffered in memory until a match decision is made. Once the current packet matching has completed, the next packet header and payload is passed to the matching modules
- Note that TCP/IP reassembly is not handled in this architecture

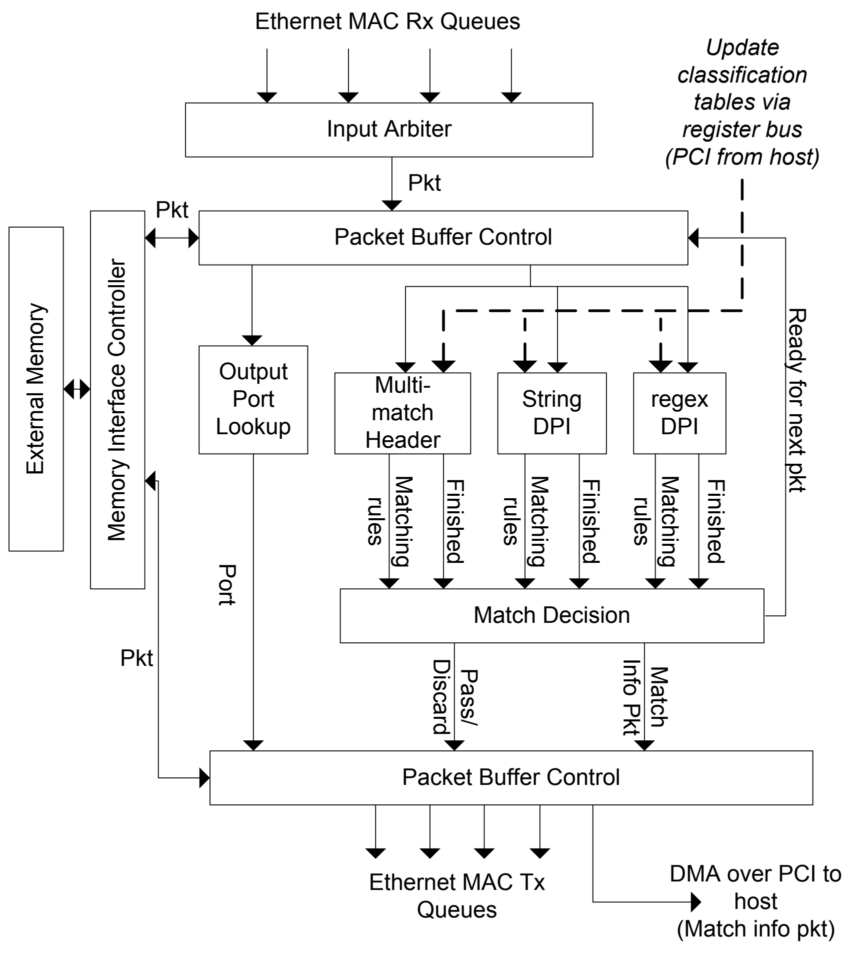


**Figure 59: Proposed overall IDS architecture on NetFPGA platform**

The Regex DPI module, as illustrated in Figure 60, consists of many regex processing engines (PEs) operating in parallel, each handling a single regex which is stored in simple dual-port block RAM. The regex data can be dynamically updated by software running on an internal or external CPU. The maximum number of engines is limited by the available memory, logic and interconnects on the FPGA.

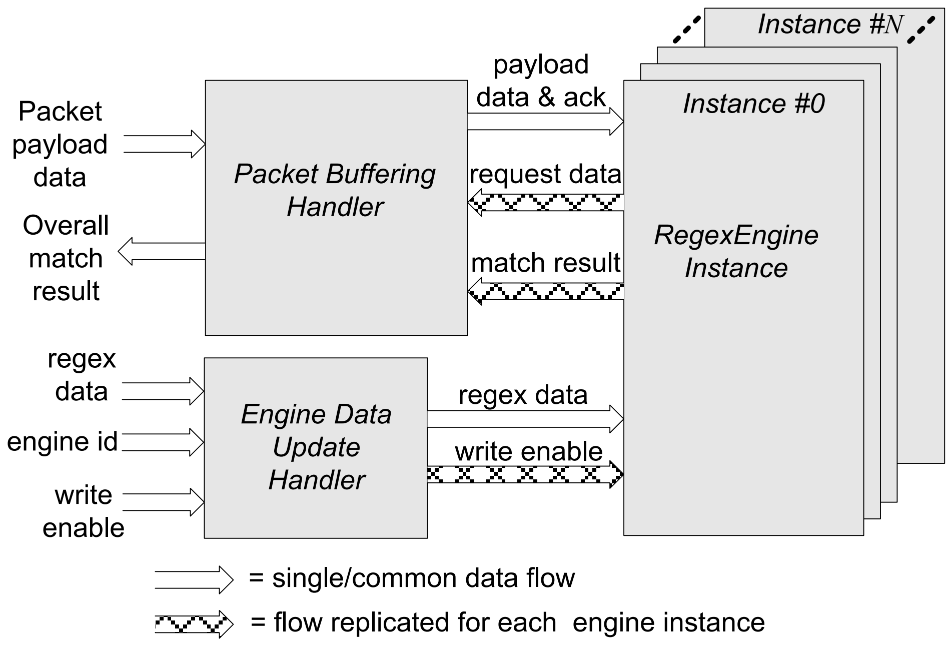


**Figure 60: Regex DPI Handling Module hardware architecture**

The Engine Data Update Handler module is responsible for writing the regex data to the PE identified by the *engine id* input. This essentially involves decoding the *engine id* and asserting the appropriate *write enable* output.

The Packet Buffer Handler module is responsible for buffering received payload symbols and handling handshaking communication with all PEs. It also monitors the *match* and *no_match* signals from all PEs in order to declare an overall match result.

The Packet Buffer Handler module implemented in order to evaluate the design is shown in Figure 61. The payload data input is a 10-bit wide input signal consisting of 8 bits of *symbol* data, a *first*-symbol-in-packet flag and a *last*-symbol-in-packet flag. Buffer memory consists of two Block RAM (BRAM)–based FIFOs, each capable of holding 2048 10-bit symbols. Each FIFO contains only the payload data from one particular packet. Once a match is found for a particular packet, the remaining content of that packet is discarded by simply resetting the associated FIFO.

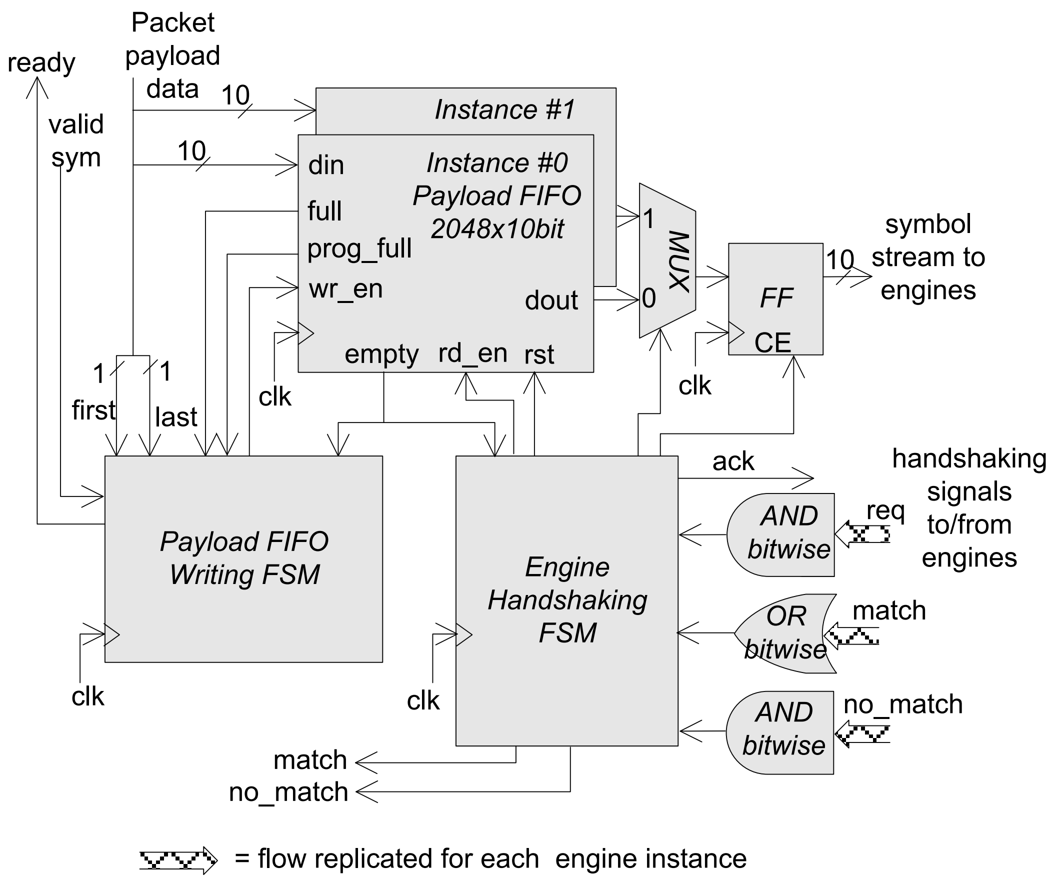


**Figure 61: Payload Buffer scheme used in evaluation**

The Payload FIFO Writing FSM is responsible for writing received symbols into the appropriate FIFO. If the *prog_full* signal indicates that the FIFO is almost full, then the FSM will deassert the *ready* output signal. Once the last symbol of the payload has been written to a FIFO, it switches to writing to the other FIFO. If that FIFO is not empty, then it will deassert the *ready* flag and pause until it is.

The Engine Handshaking FSM handles communication with all PEs and selects which payload FIFO to read from. The same payload symbols are fed to all PEs concurrently. The FSM must therefore wait for all PEs have asserted their *req* output signals, indicating that they are ready, before sending symbols from the FIFO to the PEs. Similarly, the FSM will only declare that no match has been found if all PEs assert their *no_match* outputs. It asserts the *ack* signal to indicate a valid *symbol* output signal. If any PE asserts its *match* output, the FSM will assert the overall *match* output signal and reset the FIFO in order to remove any remaining content from the successfully matched packet. The FSM switches FIFO once the overall *match* or *no_match* output has been asserted in order to read in the next packet to be processed by the PEs.

A simplified version of the logic present in each PE is illustrated in Figure 62 and a schematic for the counting block mechanism it uses is shown in Figure 63. This logic is a single symbol implementation of the algorithm outlined earlier. The logic used to handle regex features such as anchors and multi-line mode has been omitted for clarity. Each engine requires one BRAM for the ENTER table and one for each of the horizontally partitioned FOLLOW_ACTIVE tables. In the case of $n$=32, four BRAMs are required for a 4-way partition of the FOLLOW_ACTIVE table. Each Virtex5 BRAM can be configured to be simple dual port of size 512x32, i.e. each 32-bit wide dual port BRAM has 512 rows. *Simple* dual port mode means that one port is used for reading and the other for writing, i.e. one port cannot be used for both. However, each engine only uses 256 rows per BRAM. Therefore, so as to maximise space efficiency, a quad-port "wrapper" can be added around each BRAM, where each port accesses one half of the memory. This is achieved by time division multiplexing the access to the BRAM which means the normally dual port memory is clocked at twice the rate of the rest of the design. This has the disadvantage of reducing the maximum frequency of operation of the design by 50%, as well as adding some additional logic. So it boils down to the usual time-space trade-off. Xilinx provides an application note on how to implement a quad-port BRAM (Sawyer and Defossez, 2002).

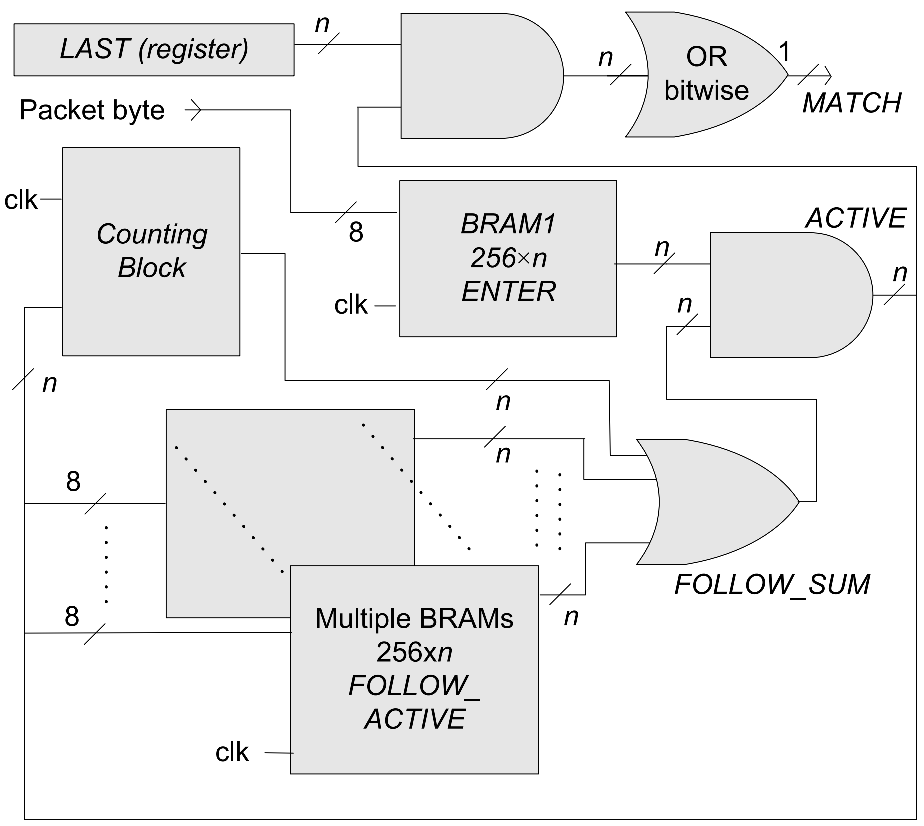


**Figure 62: Outline of Regex Processing Engine (PE)**
(for regex with up to n-1 symbols)

A single symbol is processed by a PE in each clock cycle. The input to the counting mechanism, as illustrated in Figure 63, is the current *active* bitmask. A 12-bit counter is used to in order to support a maximum counter value of 4095, the maximum value of *max* found in the regexes extracted from the Snort rule set used for the evaluation. The counting mechanism is active when the counting block state is active and, while active, its counter is incremented in each clock cycle. Otherwise the counter is cleared to zero. The output of the counting mechanism is the *n*-bit $FOLLOW_{CNT}$ bitmask that can have the following bits set:

a) No bits set because the counter value is zero.
b) Bit corresponding to counting block state, indicating that repetition is enabled. This occurs if the counter value is less than *max*.
c) Bits corresponding to states, outside of the block, that are reachable from the counting block state, i.e. allowing transition out of the block. This occurs if the counter value is at least *min*.
d) A combination of (b) and (c) if the counter is at least *min* and less than *max*.

The reader should refer to Figure 58 for a higher level view of the functionality outlined in Figure 63.

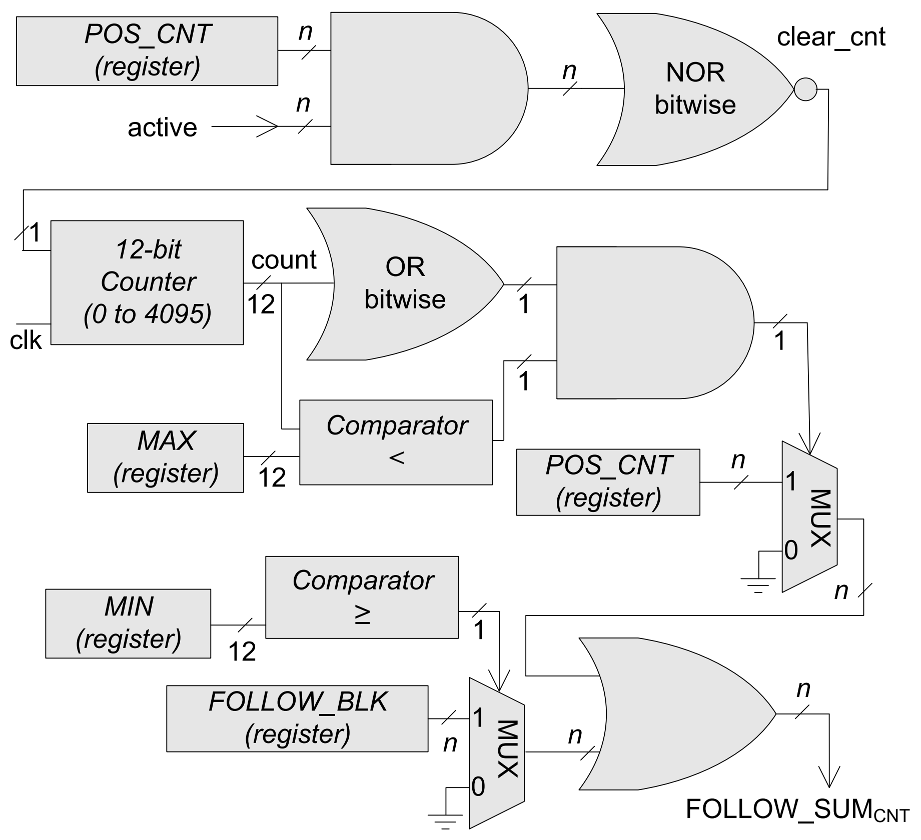


**Figure 63: Counting Block Mechanism for regex with up to *n*-1 symbols**

### 6.6.2. Bitmask Generation Software

Pre-processing software rewrites certain constrained quantifiers, as listed in Table 27, so that they can be handled by the counting G-NFA. *R{0,max}* needs to be rewritten because the counting block cannot handle a min of zero. *R{num}* at the end of a regex is rewritten as *R{num-1,num-1}R* as the regex accept state needs to be outside of the counting block.

**Table 27. Rewriting of constrained quantifiers**

| Original | Rewritten |
|---|---|
| *R{num}* | *R{num,num}* |
| *R{min,}* | *R{min,min}R** |
| *R{,max}* or *R{0,max}* | *R?\|R{1,max}* |
| *R{num}* located at end of regex | *R{num-1,num-1}R* |

Software is required to convert the Snort regexes into the Counting G-NFA and corresponding bitmasks. The CCP C program developed by Champarnaud et al. (2004) performs pattern matching based on the G-NFA. This software consists of two principal modules, the first generates the automaton from the user supplied regex pattern and the second uses the automaton to scan a user supplied text file for occurrences of the pattern. The first module was extended to handle most of the PCRE syntax found in Snort and a layer added to handle the conversion of the automata into tables of bitmasks for storage in BRAM. This layer also takes account of the single line mode and case insensitivity modifiers when generating the bitmasks. The anchor and multi-line mode modifier values, on the other hand, are programmed into a flags register in each PE so as to control its operation.

Consider the example regex /abc{2,3}defghi/. The modified CCP algorithm will parse this regex and generate an `ENTER` and a `FOLLOW` table, `LAST` bitmask = 0x00000400, in addition to extracting details of the counting block, i.e. `POS_CNT` = 0x00000008 and `FOLLOW_BLK` = 0x00000010 bitmasks, `MIN`=2 and `MAX`=3 counter values. The generated `ENTER` table is indexed by each possible 8 bit symbol value and the `FOLLOW` table by each state's position, as outlined in Table 28.

**Table 28. ENTER and FOLLOW tables as generated by modified CCP software**

| *Input Symbol* | *ENTER* |
|---|---|
| 0x00 | 0x00 00 00 00 |
| ...etc... | 0x00 00 00 00 |
| 0x61 (ASCII ‘a’) | 0x00 00 00 02 |
| 0x62 (ASCII ‘b’) | 0x00 00 00 04 |
| 0x63 (ASCII ‘c’) | 0x00 00 00 08 |
| 0x64 (ASCII ‘d’) | 0x00 00 00 10 |
| 0x65 (ASCII ‘e’) | 0x00 00 00 20 |
| 0x66 (ASCII ‘f’) | 0x00 00 00 40 |
| 0x67 (ASCII ‘g’) | 0x00 00 00 80 |
| 0x68 (ASCII ‘h’) | 0x00 00 01 00 |
| 0x69 (ASCII ‘i’) | 0x00 00 02 00 |
| .... etc... | 0x00 00 00 00 |

| State position | FOLLOW |
|---|---|
| 0 | 0x00 00 00 02 |
| 1 | 0x00 00 00 04 |
| 2 | 0x00 00 00 08 |
| 3 | 0x00 00 00 00 |
| 4 | 0x00 00 00 20 |
| 5 | 0x00 00 00 40 |
| 6 | 0x00 00 00 80 |
| 7 | 0x00 00 01 00 |
| 8 | 0x00 00 02 00 |
| 9 | 0x00 00 04 00 |
| 10 | 0x00 00 00 00 |

**Table 29. FOLLOW_ACTIVE tables**

| ACTIVE bits 7 to 0 | FOLLOW_ACTIVE [0] |
|---|---|
| 0x00 | 0x00 00 00 00 |
| 0x01 | 0x00 00 00 02 |
| 0x02 | 0x00 00 00 04 |
| 0x03 | 0x00 00 00 06 |
| 0x04 | 0x00 00 00 08 |
| 0x05 | 0x00 00 00 00 |
| 0x06 | 0x00 00 00 0C |
| 0x07 | 0x00 00 00 0E |
| 0x08 | 0x00 00 00 00 |
| 0x09 | 0x00 00 00 02 |
| etc... | |
| 0x80 | 0x00 00 01 00 |
| 0x81 | 0x00 00 01 02 |
| etc... | |
| 0xff | 0x00 00 01 EE |

| ACTIVE bits 15 to 8 | FOLLOW_ACTIVE [1] |
|---|---|
| 0x00 | 0x00 00 00 00 |
| 0x01 | 0x00 00 02 00 |
| 0x02 | 0x00 00 04 00 |
| 0x03 | 0x00 00 06 00 |
| 0x04 | 0x00 00 00 00 |
| 0x05 | 0x00 00 02 00 |
| 0x06 | 0x00 00 04 00 |
| 0x07 | 0x00 00 06 00 |
| etc... | |
| 0xff | 0x00 00 06 00 |

The new layer of software added to CCP then converts the `FOLLOW` table into four horizontally partitioned `FOLLOW_ACTIVE` tables, two of which are shown in Table 29. The final step is to convert all tables and bitmasks into a format suitable for uploading into FPGA memory. All the data is stored in a binary file based on the `VHDL STD_LOGIC` data type (Tang, 2000) which is suitable for reading by a VHDL test bench. This file is also suitable for reading by a microprocessor for programming of the DPI core via memory-mapped I/O.

One serious problem with programming the BRAMs via the test bench is that it results in an excessively long simulation run time. One way of avoiding this issue is to hard code the BRAM data into the VHDL code using the `INIT_xx` generic attributes available in each Xilinx BRAM primitive (Xilinx, 2012a). Therefore, for simulation purposes, only non-BRAM data is stored in the binary data file that is loaded dynamically via the test bench, i.e. bitmasks such as `FIRST`, `FINAL`, `POS_CNT`, etc. In the case of BRAM data, i.e. `ENTER` and `FOLLOW_ACTIVE` tables, a VHDL package file is generated which contains an array indexed by the regex engine identifier. Each element of this array contains an array of five elements, where each element represents the contents of a single BRAM. Each of these BRAM data elements consists of an array of bit vectors, where each vector corresponds to an `INIT_XX` attribute.

## 6.7. Performance Results

### 6.7.1. Synthesis and simulation

The design was implemented in VHDL for a bitmask length of 32 with the tables horizontally partitioned four times. Simulation and evaluation was performed with the Xilinx Virtex5 TX240T (Speed -2) as the target FPGA device. Longer bitmasks, and hence regex lengths, could have been implemented, but would occupy a significantly larger amount of memory. Doubling the bitmask length typically involves an increase in the storage requirement by a factor of 4. This is because both the number of horizontal partitions and the width of each table are doubled. It should be noted, however, that the majority of regexes are short, as shown in Figure 55.

Sets of regexes, each containing a single constrained repetition and each having less than 32 states in the Counting G-NFA, were randomly chosen from the Snort rule set. The bitmask generation software was then used to convert these regexes into tables of bitmasks for programming the FPGA BRAM. All tests were repeated with different sets of regexes and the results were found to be the same for a particular number of regexes. Payload data extracted from the Shmoo Group (2009) DEFCON traffic traces were used for the timing simulation. Power analysis was performed using the Xilinx XPower Analyzer.

**Table 30: Virtex5 TX240T Device Utilisation**

| *Resource* | *32 PEs* | *64 PEs* | *96 PEs* |
|---|---|---|---|
| *Block RAM* | 28% | 50% | 75% |
| *Slices* | 11% | 15% | 39% |

Xilinx XST tool gave the maximum clock frequency as 201.5 MHz on the Virtex5 and 220 MHz on the Virtex7 with corresponding throughputs of 1.6 Gb/s and 1.8 Gb/s, respectively. Resource utilisation figures for the Virtex5 are given in Table 30. It can support a maximum of 128 PEs due to its limited BRAM. A higher end FPGA such as the Virtex7 1140T has 5.8 times more BRAM and so could handle up to 750 PEs, as shown in Table 31.

**Table 31: Virtex7 1140T Device Utilisation**

| *Resource* | *256 PEs* | *512 PEs* | *736 PEs* |
|---|---|---|---|
| *Block RAM* | 34% | 68% | 97% |
| *Slices* | 8% | 14% | 20% |

The current VHDL design uses only half of each BRAM. BRAM utilisation could be maximised at the cost of reduced maximum processing speed by clocking the BRAM at twice the rate of the PE and performing time division multiplexed reading of the BRAM.

In order to minimise power consumption, the BRAM Write and Read Enable inputs were only asserted when updating the regexes and processing packets, respectively. This significantly reduces power consumption at lower packet throughput rates as illustrated in Figure 64 for an implementation with 96 PEs.

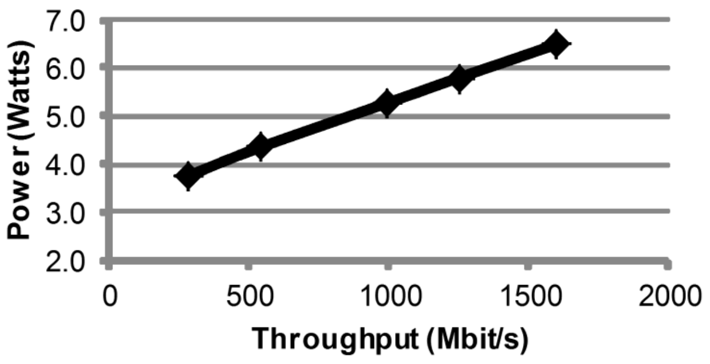


**Figure 64: Power Consumption as a function of throughput**

Power consumption was also measured with different numbers of PEs in the implementation and was found to increase linearly with the number of PEs, rising from 2.8W with 8 PEs to 6.5W with 96, measured at maximum throughput.

### 6.7.2. Memory requirements

Each regex requires 5kbytes of BRAM. 2115 regexes from the Sep. 2012 rule set are compatible with the Counting G-NFA implementation and would require a total of 10MB of BRAM. The Virtex 7 1140T has 7MB of BRAM and so two would be required to handle all compatible regexes assuming time division multiplexed reading was performed.

### 6.7.3. Memory and power savings

Consider the following regex from the Snort v2.9 rule set

*RE=\S{998}\S*

where *\S* means any non-whitespace character.

This RE requires only 3 states in the Counting G-NFA and can be easily handled in the 32-bit BP implementation which requires 5 Kbytes of BRAM per PE. Compare this to the standard G-NFA which would have 1000 states after unrolling of the repetition. A 1000-bit BP implementation with 10-bit horizontal partitioning would require 12 Mbytes of BRAM for just this regex! Clearly it is much more storage efficient to use the Counting G-NFA except for cases where the unrolled G-NFA has less than 32 states. The reduced memory results in significant power savings.

### 6.7.4. Extending to multiple counting blocks

This single counting block implementation could be extended to handle the generic 32-bit algorithm where each PE can have multiple counting blocks. For space efficiency, such an implementation would have a range of PE types, where each PE type has a different number of counting blocks. The relative numbers of each type would be determined by the typical distribution of regexes as a function of the number of constrained repetitions contained in each regex. Figure 65 shows such a distribution for the September 2012 snapshot of the Snort rule set. Table 17 showed that of the 5555 unique regexes, 1043 contained constrained repetitions. As can be seen from the chart in Figure 65, the vast majority of these contain just one constrained repetition.

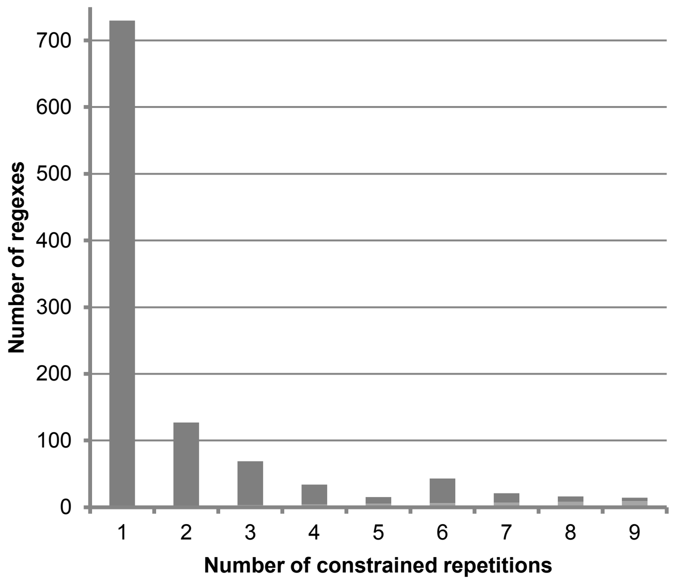


**Figure 65. Distribution of Snort regexes based on no. of constrained repetitions (v2.9.3.1, 18.09.2012 snapshot)**

## 6.8. Related Work

Existing BP algorithms (Lee, 2010; Kaneta et al., 2010) handle constrained repetitions by unrolling, which is not efficient since the SNORT rule set contains a high number of such quantifiers, many of which have high *min/max* count values. This thesis describes how a counting design can be integrated into the BP architectures proposed in existing research.

The majority of other research into the handling of constrained repetitions is based on the approach of converting the NFA into a static FPGA configuration bitstream as opposed to using memory-based architectures. The bitstream must be regenerated every time the rule set changes. Such regeneration can be quite time consuming for large rule sets, often taking several hours. Moreover, reconfiguration of the FPGA device itself will require a short system downtime, which is not ideal. Regeneration of the bitmask tables for the BP G-NFA system is performed in software for each of the modified regexes and only the modified tables are reprogrammed on the FPGA. Depending on the number of regexes to be updated, the entire procedure should be a matter of seconds.

Bispo et al. (2006) present a static FPGA configuration type system. It uses a shift register to deal with overlap but, unlike the system proposed in this thesis, it cannot handle constrained repetitions of multi-symbol sub-expressions. Faezipour and Nourani (2008) propose another static logic–based solution which adds support for multi-symbol repeated sub-expressions. They claim that they only need to deal with overlap if the repeated sub-expression is at the beginning of the regex. However overlap is also an issue when the sub-expression is located elsewhere in the regex. Yun and Lee (2009) present a similar solution but admit that it does not fully handle overlap. Long et al. (2010) add BRAM-based character matching to save logic resources.

Pao (2009) and Wang et al. (2010) carried out research into NFA-based architectures which handle constrained repetitions without unrolling. Pao's (2009) CX-NFA architecture stores its lookup tables in TCAM which typically increases cost and energy usage. Detailed evaluation figures are not provided. Wang et al. propose an NFA-based architecture, called CES (CCR regExp Scanner), which is suitable for FPGA synthesis. Its building blocks are Character Class with Constraint Repetition (CCR) modules and its operation is based on the fact that most regexes can be regarded as a sequence of character classes with repetitions, which are connected by concatenation and alternation operators, i.e. $RE = CCR_1 \bullet CCR_2 \bullet \cdots \bullet CCR_n$. Each CCR module handles the matching of a single $CCR_i$ using a MIN-MAX algorithm which uses two counters ($MIN_i$, $MAX_i$) to keep track of the minimum and maximum number of character repetitions that $CCR_i$ may have matched.  The second level of the system is the CES block or tile which consists of a mesh of interconnected CCRs as illustrated in Figure 66. By default, the interconnections are disabled and must be enabled using configurable bits in each CCR, as shown in Figure 67. A CCR can also be configured as bypassed in order to cope with an unbalanced alternation.

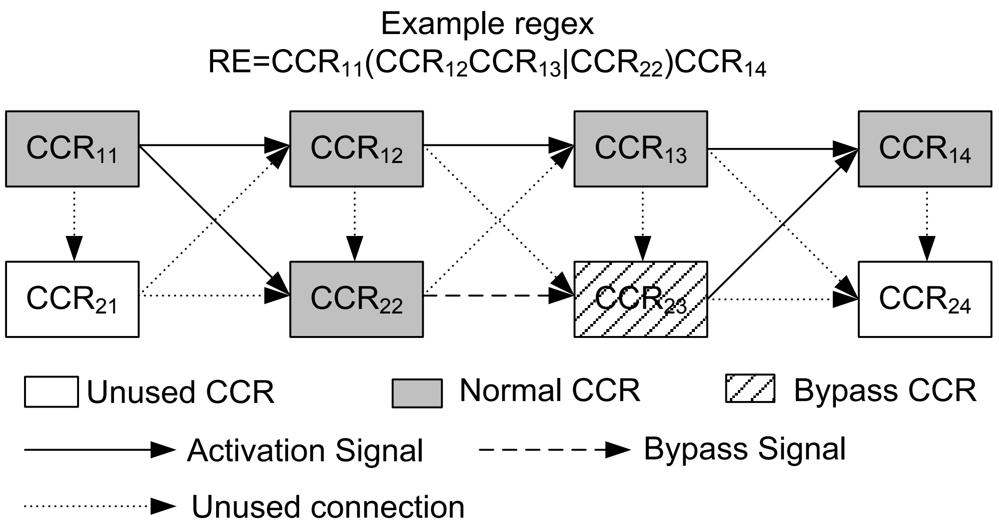


**Figure 66. CES tile mesh for example regex**

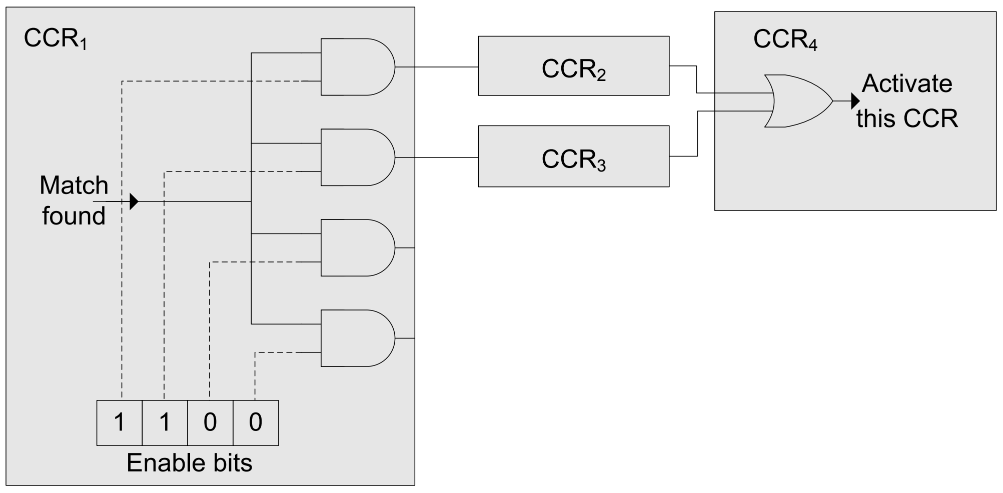


**Figure 67. CCR Interconnections**

Unlike the BP scheme proposed in this thesis, CES only uses BRAM to handle character class recognition. The paths between CCRs within each CES tile are fixed at synthesis and the interconnections are enabled or disabled using the enable bits in each CCR. Each normal CCR represents an NFA state and so the states that are reachable from those currently active are determined by the enabled connection paths between CCRs within the CES tile mesh and associated match-found signal values. A fully flexible CES tile would require an excessive number of interconnections in order to handle all possible regex concatenations and alternations. As a workaround, Wang et al. suggest that the design be synthesised with a range of CES topologies, with a portion of CES tiles optimised for each particular type of regex. The proportion of each type of CES tile would be based on an analysis of recent rule sets. Another disadvantage of the CES architecture is that it can only handle constrained repetitions of single symbol elements. Wang et al. synthesised their design for a Virtex-5 LX110T FPGA and report a clock frequency of approx. 200MHz and throughput of 1.6 Gb/s for a single 32 symbol regex implementation, very similar to that of the Counting G-NFA implementation presented in thesis.

Kaneta et al.'s (2010) Virtex-5 LX300–based BP-NFA design shows similar performance characteristics to that of the Counting G-NFA BP implementation with a maximum clock frequency of 202 MHz for an implementation with 128 regexes. However, unlike the Counting G-NFA, the BP-NFA algorithm can only handle constrained repetitions by unrolling. Lee's bitmap-based G-NFA (2009) gives a throughput of 4Gb/s, but this is for a simple implementation of just one regex and it also needs to unroll constrained repetitions.

There has been less research into FPGA-based DFA implementations as NFAs are seen as more appropriate for the parallel nature and smaller memory size of FPGAs. Becchi (2009) discusses the potential for a memory centric FPGA-based DFA implementation where, in order to harness the parallelism of the FPGA, each BRAM would hold a single DFA. A number of DFAs could be combined into one DFA in order to increase the number of regexes per BRAM, provided there is not a state blow-up as a result of the combination process. Hayes & Luo (2007) propose a system called DPICO (DPI COmpact) that uses a modified CAM-like structure, BRAM, and data packing to implement a compact DFA on an FPGA. Their first proposed improvement to a baseline DFA implementation is to combine multiple transitions between the same two states into a single default transition. All non-default transitions are referred to as labelled transitions. A modified CAM-like (mCAM) structure associated with each DFA state is used to look up labelled transitions. If no labelled transition is found, then the default transition, which is read from memory, is used. This mCAM scheme only produces a memory saving if the number of default transitions is relatively high. Otherwise, it would be better to use the standard method of storing all transitions in a 256-element array in memory indexed by the 8-bit transition label (i.e. symbol). The authors' second proposal is to take the mCAM idea and to distribute the storage of transitions over multiple parallel BRAMs. Assuming sufficient BRAMs are available to cover the maximum number of labelled transitions per state plus one default transition, all of a state's transitions can be read in one clock cycle.

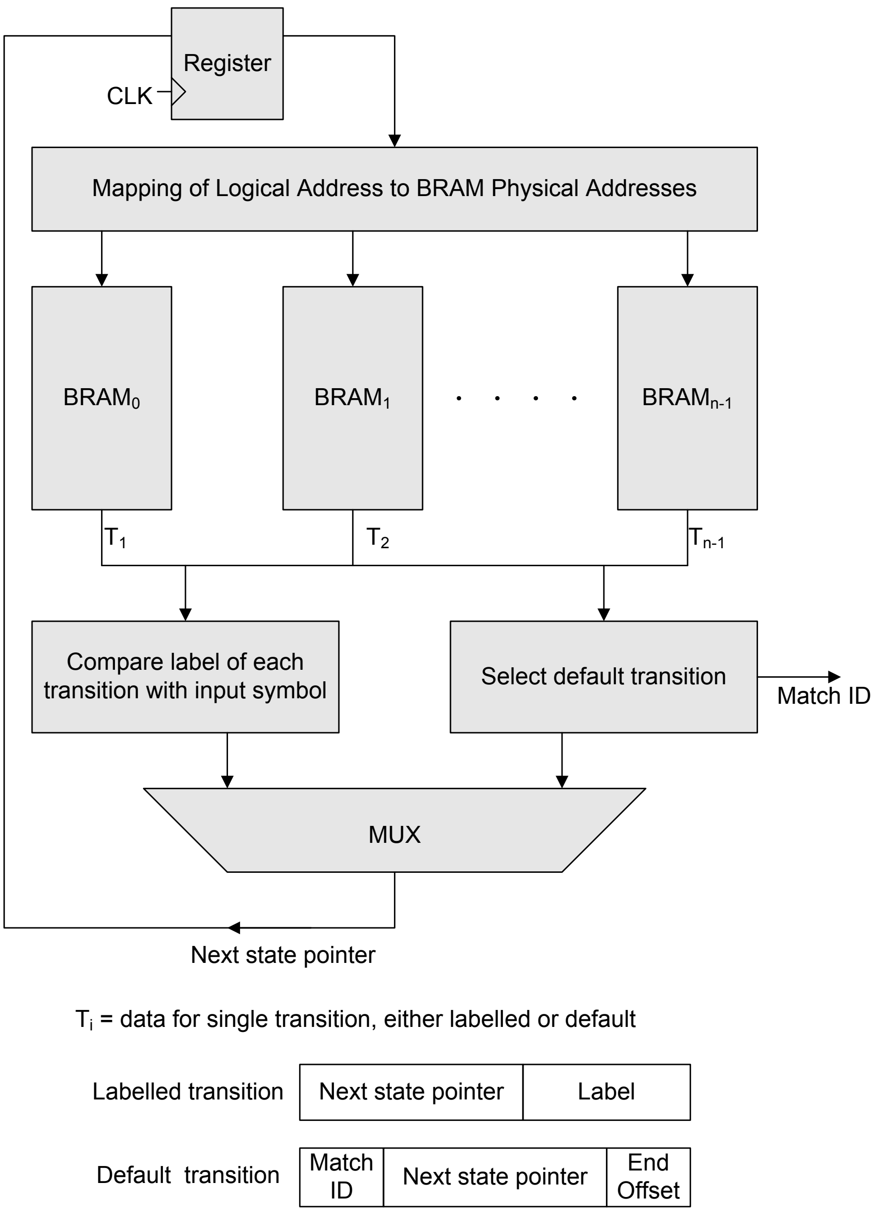

Register
CLK
Mapping of Logical Address to BRAM Physical Addresses
BRAM$_0$
BRAM$_1$
BRAM$_{n-1}$
T$_1$
T$_2$
T$_{n-1}$
Compare label of each transition with input symbol
Select default transition
Match ID
MUX
Next state pointer
T$_i$ = data for single transition, either labelled or default
Labelled transition
Next state pointer
Label
Default transition
Match ID
Next state pointer
End Offset

**Figure 68. DPICO block diagram**

A block diagram of the DPICO scheme is shown in Figure 68. Each BRAM outputs the data for a single transition, which may be either a labelled or default transition. The data for a labelled transition consists of the label and the next state pointer logical address. The data for a default transition consist of the next state pointer logical address, the match ID for the regex if this is an accept state and an offset to the last BRAM which holds transition data for this particular state. This offset is required to pinpoint the valid pieces of transition data when the number of this state's transitions is less than the number of BRAMs. The labels of all valid labelled transitions received from the BRAMs must be compared to the input symbol in order to find a match. If a match is found, then the transitions next state pointer address is used. Otherwise, the next state pointer from the default transition data is used. The next state pointer is a logical address which points to the default transition data of the next state.

One of the main downfalls of the DPICO algorithm is that it needs to perform a comparison between the input symbol and a potentially large number of transition labels which will limit the maximum clock frequency achievable in an FPGA implementation. According to Hayes & Luo's evaluation, if 16 BRAMs are sufficient to cover the maximum number of transitions, then the maximum throughput on a Virtex 4 FPGA is given as 789 Mb/s which increases to 2175 Mb/s in a pipelined design. The number of possible next-state transitions is likely to be excessively high if the DFA is constructed from regexes containing constrained repetitions with high counter values, making it unsuitable for this system. It is therefore difficult to make an accurate comparison between the DPICO and Counting G-NFA implementation as the DPICO memory requirements and performance depend very much on the form of the regex. Another possible issue with the DPICO scheme is that all DFAs need to be combined into a single DFA which may result in a state explosion. The resulting DFA may be too large to fit into the series of BRAMs.

Table 32 provides a summary of the supported features and performance of each of the dynamic memory-based architectures.

**Table 32: Comparison of dynamic memory-based hardware architectures**

| Feature | Counting G-NFA | BP-NFA | Bitmap G-NFA | CX-NFA | CES |
|---|---|---|---|---|---|
| | | (Kaneta et al.) | (Lee) | (Pao) | (Wang et al.) |
| No. of regexes in evaluated design | 96 | 128 | 1 | - | 1 |
| FPGA device used for evaluation | Virtex 5 TX240T | Virtex 5 LX330 | Virtex-II XC2VP30 | - | Virtex 5 LX110T |
| Clock freq. for 32 symbol regex (MHz) | 201.5 | 202 | - | - | 200 |
| Max. throughput (Gb/s) | 1.6 | 1.6 | 4 | - | 1.6 |
| BRAM per regex symbol (bytes) | 160 | 64 | - | TCAM-based | 29 |
| No. Virtex 5 slices per regex symbol | 4 | 3.8 | - | - | 22 |
| Power cons. per regex at max throughput | 68mW | - | - | - | - |
| BP-based solution? | Yes | Yes | Yes | No | No |
| Constrained repetitions without unrolling? | Yes | No | No | Yes | Yes |
| Full handling of counter overlap? | No* | - | - | Yes | Yes |
| Support for multi-symbol repetitions? | Yes | - | - | Unclear | No |

*Full handling of counter overlap is added to the Counting G-NFA in Chapter 7.

## 6.9. Conclusion

Network Intrusion Detection Systems (NIDS) make extensive use of regexes as attack signatures. Such expressions can be handled in hardware using a bit-parallel (BP) architecture based on the Glushkov Non-deterministic Finite Automata (NFA). However, many expressions contain constrained {min,max} repetitions which first need to be unrolled so that they can be handled by the standard BP system. Such unrolling often leads to an excessive memory requirement which makes handling of such regexes unfeasible. This chapter has presented a solution, based on the standard BP architecture, which incorporates a counting mechanism that renders unrolling unnecessary. As a result, many regexes, which were previously unsuitable for the standard BP system, can now be efficiently handled. Unlike many other approaches, this architecture is dynamically reconfigurable thanks to its memory, rather than logic, -based engine. This is important as NIDS rule sets are regularly updated. It can also handle repetition of both single and multi-symbol sub-expressions.

The BP architecture presented in this chapter shows similar performance results to the CES architecture proposed by Wang et al. (2010) but is more flexible in that it does not use static interconnections between states and it can support multi-character repetitions. Another memory-centric FPGA architecture is DPICO as proposed by Hayes & Luo (2007). The main disadvantage of DPICO is the potentially large number of comparisons that need to be performed for every input symbol, which is likely to limit the maximum clock frequency. It is also unclear how DPICO would cope with the potentially large number of next state transitions associated with constrained repetitions.

Besides the constrained {min,max} repetition examined in this chapter, other examples of complex regex syntax found in Snort rule sets  include back references and zero-width look-around assertions. Although the constrained repetition is relatively complex, its implementation in hardware is eased somewhat by the fact that the repeated sub-expression is usually a static value. Back references are more complicated because the pattern to match depends on what was actually matched earlier in the input string. This would not be easy to implement in hardware as both the length and value of the back-reference pattern only become known during the matching process for each input string. Look-around assertions would be difficult to handle in hardware because they involve pattern matching without the consumption of any symbols from the input string. Adding support for back reference and look-around assertion syntax to the BP architecture would be an interesting, albeit challenging, topic for future research.

# Chapter 7 - Pattern Overlap in case of Constrained Repetitions

As described in the previous chapter, a counter-based mechanism can be used to handle constrained repetitions without the need for inefficient unrolling of the repeated sub-expression. However, most existing proposals do not fully handle what is known as the "overlap issue" which some regexes can be prone to. This chapter presents a memory-centric Bit Parallel hardware architecture that overcomes the issue of counter overlap through the use of a bit serial First-In-First-Out (FIFO) queue. The memory-centric rather than logic-centric nature of the design has the advantage of allowing dynamic updates to individual attack signatures. The proposed solution is targeted at ASIC and FPGA platforms and experimental results for a proof-of-concept design are presented.

## 7.1. Counting Overlap Issue

The BP G-NFA inherently handles most cases of pattern overlap. However, counter overlap occurs when there is a transition into the counting block while it is actively counting. Analysis of recent SNORT rule sets shows roughly 30% of the constrained repetitions are susceptible to counter overlap. The following are examples of regexes that are susceptible:

- Unanchored regex where the repeated sub-expression is at the start, e.g. /*a{3}bcd*/ . This can be rewritten as /*^a{3}a*bcd*/ so as to avoid the overlap issue. So there is no problem in this case.
- Unanchored regex where the repeated sub-expression is preceded by a number of symbols, all of which overlap with the sub-expression, e.g. /*ab[abc]{3}d*/ or /*a.(ab){2}c*/ .
- Overlap between the constrained repeated sub-expression and a preceding repetition where the two repetitions are separated only by symbols which also overlap. The preceding repetition can be *,?,+ or a constrained {*min,max*} quantifier where *min≠max*. Regex can be anchored or unanchored. e.g. /*^xyz[ab]+[abc]{2}d*/ or /*xyz[ab]*.(ab){2}d*/ .

As an illustration of the issue, consider the example regex /*ab[abc]{3}d*/ . Reception of ab***ababcd*** would cause a problem as the counter would be reset to zero after receiving *ababa* and mismatch would occur on receiving the next *b*, as shown in Table 33.

**Table 33: Counter Overlap in case of regex /ab[abc]{3}d/**

| *Symbol received* | *Counter* | *Activate counting block?* |
|---|---|---|
| a | | |
| b | | |
| a | 1 | Yes |
| b | 2 | |
| a | 3 | Yes. But already active=>overlap issue! |
| b | Reset to 0 | |
| c | 1 | |
| d | Mismatch | |

**Table 34. Handling counter overlap with multiple counter instances in case of regex /ab[abc]{2,5}d/**

| *Symbol received* | *Cnt 1* | *Cnt 2* | *Cnt 3* | *Cnt 4* | *Description* |
|---|---|---|---|---|---|
| a | | | | | |
| b | | | | | |
| a | 1 | | | | Create counter instance 1 and activate counting block. |
| b | 2 | | | | Cnt1≥*min*, so allow transition out of block. |
| a | 3 | 1 | | | Create counter instance 2. Cnt1≥*min*, so allow transition out of block. |
| b | 4 | 2 | | | Cnt1≥*min*, so allow transition out of block. |
| a | 5 | 3 | 1 | | Create counter instance 3. Cnt1≥*min*, so allow transition out of block. |
| b | | 4 | 2 | | Cnt1=*max*, so remove Cnt1. Cnt2≥*min*, so allow transition out of block. |
| c | | 5 | 3 | 1 | Create counter instance 4. Cnt2≥*min*, so allow transition out of block. |
| b | | | 4 | 2 | Cnt2=*max*, so remove Cnt2. Cnt3≥*min*, so allow transition out of block |
| d | | | | | Valid transition out of counting block to accept state => match found. |

This issue can be solved by using multiple counter instances, as illustrated in Table 34, but this can seriously degrade performance due to the number of memory accesses required to check and update all counter instances. An *{m,n}* quantifier may potentially require *n* concurrently active counter instances which would occupy a significant amount of memory and degrade performance due to the number of memory accesses per symbol processed. Becchi and Crowley (2008) observed that only two accesses are required if differential representation is used. The use of differential representation is illustrated by the example in Table 35. Only the true value of the oldest active counter is stored. All other counters are delta values between their true value and the value of the previously created instance. These delta values do not need to be incremented in each clock cycle and only need to be converted to the true value when the previously created counter instance is removed. Although differential representation reduces the amount of memory accesses, it does not solve the problem of the memory occupied. Similarly, the TCAM-based event queue feature proposed by Pao (2009) grows in length as the number of concurrent overlaps increases.

**Table 35. Handling counter overlap with differential counters in case of regex /ab[abc]{2,5}d/**

| *Sym* | *Cnt 1* | *Cnt 2* | *Cnt 3* | *Cnt 4* | *Description* |
|---|---|---|---|---|---|
| a | | | | | |
| b | | | | | |
| a | 1 | | | | Create counter instance 1 and activate counting block. |
| b | 2 | | | | Cnt1≥*min*, so allow transition out of block. |
| a | 3 | 3-1= 2 | | | Create counter instance 2. Cnt1≥*min*, so allow transition out of block. |
| b | 4 | 2 | | | Cnt1≥*min*, so allow transition out of block. |
| a | 5 | 2 | 5-(2+1) =2 | | Create counter instance 3. Cnt1≥*min*, so allow transition out of block. |
| b | | (5-2)+1 =4 | 2 | | Cnt1=*max*, so remove Cnt1. Cnt2≥*min*, so allow transition out of block. |
| c | | 5 | 2 | 5-(2+1) =2 | Create counter instance 4. Cnt2≥*min*, so allow transition out of block. |
| b | | | (5-2)+1 =4 | 2 | Cnt2=*max*, so remove Cnt2. Cnt3≥*min*, so allow transition out of block |
| d | | | | | Valid transition out of counting block to accept state => match found. |

The good news, however, is that most constrained repetition quantifiers found in the SNORT rule set are not subject to the overlap problem. In particular, repetition quantifiers at the end of a regex always only require a single counter instance. Consider, for example, the regex /ab[abc]{4,6}/. This can be rewritten as /ab[abc]{4}/ as the repetition is at the end of the regex and there is nothing gained from matching a further two symbols when a match has already been found. Although overlapping patterns can occur, there is no benefit in having multiple counters because all counters will be reset to 0 whenever there is a mismatching symbol.

## 7.2. Counting GlushKov NFA with Overlap Handling

The storage of multiple counters as a solution to the overlap issue is not practical because of the worst case memory requirement. This thesis proposes a variant of the differential counter scheme which uses a bit serial FIFO to represent the differential counters in addition to a single counter instance which holds the oldest active counter value. For each input symbol processed, a 1 is written to the FIFO if this is the start of an overlapping pattern, and 0 otherwise. When the counter reaches its maximum value, the FIFO is repeatedly read until a 1 is encountered. The number of bits read is then subtracted from the counter value. So in the example shown in Table 36, two bits are read from the FIFO when the counter reaches its maximum value and so 2 is subtracted from the counter value of 3 to give a new counter value of 1.

Each counting block has the following additional storage elements:

- `PRE_BLOCK`: bitmask indicating states outside the counting block which have an outgoing transition into the counting block
- `FIFO`: bitmask indicating occurrences of overlaps

**Table 36: Bit serial FIFO to track overlap in case of regex /ab[abc]{3}d/**

| *Step* | *Input Symbol* | `ACTIVE` *bitmask* *4 3210* | `FOLLOW_SUM` *4 3210* | `CNT` | *Write to FIFO* | *Read from FIFO* | *FIFO contents* |
|---|---|---|---|---|---|---|---|
| | | | 0 0010 | | | | |
| 1 | *a* | 0 0011 | 0 0110 | 0 | - | - | - |
| 2 | *b* | 0 0101 | 0 1010 | 0 | - | - | - |
| 3 | *a* | 0 1011 | 0 1110 | 1 | - | - | - |
| 4 | *b* | 0 1101 | 0 1010 | 2 | 0 | - | 0 |
| 5 | *a* | 0 1011 | 1 0110 | 3 | 1 | - | 10 |
| 6 | Overlap Check | | 1 1110 | 3-2=1 | | 10 | - |
| 7 | *b* | 0 1101 | 0 1010 | 2 | 0 | - | 0 |
| 8 | *c* | 0 1001 | 1 0010 | 3 | 1 | - | 10 |
| 9 | Overlap Check | | 1 1010 | 3-2=1 | - | 10 | - |
| 10 | *d* | 1 0001 | 0 0010 | 0 | - | - | - |
| 11 | | *Match!* | | | | | |

| *Step* | *Description* |
|---|---|
| 1 | Following receipt of '*a*', state 1 becomes active. State 0 also remains active because the regex is unanchored. |
| 2 | Following receipt of '*b*', state 2 becomes active. |
| 3 | Following receipt of '*a*', states 1 and 3 becomes active.<br>Counting block is now active and the counter `CNT` is incremented to 1. |
| 4 | Following receipt of '*b*', states 2 becomes active (transition from state 1). `CNT` is incremented to 2.<br>`CNT` ≤ 3 (max), so state 3 remains active.<br>No overlap, so 0 written to FIFO. |
| 5 | Following receipt of '*a*', state 1 becomes active. `CNT` is incremented to 3. Overlap situation detected, so 1 is written to FIFO.<br>`CNT` has reached min, so transition out of counting block is enabled by setting bit 4 of `FOLLOW_SUM` bitmask to 1.<br>`CNT` ≤ 3 (max), so state 3 remains active.<br>`CNT` has reached max, so an overlap check is required. Bit 3 in `FOLLOW_SUM` bitmask is set to 0 in order to disable repetition. `ACTIVE` bitmask is kept unchanged while this check is performed. |

| | |
|---|---|
| 6 | Overlap check performed by reading FIFO until bit=1 is reached. Two bits are read and so 2 is subtracted from the current `CNT` value to give a new value of 1.<br>CNT ≤ 3 (max), so bit 3 is set back to 1 in FOLLOW_SUM bitmask in order to allow repetition. |
| 7 | Following receipt of '*b*', state 2 becomes active (transition from state 1). `CNT` is incremented to 2.<br>`CNT` ≤ 3 (max), so state 3 remains active.<br>No overlap, so 0 written to FIFO. |
| 8 | Following receipt of 'c', `CNT` is incremented to 3. Overlap situation detected, so 1 is written to FIFO.<br>`CNT` has reached min, so transition out of counting block is enabled by setting bit 4 of `FOLLOW_SUM` bitmask to 1.<br>`CNT` ≤ 3 (max), so state 3 remains active.<br>`CNT` has reached max, so an overlap check is required. Bit 3 in `FOLLOW_SUM` bitmask is set to 0 in order to disable repetition. `ACTIVE` bitmask is kept unchanged while this check is performed. |
| 9 | Overlap check performed by reading FIFO until bit=1 is reached. Two bits are read and so 2 is subtracted from the current `CNT` value to give a new value of 1.<br>`CNT` ≤ 3 (max), so bit 3 is set back to 1 in `FOLLOW_SUM` bitmask in order to allow repetition. |
| 10 | Following receipt of '*b*', state 4 becomes active. State 3 is inactive, so `CNT` is reset to zero. (A non-empty FIFO would be reset at this stage in order to empty it). |
| 11 | `ACTIVE` and `FINAL` bitmasks are ANDed together and a match is detected |

Figure 69 shows a high level view of how counter overlap is dealt with in the modified BP Counting G-NFA scheme. Algorithm 1 gives the pseudo-code for a 32-bit wide implementation (i.e. $n$=32) in which the `FOLLOW_ACTIVE` table is horizontally partitioned into four separate tables and the repeated sub-expression is a single symbol.

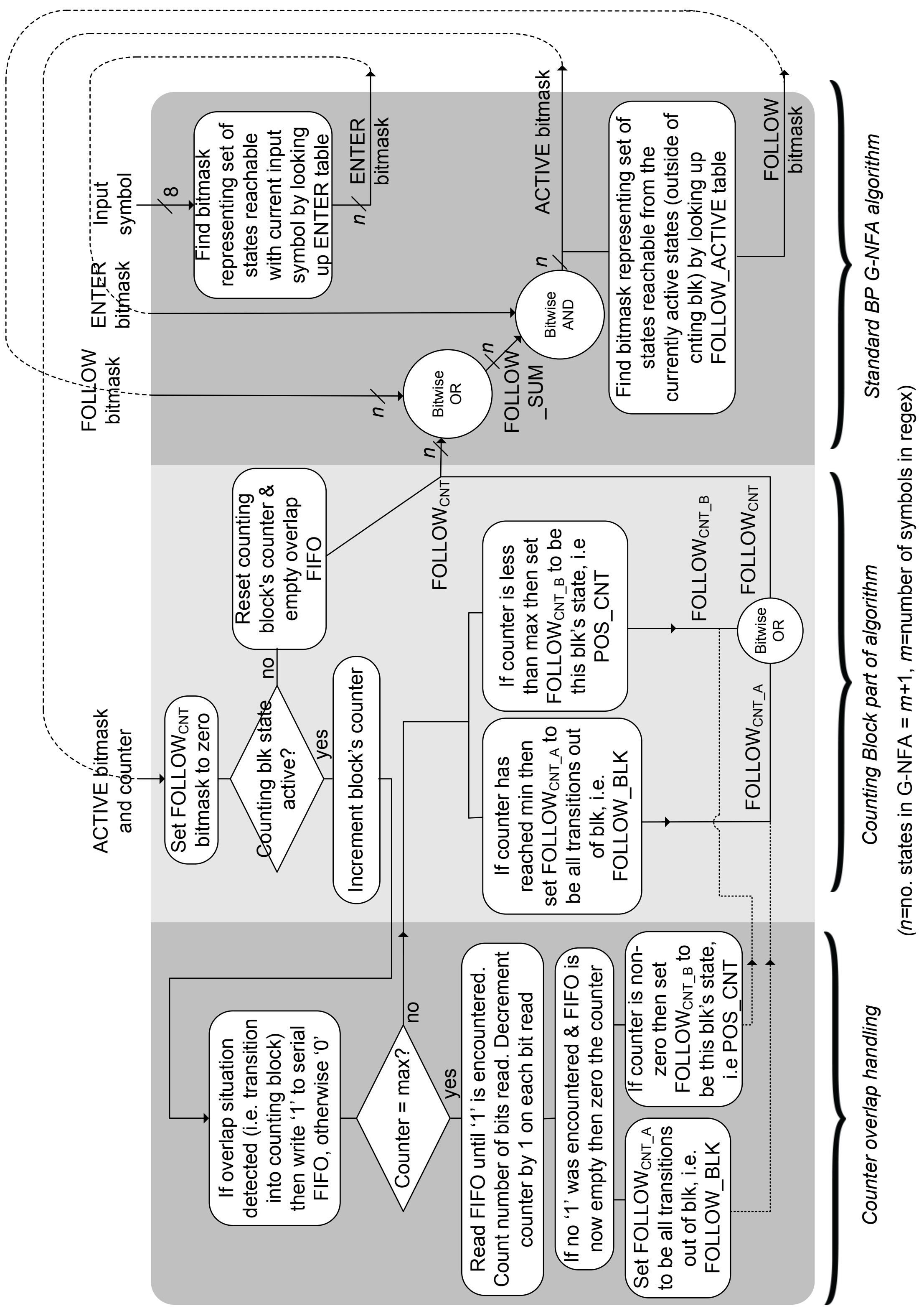

Input symbol
8
Find bitmask representing set of states reachable with current input symbol by looking up ENTER table
n
ENTER bitmask
ACTIVE bitmask
n
ENTER bitmask
Bitwise AND
n
FOLLOW bitmask
n
Bitwise OR
FOLLOW _SUM
Find bitmask representing set of states reachable from the currently active states (outside of cnting blk) by looking up FOLLOW_ACTIVE table
FOLLOW bitmask
Standard BP G-NFA algorithm
ACTIVE bitmask and counter
Set FOLLOW$_{CNT}$ bitmask to zero
Counting blk state active?
no
yes
Reset counting block's counter & empty overlap FIFO
FOLLOW$_{CNT}$
n
Increment block's counter
If counter is less than max then set FOLLOW$_{CNT\_B}$ to be this blk's state, i.e POS_CNT
FOLLOW$_{CNT\_B}$
FOLLOW$_{CNT}$
Bitwise OR
If counter has reached min then set FOLLOW$_{CNT\_A}$ to be all transitions out of blk, i.e. FOLLOW_BLK
FOLLOW$_{CNT\_A}$
FOLLOW$_{CNT}$
Counting Block part of algorithm
(n=no. states in G-NFA = m+1, m=number of symbols in regex)
If overlap situation detected (i.e. transition into counting block) then write '1' to serial FIFO, otherwise '0'
Counter = max?
no
yes
Read FIFO until '1' is encountered. Count number of bits read. Decrement counter by 1 on each bit read
If no '1' was encountered & FIFO is now empty then zero the counter
If counter is non-zero then set FOLLOW$_{CNT\_B}$ to be this blk's state, i.e POS_CNT
Set FOLLOW$_{CNT\_A}$ to be all transitions out of blk, i.e. FOLLOW_BLK
Counter overlap handling

**Figure 69. Counting G-NFA for single symbol repetition elements with overlap handling**

**ALGORITHM 1.** 32-bit BP Counting G-NFA search algorithm with overlap handling

```
Input: String, Σ, of input symbols, σ.
Output: Match and NoMatch signals.
Active ← First;
OldActive ← $0^{32}$;
for each symbol σ ∈ Σ repeat /* Process each incoming symbol */
    FollowSum ← $0^{32}$;
    for idx 1 to NumCountingBlocks repeat /* Handle each constrained repetition in regex */
        if (Active AND PosCnt(idx)) then
            if (Cnt(idx) then /* Counter value is non-zero? */
                if (OldActive AND PreBlk(idx)) then /* Transition into counting block? */
                    Write(FIFO,1);
                else
                    Write(FIFO,0);
                end if
            end if
            Cnt(idx) ← Cnt(idx) + 1;
            if (Cnt(idx) ≥ Min(idx)) then /* Allow transition out of counting block? */
                FollowSum ← FollowSum OR FollowBlk(idx);
            end if
            if (Cnt(idx) = Max(idx)) then
                z ← 0;  /* z is the number of bits read from FIFO  */
                repeat
                    z ← z+ 1;
                while (Read(FIFO)=0));
                /* Now subtract number of bits read to get new counter value */
                Cnt(idx) ← Cnt(idx) - z;
            end if
            if (Cnt(idx) < Max(idx)) then /* Allow another repetition? */
                FollowSum ← FollowSum OR PosCnt(idx);
            end if
        else
            Cnt(idx) ← 0;
            Clear(FIFO);
        end if
    end for
    /* Combine the set of follow states calculated in the counting  block section above    *
     * with the standard horizontally partitioned FOLLOW_ACTIVE values                      */
    FollowSum←  FollowSum OR FollowActive[Active(31..24)] OR
                FollowActive[Active(23..16)] OR FollowActive[Active(15..8)] OR
                FollowActive[Active(7..0)];
    OldActive ← Active;
    /* Calculate new set of active states by ANDing together the bitmask of states         *
     * reachable from the currently active states with the bitmask of states reachable     *
     * with the current input symbol                                                        */
    Active ← FollowSum AND Enter[σ];
    if (Active AND Last) then /* Accept state reached? */
        MatchFound ← 1;
        break
    end if
end for
if (MatchFound = 0) then
    NoMatchFound ← 1;
end if
```

## 7.3. Implementation

### 7.3.1. Hardware Architecture

The implementation outlined in the previous chapter was extended to include the FIFO-based counter overlap handling mechanism. The PE architecture was redesigned as an FSM, interfaced with BRAM for regex data storage and with a one bit wide FIFO for overlap tracking, as illustrated in Figure 70.

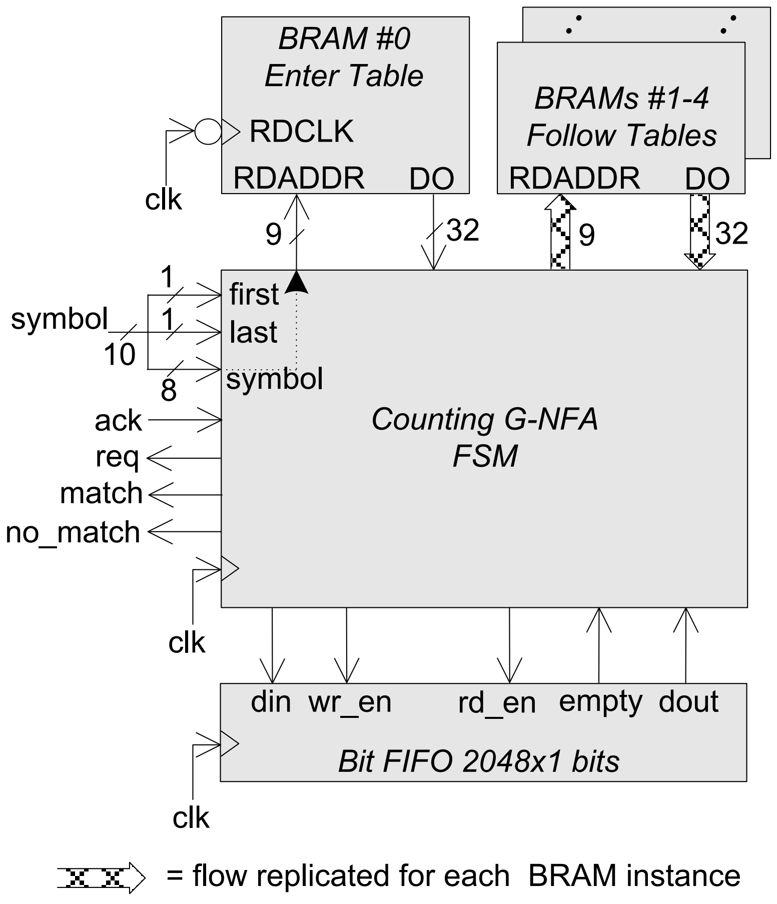


**Figure 70: Regex Engine Architecture**

The FSM implements the Counting BP G-NFA algorithm in addition to handling handshaking with the DPI packet buffer handler. The 2048x1 bit FIFO is used to deal with the overlap issue for constrained *{min,max}* quantifiers up to a *max* value of 2048. One BRAM holds the ENTER table, while the other four hold the FOLLOW table horizontally partitioned four times. Regex related bitmask values such as MIN, MAX, POS_CNT, etc., are stored in registers within the FSM design. All these bitmasks, in addition to the BRAM contents, can be dynamically updated via the Engine Data Update Handler. Details of signals related to these updates are omitted from Figure 70 for clarity.

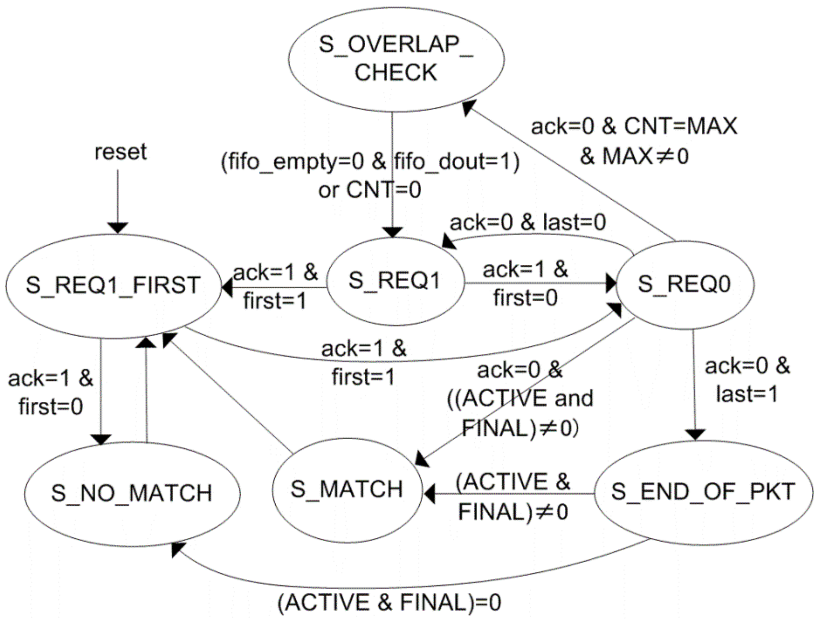


**Figure 71: Counting G-NFA FSM**

The Counting G-NFA FSM is the key block in the DPI design. A simplified Moore style state diagram showing the states and events triggering each transition is outlined in Figure 71. The following is a summary of each state's role:

- S_REQ1_FIRST:
  - FSM remains in this state until the first symbol of a new payload is received
  - *req* flag is asserted to indicate that it is waiting for a symbol
  - *ACTIVE* is initialized to the value of the stored *FIRST* bitmask
- S_REQ0:
  - input *symbol* signal is latched on transitioning into this state
  - *req* flag is deasserted
  - start-anchor and multi-mode regex flags are checked to see whether or not the *FIRST* states should be held active
  - most of the counting algorithm, as described in Algorithm 1, is performed. This involves
    - checking if any of the counting block states are now active
    - incrementing the *CNT* counter

    - writing 1 or 0 to the overlap Bit FIFO to indicate counter overlap or no overlap, respectively
    - checking if *CNT* is less than *MAX* and, if so, ORing the value of *POS_CNT* into the *FOLLOW* bitmask, thereby allowing further repetition
    - checking if *CNT* is greater than or equal to *MIN* and, if so, ORing the value of *FOLLOW_BLK* into the *FOLLOW* bitmask, thereby allowing transitions out of the counting block
    - resetting *CNT* to zero if none of the counting block states are active

- S_REQ1:
  - *req* flag is asserted to indicate that it is waiting for the next symbol
- S_OVERLAP_CHECK:
  - decrements *CNT* once *MAX* is reached, as per the algorithm outlined in Algorithm 1. This involves repeatedly reading the Bit FIFO until the output is '1' and decrementing *CNT* for every '0' read
- S_END_OF_PKT:
    - simply waits for processing of last symbol in payload to complete
- S_MATCH:
  - asserts the *match* output signal
- S_NO_MATCH:
  - asserts the *no_match* output signal

### 7.3.2. Software

The CCP program developed by Champarnaud et al. (2004), which had been extended to generate the Counting G-NFA bitmasks, was reused.

## 7.4. Performance Results

The design was implemented in VHDL and simulated with the Xilinx Virtex5 TX240T (Speed -2) as the target FPGA. Sets of regexes, varying in size from 32 to 736, were randomly chosen from the snapshot dated 18.09.2012 of the Snort v2.9.3.1 rule set and converted into the appropriate FPGA BP format using the modified CCP program. Payload data from Shmoo (2009) Group DEFCON traffic traces were used for the timing simulation and subsequent power analysis.

The Xilinx XST tool gave the maximum achievable clock frequency as 116 MHz on the Virtex5 and 146 MHz on the Virtex7. This proof-of-concept design is quite a simple implementation which does not include packet pipelining. It processes one symbol every 9 clock cycles if there is no constrained repetition, and every 9.2 cycles on average when tested with the randomly chosen rule sets and traffic traces. This gives an average throughput of 101 Mb/s and 127 Mb/s on the Virtex5 and Virtex7, respectively. Adding pipelining to the design should, in theory, result in a nine-fold increase in throughput, giving 909 Mb/s and 1.14 Gb/s. However, this is likely to be somewhat lower in practice because the increased complexity of the design will probably reduce the maximum achievable clock frequency.

**Table 37: Virtex5 TX240T Device Utilisation**

| *Resource* | *32 PE instances* | *64 PE instances* | *96 PE instances* |
|---|---|---|---|
| *Block RAM* | 25% | 49% | 74% |
| *Slices* | 12% | 25% | 38% |

Resource utilization figures for the Virtex5 and Virtex7 are shown in Table 37 and Table 38, respectively. The NetFPGA-10G platform can support a maximum of 128 PEs due to the FPGA’s limited BRAM. The Virtex7 1140T can support a much larger number of PEs as it has substantially more BRAM. Power consumption with 96 PEs is approximately 6W on the Virtex5, and 3.1W on the Virtex7, at maximum throughput. These figures include both dynamic and quiescent (static) power consumption. The much lower consumed by the Virtex7 device ties in with Xilinx claims’ that the 7 series FPGAs provide 50% power reduction compared to previous generation FPGAs (Xilinx, 2012b).

**Table 38: Virtex7 1140T Device Utilisation**

| *Resource* | *256 PE instances* | *512 PE instances* | *736 PE instances* |
|---|---|---|---|
| *Block RAM* | 25% | 49% | 74% |
| *Slices* | 12% | 25% | 38% |

## 7.5. Related Work

The BP architecture has the advantage that it allows rules to be dynamically updated whenever a new threat emerges, as well as allowing the design to be implemented on both FPGA and ASIC. The more common logic-based design requires regeneration of the FPGA bit-stream whenever the rule set is updated. The logic-based system proposed by Bispo et al. (2006) uses a shift register to handle overlap of single symbol constrained repetitions. Faezipour and Nourani (2008) added support for multi-symbol repetitions but their algorithm does not appear to handle all cases of counter overlap.

The problem of counter overlap has also been examined by those researching DFA-based solutions. Becchi and Crowley (2008) propose a counting-DFA algorithm that uses multiple counter instances to overcome the overlap issue. Differential representation can be used in order to minimize the counter size. Becchi (2009) also looks at how DFAs in general can be implemented using a memory-based FPGA design. The use of multiple counter instances, however, is more suited to a software implementation using dynamic memory allocation rather than to an FPGA design as the number of instances is potentially very high.

CES (Wang et al., 2010) is a memory-centric NFA-based algorithm which can handle counter overlap using a stack of checkpoint counter registers. This method of handling overlap is similar to the differential counter representation suggested by Becchi and Crowley (2008) and may potentially occupy a significant amount of memory. CES can only handle single symbol repetitions and some regexes may be unsuitable due to an excessive number of interconnections between CCRs within the CES tile mesh.

## 7.6. Conclusion

This chapter improves on the architecture presented in the previous chapter for the handling of constrained repetitions in regexes. The improved design overcomes the issue of counter overlap through the use of a bit serial First-In-First-Out (FIFO) queue, thereby allowing a greater number of regexes to be handled. This FIFO-based mechanism is much more space efficient than the use of multiple counter instances which are more suited to software implementation where dynamic memory allocation is available. The proposed solution is targeted at ASIC and FPGA platforms and experimental results are presented for a proof-of-concept design.

# Chapter 8 - Conclusions and Further Work

The last decade has seen enormous growth in the size of the internet and in the amount of services it offers. In recent years, much of this has been due to the meteoric rise in the use of smartphones and mobile applications. Many predict a much bigger increase in the coming years as cloud computing becomes even more significant and the Internet of Things becomes a reality. This growth is accompanied by an increase in the number and complexity of attacks. As a consequence, network security systems need to be more sophisticated while, at the same time, capable of handling ever increasing traffic rates.

NIDPS systems allow attacks to be detected and blocked before they can enter the enterprise network or sensitive parts of the intranet. The three fundamental building blocks of such systems, namely TCP/IP reassembly, multi-match classification and DPI have been studied in this thesis. DPI can be further split into fixed string and regex matching. Fixed string DPI was not looked at in this thesis as it has already been well studied by many researchers. Regexes have become more important as they are used in the signatures that describe most new attacks.

### 8.1.1. TCP/IP Reassembly

Attack patterns may bridge the boundary between IP fragments and TCP segments in a particular TCP connection flow. It is therefore essential that the DPI is performed on the reassembled payload data stream rather than simply on the payload of individual packets. Performing TCP/IP reassembly on an intermediate network node, such as an NIDPS, is complicated by the fact that there are subtle differences in how destination nodes perform the reassembly. These differences are a result of different interpretations of the TCP/IP standards by the implementers of different operating systems. Therefore, in order for an NIDPS to reassemble an IP fragment or TCP segment in the same way as the destination node, it must be aware of the OS of the destination. Although there are methods available for the automatic detection of the destination's OS, these are not always guaranteed to work. Manual configuration is often required, which is not ideal. An alternative, more maintainable, strategy is for the NIDPS to normalise traffic by removing any ambiguities which could be interpreted differently by different OS types. The downside is that a normaliser actively changes the traffic and so needs to be very robust. It may also break the operation of traceroute and PMTUD (Handley et al., 2001).

TCP/IP reassembly is typically implemented in software due to the amount of decision making and connection tracking that is required. However, NIDPS systems need to handle much higher traffic rates than most end host systems. This has prompted research into implementing reassembly in hardware. Besides the need for target-based reassembly or traffic normalisation, an added complication is the handling of 'holes' due to of out-of-order and missing fragments or segments and the resultant buffering requirement. This thesis surveyed a number of solutions and compared the techniques used for dealing with 'holes' and ambiguities in the fragmented traffic. A hardware-based reassembly system was proposed that takes advantage of the fact that out-of-sequence packets are rare under normal circumstances by carrying out target-based reassembly of the affected streams in software while dealing with the normal in-sequence streams directly in FPGA programmable logic.

### 8.1.2. Multi-match Packet Header Classification

The most obvious ways to perform multi-match packet header classification in hardware involve the use of TCAM. This thesis investigated alternative algorithmic solutions that use SRAM instead of TCAM in order to save energy. Software implementations of a number of single match classification algorithms were modified to perform multi-matching and their performance evaluated and compared. The EGT-PC and ART algorithms were found to perform quite well, but not as well as the very simple FSBV and StrideBV which perform well by virtue of the fact that the number of unique headers is quite low due to many rules sharing the same header.

### 8.1.3. Regular Expression DPI

While DPI has many applications, this thesis studied it in the context of NIDPS systems where regex matching is required in order to detect complex attacks. Regex matching also has other uses such as the matching of DNA and protein sequences, and text retrieval.

This thesis focused on finding an improved hardware design for a bit-parallel memory-centric architecture based on the Glushkov-NFA which could handle constrained repetitions, one of the more complicated features of regex syntax. Such an architecture has the advantage in that the regexes are stored in memory rather than hardcoded into the logic of the FPGA or ASIC, thereby enabling dynamic rule updates on live systems. The efficient handling of repetitions without unrolling of the repeated sub-expression leads to significant memory and hence energy savings.

Unfortunately, the format of some regexes make them susceptible to the problem of overlapping patterns rendering a single counter inadequate for tracking the repetition. An additional FIFO-based mechanism, which is relatively easy to implement in hardware, was proposed to deal with this issue. Ideally, this additional mechanism should only be used for regexes susceptible to counter overlap. Pre-processing software should therefore be used to split the rule set's regexes into three categories:

- those without any constrained repetitions which can be handled by the basic BP processing engine
- those with repetitions, but not susceptible to the overlap issue. These can be handled by engines which have the basic counting mechanism
- those susceptible to overlap that need the FIFO-based mechanism

Prototype designs with and without the overlap handling mechanism were implemented in VHDL and evaluated on the Xilinx Virtex5 and Virtex7 FPGAs. The designs performed quite well compared to other memory-based solutions. The relatively simple designs did not use any packet pipelining which should yield significant performance improvements.

## 8.2. Future Directions

This thesis has studied the various building blocks of an NIPDS system in relative isolation from each other. Looking at each in the context of a fully integrated solution may yield ideas on how to improve the performance and efficiency of each. Use of packet pipelining to give improved throughput performance and clock gating to reduce energy consumption could also be looked at. Finally, the huge increase in mobile internet data has opened up new applications for DPI which could provide interesting follow-on research opportunities.

### 8.2.1. Improving Performance

The design used to evaluate the bit-parallel regex matching algorithm was a simple prototype implementation that did not include packet pipelining. The next step would be to extend the design to process multiple data streams in parallel. This would not only improve performance, but would also allow handling of a much larger rule set on a single FPGA as the regex data could be stored in off-chip memory instead of in the limited internal BRAM.

The performance could also be improved by performing multi-stride matching (Avalle et al. 2012), i.e. processing of multiple symbols in each iteration of the algorithm. Compression of the BP tables is required in order to make this feasible in practice. For example, many symbol combinations are equivalent in that they are always used together. However, this compression means that simple SRAM-based tables need to be replaced by more expensive and less energy efficient TCAM.

### 8.2.2. Fixed String Pre-Filter

Most Snort rules include both a fixed string and a regex pattern. Even if no fixed string is specified, it is usually possible to extract one or more fixed strings from the regex. To reduce the load on the regex matching part of the system it's possible to pre-filter the traffic using fixed string matching as follows:

- traffic which does not match the fixed string is not forwarded for regex matching
- traffic which does match is only checked against regexes from rules with a matching fixed string. Some regex matching algorithms cannot take advantage of this as they are constrained to match against all patterns, e.g. NFA implemented in FPGA/ASIC logic, multiple DFAs combined into a single or small number of DFAs

The catch is that this should be performed on a reassembled PDU which can have a very large size (up to the maximum size of a socket buffer). The solution to this would be to reassemble IP fragments and TCP segments into pseudo-packets of a certain maximum length in a similar fashion to Snort. An added complication is that each Snort rule can contain multiple fixed string patterns and the pre-filter stage would need to logically AND the match results in these cases

### 8.2.3. Improving Power Efficiency

In a pipelined architecture incorporating a fixed string pre-filter, it would be possible to change the DPI stage design so that each engine is dynamically assigned a regex to process based on the matches found by the pre-filter. This could be done by associating an SRAM offset address with each regex and this offset is passed to each processing engine. The clock signal to unused engines could be gated in order to save energy.

More energy could be saved by using frequency scaling to dynamically adjust the clock frequency depending on the volume of traffic being processed. As most internet applications have a typical traffic distribution, application detection could be used to predict future traffic volumes.

### 8.2.4. Mobile Internet DPI

The roll-out of 4G, which is based on an all-IP core network, is opening up new possible applications for DPI such as mobile internet traffic management, Quality of Service and security. The role of DPI in this context is primarily to accurately detect the application corresponding to each traffic flow. This information can then be used to manage bandwidth, to allow operators to charge based on application, to provide itemised billing, etc. For example, German company Ipoque (2013) recently announced that researchers at Intel Labs have integrated Ipoque's Protocol & Application Classification Engine (PACE) DPI software library into a "smart pipe" server that can allocate bandwidth across multiple wireless networks to high priority applications. The smart pipe also allows for the seamless handover of VoIP connections between Wi-Fi and cellular networks.

The ideal mobile DPI solution consists of both application detection and intrusion detection/prevention. The main focus of future work would be on finding more effective methods for application detection in hardware in order to provide fast yet energy and memory efficient solutions. It is likely that application detection would be performed using regex matching in the case of unencrypted traffic and behavioural analysis in the case of encrypted traffic.

# References

Aho, A.V. and Corasick, M.J. (1975). Efficient String Matching: An Aid to Bibliographic Search. *Communications of ACM*, 18(6) pp.333-340.
DOI: http://dx.doi.org/10.1145/360825.360855

Albin, E. and Rowe, N.C. (2012). A Realistic Experimental Comparison of the Suricata and Snort Intrusion-Detection Systems. *26th International Conference on Advanced Information Networking and Applications Workshops (WAINA),* pp.122-127.
DOI: http://dx.doi.org/10.1109/WAINA.2012.29

Antonello, R., Fernandes, S., Kamienski, C., Sadok, D., Kelner, J., Gódor, I., Szabó, G., and Westholm, T. (2012). Deep packet inspection tools and techniques in commodity platforms: Challenges and trends. *Elsevier Journal of Network and Computer Applications*, 35(6), pp.1863-1878.
DOI: http://dx.doi.org/10.1016/j.jnca.2012.07.010

Avalle, M., Risso, F. and Sisto, R. (2012). Efficient multistriding of large non-deterministic finite state automata for deep packet inspection. *2012 IEEE International Conference on Communications* (ICC), pp.1079-1084.
DOI: http://dx.doi.org/10.1109/ICC.2012.6364235

AV-Test Institute (2013). Malware statistics. AV-Test Institute, Germany; Obtained through the internet:
Web: http://www.av-test.org/en/statistics/malware/ [accessed 1 Oct. 2013]

Baboescu, F., Singh, S. and Varghese, G. (2003). Packet Classification for Core Routers: Is there an alternative to CAMs? *IEEE INFOCOM 2003.* pp.53-63.
DOI: http://dx.doi.org/10.1109/INFCOM.2003.1208658

Baeza-Yates, R. and Gonnet, G.H. (1992). A new approach to text searching. *Communications of the ACM* , 35(10), pp.74-82.
DOI: http://dx.doi.org/10.1145/135239.135243

Becchi, M. (2009). Data Structures, Algorithms and Architectures for Efficient Regular Expression Evaluation. *Ph.D. Dissertation*, Technical Report Number 2009-55, Washington University, St. Louis, MO, USA. Advisor Patrick J. Crowley; Obtained through the internet:
Web:http://cse.wustl.edu/Research/Lists/Technical%20Reports/Attachments/887/Michela Becchi_Dissertation.pdf
[accessed 1 Oct. 2013]

Becchi, M. and Crowley, P. (2007a). A hybrid finite automaton for practical deep packet inspection. In *Proceedings of the 2007 ACM CoNEXT conference* (CoNEXT '07). Article 1, 12 pages.
DOI: http://dx.doi.org/10.1145/1364654.1364656

Becchi M. and Crowley, P. (2007b). An improved algorithm to accelerate regular expression evaluation. *Proceedings of the 3$^{rd}$ ACM/IEEE Symposium on Architecture for networking and communications systems (ANCS '07)*, pp.145-154.
DOI: http://dx.doi.org/10.1145/1323548.1323573

Becchi, M. and Crowley, P. (2008a). Extending finite automata to efficiently match Perl-compatible regular expressions. In *Proceedings of the 2008 ACM CoNEXT Conference* (CoNEXT '08). Article 25, 12 pages.
DOI: http://dx.doi.org/10.1145/1544012.1544037

Becchi, M. and Crowley, P. (2008b). Efficient Regular Expression Evaluation : Theory to Practice. *Proceedings of the 4th ACM/IEEE Symposium on Architecture for networking and communications systems (ANCS'08)*, pp.50-59.
DOI: http://dx.doi.org/10.1145/1477942.1477950

Becchi, M., Wiseman, C. and Crowley, P. (2009). Evaluating regular expression matching engines on network and general purpose processors. *Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems* (ANCS), pp.30-39.
DOI: http://dx.doi.org/10.1145/1882486.1882495

Berry, G. and Sethi, R. (1986). From regular expressions to deterministic automata. *Theoretical Computer Science*, 48(1) pp.117-126.
DOI: http://dx.doi.org/10.1016/0304-3975(86)90088-5

Bispo, J., Sourdis, I., Cardoso, J.M.P. and Vassiliadis, S. (2006). Regular Expression Matching for Reconfigurable Packet Inspection. *Proceedings of the IEEE Int. Conf. on Field Programmable Technology (FPT 2006),* pp. 119-216.
DOI: http://dx.doi.org/10.1109/FPT.2006.270302

Boyer R.S. and Moore, J.S. (1977). A Fast String Searching Algorithm. *Communications of ACM*, 20(10), pp.762-772.
DOI: http://dx.doi.org/10.1145/359842.359859

Bloom, B.H. (1970). Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*. 13(7), pp.422-426.
DOI: http://dx.doi.org/10.1145/362686.362692

Brodie, B., Cytron R. and Taylor, D. (2006). A Scalable Architecture For High-Throughput Regular-Expression Pattern Matching. *Proceedings of the international symposium on Computer Architecture (ISCA'06)*, pp.191-202.
DOI: http://dx.doi.org/10.1145/1150019.1136500

Cadambi, S., Chakradhar, S.T. and Becchi, M. (2007). *Fast and scalable process for regular expression search*, US Patent Application filed 2007, Pub. No. US 2008/0034427 A1
Web: http://www.freepatentsonline.com/20080034427.pdf

Champarnaud, J-M., Coulon, F. and Paranthoën, T. (2004). Compact and fast algorithms for safe regular expression search. In *Int. Journal of Computer Mathematics*, 81(4), pp.383-401.
DOI: http://dx.doi.org/10.1080/00207160310001650025
Software: http://elm.eeng.dcu.ie/~croninb/ccp/ccp-0.3.tar.gz

Cheng, Y., Chu, J., Radhakrishnan, S. and Jain A. (2013). TCP Fast Open. *IETF Internet Draft*, 2013; draft-ietf-tcpm-fastopen-05; Obtained through the internet:
Web: https://ietf.org/doc/draft-ietf-tcpm-fastopen/ [accessed 1 Dec. 2013]

Chomsky, N. (1956). Three models for the description of language. *IRE Transactions on Information Theory*. 2(3), pp.113-124
DOI: http://dx.doi.org/10.1109/TIT.1956.1056813

Cisco (2013). The Zettabyte Era – Trends and Analysis. *White paper, Cisco Systems Inc*. May 29, 2013; Obtained through the internet:
Web: www.cisco.com
[accessed 1 Oct. 2013]

Commentz-Walter, B. (1979). A String Matching Algorithm Fast on the Average. *Lecture Notes in Computer Science*. 71, pp.118-132.
DOI: http://dx.doi.org/10.1007/3-540-09510-1_10

Day, D. and Burns, B. (2011). A performance analysis of snort and suricata network intrusion detection and prevention engines. *Proceedings of the 5th International Conference on Digital Society* (ICDS), pp.187-192; Obtained through the internet:
Web: http://www.thinkmind.org/download.php?articleid=icds_2011_7_40_90007
[accessed 1 Oct. 2013]

Dharmapurikar, S., Krishnamurthy, P., Sproull, T. and Lockwood, J.W. (2003). Deep Packet Inspection Using Parallel Bloom Filters. *Proceedings 11th Symposium on High Performance Interconnects (HotInterconnects)*. pp.44-51.
DOI: http://dx.doi.org/10.1109/CONECT.2003.1231477

Dharmapurikar, S. and Paxson, V. (2005). Robust TCP stream reassembly in the presence of adversaries. *Proceedings of the 14th conference on USENIX Security Symposium*, (SSYM'05), Vol. 14; Obtained through the internet
Web:https://www.usenix.org/events/sec05/tech/full_papers/dharmapurikar/dharmapurikar.pdf
[accessed 1 Oct. 2013]

Dorfinger, P. Strohmeier, F., Moosbrugger, A., Gojmerac, I., Trammell, B., Boschi, et. al. (2009). PRISM: Final monitoring applications specification and analysis, Chapter 3; Obtained through the internet:
Web: http://telscom.ch/wp-content/uploads/Prism/FP7-PRISM-WP3.2-D3.2.3.pdf
[accessed 1 Oct. 2013]

Eatherton, W. (1998). Hardware-based Internet Protocol Prefix Lookups. *M.S. thesis, Electr. Eng. Dept.,* Washington Univ., St. Louis, MO, USA; Obtained through the internet:
Web: http://www.arl.wustl.edu/~jst/studentTheses/wEatherton-1999.pdf
[accessed 1 Oct. 2013]

Faezipour, M. and Nourani, M. (2008). Regular Expression Matching for Reconfigurable Constraint Repetition Inspection. *Proceedings of the IEEE Global Telecommunication Conf.* (GLOBECOM). pp.1–5.
DOI: http://dx.doi.org/10.1109/GLOCOM.2008.ECP.403

Ficara, D., Giordano, S., Procissi, G., Vitucci, F., Antichi, G. and Di Pietro, A. (2008). An improved DFA for fast regular expression matching. *SIGCOMM Comput. Commun.* 38(5), pp.29-40.
DOI: http://dx.doi.org/10.1145/1452335.1452339

Ganegedara, T. and Prasanna, V.K. (2012). *StrideBV: Single chip 400G+ packet classification*, IEEE 13th International Conference on High Performance Switching and Routing (HPSR), pp.1-6, 24-27.
DOI: http://dx.doi.org/10.1109/HPSR.2012.6260820

Glushkov, V.M. (1961). The Abstract Theory of Automata, *Russian Mathematical Surveys*, 16, 1-53.
DOI: http://dx.doi.org/10.1070/RM1961v016n05ABEH004112

Handley, M., Kreibich C., and Paxson, V. (2001). Network Intrusion Detection: Evasion, Traffic Normalization. *Proc. 10th USENIX Security Symposium*; Obtained through the internet:
Web: http://www.icir.org/vern/papers/norm-usenix-sec-01.pdf [accessed 1 Oct. 2013]

Hariguchi, Y. (2002). ART – Allotment Routing Table – A Fast Free Multibit Trie Based Routing Table; Obtained through the internet:
Webpage: www.hariguchi.org/art [accessed 1 Oct. 2013]

Hayes, C.L. and Luo, Y. (2007). DPICO: a high speed deep packet inspection engine using compact finite automata. *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems* (ANCS). pp. 195-203
DOI: http://dx.doi.org/10.1145/1323548.1323579

Hoffman, D. and Strooper, P. (1997). Classbench: A Framework for Automated Class Testing. *Software Practice and Experience.* 27(5).
DOI: dx.doi.org/10.1002/(SICI)1097-024X(199705)27:5%3c573::AID-SPE98%3e3.0.CO;2-3
Web: http://www.arl.wustl.edu/classbench/ [accessed 1 Oct. 2013]

Hopcroft, J.E., Motwani, R. and Ullman, J.D. (2006). *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Horspool, R.N. (1980). Practical fast searching in strings. Software: Practice and Experience. 10(6), pp.501-506.
DOI: http://dx.doi.org/10.1002/spe.4380100608

Intilop Corp. (2012). 10 G Bit TCP Offload Engine (TOE) – Hardware IP Core, INT 20012. *Top Level Product Specifications*, 2012; Obtained through the Internet:
Web: http://www.intilop.com [accessed 1 Oct. 2013].

Ipoque (2013). Intel Labs using DPI from Ipoque. *Case Study*; Obtained through the internet:
Web:
http://www.ipoque.com/sites/default/files/mediafiles/documents/CS_Intel_1308.pdf
[accessed 1 Dec. 2013].

ITU (2013). Key ICT indicators for developed and developing countries and the world (totals and penetration rates). *International Telecommunications Union (ITU)*, Geneva, 27 Feb. 2013; Obtained through the internet:
Web: www.itu.int/en/ITU-D/Statistics/Documents/statistics/2013/ITU_Key_2005-2013_ICT_data.xls
[accessed 1 Oct. 2013].

Jaiswal, S., Iannaccone, G., Diot, C., Kurose, J. and Towsley, D. (2007). Measurement and Classification of Out-of-Sequence Packets in a Tier-1 IP Backbone. *IEEE/ACM Transactions on Networking (TON)* 2007; 15(1), pp.54-66.
DOI: http://dx.doi.org/10.1109/TNET.2006.890117

Jiang, W. and Prasanna, V.K. (2009). Field-Split Parallel Architecture for High Performance Multi-Match Packet Classification Using FPGAs. *Proceedings of 21st ACM Symposium on Parallelism in Algorithms and Architectures (**SPAA** '09).* pp.188-196.
DOI: http://dx.doi.org/10.1145/1583991.1584044

Jones, G. (2006). The 10 Most Destructive PC Viruses Of All Time, CRN Magazine, 5 July 2006; Obtained through the internet:
Web: www.crn.com/news/security/190300322/the-10-most-destructive-pc-viruses-of-all-time.htm
[accessed 1 Oct. 2013].

Kaneta, K., Yoshizawa, S., Minato, S., Arimura, H. and Miyanaga. Y. (2010). Dynamic reconfigurable bit-parallel architecture for large-scale regular expression matching. In *Proceedings of the International Conference on Field-Programmable Technology (FPT 2010),* pp.21-28.
DOI: http://dx.doi.org/10.1109/FPT.2010.5681536

Kennedy, A., Wang, X. and Liu, B. (2008). Energy Efficient Packet Classification Hardware Accelerator. *Proceedings of 2008 IEEE International Parallel & Distributed Processing Symposium (IPDPS 2008).* pp.1-8.
DOI: http://dx.doi.org/10.1109/IPDPS.2008.4536216

Kennedy, A., Wang, X., Liu, Z. and Liu, B. (2010). Ultra-high throughput string matching for Deep Packet Inspection. *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '10).* pp.399-404.
DOI: http://dx.doi.org/10.1109/DATE.2010.5457172

Kinane, A. (2006). Energy Efficient Hardware Acceleration of Multimedia Processing Tools. *PhD thesis,* Dublin City University, Ireland. Advisor: Noel E. O'Connor.
Webpage: http://doras.dcu.ie/17985/

Knuth, D.E., Morris, J. and Pratt, V.R. (1977). Fast Pattern Matching in Strings. *SIAM Journal on Computing*. 6(2), pp.323-350
DOI: http://dx.doi.org/10.1137/0206024

Kumar, S., Dharmapurikar, S., Yu, F., Crowley, P. and Turner, J. (2006a). Algorithms to accelerate multiple regular expressions matching for deep packet inspection. *SIGCOMM Comput. Commun. Rev.* 36(4), pp.339-350.
DOI: http://dx.doi.org/10.1145/1151659.1159952

Kumar, S., Turner, J. and Williams, J. (2006b). Advanced algorithms for fast and scalable deep packet inspection. *Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems* (ANCS '06). pp.81-92.
DOI: http://dx.doi.org/10.1145/1185347.1185359

Kumar, S., Chandrasekaran, B., Turner, J. and Varghese, G. (2007). Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia. *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems(ANCS '07)*. pp.155-164
DOI: http://dx.doi.org/10.1145/1323548.1323574

Lanzisera, S., Nordman, B. and Brown, R.E. (2012). Data network equipment energy use and savings potential in buildings. Springer Energy Efficiency. 5(2), pp.149-162.
DOI: http://dx.doi.org/10.1007/s12053-011-9136-4

Lee, T.H. (2009). Hardware Architecture for High-Performance Regular Expression Matching. *IEEE Transactions on Computers*, 58(7), pp.984-993.
DOI: http://dx.doi.org/10.1109/TC.2008.145

Long, L.H., Hieu, T.T., Tai, V.T., Hung, N.H., Thinh, T.N., Anh Vu, D.D. (2010). Enhanced FPGA-based architecture for regular expression matching in NIDS. *Proceedings International Conference on Electrical Engineering/Electronics Computer Telecommunications and Information Technology* (ECTI-CON) 2010, pp. 666–670; Obtained through the internet:
Web: http://www.ecti-thailand.org/assets/papers/1083_pub_34.pdf [accessed 1 Oct. 2013].

McDonald, J. (1998). Defeating Sniffers and Intrusion Detection Systems. *Phrack Magazine*, 8(54); Obtained through the internet:
Web: http://www.phrack.org/issues.html?issue=54&id=10 [accessed 1 Oct. 2013].

McNaughton, R., Yamada, H. (1960). Regular Expressions and State Graphs for Automata. *IRE Transactions on Electronic Computers,* EC-9(1), pp.39-47.
DOI: http://dx.doi.org/10.1109/TEC.1960.5221603

Malan, G.R., Watson, D., Jahanian, F. and Howell, P. (2000). Transport and application protocol scrubbing. *Proceedings of the 19th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, 2000; 3:1381-1390.
DOI: dx.doi.org/10.1109/INFCOM.2000.832535

Markatos, E., Antonatos, S., Polychronakis, M. and Anagnostakis, K. (2002). Exclusion-based Signature Matching for Intrusion Detection. In *Proceedings of IASTED International Conference on Communications and Computer Networks* (CCN 2002); Obtained through the internet:
Web: http://www.ics.forth.gr/carv/papers/2002.CCN02.ExB.ps.gz
[accessed 1 Oct. 2013].

Mathis, J. and Heffner, J. (2007). Packetization Layer Path MTU Discovery. *IETF RFC 4821*.
Web: http://www.ietf.org/rfc/rfc4821.txt

Meiners, C.R., Patel, J., Norige, E., Torng, E. and Liu., A.X. (2010). Fast regular expression matching using small TCAMs for network intrusion detection and prevention systems. In *Proceedings of the 19th USENIX conference on Security* (USENIX Security'10). pp111-126; Obtained through the internet:
Web: http://www.usenix.org/events/sec10/tech/full_papers/Meiners.pdf
[accessed 1 Oct. 2013].

mi2g (2004). MyDoom becomes most damaging malware as SCO is paralysed. *mi2g news alert*. 1 Feb. 2004; Obtained through the internet:
Web: http://www.mi2g.com/cgi/mi2g/press/010204.php [accessed 1 Oct. 2013].

Mogul, J. and Deering, S. (1990) Path MTU Discovery. *IETF RFC 1191*.
Web: http://www.ietf.org/rfc/rfc1191.txt

Morrison, D.R. (1968). PATRICIA—Practical Algorithm To Retrieve Information Coded in Alphanumeric. *Journal of the ACM*, 15(4), pp.514-534.
DOI: http://dx.doi.org/10.1145/321479.321481

Mudge, T., Austin, T., Grunwald, D. (2004). Sim-Panalyzer: The SimpleScalar-Arm Power Modeling Project; Obtained through the internet
Web: http://www.eecs.umich.edu/~panalyzer/ [accessed 1 Jan. 2010]
Web: http://elm.eeng.dcu.ie/~croninb/multimatch/sim-panalyzer.htm [accessed 1 Oct. 2013]

Murray, D. and Koziniec, T. (2012). The state of enterprise network traffic in 2012. *18th Asia-Pacific Conference on Communications (APCC)*, 2012; 179-184.
DOI: http://dx.doi.org/10.1109/APCC.2012.6388126

Naous, J., Gibb, G., Bolouki, S. and McKeown N. (2008). NetFPGA: reusable router architecture for experimental research. In *Proceedings of the ACM workshop on Programmable routers for extensible services of tomorrow (PRESTO '08)*. pp.1-7.
DOI: http://dx.doi.org/10.1145/1397718.1397720

Navarro, G. and Raffinot, M. (2002). *Flexible Pattern Matching in Strings: Practical On-Line Search Algorithms for Texts and Biological Sequences*. Cambridge University Press, New York, NY, USA.
ISBN: 0-521-81307-7

Necker, M., Contis, D. and Schimmel, D. (2002). TCP-Stream Reassembly and State Tracking in Hardware. *Proceedings 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2002)*. pp.286-287.
DOI: http://dx.doi.org/10.1109/FPGA.2002.1106687

Netronome (2012). Network Flow Management Software: IPS/IDS Application Kit; Obtained through the internet:
Web: http://www.netronome.com
[accessed 1 Oct. 2013]

Norton, M. (2004). Optimizing Pattern Matching for Intrusion Detection, *Sourcefire Inc.*; Obtained through the internet:
Web: http://pdf.aminer.org/000/309/890/optimizing_pattern_matching.pdf
[accessed 1 Oct. 2013]

Novak, J. and Sturges, S. (2007). Target-Based TCP Stream Reassembly. Sourcefire Inc.; Obtained through the internet:
Web:
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.73.9007&rep=rep1&type=pdf
[accessed 1 Oct. 2013]

NQ Mobile (2013). 2012 Security Report. *NQ Mobile's Security Lab*; Obtained through the internet:
Web: http://www.nq.com/2012_NQ_Mobile_Security_Report.pdf
[accessed 1 Oct. 2013]

Open Information Security Foundation, n.d.;
Web: http://www.openinfosecfoundation.org/ [accessed 1 Oct. 2013]

Pao, D. (2009). A NFA-based programmable regular expression match engine. In *Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS '09)*. pp.60-61.
DOI: http://dx.doi.org/10.1145/1882486.1882499

Paxson, V. (1999). Bro: a system for detecting network intruders in real-time. *Computer Networks: The International Journal of Computer and Telecommunications Networking*. 31(23-24), pp.2435-2463.
DOI: http://dx.doi.org/10.1016/S1389-1286(99)00112-7

PLDA. (2012) QuickTCP for Xilinx. Product Spec. 2012; Obtained through the internet:
Web: http://www.plda.com/products/fpga-ip/xilinx/fpga-ip-tcpip/quicktcp-xilinx [accessed 1 Oct. 2013].

Ptacek, T.H and Newsham, T.N. (1998). Insertion, Evasion and Denial of Service: Eluding Network Intrusion Detection, *Secure Networks Inc*.
Web: http://insecure.org/stf/secnet_ids/secnet_ids.pdf

Rabin, M.O. and Scott, D. 1959. Finite automata and their decision problems. *IBM J. Res. Dev.* 3(2), pp.114-125.
DOI: http://dx.doi.org/10.1147/rd.32.0114

Raghavan, B. and Ma, J. (2011). The Energy and Emergy of the Internet. *Proceedings of the ACM Workshop on Hot Topics in Networks* (HotNets). Art. 9, 6 pages.
DOI: http://dx.doi.org/10.1145/2070562.2070571

Roesch, M., Green, C. and Sourcefire Inc. (2012), SNORT Users' Manual (November 2012). Retrieved March 1, 2013 from http://manual.snort.org.

Ruggiero, P. and Foote, J. (2011). Cyber Threats to Mobile Phones. *Carnegie Mellon University,* Paper produced for *United States Computer Emergency Readiness Team*; Obtained through the internet:
Web: http://www.us-cert.gov/reading_room/cyber_threats_to_mobile_phones.pdf [accessed 1 Oct. 2013].

Sanny, A., Ganegedara, T. and Prasanna, V.K. (2013). *A Comparison of Ruleset Feature Independent Packet Classification Engines on FPGA*, IEEE 27th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), pp.124-133.
DOI: http://dx.doi.org/10.1109/IPDPSW.2013.249

Sawyer, N. and Defossez, M. (2002). Quad-Port Memories in Virtex Devices, *Xilinx Application Note 228*; Obtained through the internet:
Web: http://www.xilinx.com/support/documentation/application_notes/xapp228.pdf [accessed 1 Oct. 2013].

Schuehler, D.V. and Lockwood, J. (2002). TCP-Splitter: A TCP/IP flow monitor in reconfigurable hardware. *Proceedings of 10th Symposium on High Performance Interconnects*. pp.127-131.
DOI: http://dx.doi.org/10.1109/CONECT.2002.1039268

Schuehler, D.V. and Lockwood, J. (2004). A Modular System for FPGA-Based TCP Flow Processing in High-Speed Networks. *4th International Conference on Field Programmable Logic and Application (FPL)*, Springer LNCS 3203, pp.301-310.
DOI: http://dx.doi.org/10.1007/978-3-540-30117-2_32

Shankar, U. and Paxson, V. (2003). Active mapping: resisting NIDS evasion without altering traffic. *Proceedings of the 2003 Symposium on Security and Privacy*, 2003; 44-61.
DOI: http://dx.doi.org/10.1109/SECPRI.2003.1199327

Shmoo Group, (2009). DEF CON 9.0 capture the flag contest data sets; Obtained through the internet:
`Web: http://ictf.cs.ucsb.edu/data/defcon_ctf_09/ [accessed 1 Feb. 2012].`

Sidhu, R. and Prasanna, V.K. (2001). Fast Regular Expression Matching Using FPGAs. In *Proceedings of the the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines* (FCCM '01). pp.227-238.
`DOI: http://dx.doi.org/10.1109/FCCM.2001.22`

Sinnappan, R. and Hazelhurst, S. (2001). A Reconfigurable Approach to Packet Filtering. *Proceedings of the 11th International Conference on Field-Programmable Logic and Applications* (FPL'01). pp.638-642.
`DOI: http://dx.doi.org/10.1007/3-540-44687-7_70`

Singh, S. and Baboescu, F. (2002). EGT-PC single-match implementation.
`Web: http://elm.eeng.dcu.ie/~croninb/multimatch/PacketClassficiationRepository/trie.c Originally available from www.ial.ucsd.edu/classification`

Singh, S., Baboescu, F., Varghese, G. and Wang J. (2003). Packet Classification Using Multidimensional Cutting. *Proceedings of the 2003 conference on Applications,* technologies, architectures, and protocols for computer communications (SIGCOMM '03). pp.213-224.
`DOI: http://dx.doi.org/10.1145/863955.863980`

Smith, R., Estan, C., Jha, S. and Kong S. (2008). Deflating the big bang: fast and scalable deep packet inspection with extended finite automata. *SIGCOMM Comput. Commun. Rev.* 38(4), pp.207-218.
`DOI: http://dx.doi.org/10.1145/1402958.1402983`

SNORT Frag3 & Stream5 preprocessor documentation. README.frag3 and README.stream5 located in doc folder of the source code tar-ball.
`Web: http://www.snort.org/snort-downloads [accessed 1 Oct. 2013].`

Sommer, R. (2011). The Open Source Bro IDS: Overview and Recent Developments. *CACR Higher Education Cybersecurity Summit*, Presentation, April 2011; Obtained through the internet:
`Web: http://www.icir.org/robin/slides/Bro-CACR-Indianapolis.pdf [accessed 1 Oct. 2013].`

Song, H. and Lockwood, J.W. (2005a). Efficient Packet Classification for Network Intrusion Detection using FPGA. *FPGA '05: Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*. pp.238-245.
`DOI: http://dx.doi.org/10.1145/1046192.1046223`

Song, H. and Lockwood, J.W. (2005b). Multi-pattern Signature Matching for Hardware Network Intrusion Detection Systems. *IEEE Global Telecommunications Conference GLOBECOM'05*. vol.3, 5 pages.
`DOI: http://dx.doi.org/10.1109/GLOCOM.2005.1577937`

Srinivasan, V. and Varghese, G. (1998). Faster IP lookups using controlled prefix expansion. *ACM SIGMETRICS Performance Evaluation Review*. 26(1), pp.1-10.
`DOI: http://dx.doi.org/10.1145/277858.277863`

Srinivasan, V., Varghese, G., Suri, S. and Waldvogel, M (1998). Fast and Scalable Layer-4 Switching. *Proceedings of the ACM SIGCOMM '98 conference on Applications, technologies, architectures, and protocols for computer communication*. pp.191-202.
`DOI: http://dx.doi.org/10.1145/285237.285282`

Sundaramoorthy N, Rao N, Hill T. AXI4 Interconnect Paves the Way to Plug-and-Play IP. *Xilinx White Paper*, 2010; WP379 (v1.0); Obtained through the internet:
`Web: http://www.xilinx.com/support/documentation/white_papers/wp379_AXI4_Plug_and_Play_IP.pdf [accessed 1 Dec. 2013]`

Sung, J., Kang, S., Y. Lee, T. Kwon, and B. Kim (2005). A Multi-gigabit Rate Deep Packet Inspection Algorithm using TCAM. *IEEE Global Telecommunications Conference GLOBECOM'05*. vol.1, 5 pages.
DOI: http://dx.doi.org/10.1109/GLOCOM.2005.1577667

Taleck G. Ambiguity Resolution via Passive OS Fingerprinting. (2003) *Proceedings of the International Conference on Recent Advances in Intrusion Detection (RAID), Lecture Notes in Computer Science, Springer* 2003; 2820:192-206.
DOI: http://dx.doi.org/10.1007/978-3-540-45248-5_11

Tan, L. and Sherwood, T. (2005). A High Throughput String Matching Architecture for Intrusion Detection and Prevention. In *Proceedings of the 32nd annual international symposium on Computer Architecture* (ISCA '05). pp.112-122.
DOI: http://dx.doi.org/10.1109/ISCA.2005.5

Tang, S. (2000). Using Binary Files in VHDL Test Benches. *Application Note, Dept. of Electrical and Computer Engineering, University of Alberta*.
Web:http://www.ece.ualberta.ca/~elliott/ee552/studentAppNotes/2000_w/vhdl/BinaryFileTestbenching/binary.html

Taylor, D.E. (2005). Survey and Taxonomy of Packet Classification Techniques. *ACM Computing Surveys,* 37(3), pp.238-275.
DOI: http://dx.doi.org/10.1145/1108956.1108958

ThinkQuest (2004). Sasser Worm. *Oracle ThinkQuest, Cybercrime – Piercing the darkness*. Oct. 2004.
DOI:http://library.thinkquest.org/04oct/00460/sasser.html

Thompson, K. (1968). Programming Techniques: Regular expression search algorithm. *Communications of the ACM*, 11(6), pp.419-422.
DOI: http://dx.doi.org/10.1145/363347.363387

Tuck, N., Sherwood, T., Calder, B. and Varghese, G. (2004). Deterministic memory-efficient string matching algorithms for intrusion detection. *INFOCOM 2004*. 4, pp.2628-2639. March 2004.
DOI: http://dx.doi.org/10.1109/INFCOM.2004.1354682

Vamosi, R. Sasser.a and Sasser.b prevention and cure. *CNET Reviews*, 3 May 2004.
Web: http://reviews.cnet.com/4520-6600_7-5133023-1.html

Vasiliadis, G., Polychronakis, M., Antonatos, S., Markatos, E.P. and Ioannidis, S. (2009). Regular Expression Matching on Graphics Hardware for Intrusion Detection. *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection* (RAID '09). pp.265-283.
DOI: http://dx.doi.org/10.1007/978-3-642-04342-0_14

Vutukuru, M., Balakrishnan, H. and Paxson, V. (2008). Efficient and Robust TCP Stream Normalization. *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, 2008; 96-110.
DOI: http://dx.doi.org/10.1109/SP.2008.27

Wang, H., Pu, S., Knezek, G. and Liu, J-C. (2010). A modular NFA architecture for regular expression matching. *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays* (FPGA '10). pp.209-218.
DOI: http://dx.doi.org/10.1145/1723112.1723149

Wang, X., Xu, Y., Jiang, J., Ormond, O., Liu, B. and Wang, X. (2013). StriFA: Stride Finite Automata for High-Speed Regular Expression Matching in Network Intrusion Detection Systems. *IEEE Systems Journal*, 7(3), pp.374,384.
DOI: http://dx.doi.org/10.1109/JSYST.2013.2244791

White, J.S., Fitzsimmons, T. and Matthews, J.N. (2013). Quantitative analysis of intrusion detection systems: Snort and Suricata. *Proc. SPIE* 8757, *Cyber Sensing 2013*, 875704.
DOI: http://dx.doi.org/10.1117/12.2015616

Wu, S. and Manber, U. (1992). Fast text searching: allowing errors. *Commun. ACM*, 35(10), pp.83-91.
DOI: http://dx.doi.org/10.1145/135239.135244

Wu, S. and Manber, U. (1994). A fast algorithm for multi-pattern searching. *Technical Report TR94-17, Department of Computer Science, University of Arizona*, 1994.
Web: ftp://ftp.cs.arizona.edu/reports/1994/TR94-17.ps

Wun, B., Crowley, P. and Raghunth, A. (2009). Parallelization of Snort on a multi-core platform. *Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems* (ANCS), pp.173-174.
DOI: http://dx.doi.org/10.1145/1882486.1882528

Xilinx (2012a). Virtex-5 FPGA User Guide. *UG190 (v5.4) March 16, 2012*, Obtained through the internet:
Web: http://www.xilinx.com/support/documentation/user_guides/ug190.pdf
[accessed 1 Dec. 2013].

Xilinx (2012b). Xilinx 7 Series FPGAs: Breakthrough Power and Performance, Dramatically Reduced Development Time. *Xilinx 7 Series Product Brief*; Obtained through the internet:
Web: http://www.xilinx.com/publications/prod_mktg/7-Series-Product-Brief.pdf
[accessed 1 Dec. 2013].

Xilinx (2013). Zynq-7000 All Programmable SoC First Generation Architecture. *Preliminary Product Specification*, 2013; DS190 (v1.6), Obtained through the internet:
Web: http://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf
[accessed 24 Dec. 2013]

Yang, L., Karim, R., Ganapathy, V. and Smith, R. (2010). Improving NFA-based signature matching using ordered binary decision diagrams. *Proceedings of the 13th international conference on Recent advances in intrusion detection* (RAID'10). pp.58-78.
DOI: http://dx.doi.org/10.1007/978-3-642-15512-3_4

Young, G. and Pescatore, J. (2012). Magic Quadrant for Intrusion Prevention Systems. *Gartner Research, 5 July 2012*; Obtained through the internet:
Web: http://www.gartner.com/id=2073115 [accessed 1 Oct. 2013].

Yu, F. (2006). High Speed Deep Packet Inspection with Hardware Support. *Ph.D. thesis*, University of California, Berkeley, CA, USA. Advisor: Randy Katz; Obtained through the internet:
Web: http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-156.pdf
[accessed 1 Jan. 2010].

Yu, F., Chen, Z., Diao, Y., Lakshman, T.V. and Katz, R.H. (2006). Fast and memory-efficient regular expression matching for deep packet inspection. *Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems* (ANCS '06). pp.93-102.
DOI: http://dx.doi.org/10.1145/1185347.1185360

Yu, F. and Katz, R.H. (2004). Efficient Multi-Match Packet Classification with TCAM. *Proceedings 12$^{th}$ IEEE Symposium on Hot Interconnects* (HOTI'04). pp.28-34.
DOI: http://dx.doi.org/10.1109/CONECT.2004.1375197

Yu, F., Katz, R.H. and Lakshman, T.V. (2004). Gigabit Rate Pattern-Matching Using TCAM, *Proceedings of the 12th IEEE International Conference on Network Protocols* (ICNP '04). pp.174-183.
DOI: http://dx.doi.org/10.1109/ICNP.2004.1348108

Yu, F., Lakshman, T.V., Motoyama, M.A. and Katz, R.H. 2005. SSA: A Power and Memory Efficient Scheme to Multi-Match Packet Classification. *Proceedings of the 2005 ACM Symposium on Architectures for Networking and Communications Systems (ANCS)*. pp.105-113.
DOI: http://dx.doi.org/10.1145/1095890.1095905

Yun, S. and Lee, K. (2009). Regular Expression Pattern Matching Supporting Constrained Repetitions. *Proceedings International Workshop on Reconfigurable Computing: Architectures, Tools and Applications* (ARC), (LNCS, 5453), pp. 300–305
DOI: http://dx.doi.org/10.1007/978-3-642-00641-8_32

Zhao, Y., Yuan, R., Wang, W., Meng D., Zhang, S. and Li, J. (2012). A Hardware-based TCP Stream State Tracking and Reassembly Solution for 10G Backbone Traffic. *IEEE 7th International Conference on Networking, Architecture and Storage* (NAS). pp.154-163.
DOI: http://dx.doi.org/10.1109/NAS.2012.24

Zhou, Y. and Wang, X. 2010. Efficient Pattern Matching with Counting Bloom Filter. *CIICT2010*.
ISBN: 978-1-935068-30-3

Zu, Y., Yang, M., Xu, Z., Wang, L., Tian, X., Peng, K. and Dong Q. (2012). GPU-based NFA implementation for memory efficient high speed regular expression matching. *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming* (PPoPP). pp.129-140.
DOI: http://dx.doi.org/10.1145/2145816.2145833