ΠΟΛΥΤΕΧΝΕΙΟ ΚΡΗΤΗΣ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Εργαστήριο Μικροεπεξεργαστών και Υλικού

Διπλωματική Εργασία

Ανίχνευση Πακέτων σε αρχιτεκτονική Maxeler

Μαρία Καπελλάκη

Επιβλέπων καθηγητής : Διονύσιος Ν. Πνευματικάτος

Καθηγητής Πολυτεχνείου Κρήτης

Χανιά, Σεπτέμβριος 2016

ΠΟΛΥΤΕΧΝΕΙΟ ΚΡΗΤΗΣ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Εργαστήριο Μικροεπεξεργαστών και Υλικού

Διπλωματική Εργασία

Ανίχνευση Πακέτων σε αρχιτεκτονική Maxeler

Μαρία Καπελλάκη

Επιβλέπων καθηγητής : Διονύσιος Ν. Πνευματικάτος

Καθηγητής Πολυτεχνείου Κρήτης

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή:

Διονύσιος Ν. Πνευματικάτος, Καθηγητής Πολυτεχνείου Κρήτης

Πολυχρόνης Κουτσάκης, Αναπληρωτής Καθηγητής Πολυτεχνείου Κρήτης

Ιωάννης Παπαευσταθίου, Αναπληρωτής Καθηγητής Πολυτεχνείου Κρήτης

Χανιά, Σεπτέμβριος 2016

Περίληψη

Τα τελευταία χρόνια το διαδίκτυο αποτελεί ένα μεγάλο κομμάτι της καθημερινής μας ζωής καθώς η πρόσβαση σε αυτό είναι πλέον πολύ εύκολη, μέσω διάφορων συσκευών. Βάσει αυτού έχει δημιουργηθεί η ανάγκη για αποτελεσματικά συστήματα προστασίας και ασφάλειας. Η ανάπτυξη των συστημάτων **Ανίχνευσης Εισβολών Δικτύου** (Network Intrusion Detection Systems – NIDS) είναι ραγδαία και αυτά αποτελούν την εξέλιξη των firewalls, μιας και εκτός από το να ελέγχουν τις επικεφαλίδες των διερχόμενων πακέτων ελέγχουν και τα περιεχόμενα (payload) του πακέτου για την ύπαρξη συγκεκριμένων ύποπτων συμβολοσειρών. Ένα από τα πιο διαδεδομένα συστήματα ανίχνευσης εισβολών δικτύου είναι το **Snort**, το οποίο είναι ένα εργαλείο ανοιχτού κώδικα και διατίθεται δωρεάν. Η αναζήτηση συμβολοσειρών στα δεδομένα ενός πακέτου δεν είναι απλή διαδικασία και μπορεί να γίνει πολύ απαιτητική τόσο σε χρόνο, όσο και σε πόρους συστήματος, όταν μιλάμε για υψηλούς ρυθμούς μετάδοσης. Μία λύση σε αυτό μπορεί να προσφέρει η χρήση υλικού, καθώς μπορεί να παρέχει ρυθμούς επεξεργασίας πολύ μεγαλύτερους από εκείνους του λογισμικού. Η εργασία επικεντρώθηκε στην δημιουργία ενός συστήματος ανίχνευσης εισβολών δικτύου, έχοντας ως σύστημα αναφοράς, το σύστημα Snort, στον υβριδικό υπερυπολογιστή της εταιρείας Maxeler Technologies. Ο συγκεκριμένος υπερυπολογιστής, κάνοντας χρήση αρχιτεκτονικών που συνδυάζουν επεξεργαστές γενικού σκοπού με αναδιατασσόμενη λογική, κάνει δυνατή την κατασκευή συστημάτων με εξαιρετικές δυνατότητες. Στην παρούσα διπλωματική εργασία δημιουργήθηκε το σύστημα ταυτοποίησης των περιεχομένων του πακέτου, το σύστημα ταυτοποίησης της κεφαλίδας του πακέτου, το σύστημα ταυτοποίησης του κανόνα, καθώς και ένα πρόγραμμα αυτόματης παραγωγής κώδικα για τις ανάγκες του compiler της Maxeler, συνθέτοντας έτσι ένα ολοκληρωμένο σύστημα ανίχνευσης εισβολών δικτύου, συμβατό με τους κανόνες Snort.

TECHNICAL UNIVERSITY OF CRETE, GREECE

SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

Microprocessor and Hardware Laboratory

Diploma Thesis

Packet Detection on Maxeler Architecture

Maria Kapellaki

Supervisor : Dionisios N. Pnevmatikatos

Professor of Technical University of Crete

Chania, September 2016

Abstract

Recent years, the Internet is a big part of our everyday life because access to it is very easy, through various devices. The need for effective protection and safety systems is necessary. Recently, **Network Intrusion Detection Systems - NIDS**, have been developed, often as an upgrade of firewalls; NID Systems not only check the headers of passing packet, but they also check the payload of the packet for the existence of specific suspicious strings. One of the most popular Network Intrusion Detection systems is **Snort**, which is a free and open source tool. We can easily understand that one string search in the data of a packet is not straightforward and can be very demanding in terms of time, and on system resources when we talk about high bit rates. A solution to this can be the use of hardware, as it can provide much higher processing rates than those of the software. This thesis is focused on the creation of such a system, on a hybrid supercomputer of the Maxeler Technologies Company, having SNORT as a reference system. Maxeler supercomputers, use an architecture that combines a general purpose processor with reconfigurable logic and leads to the construction of systems with excellent potentialities. In this thesis, we created a system for content matching, header matching, rule matching, as well as an automatic generator of code needed by Maxeler's compiler, thus composing a complete network intrusion detection system, following the principles and syntax of the Snort rules.

Στους γονείς και την αδερφή μου...

Ευχαριστίες

Αρχικά, θα ήθελα να ευχαριστήσω τον επιβλέπων καθηγητή μου, κ. **Διονύσιο Πνευματικάτο** που με βοήθησε να ασχοληθώ με ένα θέμα που να μου αρέσει, καθώς και για την καθοδήγηση του καθ΄ όλη την διάρκεια εκπόνησης της διπλωματικής μου εργασίας.

Θα ήθελα επίσης, να ευχαριστήσω τους κ. **Κουτσάκη Πολυχρόνη** και **Παπαευσταθίου Ιωάννη** για τη συμβολή τους στην εκπόνηση της εργασίας μου ως μέλη της τριμελούς εξεταστικής επιτροπής.

Ένα πολύ μεγάλο ευχαριστώ στον Δρ. **Γρηγόρη Χρυσό** που σε όλη την διάρκεια των σπουδών μου φρόντιζε να μου δίνει όση περισσότερη από τη γνώση του και που κατά τη διάρκεια εκπόνησης της διπλωματικής μου εργασίας ήταν πάντα εκεί, σε ό,τι και αν χρειαζόμουν, ξοδεύοντας πολύ από τον πολύτιμο χρόνο του πάνω από τον κώδικα μου. Επίσης, ευχαριστώ τον κ. **Π. Μαλακωνάκη** για τη συμβολή του στο κομμάτι της Maxeler. Ακόμη, θέλω να ευχαριστήσω τον κ. **Ε. Κουτρούλη** που ως σύμβουλος σπουδών μου, μου έδωσε θάρρος και με βοήθησε να βάλω στόχους. Τον ευχαριστώ επίσης, που με μεγάλη προθυμία δέχτηκε να στείλει συστατική επιστολή όταν τη χρειάστηκα.

Θα ήταν παράληψη να μην ευχαριστήσω την μικρή μου αδερφή **Εύα** που είναι πάντα στο πλευρό μου. Μπορώ επίσης, να τη συγχωρήσω που πήρε το πτυχίο της πριν από εμένα. Κάπου εδώ πρέπει να ευχαριστήσω όλους τους φίλους μου για τα όμορφα φοιτητικά χρόνια που είχαμε, τις αναμνήσεις που δημιουργήσαμε στα Χανιά, τα γέλια και τα κλάματα. (Ιδιαιτέρα τις Δ.Λ., Ν.Ξ. και Α.Ν.) Επίσης, τους Α.Α., Ω.Π. και Δ.Α. για τις πρόβες της παρουσίασης.

Τέλος, το μεγαλύτερο ευχαριστώ είναι για τους γονείς μου, **Χάρη** και **Αδριάνη**. Για την αμέριστη αγάπη που μου δίνουν, που ήταν δίπλα μου σε κάθε όμορφη και άσχημη στιγμή όλα αυτά χρόνια, που συνεχίζουν να με στηρίζουν σε όποια απόφαση και αν πάρω, που μου έδωσαν σωστά εφόδια για να πορευτώ στη ζωή μου και που χωρίς αυτούς θα ήταν αδύνατο να φτάσω ως εδώ, αλλά και να συνεχίσω.

Χανιά, Σεπτέμβριος 2016

Μαρία.

Κατάλογος Σχημάτων

2.1 Λογότυπο Snort 19
2.2 Πεδία κεφαλίδας κανόνα 21

3.1 FPGA 31
3.2 Λογότυπο της εταιρείας Maxeler Technologies 32
3.3 Η επαναχρησιμοποίηση των λειτουργικών μονάδων μίας CPU στο χρόνο . 33
3.4 Ένα dataflow πρόγραμμα. 34
3.5 Αρχιτεκτονική της DFE 36
3.6 Η αρχιτεκτονική του συστήματος της Maxeler 37
3.7 Διάγραμμα ροής της πορείας σχεδίασης 38

4.1 Block diagram κανόνα 40
4.2 Block diagram Μονάδας ελέγχου δεδομένων πακέτου 40
4.3 Σχηματική αναπαράσταση παραδείγματος 42
4.4 Λογικό διάγραμμα μονάδας ελέγχου συμβολοσειράς μήκους 3 43
4.5 Block diagram Μονάδας ελέγχου κεφαλίδας 44
4.6 Μορφή κεφαλίδας IPv4 πακέτου 45
4.7 Λογικό διάγραμμα ελέγχου πρωτοκόλλου 46
4.8 Λογικό διάγραμμα ελέγχου διεύθυνσης IP αποστολέα 47
4.9 Λογικό διάγραμμα ελέγχου θύρας αποστολέα 48
4.10 Λογικό διάγραμμα τελικού ελέγχου κεφαλίδας πακέτου 49
4.11 Block diagram τελικής μονάδας ελέγχου κανόνα 50
4.12 Λογικό διάγραμμα τελικής μονάδας ελέγχου κανόνα 50
4.13 Σχηματική αναπαράσταση της State Machine 51
4.14 Ολοκληρωμένο Block Diagram 58

5.2 Οι γραφικές παραστάσεις του ρυθμού επεξεργασίας όπως προέκυψε πειραματικά (rate), του ρυθμού επεξεργασίας μετά την αφαίρεση του overhead (actual rate) και ο μέσος όρος αυτού, καθώς είδαμε παραμένει σχεδόν σταθερός (avg rate) 61
5.3 Σχηματικά Διαγράμματα δοκιμαστικών σχεδιάσεων 64

Κατάλογος Πινάκων

5.1 Πίνακας χωρικής επίδοσης για 48 κανόνες 59
5.2 Πίνακας χρονικής επίδοσης του συστήματος 60
5.3 Πίνακας χρονικής επίδοσης του συστήματος 60
5.4 Συγκριτικός πίνακας χωρικής επίδοσης διαφορετικών σχεδιάσεων, 1 . . . 64
5.5 Συγκριτικός πίνακας χωρικής επίδοσης διαφορετικών σχεδιάσεων, 2 . . . 65
5.6 Συγκριτικός πίνακας χωρικής επίδοσης δεύτερης σχεδίασης ανάμεσα στις δύο προσεγγίσεις 65

Περιεχόμενα

Κατάλογος Σχημάτων **8**

Κατάλογος Πινάκων **10**

1 Εισαγωγή **13**

1.1 Κίνητρα 13

1.2 Συνεισφορά 15

1.3 Διάρθρωση κειμένου 16

2 Σχετικές εργασίες-Λύσεις **18**

2.1 Λύσεις σε λογισμικό 18

2.1.1 Κανόνες Snort 19

2.1.2 Περιγραφή των κανόνων Snort 19

2.2 Λύσεις βασισμένες σε υλικό 23

2.2.1 Content Addressable Memory 23

2.2.2 FPGA Implementations 24

2.2.3 Multi-core network processors (NP) 26

2.2.4 Hardware DPI Systems with Network Interface 27

2.2.5 Κάρτες γραφικών 28

3 Η αρχιτεκτονική του συστήματος Maxeler **30**

3.1 Υπερυπολογιστές με FPGA 30

3.1.1 Αναδιατασσόμενη λογική-FPGA 30

3.1.2 Υβριδικοί Υπερυπολογιστές 31

3.2 Maxeler 32

3.2.1 Dataflow vs Control flow model 33

3.2.2 DFEs 35
3.2.3 Η αρχιτεκτονική του συστήματος 36

4 Αρχιτεκτονική και Υλοποίηση **39**
4.1 Το Block Diagram του Πυρήνα 40
4.2 Content Match 40
4.3 Header Match 44
4.3.1 Έλεγχος πρωτοκόλλου 46
4.3.2 Έλεγχος διεύθυνσης IP αποστολέα 47
4.3.3 Έλεγχος διεύθυνσης IP προορισμού 47
4.3.4 Έλεγχος θύρας αποστολέα 48
4.3.5 Έλεγχος θύρας προορισμού 48
4.3.6 Τελικός έλεγχος κεφαλίδας πακέτου 48
4.4 Rule Match 49
4.5 Εναλλαγή πακέτων με State Machine 51
4.6 Αυτόματη παραγωγή κώδικα 54
4.7 CPU 55
4.8 Ολοκληρωμένο Block Diagram 55

5 Αποτελέσματα **59**
5.1 Αποτελέσματα 59
5.2 Διδάγματα που αποκομίστηκαν 61

6 Συμπεράσματα-Μελλοντικές Εργασίες **67**
6.1 Συμπεράσματα 67
6.2 Μελλοντικές Εργασίες 68

Παράρτημα **70**

Βιβλιογραφία **75**

Κεφάλαιο 1

Εισαγωγή

1.1 Κίνητρα

Στη δεκαετία του 1980, η έρευνα στο CERN στην Ελβετία, είχε ως αποτέλεσμα το World Wide Web. Από τα μέσα της δεκαετίας του 1990, το Internet έχει έναν επαναστατικό αντίκτυπο στον πολιτισμό, το εμπόριο και την τεχνολογία, συμπεριλαμβανομένης της αύξησης της σχεδόν στιγμιαίας επικοινωνίας μέσω ηλεκτρονικού ταχυδρομείου, instant messaging, VoIP τηλεφωνικές κλήσεις, αμφίδρομες διαδραστικές βίντεο-κλήσεις, και ο παγκόσμιος ιστός με φόρουμ συζητήσεων, ιστολόγια, κοινωνική δικτύωση, ηλεκτρονικές αγορές. Πλέον, το διαδίκτυο αποτελεί ένα μεγάλο κομμάτι της καθημερινής μας ζωής καθώς η πρόσβαση σε αυτό είναι πολύ εύκολη, μέσω διάφορων συσκευών. Παράλληλα, η ταχύτητα πρόσβασης σε αυτό έχει οδηγήσει σε βελτίωση της ποιότητας των παρεχόμενων υπηρεσιών σε ενημέρωση, ψυχαγωγία, αγορές, επιχειρηματικές δραστηριότητες κ.α.

Τα παραπάνω έχουν δημιουργήσει την ανάγκη για αποτελεσματικά συστήματα προστασίας και ασφάλειας. Συστήματα που θα επιτρέπουν/αποτρέπουν σε χρήστες να εκτελούν εργασίες ή θα τους εξουσιοδοτούν για αυτές. Δεν είναι λίγα τα παραδείγματα των επιτυχημένων εισβολών σε δίκτυα και διαδικτυακές υπηρεσίες υψηλού προφίλ εταιρειών. Για τον λόγο αυτόν, έχουν αναπτυχθεί πολλές μέθοδοι για να διασφαλίσουν την ασφάλεια των υποδομών και της επικοινωνίας μέσω του διαδικτύου.

Ένα από τα πιο διαδεδομένα συστήματα προστασίας είναι τα **τείχη προστασίας** (firewalls) τα οποία συνήθως υλοποιούνται σε λογισμικό, αλλά και σε υλικό. Η λειτουργία αυτών των συστημάτων είναι να εξετάζουν τις επικεφαλίδες (headers) των διερχομένων

πακέτων και να επιτρέπουν ή να απαγορεύουν την πρόσβαση σε συγκεκριμένους Η/Υ, χρήστες κ.α., ενώ μπορούν να ελέγχουν και το είδος της επικοινωνίας που γίνεται. Τα πιο γνωστά είναι από Cisco[7] , Netscreen[8] και Checkpoint[9], ενώ υπάρχει και το ανοιχτού κώδικα netfilter[10].

Άλλα διαδεδομένα συστήματα προστασίας είναι τα **Εργαλεία Αξιολόγησης Ευπάθειας Δικτύου**, τα οποία ουσιαστικά ανιχνεύουν ελλείψεις στην προστασία του δικτύου. Οι πληροφορίες που συλλέγονται από αυτά τα εργαλεία καθορίζουν τους κανόνες προστασίας του δικτύου. Παράδειγμα τέτοιων εργαλείων είναι το nMap [11], το οποίο είναι ένα ανοικτού κώδικα εργαλείο για την εξερεύνηση του δικτύου και τον έλεγχο της ασφάλειας. Λειτουργεί ως σαρωτής ασφαλείας και χρησιμοποιείται για να ανακαλύψει κεντρικούς υπολογιστές και τις υπηρεσίες σε ένα δίκτυο υπολογιστών, δημιουργώντας έτσι ένα ¨χάρτη' του δικτύου. Η λειτουργία του παρέχει στο χρήστη μία αναλυτική εικόνα, του προς ελέγχου δικτύου φανερώνοντας πιθανά προβλήματα και ελλείψεις ασφάλειας.

Τα τελευταία χρόνια έχουν αναπτυχθεί τα συστήματα **Ανίχνευσης Εισβολών Δικτύου** (Network Intrusion Detection Systems – NIDS), τα οποία αποτελούν την εξέλιξη των firewalls, μιας και εκτός από το να ελέγχουν τις επικεφαλίδες των διερχόμενων πακέτων ελέγχουν και τα περιεχόμενα (payload) του πακέτου για την ύπαρξη συγκεκριμένων ύποπτων συμβολοσειρών. Ένα από τα πιο διαδεδομένα συστήματα ανίχνευσης εισβολών δικτύου είναι το **Snort**, το οποίο είναι ένα εργαλείο ανοιχτού κώδικα και διατίθεται δωρεάν.

Ταυτόχρονα, η ταχύτητα μετάδοσης στα δίκτυα συνεχώς αυξάνεται. Το να ακολουθήσει η ταχύτητα επεξεργασίας τις τάσεις της ταχύτητας μετάδοσης, αποτελεί πρόκληση. Ειδικά όταν πλέον μιλάμε για ταχύτητες στο επίπεδο των 40 και 100Gbps.

Τα συστήματα ανίχνευσης εισβολών δικτύου (NIDS) παρέχουν μεγάλη ευελιξία στη χρήση τους, παράδειγμα το σύστημα Snort δίνει τη δυνατότητα στους χρήστες να το παραμετροποιήσουν και να αλλάξουν τους κανόνες, βάσει των αναγκών τους. Ακολουθεί ένα παράδειγμα κανόνα.

```
alert tcp $HOME_NET 2589->$EXTERNAL_NET any
(msg:"MALWARE-BACKDOOR-Dagger_1.4.0"; flow:to_client,established;
content:"2 | 00 00 00 06 00 00 00|"; depth:16;
metadata:ruleset community;classtype:misc-activity; sid:105; rev:14;)
```

Μπορεί εύκολα να καταλάβει κανείς ότι η αναζήτηση συμβολοσειρών στα δεδομένα ενός πακέτου δεν είναι απλή υπόθεση και μπορεί να γίνει πολύ απαιτητική τόσο σε χρόνο, όσο και σε πόρους συστήματος. Αν λάβουμε υπόψιν μας και την περίπτωση που η διερχόμενη κίνηση πακέτων από το δίκτυο είναι μεγαλύτερη από αυτή που μπορεί να επεξεργαστεί το σύστημα, μπορεί το σύστημα να γίνει εύκολα αναξιόπιστο, καθώς υπάρχει το ενδεχόμενο ορισμένα επικίνδυνα πακέτα να καταφέρουν να διαφύγουν από τον έλεγχο του. Επίσης, λόγω ταχύτητας επεξεργασίας μπορεί να έχουμε μεγάλες καθυστερήσεις των πακέτων και μεγάλο ποσοστό απώλειας αυτών.

Μία λύση στο παραπάνω πρόβλημα μπορεί να προσφέρει η χρήση υλικού, καθώς μπορεί να παρέχει ρυθμούς επεξεργασίας πολύ μεγαλύτερους από εκείνους του λογισμικού.

Τα παραπάνω γεγονότα και τάσεις αποτέλεσαν και το κίνητρο αυτής της διπλωματικής εργασίας, στην οποία εξετάζονται τρόποι επιτάχυνσης και διευκόλυνσης της λειτουργίας των συστημάτων ανίχνευσης εισβολών δικτύου, βασιζόμενοι στο σύστημα κανόνων Snort, με τη χρήση κατάλληλων διατάξεων υλικού.

1.2 Συνεισφορά

Το εργαστήριο Μικροεπεξεργαστών και Υλικού του Πολυτεχνείου Κρήτης έχει επανειλημμένα ασχοληθεί με την επιτάχυνση των συστημάτων ανίχνευσης εισβολών δικτύου, έχοντας πληθώρα δημοσιεύσεων σε αυτόν τον τομέα. Η συγκεκριμένη διπλωματική εργασία αποτελεί το έναυσμα για την ενασχόληση και συσχέτιση των συστημάτων ανίχνευσης εισβολών κάνοντας χρήση υβριδικών υπερυπολογιστών βασιζόμενων σε αναδιατασσόμενη λογική.

Η εργασία επικεντρώθηκε στην δημιουργία ενός τέτοιου συστήματος, στον υβριδικό

υπερυπολογιστή της εταιρείας Maxeler Technologies, έχοντας ως σύστημα αναφοράς, το σύστημα Snort. Ο συγκεκριμένος υπερυπολογιστής, όπως και της Convey computers[15], κάνοντας χρήση αρχιτεκτονικών που συνδυάζουν τους επεξεργαστές γενικού σκοπού με αναδιατασσόμενη λογική, μπορεί να οδηγήσει στην κατασκευή συστημάτων με εξαιρετικές δυνατότητες, τις οποίες και θα προσπαθήσουμε να αναλύσουμε παρακάτω.

Στην παρούσα διπλωματική εργασία δημιουργήθηκε το σύστημα ταυτοποίησης των περιεχομένων του πακέτου, το σύστημα ταυτοποίησης της κεφαλίδας του πακέτου, το σύστημα ταυτοποίησης του κανόνα, καθώς και ένα πρόγραμμα αυτόματης παραγωγής κώδικα για τις ανάγκες του compiler της Maxeler. Συνθέτοντας έτσι, ένα ολοκληρωμένο σύστημα ανίχνευσης εισβολών δικτύου, ακολουθώντας τις αρχές των κανόνων Snort.

Όσον αφορά τα αποτελέσματα αυτής της εργασίας, το κόστος αναζήτησης ανά χαρακτήρα που προέκυψε είναι 0.022% της λογικής, 0.016% των LUT's, 0.019% των Primary Fifos, 0.008% των Secondary Fifos και 0.006% της Block Memory, που παρέχει ο υπερυπολογιστής της Maxeler. Η επίδοση του συστήματος μας στο χρόνο είναι 13.05MB/sec για είσοδο 0.5MB, φτάνει τα 84.65MB/sec για είσοδο 256MB και φυσικά αυξάνεται για μεγαλύτερη είσοδο.

Μέσω της παρούσας εργασίας έγινε βαθύτερη εξοικείωση με τη χρήση του υβριδικού υπερυπολογιστή της Maxeler Technologies και των δυνατοτήτων που αυτός προσφέρει.

1.3 Διάρθρωση κειμένου

ε Σε αυτή την εργασία θα αναλύσουμε τα στάδια για τη δημιουργία του συστήματος μας, καθώς και θα αναφερθούμε με περισσότερες λεπτομέρειες στην σχεδίαση και την αρχιτεκτονική που υλοποιήθηκαν. Παράλληλα θα παρουσιάσουμε τα εργαλεία που χρησιμοποιήθηκαν για τη δημιουργία του και τη βιβλιογραφία που χρησιμοποιήσαμε.

Στο επόμενο Κεφάλαιο θα αναφερθούμε σε διάφορες άλλες λύσεις που έχουν υλοποιηθεί, τι προσφέρουν και ποιες ανάγκες καλύπτουν.

Στο τρίτο Κεφάλαιο θα παρουσιάσουμε τους κανόνες που χρησιμοποιούμε, ως σύστημα αναφοράς.

Στο τέταρτο Κεφάλαιο θα παρουσιάσουμε την σχεδίαση του συστήματος μας. Θα κάνουμε μια εισαγωγή στους υβριδικούς υπερυπολογιστές και την αναδιατασσόμενη λογική, ενώ θα παρουσιάσουμε και την πλατφόρμα που χρησιμοποιήσαμε για τη σχεδίαση

μας.

Στο πέμπτο Κεφάλαιο θα παρουσιάσουμε την αρχιτεκτονική του συστήματος μας. Θα δούμε το ολοκληρωμένο σχηματικό διάγραμμα, παρουσιάζοντας και τα επιμέρους κομμάτια (υποσυστήματα) από τα οποία αποτελείται το σύστημά μας.

Στο έκτο Κεφάλαιο θα παρουσιαστούν τα αποτελέσματα της δουλειάς μας, ενώ θα γίνει αναφορά σε όλους τους ελέγχους και τις δοκιμές που έγιναν. Σημαντικό σημείο αποτελεί και η παράγραφος με τα διδάγματα που αποκομίστηκαν από αυτή τη δουλειά.

Στο έβδομο και τελευταίο Κεφάλαιο θα μιλήσουμε για τα συμπεράσματα στα οποία καταλήξαμε μετά την υλοποίηση του συστήματος και τέλος θα αναφερθούμε σε διάφορες προτάσεις για τη μελλοντική βελτίωση του συστήματος μας.

Κεφάλαιο 2

Σχετικές εργασίες-Λύσεις

Σε αυτό το κεφάλαιο θα γίνει μία παρουσίαση των υπαρχόντων λύσεων στα συστήματα ανίχνευσης εισβολών δικτύου. Δεδομένου ότι η λύση που εμείς προτείνουμε είναι σε υλικό, θα δοθεί μεγαλύτερη έμφαση σε αυτές, ώστε να έχουμε καλύτερη εικόνα των εργασιών που έχουν γίνει και να μπορέσουμε να κάνουμε τη σύγκριση στο τέλος.

2.1 Λύσεις σε λογισμικό

Πολλές εφαρμογές ελέγχου πακέτων χρησιμοποιούν Deep Packet Inspection Systems. Οι πιο διάσημες είναι το Snort[13], Bro[14] και L7-filter[12] για Linux. Το Snort και το Bro είναι συστήματα ανίχνευσης εισβολών δικτύου, ενώ το L7-filter είναι μια εφαρμογή για τα πρωτόκολλα στο επίπεδο εφαρμογής, το οποίο κάνει κατηγοριοποίηση πακέτων βάσει των δεδομένων στο επίπεδο εφαρμογής (επίπεδο 7 στο OSI). Όλα αυτά τα συστήματα είναι ανοιχτού κώδικα και μας επιτρέπουν μια λεπτομερή ανάλυση δείχνοντας τις δυνατότητες και τους περιορισμούς τους. Στη συνέχεια θα καταπιαστούμε με το Snort, δεδομένου ότι είναι ένα από τα διασημότερα συστήματα ανίχνευσης εισβολών δικτύου.

2.1.1 Κανόνες Snort

Σχήμα 2.1: Λογότυπο Snort

Το Snort πρόκειται για ένα σύστημα ανίχνευσης εισβολών, ικανό για ανάλυση της κυκλοφορίας και καταγραφής των πακέτων σε πραγματικό χρόνο σε Πρωτόκολλο Διαδικτύου (Internet Protocol-IP). Επίσης, είναι ικανό για ανάλυση πρωτοκόλλου, αναζήτηση/ταίριασμα περιεχομένου και μπορεί να χρησιμοποιηθεί για την ανίχνευση πληθώρας επιθέσεων, όπως υπερχείλιση καταχωρητών προσωρινής μνήμης (buffer overflow), κλοπή πόρτας (stealth scan port), επιθέσεις κοινής πυλαίας διεπαφής (Common Gateway Interface-CGI attacks), ανιχνεύσεις παραβίασης πρωτοκόλλου εξυπηρετητή (Server Message Block-SMB detection), απόπειρες υποκλοπής αποτυπώματος του λειτουργικού συστήματος (OS fingerprint attempts), και πολλά άλλα. Το Snort έχει τρεις πρωταρχικές χρήσεις. Μπορεί να χρησιμοποιηθεί:

1. Απευθείας ως αναλυτής πακέτων, όπως το tcpdump
2. Ως καταγραφέας πακέτων (χρήσιμο για αποσφαλμάτωση της κίνησης στο δίκτυο)
3. Ως ένα πλήρες σύστημα αποτροπής εισβολών δικτύου

Όπως αναφέραμε και νωρίτερα, όντας ένα ανοιχτού κώδικα σύστημα δίνει τη δυνατότητα στους χρήστες να αλλάξουν τους κανόνες που το απαρτίζουν και να εξατομικεύσουν το σύστημα σύμφωνα με τις προσωπικές τους ανάγκες. Αυτή τη στιγμή κυκλοφορεί η σταθερή έκδοση 2.9.8.0, με ημερομηνία κυκλοφορίας 17/11/2015, διαθέσιμη για πληθώρα λειτουργικών συστημάτων (Linux, Windows, Mac OS, Solaris etc).

2.1.2 Περιγραφή των κανόνων Snort

Οι κανόνες Snort είναι γραμμένοι σε απλή γλώσσα περιγραφής κανόνων προσφέροντας, όμως, πληθώρα επιλογών που την κάνουν αρκετά ισχυρή και ευέλικτη. Κάθε κανόνας μας λέει όχι μόνο τι θέλουμε να ψάξει το σύστημα στα εισερχόμενα πακέτα, αλλά και τι ενέργεια θέλουμε να εκτελέσει σε περίπτωση που βρεθεί πακέτο που να ικανοποιεί όλες τις συνθήκες του κανόνα.

Σχεδόν όλοι οι κανόνες έχουν γραφτεί με στόχο την ανίχνευση και απαγόρευση διέλευσης σε πακέτα που περιέχουν «υπογραφές εισβολής» (intrusion signatures), δηλαδή ορισμένα χαρακτηριστικά στοιχεία ενός συγκεκριμένου τύπου επίθεσης, τα οποία μπορεί να είναι καθορισμένες παράμετροι στην επικεφαλίδα του πακέτου (packet header) ή/και στα δεδομένα του πακέτου (payload). Δίνεται, όμως, και η δυνατότητα συγγραφής κανόνων οι οποίοι έχουν ως στόχο να επιτρέπουν τη διέλευση πακέτων που πληρούν συγκεκριμένες προϋποθέσεις. Όλοι οι κανόνες αποτελούνται από δύο λογικές ενότητες, την επικεφαλίδα του κανόνα (rule header) και τις επιλογές/ρυθμίσεις του κανόνα (rule option). Στην επικεφαλίδα του κανόνα περιέχονται οι πληροφορίες για την ενέργεια που θα εκτελέσει το Snort σε περίπτωση που επαληθευτεί ο κανόνας, το πρωτόκολλο, οι διευθύνσεις IP του αποστολέα και του παραλήπτη μαζί με τις κατάλληλες μάσκες δικτύου(netmasks) καθώς και οι θύρες (ports) αποστολέα και παραλήπτη. Το τμήμα των επιλογών/ρυθμίσεων του κανόνα περιλαμβάνει τα μηνύματα προειδοποίησης (alerts) που θα παράγει το Snort σε περίπτωση επαλήθευσης καθώς και πληροφορίες για το ποια τμήματα και με τι περιεχόμενο του εισερχόμενου πακέτου πρέπει να εξεταστούν, ώστε να αποφασίσουμε αν το πακέτο είναι ύποπτο, οπότε και να ενεργήσουμε κατάλληλα. Ένα τυπικό παράδειγμα κανόνα φαίνεται παρακάτω:

```
alert tcp $HOME_NET 2589->$EXTERNAL_NET any
(msg:"MALWARE-BACKDOOR-Dagger_1.4.0"; flow:to_client,established;
content:"2 | 00 00 00 06 00 00 00|"; depth:16;
metadata:ruleset community;classtype:misc-activity; sid:105; rev:14;)
```

Rule Header

Το πρώτο πεδίο στην κεφαλίδα του πακέτου είναι η **ενέργεια (action)**, τι θα συμβεί δηλαδή στην περίπτωση που ο κανόνας επαληθευτεί. Οι τιμές που μπορεί να πάρει είναι: log, alert, pass, activate και dynamic. Στην περίπτωση του log απλά καταγράφεται το πακέτο. Στην περίπτωση του alert, ο διαχειριστής ενημερώνεται με προειδοποίηση, η οποία καταγράφεται σε ένα αρχείο log. Στην επιλογή pass αγνοείται το πακέτο. Στην περίπτωση

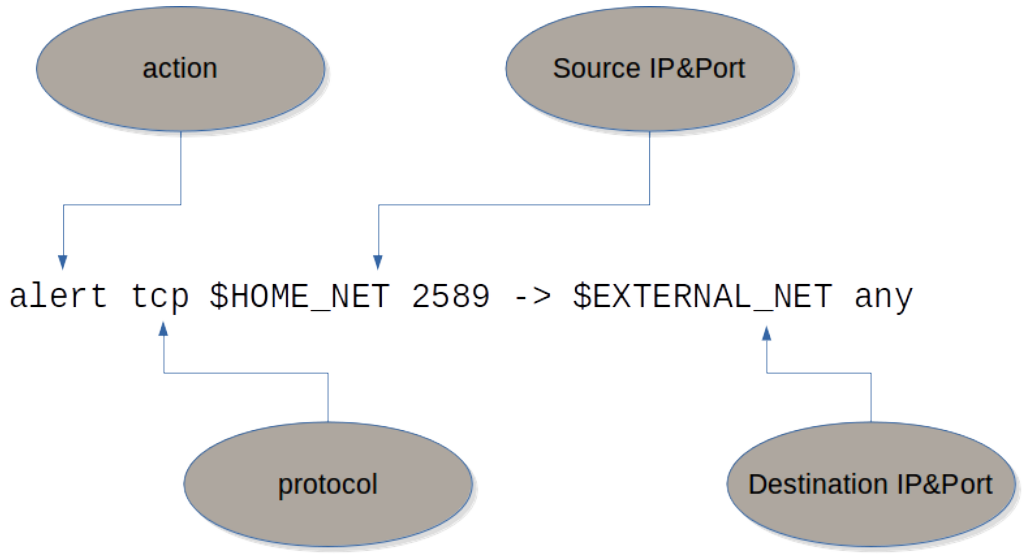
activate εμφανίζεται προειδοποίηση και έπειτα ενεργοποιείται κάποιος δυναμικός κανόνας, ενώ στην περίπτωση dynamic το σύστημα είναι αδρανές έως ότου ενεργοποιηθεί ο κανόνας και στη συνέχεια λειτουργεί όπως η επιλογή log.

Επόμενο πεδίο είναι το **πρωτόκολλο (protocol)** που πρέπει να χρησιμοποιεί το πακέτο. Αυτή τη στιγμή οι κανόνες υποστηρίζουν τα πρωτόκολλα TCP, UDP, IP, ICMP ενώ υπάρχει και η επιλογή να χρησιμοποιηθεί η λέξη κλειδί any, της οποίας η χρήση επιτρέπει οποιοδήποτε από τα υποστηριζόμενα πρωτόκολλα.

Στη συνέχεια συναντάμε το πεδίο της **διεύθυνσης IP αποστολέα (source IP)** καθώς και την θύρα αποστολέα (port). Στο παράδειγμα μας βλέπουμε ότι στο πεδίο της διεύθυνσης είναι δηλωμένη μία μεταβλητή. Οι κανόνες αφήνουν τον χρήστη να παραμετροποιήσει αυτές τις μεταβλητές από ένα αρχείο .conf. Επίσης, μπορούν να πάρουν τιμές εύρους, ξεχωριστές διευθύνσεις ή και λίστα από αυτές.

Ακολουθεί ο **τελεστής κατεύθυνσης**, ο οποίος δείχνει την κατεύθυνση κίνησης των πακέτων. Δίνεται επίσης η δυνατότητα για αμφίδρομη κίνηση με χρήση του τελεστή ⟨⟩.

Τελευταίο πεδίο είναι **η διεύθυνση IP και θύρα προορισμού**, με ιδιότητες ίδιες με το πεδίο source IP & Port.



Σχήμα 2.2: Πεδία κεφαλίδας κανόνα

Rule Options

Οι ρυθμίσεις του κανόνα περιέχονται πάντα σε αγκύλες (), ενώ οι επιλογές διαχωρίζονται με ερωτηματικό (;). Χάριν πληρότητας θα παρουσιάσουμε όλα τα πεδία των ρυθμίσεων του κανόνα, στην παρούσα εργασία, όμως, εκμεταλλευόμαστε μονάχα τα πεδία περιεχομένου (content) και μοναδικού κωδικού (sid). Οι ρυθμίσεις χωρίζονται σε τέσσερις κατηγορίες, τις βοηθητικές, περιεχομένου, μη-περιεχομένου και ολοκληρωμένης αναζήτησης.

Οι **βοηθητικές ρυθμίσεις** παρέχουν πληροφορίες που δεν επηρεάζουν, όμως, την εκτέλεση του κανόνα. Βοηθητική ρύθμιση είναι για παράδειγμα, το μήνυμα (msg) που βλέπουμε στο παράδειγμα μας, το οποίο καθορίζει το προειδοποιητικό μήνυμα που θα διαβάσει ο χρήστης.

Οι **ρυθμίσεις περιεχομένου** (content), είναι η ουσία του κανόνα. Αποτελούν τα δεδομένα τα οποία αναζητούμε μέσα στο σώμα του πακέτου. Ο χρήστης μπορεί να ορίσει τις συμβολοσειρές που αναζητεί. Η προεπιλεγμένη επιλογή για την αναζήτηση του περιεχομένου είναι case sensitive, δίνεται βέβαια η δυνατότητα να αλλάξει η εξόρισμού λειτουργία με λέξεις κλειδιά. Τέτοια είναι το non-case sensitive, depth, offset, distance, within, rawbytes.

Οι ρυθμίσεις **μη-περιεχομένου** αναζητούν για δεδομένα εκτός του σώματος του πακέτου.

Οι ρυθμίσεις **ολοκληρωμένης αναζήτησης** ορίζουν κάποια γεγονότα μετά την επαλήθευση του κανόνα.

```
(msg:"MALWARE-BACKDOOR-Dagger_1.4.0"; flow:to_client,established;
        content:"2 | 00 00 00 06 00 00 00"; depth:16;
metadata:ruleset community;classtype:misc-activity; sid:105; rev:14;)
```

Παραπάνω παραθέτουμε τις ρυθμίσεις ενός κανόνα, αυτό με κωδικό 105, οι οποίες μας λένε ότι θα λάβουμε το μήνυμα "MALWARE BACKDOOR Dagger_ 1.4.0", αν ανιχνευθεί το ′2′ ή το "00 00 00 06 00 00 00" στους πρώτους 16 χαρακτήρες του πακέτου.

2.2 Λύσεις βασισμένες σε υλικό

Όπως αναφέραμε και παραπάνω, συχνά λόγω των απαιτήσεων του δικτύου, τα συστήματα λογισμικού αδυνατούν να δώσουν λύση. Έτσι, έρχονται τα συστήματα υλικού να δώσουν λύσεις και πολλές φορές να βελτιώσουν κατά πολύ την επίδοση.

Διάφοροι ερευνητές και φοιτητές έχουν προτείνει διαφορετικές προσεγγίσεις, τις οποίες μπορούμε να χωρίσουμε σε 4 κατηγορίες ανάλογα με τον τύπο του υλικού που χρησιμοποιούν.

- Content Addressable Memory (CAM)
- FPGA implementations
- Multi-core network processors (NP)
- Hardware DPI Systems with Network Interface
- Κάρτες Γραφικών

Παρακάτω θα παρουσιάσουμε αυτές τις κατηγορίες και διάφορες εργασίες πάνω στην καθεμία.

2.2.1 Content Addressable Memory

Η συσχετιστική μνήμη Content addressable memory (CAM) είναι μια μνήμη που χρησιμοποιείται από πολλές εφαρμογές υψηλών ταχυτήτων για ταυτοποίηση μοτίβων. Σε μια RAM ο χρήστης δίνει μια διεύθυνση και εκείνη επιστρέφει τη λέξη από αυτή τη θέση. Η CAM λειτουργεί ανάποδα. Ο χρήστης δίνει μια λέξη και η μνήμη επιστρέφει την διεύθυνση στην οποία είναι αποθηκευμένη αυτή η λέξη. Σε αντίθεση με ένα τσιπ μνήμης RAM, η οποία έχει απλά θέσεις μνήμης, κάθε ξεχωριστό bit μνήμης σε μια πλήρως παράλληλη CAM πρέπει να έχει το δικό του κύκλωμα σύγκρισης για να εντοπίσει μια ταύτιση μεταξύ του αποθηκευμένου bit και του bit εισόδου. Επιπρόσθετα, οι έξοδοι που έχουν κάνει match σε κάθε κελί στη λέξη δεδομένων πρέπει να συνδυάζονται για να δώσουν μια πλήρη έξοδο match για τη λέξη. Το πρόσθετο κύκλωμα αυξάνει το φυσικό μέγεθος του τσιπ της CAM κάτι που αυξάνει το κόστος κατασκευής. Το επιπλέον κύκλωμα αυξάνει επίσης την ενέργεια που δαπανάται μιας και κάθε κύκλωμα σύγκρισης είναι ενεργή για κάθε

κύκλο ρολογιού. Κατά συνέπεια, η CAM χρησιμοποιείται μόνο σε εξειδικευμένες εφαρμογές όπου η ταχύτητα αναζήτησης δεν μπορεί να επιτευχθεί χρησιμοποιώντας μια λιγότερο δαπανηρή μέθοδο. Στα συστήματα Ανίχνευσης εισβολών δικτύου χρησιμοποιείται για τα match των κανόνων. Συχνά επίσης χρησιμοποιείται σε switches, routers, firewalls και για μετάφραση διευθύνσεων δικτύου.

2.2.2 FPGA Implementations

Οι υλοποιήσεις σε FPGA έχουν το πλεονέκτημα ότι δίνουν την ευελιξία στην αναζήτηση συμβολοσειρών με πολλές εναλλακτικές, μπορούν αναπροσαρμόσουν την σχεδίαση και να κρατήσουν την διεπαφή χωρίς αλλαγές. Ήδη πολλοί αλγόριθμοι έχουν υλοποιηθεί, αυτοί περιλαμβάνουν συγκριτές, φίλτρα, ταυτοποίηση κανονικών εκφράσεων με χρήση ντετερμινιστικών και μη-ντετερμινιστικών αυτόματων και ο αλγόριθμος Knuth - Morris - Pratt.

NFA Reduction for Regular Expressions Matching Using FPGA

Σε αυτή τη δημοσίευση [19] γίνεται προσπάθεια να μειωθούν οι καταστάσεις και οι μεταβάσεις των μη-ντετερμινιστικών πεπερασμένων αυτόματων που χρησιμοποιούνται για την αναζήτηση και ταυτοποίηση κανονικών εκφράσεων, παρουσιάζοντας αρκετές τεχνικές. Η αξιολόγηση έγινε με SNORT. Η καλύτερη τεχνική πέτυχε μείωση 66% των καταστάσεων και άρα των LUT's και FIFO's της FPGA.

ECEB: Enhanced Constraint Repetition Block for Regular Expression Matching on FPGA

Η συγκεκριμένη δημοσίευση [21] δίνει μια λύση βασισμένη σε υλικό για αναζήτηση και ταυτοποίηση Κανονικών εκφράσεων συμβατών με Perl, όπως αυτές του Snort. Η ταυτοποίηση των χαρακτήρων γίνεται με μία Block memory (BRAM), πράγμα που εξοικονομεί πολλούς LUT'S και βελτιώνει τη συνολική επίδοση του συστήματος. Τα πειράματα έγιναν με XC2VP50 Xilinx Virtex II Pro chip και τους κανόνες SNORT. Τα αποτελέσματα ήταν ότι εξοικονόμησαν 90% των πόρων και η επίδοση έφτασε στο 1Gbps.

Scalable Multigigabit Pattern Matching for Packet Inspection

Στη δημοσίευση [22] τους οι κ. *Σούρδης, Πνευματικάτος και Βασιλειάδης*, το 2008, παρουσιάζουν δύο τεχνικές για pattern matching για τον εντοπισμό επικίνδυνων περιεχομένων. Και οι δύο τεχνικές είναι κατάλληλα φτιαγμένες, ώστε να υποστηρίζουν 2200 κανόνες, ενώ αυτή τη στιγμή οι κανόνες ξεπερνούν τους 3000, χρησιμοποιώντας μία Virtex2 FPGA. Η πρώτη τεχνική επιτυγχάνει throughput μεταξύ 2 και 8 Gb/s και απαιτεί 0,58 έως 2,57 λογική ανά αναζήτηση χαρακτήρα. Από την άλλη η δεύτερη τεχνική μπορεί να υποστηρίξει από 2 έως 5,7 Gb/s, χρησιμοποιώντας μερικές δεκάδες block RAMs(630-1404 kb) και μόνο 0,28-0,65 λογική ανά χαρακτήρα. Δείχνουν τέλος, ότι πετυχαίνουν 30% μεγαλύτερη επίδοση σε σχέση με παλαιότερες εργασίες.

A Reconfigurable Perfect-Hashing Scheme for Packet Inspection

Σε αυτή τη δημοσίευση [25] εισάγεται μια τεχνική τέλειου κατακερματισμού βασισμένη σε υλικό για πρόσβαση στη μνήμη που περιέχει τα μοτίβα που αναζητούμε. Μια σύγκριση των διερχόμενων δεδομένων με τα περιεχόμενα της μνήμης καθορίζει αν είχαμε ταυτοποίηση. Τα αποτελέσματα είναι μεταξύ 1.7 και 5.7 Gbps, ενώ απαιτούνται μερικές δεκάδες blocks μνήμης και 0.30 έως 0.57 λογικές μονάδες ανά χαρακτήρα.

Efficient and High-Speed FPGA-based String Matching for Packet Inspection

Στην μεταπτυχιακή του διατριβή [31] ο κ. *Ιωάννης Σούρδης*, το 2004, παρουσίασε νέες μικρο-αρχιτεκτονικές για την ταυτοποίηση συμβολοσειρών, βασισμένες σε FPGA, μελετώντας παράλληλα και την επίδοση τους. Μέσα από την εργασία του, καταλήγει ότι οι FPGAs δίνουν την ευελιξία και την ταχύτητα που χρειάζονται τα συστήματα ανίχνευσης εισβολών δικτύου. Τα αποτελέσματα της υλοποίησης του είναι πιο μπροστά από τις δημοσιευμένες έως τότε εργασίες.

Αξιολόγηση Λογισμικού Ανίχνευσης ικτυακών Εισβολών με Υποβοήθηση από Υλικό

Στην διπλωματική του εργασία [28] ο κ. *Βασίλειος Δημόπουλος*, το 2004, παρουσίασε ένα σύστημα που συνδυάζει λογισμικό ανίχνευσης δικτυακών εισβολών με δύο φίλτρα υλικού.

Για μέσο μήκος πακέτου 300 byte το φίλτρο κατηγοριοποίησης μειώνει τον απαιτούμενο χρόνο από 28% μέχρι και 100%, ενώ για τις περισσότερες ακολουθίες πακέτων παρουσιάζει μείωση περισσότερο από 50%. Ενώ το φίλτρο προσεγγιστικής αναζήτησης συμβολοσειρών προσφέρει μείωση του αρχικού χρόνου επεξεργασίας κατά 21% στην καλύτερη περίπτωση.

Pre-decoded CAMs for Efficient and High-Speed NIDS Pattern Matching

Αυτή η εργασία [26] υποστηρίζει την χρήση προ-αποκωδικοποίησης σε ταυτοποίηση μοτίβου βασισμένου σε Content-addressable memory (CAM). Υλοποιείται ένα σύστημα βασισμένο στο Snort χρησιμοποιώντας διάφορες τεχνικές. Τα αποτελέσματα δείχνουν ότι για αναζήτηση 18000 χαρακτήρων, δηλαδή το σύνολο του SNORT, το κόστος είναι μικρότερο από 1,1 λογικές μονάδες ανά χαρακτήρα επιτυγχάνοντας συχνότητα περίπου 375MHz (3 Gbps) σε μια Virtex2 συσκευή. Χρησιμοποιώντας παραλληλισμό τεσσάρων επιπέδων το κόστος ανά χαρακτήρα πέφτει κάτω από μία λογική μονάδα και η επίδοση φτάνει περίπου στα 10Gbps.

Fast, Large-Scale String Match for a 10Gbps FPGA-based Network Intrusion Detection System

Σε αυτή τους την εργασία [27] οι κ. *Σούρδης και Πνευματικάτος* επεκτείνουν παλαιότερη δουλειά τους πετυχαίνοντας ταχύτητες έως 11Gbps. Παρουσιάζουν μια κλιμακούμενη, χαμηλών καθυστερήσεων αρχιτεκτονική, χρησιμοποιώντας στο έπακρον σωλήνωση για να αντιμετωπίσουν τα bottlenecks και να πετύχουν συχνότητες λειτουργίας 340MHz. Για να αυξήσουν τη χωρητικότητα χρησιμοποιούν πολλαπλούς συγκριτές και επιτρέπουν την παράλληλη ταυτοποίηση χαρακτήρων για πολλαπλές αναζητήσεις. Το κόστος είναι 4 εως 5 λογικές μονάδες ανά χαρακτήρα που αναζητούμε.

2.2.3 Multi-core network processors (NP)

Οι επεξεργαστές δικτύου είναι τυπικά συσκευές, όπως οι οι γενικού σκοπού επεξεργαστές, που προγραμματίζονται από λογισμικό με κάποιες πρόσθετες λειτουργίες, όπως αντιστοίχιση μοτίβων, αναζήτηση κλειδιών (π.χ. αναζήτηση διεύθυνσης), επεξεργασία δεδομένων σε επίπεδο bit και διαχείριση ουράς πακέτων.

Ο παραλληλισμός με πολλαπλούς πυρήνες μπορεί να πετύχει υψηλές επιδόσεις. Το

βασικό πλεονέκτημα είναι η ευελιξία να προσαρμοστούν σε ένα νέο μοτίβο αναζήτησης. Ωστόσο, η επίδοση τους είναι μέτρια, αποδίδοντας μερικές εκατοντάδες Mbps.

MIDeA: A Multi-Parallel Intrusion Detection Architecture

Το MIDeA [20] παραλληλοποιεί την επεξεργασία και ανάλυση της κίνησης του δικτύου σε τρία επίπεδα, χρησιμοποιώντας multi-queue NICs, πολλαπλές CPUs, και GPUs. Η συγκεκριμένη σχεδίαση αποφεύγει το locking, βελτιστοποιεί τη μεταφορά δεδομένων μεταξύ των διαφορετικών μονάδων επεξεργασίας και ανάλογα την επεξεργασία χρησιμοποιεί την καταλληλότερη μονάδα. Η αξιολόγηση του συστήματος έδειξε ότι μπορεί η ταχύτητα επεξεργασίας να φτάσει και τα 5,2Gbps, με μηδενική απώλεια πακέτων σε πραγματικό δίκτυο. Η ταυτοποίηση χαρακτήρων πετυχαίνει ταχύτητα ως 70Gbps, σχεδόν τετραπλάσια αύξηση σε σχέση με προηγούμενες δουλειές.

2.2.4 Hardware DPI Systems with Network Interface

Πιο κοντά στη δική μας υλοποίηση βρίσκονται τα συστήματα που μπορούν να διαχειρίζονται πραγματική κίνηση δικτύου. Μια τέτοια υλοποίηση παρουσίασε ο C. Clark[16]. Χρησιμοποίησαν μια Xilinx ML300 με V2P7 FPGA για να υλοποιήσουν περίπου 70 κανόνες Snort και πέτυχαν επίδοση 4Gbps. Ο αποκωδικοποιητής κεφαλίδας υποστηρίζει IP, ARP, ICMP, TCP και UDP πρωτόκολλα. Υπάρχει ένας επεξεργαστής που με μη-ντετερμινιστικά πεπερασμένα αυτόματα ελέγχει τα δεδομένα του πακέτου για να ταυτοποιήσει συγκεκριμένα μοτίβα και φτάνει να ελέγχει έως 8 χαρακτήρες ανά κύκλο. Το μειονέκτημα αυτής της υλοποίησης είναι το υψηλό κόστος σε πόρους για ένα μικρό σετ κανόνων.

Snort DPI on FPGA with GigE

Στην διπλωματική του εργασία [29] ο κ. *Δημήτριος - Στέφανος Δασκαλάκης*, το 2012, παρουσίασε ένα ολοκληρωμένο Deep Packet Inspection σύστημα που δουλεύει με Gigabit Ethernet συμβατό με το σύστημα κανόνων Snort. Τα αποτελέσματα αυτής της δουλειάς είναι για ένα σύνολο 750 κανόνων, υλοποίηση σε ένα board με μία Virtex-5 LX50T FPGA χρήση του 97,3% των διαθέσιμων LUTs.

2.2.5 Κάρτες γραφικών

Τα τελευταία χρόνια λόγω της ραγδαίας ανάπτυξης των καρτών γραφικών πολλές λύσεις έχουν δοθεί βασισμένες σε αυτό. Οι σύγχρονες κάρτες γραφικών για να ικανοποιήσουν τις πολύ αυξημένες απαιτήσεις της τεχνολογίας, όπως για παράδειγμα τρισδιάστατες απεικονίσεις σε πραγματικό χρόνο, χρησιμοποιούν τη τεχνολογία massively parallel για τη πραγματοποίηση των επιστημονικών υπολογισμών. Έτσι, μέσω κατάλληλων παράλληλων αλγορίθμων έχουν καταφέρει να είναι αποδοτικότερες από τους επεξεργαστές γενικού σκοπού. Αν και συνήθως οι κάρτες γραφικών εκτελούν υπολογισμούς για γραφικά, μπορούμε να εκμεταλλευτούμε την αρχιτεκτονική τους και να διεξάγουμε υπολογισμούς διάφορων εφαρμογών. Η παραπάνω χρήση μιας κάρτας γραφικών ονομάζεται προγραμματισμός γενικής χρήσης σε μονάδες γραφικών (GP GPU). Η ικανότητα μιας κάρτας γραφικών να εκτελεί παράλληλους αλγορίθμους μπορεί να αυξηθεί κάνοντας χρήση πολλαπλών καρτών γραφικών ή μιας κάρτας γραφικών με μεγάλο πλήθος επεξεργαστών (cores). Τα εργαλεία ελεύθερης χρήσης που έχουμε στη διάθεση μας για αυτή τη δουλειά είναι εξειδικευμένες γλώσσες για κάρτες γραφικών, δηλαδή επεκτάσεις της C με το πρότυπο (API) CUDA ή της OpenCL.

PixelSnort

Το PixelSnort [17] αποτελεί μια μεταφορά του Snort στην κάρτα γραφικών 6800GT της NVIDIA με χρήση της γλώσσας προγραμματισμού Cg. Τα αποτελέσματα αυτής της δουλειάς ξεπερνούν την επίδοση του Snort κατά 40%.

Gnort

Το Gnort [18] αποτελεί μία υλοποίηση του αλγορίθμου Aho Corasick στην κάρτα γραφικών GeForce 8600GT που υποστηρίζει το μοντέλο της CUDA και πετυχαίνει ρυθμό 2,3 Gbit/s.

Παράλληλη κατηγοριοποίηση κεφαλίδας πακέτων με χρήση γραφικού επεξεργαστή

Στην διπλωματική του εργασία [30] ο κ. *Ιωάννης Μακρής*, το 2010, υλοποίησε τη λειτουργία κατηγοριοποίησης κεφαλίδων πακέτων που χρησιμοποιεί το Snort σε μία κάρτα

γραφικών της NVIDIA που υποστηρίζει το μοντέλο της CUDA. Από τα πειραματικά του αποτελέσματα προέκυψε ότι μπορεί να έχει επιτάχυνση έως και 100 φορές περισσότερο, έναντι της κατηγοριοποίησης που εκτελεί το Snort.

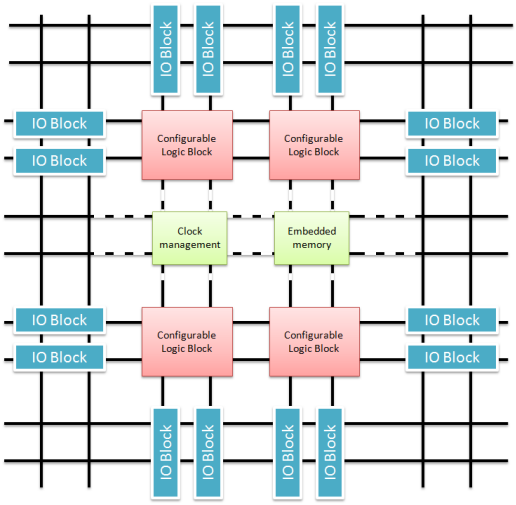
Κεφάλαιο 3

Η αρχιτεκτονική του συστήματος Maxeler

3.1 Υπερυπολογιστές με FPGA

3.1.1 Αναδιατασσόμενη λογική-FPGA

Με τον όρο "αναδιατασσόμενη λογική" ή Field Programmable Gate Array-FPGA εννοούμε ολοκληρωμένα κυκλώματα, τα οποία δεν διαθέτουν από κατασκευής τους κάποια αρχική σχεδίαση μέσα τους, όπως έχουν οι μικροεπεξεργαστές των υπολογιστών μας, αλλά έχουν πόρους (λογικές πύλες, μετρητές, καταχωρητές μνήμης και καλωδιώσεις), που κάθε σχεδιαστής μπορεί να διασυνδέσει όπως θέλει, με την χρήση κατάλληλων εργαλείων σε υπολογιστή. Με αυτόν τον τρόπο, το ίδιο φυσικό κύκλωμα μπορεί να γίνει ό,τι ο σχεδιαστής επιθυμεί, ανάλογα με τη σχεδίαση που απεικονίζεται σε αυτό. Οι FPGA's προγραμματίζονται με γλώσσες περιγραφής υλικού (HDL), όπως είναι η Verilog και η VHDL.



Τυπικό block-diagram μίας FPGA

Virtex UltraScale

Σχήμα 3.1: FPGA

3.1.2 Υβριδικοί Υπερυπολογιστές

Ιστορικά ο όρος "Υπολογιστές υψηλών επιδόσεων" [High Performance Computing-HPC (supercomputer)] είχε συνδέσει το όνομα του με κάτι πολύ ακριβό, με μεγάλη κατανάλωση ενέργειας και που κατέχει πολύ χώρο. Η ανάπτυξη της τεχνολογίας και της βιομηχανίας οδήγησε σε αλλαγές στην προσέγγιση των υπερυπολογιστών. Έγιναν αποδοτικότεροι σε θέματα ενέργειας, χώρου και χρημάτων.

Οι εποχές που μόνο οι μεγάλοι οργανισμοί μπορούσαν να έχουν υπερυπολογιστές έχουν περάσει. Φυσικά, αυτό οφείλεται στο ότι έγιναν φτηνότεροι και αποτελεσματικότεροι. Οι παράλληλες τεχνολογίες έγιναν ιδιαίτερα δημοφιλείς στον τομέα της έρευνας και αυτό οδήγησε όλο και περισσότεροι κατασκευαστές να προσαρμόζουν το λογισμικό τους, ώστε να δουλεύει σε πλατφόρμες με τέτοια αρχιτεκτονική.

Τα τελευταία χρόνια η ανάπτυξη του ενδιαφέροντος για τους υπερυπολογιστές έχει συνδεθεί με τους υβριδικούς υπερυπολογιστές. Οι τελευταίοι είναι υπερυπολογιστές που χρησιμοποιούν επεξεργαστές καθώς και αρχιτεκτονικές κεντρικών μονάδων επεξεργασίας (CPU) για τους υπολογισμούς τους. Με αυτόν τον τρόπο παρέχονται περισσότερες δυνατότητες και πόροι από τις απλές FPGA, αλλά και από τους συμβατικούς υπολογιστές, με αποτέλεσμα να αυξάνεται σε μεγάλο βαθμό η υπολογιστική τους ισχύ. Ένας τέτοιος υβρι-

δικός υπερυπολογιστής είναι και αυτός που χρησιμοποιήθηκε στην παρούσα διπλωματική εργασία, της εταιρείας Maxeler Technologies.

3.2 Maxeler

Σχήμα 3.2: Λογότυπο της εταιρείας Maxeler Technologies

Οι πλατφόρμες της Maxeler είναι περιβάλλοντα που επιτρέπουν τη διαχείριση ροών δεδομένων, αυξάνοντας την ταχύτητα και μειώνοντας την ενεργειακή κατανάλωση. Αυτό γίνεται μέσω μίας γλώσσας προγραμματισμού υψηλού επιπέδου, πράγμα που κάνει τη χρήση τους ευκολότερη για περισσότερο κόσμο.

Ο πολυκλίμακος προγραμματισμός ροών δεδομένων της Maxeler είναι ένας συνδυασμός σύγχρονων ροών δεδομένων, διανυσματικών και διατεταγμένων επεξεργαστών. Εκμεταλλεύεται παραλληλισμό σε επίπεδο βρόγχου στο χώρο, τη σωλήνωση (pipeline), όπου μεγάλες ροές δεδομένων ρέουν σε πολλές αριθμητικές μονάδες, συνδεδεμένες ώστε να ταιριάζουν σε αυτό που υπολογίζεται.

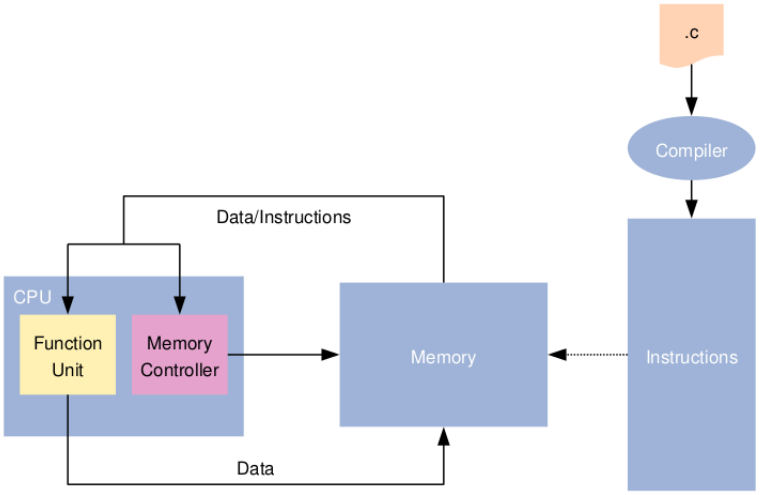
Μικρές μνήμες επάνω στο ολοκληρωμένο κύκλωμα αποτελούν το αρχείο καταχωρητών (register file) με τόσες πόρτες επικοινωνίας, όσες χρειάζονται. Ο πολυκλίμακος προγραμματισμός ροών δεδομένων χρησιμοποιεί ροές δεδομένων σε πολλά αφαιρετικά επίπεδα:

- Στο επίπεδο συστήματος
- Στο επίπεδο αρχιτεκτονικής
- Σε επίπεδο αριθμού
- Σε επίπεδο bit.

Στο επίπεδο συστήματος πολλές Data Flow Engines (DFEs) συνδέονται, ώστε να φτιάξουν έναν υπερυπολογιστή. Στο επίπεδο αρχιτεκτονικής αποσυνδέεται η πρόσβαση στη μνήμη από τις αριθμητικές πράξεις, ενώ σε επίπεδο αριθμού και bit μπορεί να βελτιστοποιηθεί η αναπαράσταση των δεδομένων και να εξισορροπηθεί ο υπολογισμός με την επικοινωνία.

3.2.1 Dataflow vs Control flow model

Σε μία εφαρμογή λογισμικού, ο πηγαίος της κώδικας μετατρέπεται σε μία λίστα εντολών για έναν συγκεκριμένο επεξεργαστή **(control flow core)**, η οποία φορτώνεται στη μνήμη. (Σχήμα 3.3). Οι εντολές περνάνε στον επεξεργαστή και περιστασιακά γράφει ή και διαβάζει από τη μνήμη. Οι σύγχρονοι επεξεργαστές περιλαμβάνουν πολλαπλά επίπεδα προσωρινής αποθήκευσης (caching), προώθησης(forwarding) και λογική πρόβλεψης για την βελτίωση της επίδοσης, ωστόσο αυτό το μοντέλο προγραμματισμού είναι ακολουθιακό και η επίδοση του εξαρτάται από τις καθυστερήσεις της μνήμης και το χρόνο ενός κύκλου του ρολογιού της CPU.

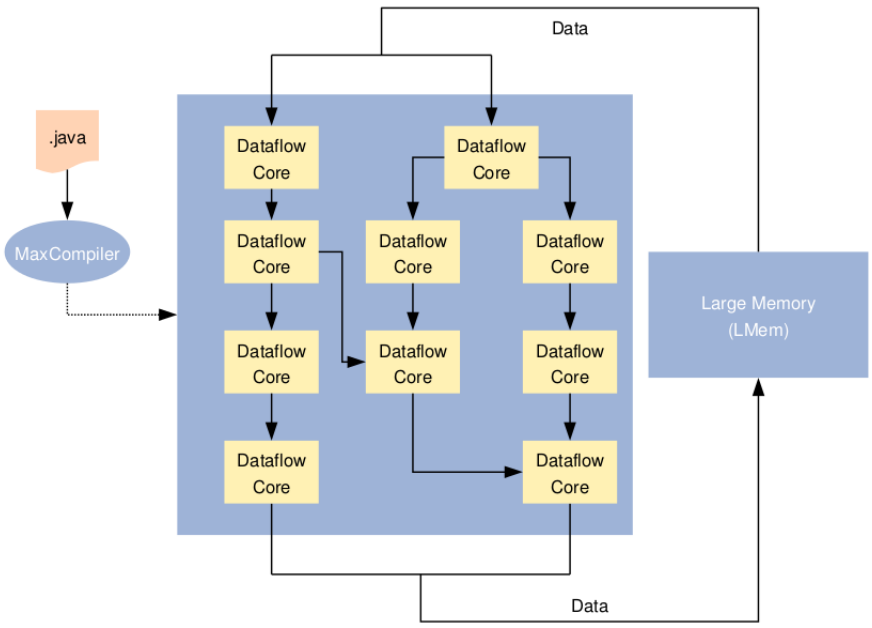


Σχήμα 3.3: Η επαναχρησιμοποίηση των λειτουργικών μονάδων μίας CPU στο χρόνο

Σε ένα **dataflow** πρόγραμμα η διαδρομή που ακολουθούν η λειτουργικότητα και τα δεδομένα περιγράφονται από ένα συγκεκριμένο αλγόριθμο (Σχήμα 3.4). Σε μία Data Flow Engine(DFE) οι ροές δεδομένων προωθούνται άμεσα από το ένα core στο άλλο μέχρι να ολοκληρωθεί η αλυσιδωτή επεξεργασία. Μόλις ένα dataflow πρόγραμμα έχει επεξεργαστεί τις ροές δεδομένων, Data Flow Engine μπορεί να επαναρυθμιστεί για μία νέα εφαρμογή σε λιγότερο από ένα δευτερόλεπτο. Κάθε core υπολογίζει συγκεκριμένο τύπο αριθμητικής πράξης, για παράδειγμα μία πρόσθεση ή έναν πολλαπλασιασμό κι έτσι είναι εύκολο να χωρέσουν χιλιάδες σε μία DFE. Σε μία pipeline DFE επεξεργασία κάθε core υπολογίζει ταυτόχρονα γειτονικά δεδομένα σε ένα Stream. Σε αντίθεση με τα control flow cores όπου οι πράξεις υπολογίζονται σε διαφορετικούς χρόνους στις ίδιες μονάδες

(computing in time), ένας dataflow υπολογισμός απλώνεται στον χώρο (computing in space).

Οι εξαρτήσεις σε ένα dataflow πρόγραμμα επιλύονται στατικά κατά τη μεταγλώττιση. Μία απλή αναλογία για την μεταφορά από το control flow model στο dataflow model είναι το παράδειγμα που ακολουθεί. Σε ένα εργοστάσιο κάθε εργαζόμενος παίρνει μία απλή εργασία και όλοι οι εργαζόμενοι λειτουργούν παράλληλα στις ροές αυτοκινήτων και ανταλλακτικών. Ακριβώς όπως στην κατασκευή, έτσι και το dataflow είναι μία μέθοδος για να αναβαθμιστεί ένα υπολογισμός σε μεγάλη κλίμακα. Η ίδια η δομή της DFE αποτελεί τον υπολογισμό, συνεπώς, δεν υπάρχει ανάγκη για εντολές. Οι εντολές αντικαθίστανται από τις αριθμητικές μονάδες που ορίζονται στο χώρο και συνδέονται για ένα συγκεκριμένο task. Ακριβώς επειδή δεν υπάρχουν εντολές δεν υπάρχει καμία ανάγκη για την λογική αποκωδικοποίηση αυτών, κρυφές μνήμες εντολών, πρόβλεψη διακλάδωσης, ή δυναμικό προγραμματισμό εκτός σειράς. Σε επίπεδο συστήματος, η DFE χειρίζεται την επεξεργασία δεδομένων μεγάλης κλίμακας, ενώ οι CPUs που τρέχουν Linux διαχειρίζονται παράτυπες και σπάνιες λειτουργίες, το I/O και την επικοινωνία μεταξύ των κόμβων του προγράμματος.



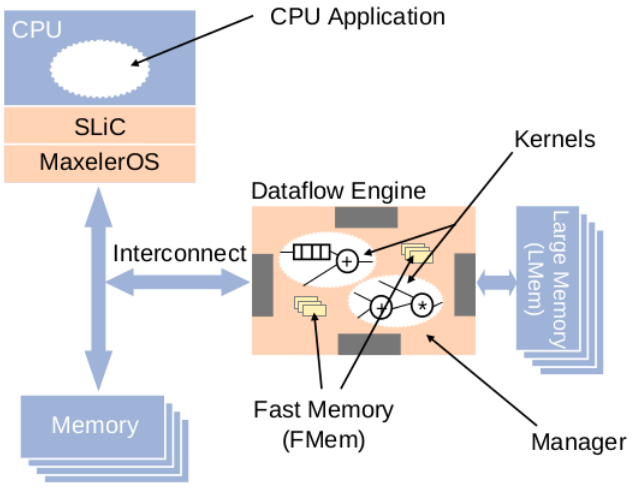
Σχήμα 3.4: Ένα dataflow πρόγραμμα.

3.2.2 DFEs

Στο Σχήμα 3.5 απεικονίζεται η αρχιτεκτονική του συστήματος επεξεργασίας ροών δεδομένων της Maxeler, το οποίο περιλαμβάνει DFEs με τις μνήμες τους να συνδέονται σε μία CPU. Κάθε CPU μπορεί να έχει πολλαπλούς πυρήνες (Kernels), οι οποίοι εκτελούν υπολογισμούς όσο τα δεδομένα ρέουν μεταξύ CPU, DFE και των μνημών που σχετίζονται με αυτά.

Η DFE έχει δύο τύπους μνημών, την γρήγορη μνήμη (FMEM), η οποία μπορεί να αποθηκεύσει αρκετά megabyte δεδομένων on-chip με προσπέλαση σε terabyte/δευτερόλεπτο και την μεγάλη μνήμη (LMEM), η οποία μπορεί να αποθηκεύσει πολλά gigabyte δεδομένων off-chip. Το εύρος ζώνης και η ευελιξία της FMem είναι ένας βασικός λόγος για τον οποίο οι DFEs είναι σε θέση να πετυχαίνει τέτοια υψηλή επίδοση σε σύνθετες εφαρμογές. Οι εφαρμογές μπορούν να αξιοποιούν πλήρως τη χωρητικότητα της FMem μίας και τα δεδομένα κρατούνται σε μνήμη κοντά στους υπολογισμούς. Αυτό έρχεται σε αντίθεση με τις παραδοσιακές αρχιτεκτονικές CPU, με πολυεπίπεδες caches, όπου μόνο η γρηγορότερη ή μικρότερη cache είναι κοντά στις μονάδες υπολογισμού και τα δεδομένα διπλογράφονται στην ιεραρχία της cache.

Η DFE προγραμματίζεται, ώστε να έχει έναν ή περισσότερους Kernel και έναν διαχειριστή (Manager). Οι Kernels υλοποιούν τους υπολογισμούς, ενώ ο Manager ρυθμίζει την κίνηση των δεδομένων σε μία DFE. Δοσμένου Kernel και Manager, ο μεταγλωττιστής της Maxeler-MaxCompiler δημιουργεί υλοποιήσεις που μπορούν να κληθούν από τη CPU μέσω της διεπαφής Simple Live CPU-SLiC. Η διεπαφή SLiC είναι μία αυτόματα δημιουργούμενη διεπαφή στο dataflow πρόγραμμα, που κάνει εύκολο να "κληθούν" DFEs από την αντίστοιχη CPU. Το συνολικό σύστημα διαχειρίζεται από το MaxelerOS, το οποίο είναι Linux based σύστημα. Το MaxelerOS διαχειρίζεται τη μεταφορά δεδομένων και τη δυναμική βελτιστοποίηση κατά τον χρόνο εκτέλεσης.

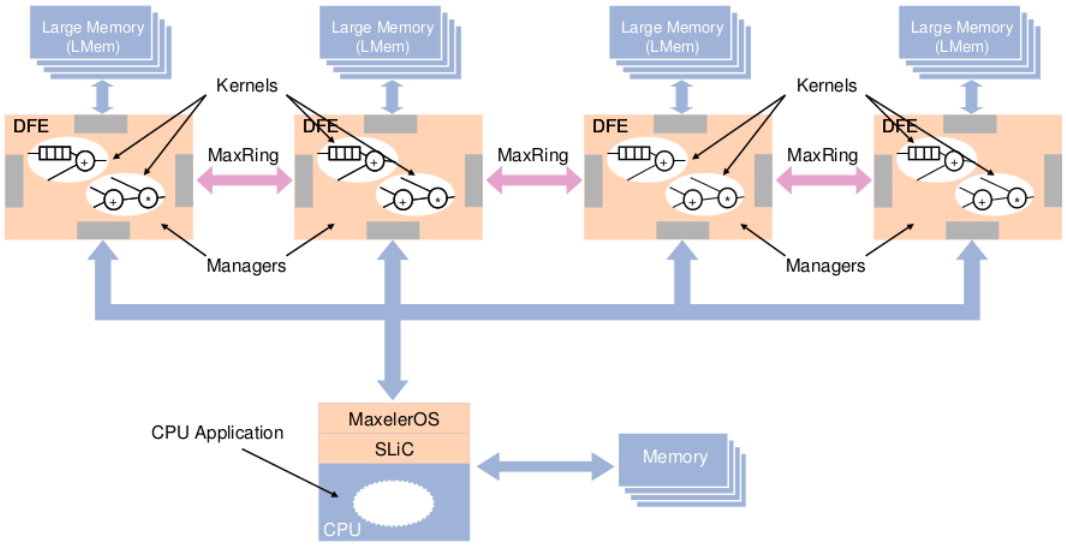


Σχήμα 3.5: Αρχιτεκτονική της DFE

3.2.3 Η αρχιτεκτονική του συστήματος

Στο Σχήμα 3.6 παρουσιάζεται η αρχιτεκτονική του υπερυπολογιστικού συστήματος Maxeler, το οποίο περιλαμβάνει Data Flow Engines (DFEs) (υποστηρίζονται έως 8 τέτοιες) συνδεδεμένες απευθείας σε τοπικές μνήμες και CPU. Στα συστήματα που είναι φτιαγμένα στη Maxeler μπορούν να υπάρχουν πολλαπλά DFEs, τα οποία συνδέονται μεταξύ τους με διασύνδεση υψηλού εύρους ζώνης.

Μία διάταξη DFEs μπορεί να αποτελείται από έναν ή και παραπάνω πυρήνες (Kernels) και έναν διαχειριστή (Manager). Στον Kernel οι ροές δεδομένων(Streams) παράγουν ένα περιβάλλον από ροές δεδομένων και αριθμούς. Ο Manager παράγει μία διεπαφή για τον Kernel με Streams εισόδου και Streams εξόδου.

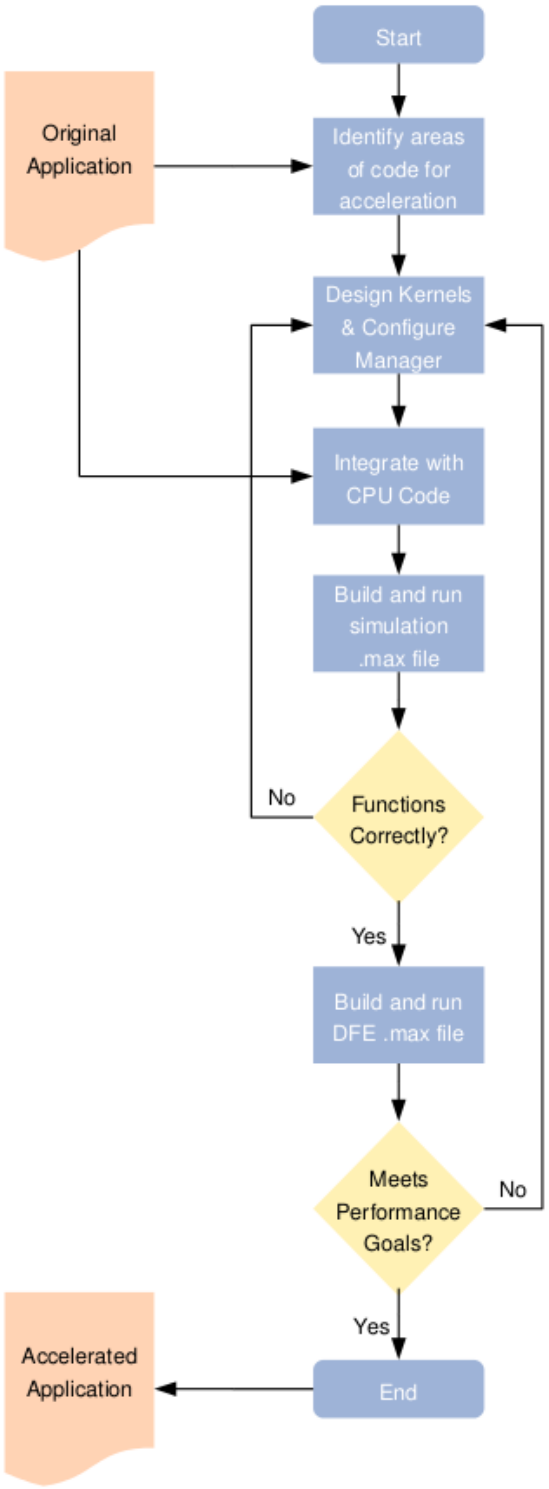


Σχήμα 3.6: Η αρχιτεκτονική του συστήματος της Maxeler

Ο Manager παρέχει μία εφαρμογή Java για:

- την ρύθμιση της επικοινωνίας μεταξύ των Kernel και I/O
- τον έλεγχο της διαδικασίας χτισίματος του συστήματος

Στην παρούσα εργασία χρησιμοποιήθηκε ένα προϊόν από τη Maxeler Technologies, το οποίο ανήκει στη σειρά MPC C. Η συγκεκριμένη σειρά επιτρέπει τη χρήση μόνο DFEs ή και CPU. Συγκεκριμένα η DFE που δημιουργήσαμε, χτίστηκε σε μία Virtex 6 FPGA. Ακολουθεί ένα διάγραμμα ροής για την κατανόηση της ροής της σχεδίασης (Σχήμα 3.7).



Σχήμα 3.7: Διάγραμμα ροής της πορείας σχεδίασης

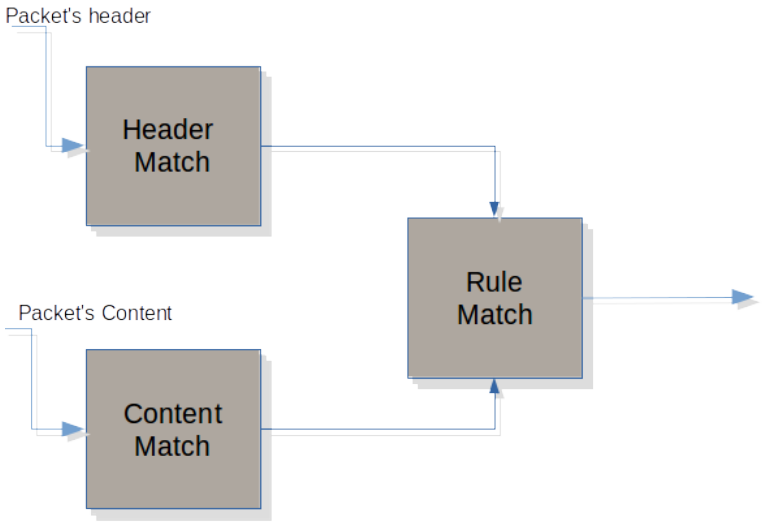
Κεφάλαιο 4

Αρχιτεκτονική και Υλοποίηση

Στον πυρήνα του συστήματος μας, βρίσκονται διάφορα υποσυστήματα καθένα από τα οποία, εκτελεί και μία διαφορετική λειτουργία.

Στο κεφάλαιο που ακολουθεί, θα εξηγήσουμε τον τρόπο λειτουργίας καθενός υποσυστήματος, ώστε να συνθέσουμε τελικά το ολοκληρωμένο σύστημα. Στο Σχήμα 4.1 βλέπουμε μια αρκετά αφαιρετική αναπαράσταση του συστήματος μας. Αξίζει να σημειώσουμε ότι η περιγραφή που γίνεται παρακάτω αφορά την επαλήθευση ενός κανόνα, ώστε να γίνει ευκολότερα κατανοητή η αρχιτεκτονική μας. Στην τελευταία παράγραφο αυτού του κεφαλαίου θα δούμε το συνολικό Σχηματικό διάγραμμα της σχεδίασης μας και θα παρουσιάσουμε και την αναζήτηση πολλαπλών κανόνων.

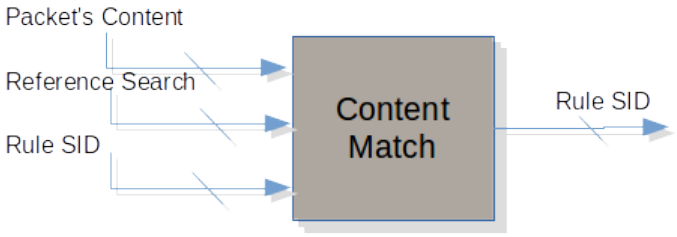
4.1 Το Block Diagram του Πυρήνα



Σχήμα 4.1: Block diagram κανόνα

Το σύστημα που δημιουργήσαμε βασιζόμενο στον τρόπο λειτουργίας των κανόνων Snort, αποτελείται από τρία υποσυστήματα για κάθε έναν από τους κανόνες που χρησιμοποιήσαμε. Όπως φαίνεται και στο Σχήμα 4.1 για να ελέγξουμε αν ο κανόνας επαληθεύεται, ενεργοποιούμε το Header Match, το Content Match και στη συνέχεια το Rule Match υποσύστημα. Παρακάτω θα αναλύσουμε κάθε ένα από αυτά, εξηγώντας την αρχιτεκτονική τους και τον τρόπο λειτουργίας τους.

4.2 Content Match



Σχήμα 4.2: Block diagram Μονάδας ελέγχου δεδομένων πακέτου

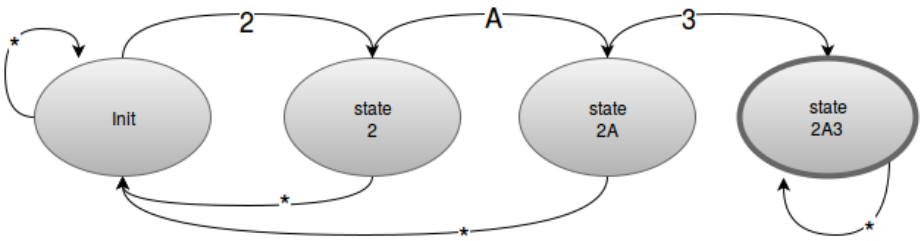
Όπως αναφέραμε και νωρίτερα, η αναζήτηση συμβολοσειρών στα δεδομένα ενός πακέτου δεν είναι απλή υπόθεση και μπορεί να γίνει πολύ απαιτητική τόσο σε χρόνο, όσο και σε πόρους συστήματος. Θέλοντας, λοιπόν, να ελαχιστοποιήσουμε τις απαιτήσεις τόσο σε πόρους συστήματος, όσο και σε χρόνο, δημιουργήσαμε μονάδες ελέγχου βάσει του μήκους της συμβολοσειράς που αναζητάμε. Έτσι, ομαδοποιήσαμε ουσιαστικά και τους κανόνες. Για παράδειγμα, έχουμε μονάδες ελέγχου για μήκος συμβολοσειράς δύο, τριών, τεσσάρων κλπ χαρακτήρων. Με αυτόν τον τρόπο, χρησιμοποιούμε κάθε φορά το κατάλληλο στοιχείο της σχεδίασης μας.

Ας δούμε όμως, πιο αναλυτικά τον τρόπο με τον οποίο έγινε η αναζήτηση των συμβολοσειρών. Η είσοδος του υποσυστήματος αυτού είναι ένα Stream από χαρακτήρες, η συμβολοσειρά που αναζητάμε βάσει του κανόνα που ελέγχουμε καθώς και ο μοναδικός κωδικός του εκάστοτε κανόνα. Η έξοδος του είναι μηδέν, αν δεν έγινε ταυτοποίηση του κανόνα ή ο μοναδικός κωδικός του κανόνα, εάν έγινε ταυτοποίηση και θα εξηγήσουμε παρακάτω τον λόγο.

Θα παρουσιάσουμε πως γίνεται η αναζήτηση μιας συμβολοσειράς μήκους τριών χαρακτήρων, μιας και είναι αντιπροσωπευτικό παράδειγμα για όλες τις μονάδες αυτού του υποσυστήματος. Η είσοδος τώρα είναι data, οι χαρακτήρες που αναζητάμε με τη σειρά που μας τους δίνει ο κανόνας, και ο μοναδικός κωδικός του κανόνα. Μιας και αναζητάμε οι χαρακτήρες να εμφανίζονται με συγκεκριμένη σειρά, πρέπει να ελέγξουμε και με συγκεκριμένη σειρά τα δεδομένα. Αν για παράδειγμα ψάχνουμε την συμβολοσειρά "2A3" θα μπορούσε να βρίσκεται στην πρώτη, δεύτερη και τρίτη θέση του Stream, ή στην δεύτερη, τρίτη και τέταρτη θέση κ.ο.κ. Αυτό θα μπορούσε να λυθεί αν αναζητούσαμε τον χαρακτήρα "2", αν τον βρίσκαμε αναζητούσαμε στον επόμενο τον χαρακτήρα "A" και στη συνέχεια τον χαρακτήρα "3" κάτι, όμως, που αυξάνει την πολυπλοκότητα των πράξεων στη Maxeler, τον χρόνο αναζήτησης και τους πόρους.

Στο Σχήμα 4.3 φαίνεται ένα ντετερμινιστικό πεπερασμένο αυτόματο (DFA), που υλοποιεί την παραπάνω αναζήτηση. Όπως είδαμε και στο Κεφάλαιο 2.2, έχουν προταθεί αρκετές λύσεις τόσο με DFA όσο και με NFA. Στο [32], αποδείχθηκε ότι ένα NFA μπορεί να υλοποιηθεί αποτελεσματικά με προγραμματιζόμενη λογική διάταξη. Μια υψηλής επίδοσης σε χώρο FPGA υλοποίηση της NFA προτάθηκε στο [33]. Σε αυτή την εργασία, το NFA άμεσα μετατρέπεται σε λογικές πύλες και καταχωρητές. Το μειονέκτημα ενός τέτοιου σχεδιασμού είναι ότι το κύκλωμα πρέπει να κάνει σύνθεση, κάθε φορά που η

κανονική έκφραση αλλάζει. Μία εφαρμογή DFA παρουσιάστηκε στο [34], επιτυγχάνει σημαντική βελτίωση στην επίδοση, αλλά μπορεί να απαιτήσει μεγάλο χώρο μνήμης. Στο [35], μία καθυστέρηση της εισόδου του DFA η οποία χρησιμοποιεί προεπιλεγμένες μεταβάσεις, μια ιδέα παρόμοια με την αποτυχία μετάβασης του αλγορίθμου Aho-Corasick, προτάθηκε για να μειωθεί ο αριθμός των μεταβάσεων και ως εκ τούτου η απαίτηση χώρου του DFA. Η pattern matching engine αυτού, χρησιμοποιεί μνήμη και όχι λογική. Η μείωση των καταστάσεων μετάβασης για περισσότερο από 95 τοις εκατό, επιτεύχθηκε με διαφορετικά σύνολα κανονικών εκφράσεων που χρησιμοποιούνται σε πραγματικά προϊόντα. Ως εκ τούτου, ο αριθμός των εκφράσεων που μπορούν να υποστηριχθούν από ένα μόνο τσιπ σε μεγάλο βαθμό αυξάνεται. Αν και η ιδέα λειτουργεί για επιλεγμένες σειρές κανονικών εκφράσεων, εξακολουθεί να έχει τον κίνδυνο ότι μπορεί να προκύψει ένας τεράστιος αριθμός καταστάσεων.



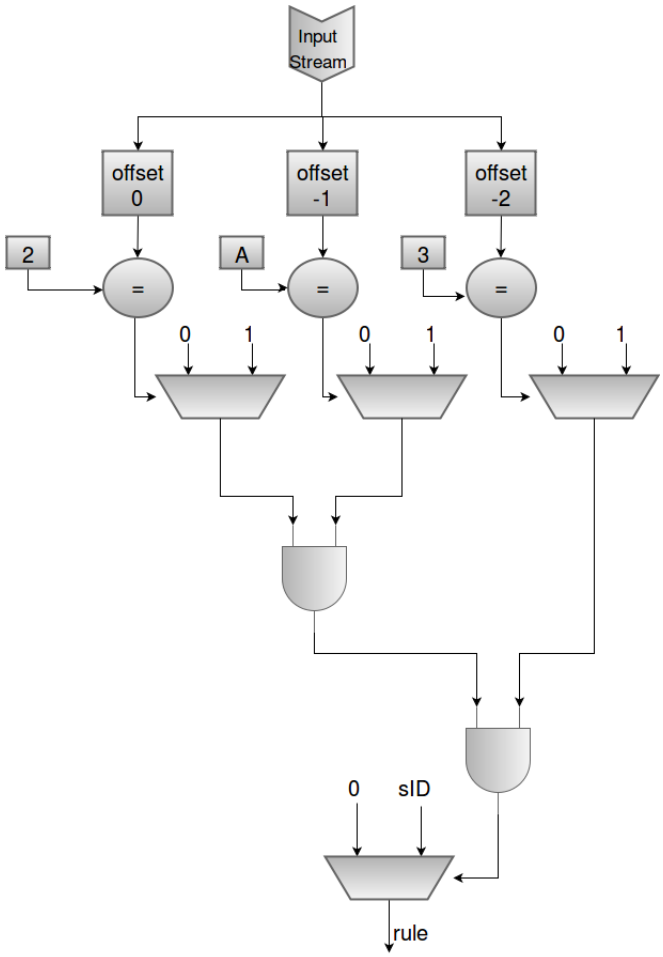
Σχήμα 4.3: Σχηματική αναπαράσταση παραδείγματος

Βάσει των παραπάνω, της πολυπλοκότητας, αλλά και της έλλειψης τεχνογνωσίας στον υπερυπολογιστή της Maxeler, αποφασίσαμε να χρησιμοποιήσουμε τη μέθοδο Windows into streams, μία μέθοδος που χρησιμοποιείται από τη Maxeler. Ουσιαστικά κομμάτια-/παράθυρα(windows) της ροής των δεδομένων αποθηκεύονται σε on-chip μνήμη, ελαχιστοποιώντας έτσι, το κόστος μεταφοράς δεδομένων. Χρησιμοποιώντας Stream Offset μπορούμε να πλοηγηθούμε σε θέσεις του Stream σχετικές με την παρούσα του θέση. Η απόσταση από το μεγαλύτερο στο μικρότερο Offset ουσιαστικά ρυθμίζει το μήκος του παραθύρου των δεδομένων που κρατιούνται στην on-chip μνήμη.

Στο παράδειγμα με την αναζήτηση συμβολοσειράς μήκους τριών χαρακτήρων, χρησιμοποιούμε Offset για την τωρινή, την επόμενη και μεθεπόμενη θέση στο Stream. Αν θέλαμε, λοιπόν, να παρουσιάσουμε τον ψευδοκώδικα για αυτό το παράδειγμα, αυτός θα ήταν:

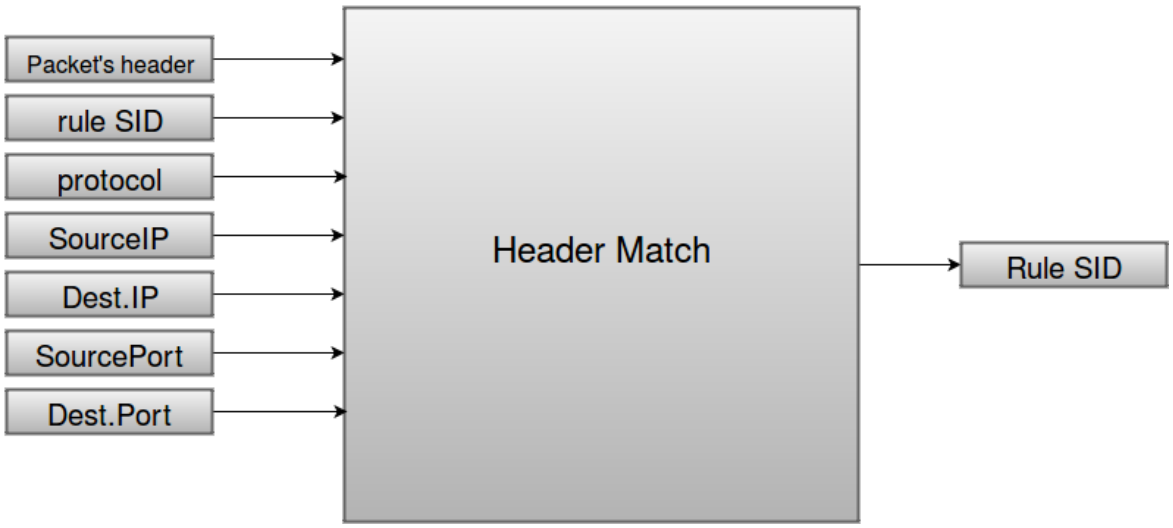
```
Var0=data(offset=0)
Var1=data(offset=-1)
Var2=data(offset=-2)
if ((Var0==2) && (Var1==A) && (Var2==3))
    return sid
else
    return 0
```

Το παραπάνω όταν μιλάμε για υλικό, μετατρέπεται σε μερικούς πολυπλέκτες και πύλες and, οπότε στο Σχήμα 4.4 παρουσιάζουμε το λογικό διάγραμμα τους. Στο παράρτημα 6.3 μπορεί να βρεθεί και ένας ενδεικτικός κώδικας για έλεγχο δύο κανόνων.



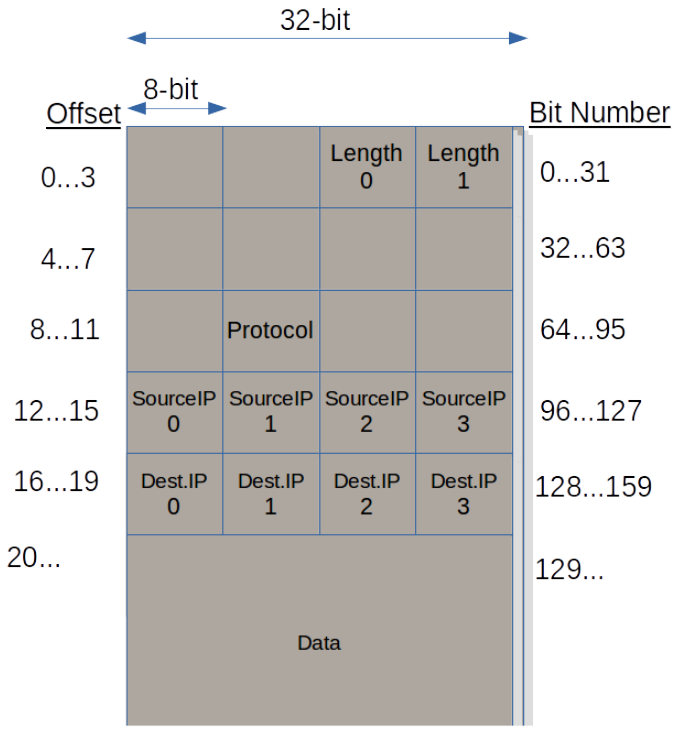
Σχήμα 4.4: Λογικό διάγραμμα μονάδας ελέγχου συμβολοσειράς μήκους 3

4.3 Header Match



Σχήμα 4.5: Block diagram Μονάδας ελέγχου κεφαλίδας

Στο Σχήμα 4.5 βλέπουμε μια σχηματική αναπαράσταση της μονάδας ελέγχου της κεφαλίδας. Οι είσοδοι είναι τα δεδομένα της κεφαλίδας του πακέτου, ο μοναδικός κωδικός του κανόνα, το πρωτόκολλο που αναζητάμε βάσει του κανόνα, οι διευθύνσεις IP αποστολέα και προορισμού καθώς και οι θύρες. Το συγκεκριμένο υποσύστημα βγάζει σαν έξοδο τον μοναδικό κωδικό του κανόνα, σε περίπτωση ταυτοποίησης και μηδέν σε περίπτωση μη ταυτοποίησης. Αξίζει να σημειωθεί ότι οι διευθύνσεις IP αποτελούνται από τέσσερις μεταβλητές η κάθε μία, ώστε να υπάρχει συμβατότητα στον αριθμό των bit, στην αναζήτηση των περιεχομένων της κεφαλίδας του πακέτου.



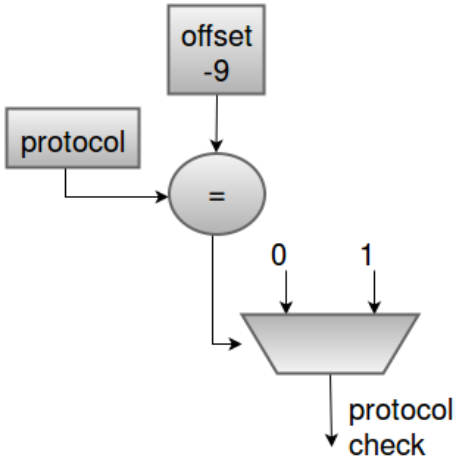
Σχήμα 4.6: Μορφή κεφαλίδας IPv4 πακέτου

Τα δεδομένα της κεφαλίδας του πακέτου χωρίζονται σε διάφορα πεδία, ακολουθώντας τη δομή της κεφαλίδας του IPv4 πρωτοκόλλου, στο επίπεδο 3-Δικτύου (πρότυπο OSI). Στο Σχήμα 4.6 φαίνεται η δομή της κεφαλίδας του πακέτου.

Για τον έλεγχο όλης της πληροφορίας εκτελούνται ξεχωριστοί έλεγχοι και στο τέλος συνδυάζονται τα αποτελέσματα. Τα δεδομένα της κεφαλίδας τα λαμβάνουμε σαν 8-bit μη προσημασμένους ακέραιους. Χρησιμοποιώντας τα Offsets ξεχωρίζουμε τα πεδία της κεφαλίδας του πακέτου. Για παράδειγμα, αν θέλουμε να χρησιμοποιήσουμε την πληροφορία για το πρωτόκολλο στο επίπεδο 4-Μεταφοράς (πρότυπο OSI), θα θέσουμε Offset=-9. Παρακάτω θα δούμε σε ξεχωριστές παραγράφους, τους ελέγχους που πραγματοποιούνται για κάθε πεδίο. Στο παράρτημα 6.2 μπορεί να βρεθεί και ολόκληρος ο κώδικας για το Header Match.

4.3.1 Έλεγχος πρωτοκόλλου

Σε αυτό το σημείο ελέγχουμε αν το πρωτόκολλο (του επιπέδου 4) στα δεδομένα της κεφαλίδας του πακέτου, συμπίπτει με το πρωτόκολλο που απαιτεί ο κανόνας που ελέγχουμε τη δεδομένη στιγμή. Οι κανόνες υποστηρίζουν τα πρωτόκολλα TCP και UDP. Στο Σχήμα 4.7 φαίνεται το λογικό διάγραμμα του παραπάνω ελέγχου.



Σχήμα 4.7: Λογικό διάγραμμα ελέγχου πρωτοκόλλου

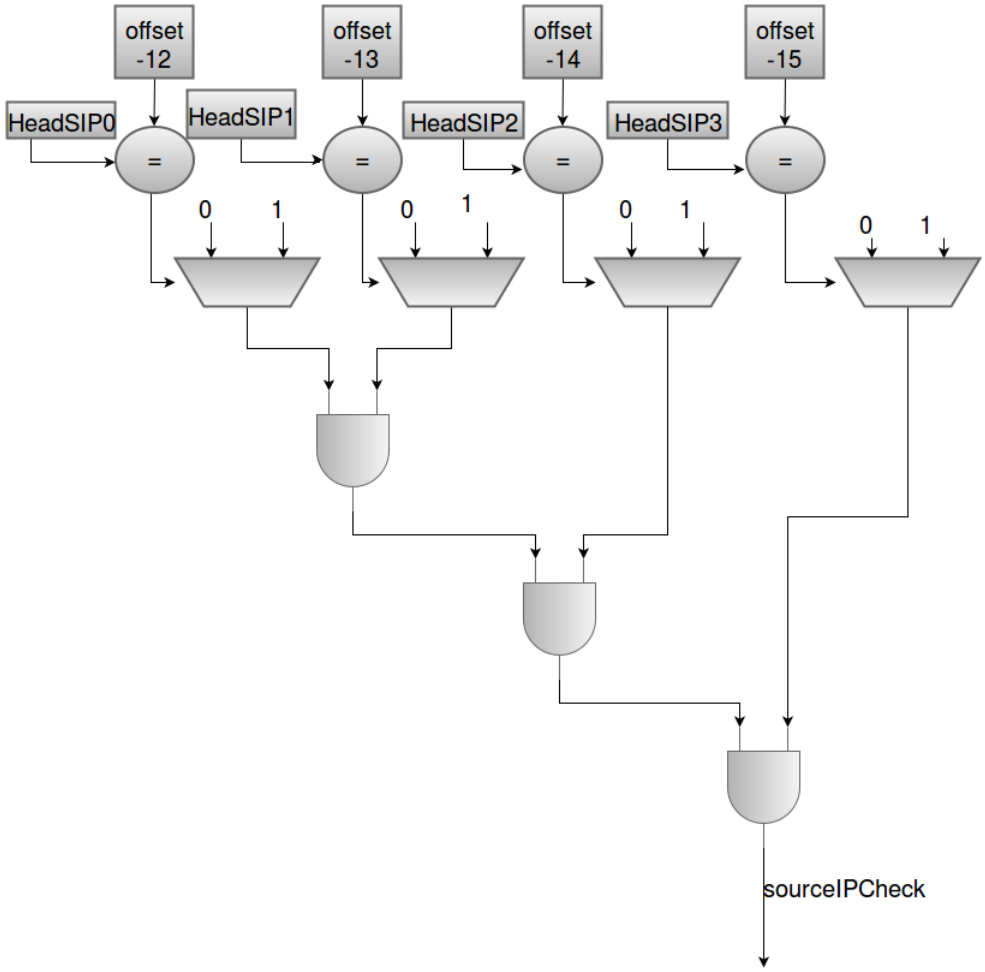
Επειδή ανάλογα με το πρωτόκολλο διαφοροποιούνται και τα πεδία των δεδομένων της κεφαλίδας του πακέτου, σε αυτό το σημείο διαμορφώνουμε και ένα select, το οποίο θα το χρησιμοποιήσουμε αργότερα. Ακολουθεί ο ψευδοκώδικας του:

```
protHead=header(offset=-9)
if (protHead==11)
    select = 1
else if (protHead==06)
    select = 2
else
    select = 0
```

Να σημειωθεί ότι οι αριθμοί 06 και 11 είναι οι κωδικοί σε δεκαεξαδική αναπαράσταση για το TCP και το UDP αντίστοιχα.

4.3.2 Έλεγχος διεύθυνσης IP αποστολέα

Όπως αναφέραμε και νωρίτερα, τα δεδομένα της κεφαλίδας του πακέτου τα δεχόμαστε σαν 8-bit μη προσημασμένους ακέραιους. Οι διευθύνσεις IP έχουν μήκος 32 bit. Έτσι, για να μπορεί να υπάρχει συμβατότητα, χρησιμοποιούμε 4 μεταβλητές για την κάθε διεύθυνση IP με offsets από -12 έως -15. Ακολουθεί το λογικό διάγραμμα για τον έλεγχο.



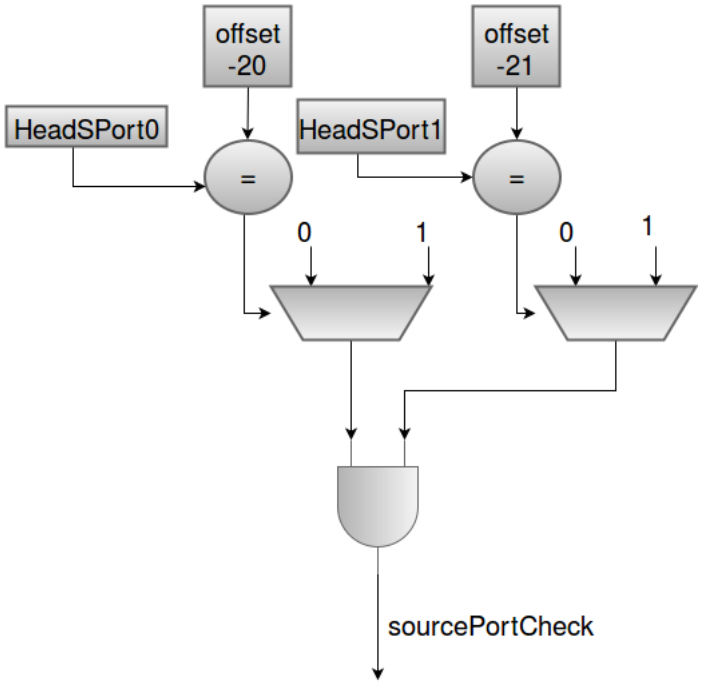
Σχήμα 4.8: Λογικό διάγραμμα ελέγχου διεύθυνσης IP αποστολέα

4.3.3 Έλεγχος διεύθυνσης IP προορισμού

Η αρχιτεκτονική αυτού του ελέγχου δεν διαφοροποιείται από την αρχιτεκτονική του ελέγχου της διεύθυνσης αποστολέα. Η μόνη διαφορά τους είναι τα offsets που χρησιμοποιούμε και αυτά είναι από -16 εως -19.

4.3.4 Έλεγχος θύρας αποστολέα

Για τον έλεγχο της θύρας αποστολέα, χρειαζόμαστε το select που δημιουργήσαμε νωρίτερα. Το select χρησιμοποιείται για να επιλέξουμε τα offsets που θα χρησιμοποιήσουμε επειδή ανάλογα με το πρωτόκολλο η θύρα αποστολέα βρίσκεται σε διαφορετική θέση στην κεφαλίδα του πακέτου. Η θύρα έχει μήκος 16 bit, έτσι για λόγους συμβατότητας χρησιμοποιούμε δύο μεταβλητές. Η αρχιτεκτονική μας δεν διαφοροποιείται πολύ από τους προηγούμενους ελέγχους. Ακολουθεί, όμως, το λογικό διάγραμμα στο Σχήμα 4.9.



Σχήμα 4.9: Λογικό διάγραμμα ελέγχου θύρας αποστολέα

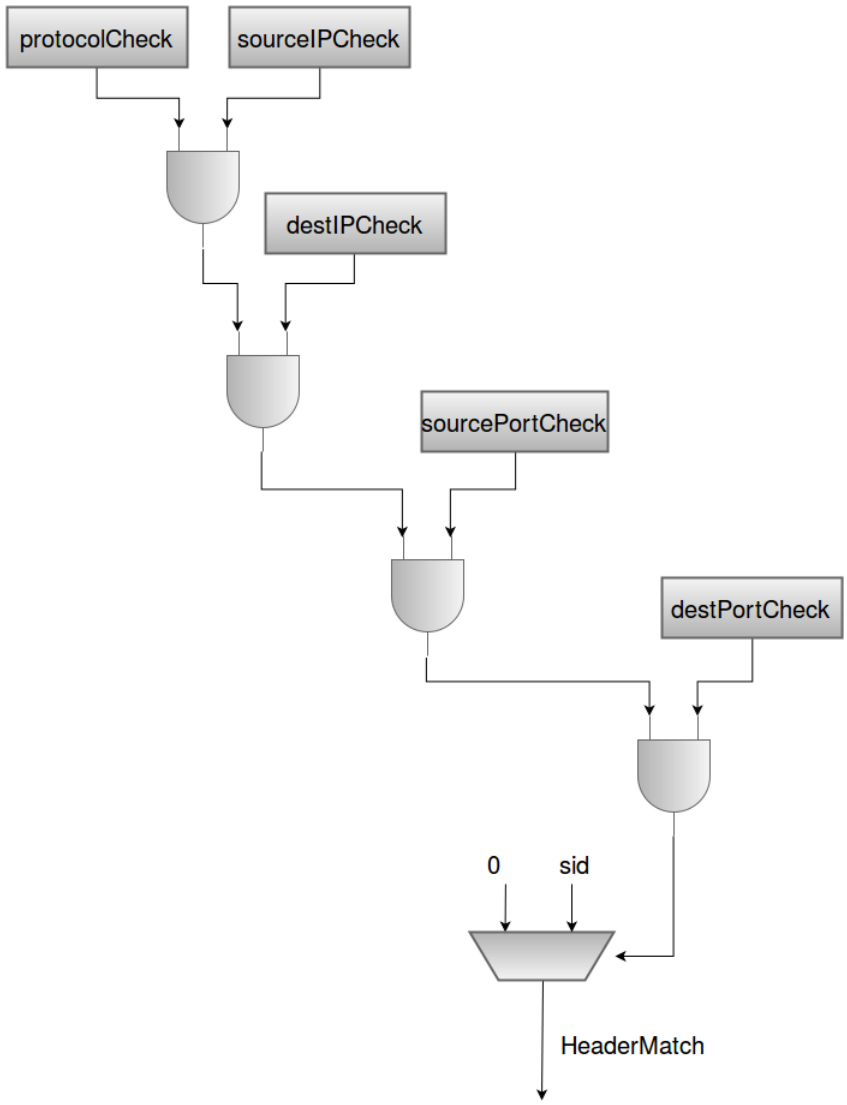
4.3.5 Έλεγχος θύρας προορισμού

Η αρχιτεκτονική αυτού του ελέγχου δεν διαφοροποιείται από την αρχιτεκτονική του ελέγχου της θύρας αποστολέα. Η μόνη διαφορά τους είναι τα offsets που χρησιμοποιούμε και αυτά είναι -22 και -23.

4.3.6 Τελικός έλεγχος κεφαλίδας πακέτου

Ο ρόλος αυτού του ελέγχου είναι να συνδυάσουμε τα αποτελέσματα όλων των προηγούμενων. Εκπληρώνονται δηλαδή, όλες οι απαιτήσεις του κανόνα για την κεφαλίδα του

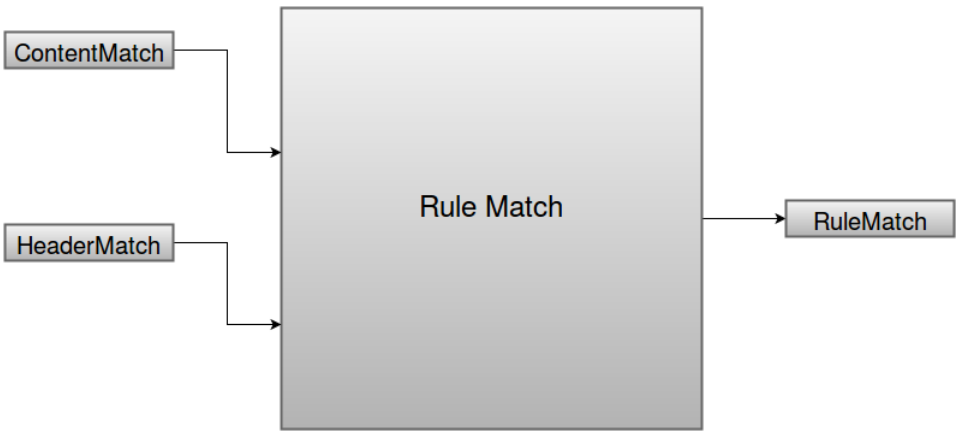
πακέτου; Η αρχιτεκτονική του είναι αρκετά απλή και αποτελείται μονάχα από πύλες and και έναν πολυπλέκτη. Ακολουθεί το λογικό διάγραμμα.



Σχήμα 4.10: Λογικό διάγραμμα τελικού ελέγχου κεφαλίδας πακέτου

4.4 Rule Match

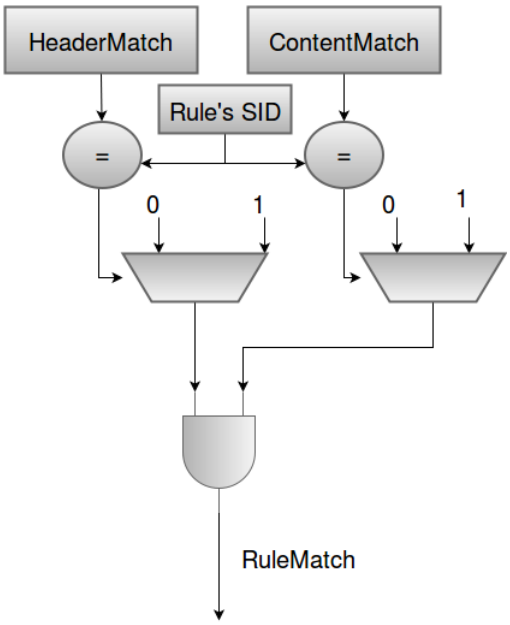
Η μονάδα ελέγχου κανόνα δέχεται ως εισόδους την έξοδο της μονάδας ελέγχου περιεχομένου και την έξοδο της μονάδας ελέγχου κεφαλίδας. Η έξοδος της είναι αν τελικά ο κανόνας επαληθεύεται ή όχι. Στο Σχήμα 4.11 παρουσιάζεται το σχηματικό διάγραμμα αυτής της μονάδας.



Σχήμα 4.11: Block diagram τελικής μονάδας ελέγχου κανόνα

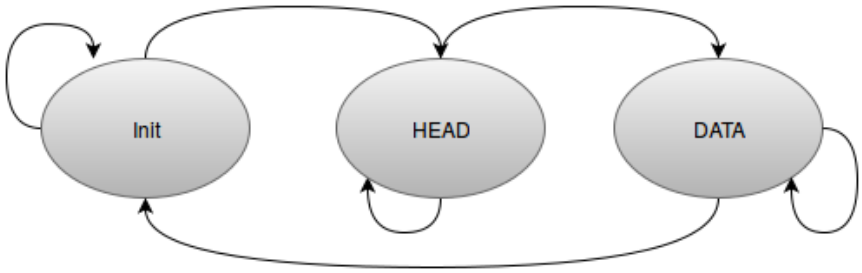
Η έξοδος της μονάδας ελέγχου περιεχομένου και κεφαλίδας είναι ο μοναδικός κωδικός του κανόνα που ταυτοποιείται και αυτό γιατί θέλουμε να ελέγχουμε πολλούς κανόνες παράλληλα και να γνωρίζουμε στο τέλος ποιοί από αυτούς ταυτοποιήθηκαν. Έτσι, σε αυτό το σημείο ελέγχουμε αν οι δύο έξοδοι έχουν την ίδια τιμή και αυτή είναι ο κωδικός του τρέχοντος κανόνα. Ακολουθεί το λογικό διάγραμμα αυτής της μονάδας ελέγχου.

Ολόκληρος ο κώδικας στο Παράρτημα 6.4.



Σχήμα 4.12: Λογικό διάγραμμα τελικής μονάδας ελέγχου κανόνα

4.5 Εναλλαγή πακέτων με State Machine



Σχήμα 4.13: Σχηματική αναπαράσταση της State Machine

Ως τώρα δεν έχουμε αναφερθεί στα διαφορετικά πακέτα που έρχονται και αν αυτά υποστηρίζονται από το σύστημα μας. Για να εξυπηρετήσουμε αυτήν ακριβώς τη λειτουργία, δημιουργήσαμε μία Μηχανή Πεπερασμένων Καταστάσεων -State Machine- η οποία λειτουργεί παράλληλα με την υπόλοιπη σχεδίαση μας, δέχεται σαν είσοδο, τα πακέτα που δέχεται και η υπόλοιπη σχεδίαση και βγάζει ως έξοδο έναν μοναδικό κωδικό για κάθε πακέτο. Με αυτόν τον τρόπο μπορούμε να γνωρίζουμε ποιο πακέτο βρέθηκε ύποπτο και από ποιους κανόνες.

Η **State Machine** έχει τρεις καταστάσεις: Init, Head και Data. Χρησιμοποιώντας έναν μετρητή, μετράμε πόσα δεδομένα έχουμε διαβάσει. Γνωρίζοντας, λοιπόν, ότι το συνολικό μήκος του πακέτου περιέχεται ως πληροφορία μέσα στην κεφαλίδα του πακέτου και συγκεκριμένα στις θέσεις 2 και 3, όπως φαίνεται και στο Σχήμα 4.6, το διαβάζουμε και το κρατάμε σε δύο registers. Ενώνοντας τις τιμές των δύο καταχωρητών, κάνοντας δηλαδή concatenate, έχουμε το συνολικό μήκος, το οποίο κρατάμε στον τελικό μας καταχωρητή.

Η **Init** είναι η αρχική μας κατάσταση στην οποία διαβάζεται το μήκος του πακέτου, όπως περιγράψαμε προηγουμένως. Στην κατάσταση αυτή μένουμε μέχρι να διαβαστεί το συνολικό μήκος του πακέτου. Μόλις διαβαστεί αυτό, αλλάζουμε κατάσταση και πηγαίνουμε στην κατάσταση Head.

Στην κατάσταση **Head** παραμένουμε τόσο όσο είναι το μήκος της κεφαλίδας του πακέτου, μόλις λοιπόν, ο μετρητής μας μετρήσει ως το μήκος της κεφαλίδας του πακέτου, αλλάζουμε κατάσταση και πηγαίνουμε στην κατάσταση Data.

Στην κατάσταση **Data** μένουμε όσο η τιμή του μετρητή μας είναι μικρότερη του μήκους του πακέτου που διαβάσαμε νωρίτερα. Όταν ο μετρητής περάσει αυτή την τιμή σημαίνει

ότι είμαστε στο πρώτο στοιχείο του επόμενου πακέτου. Οπότε, ο μετρητής μας πλέον, αρχικοποιείται στο ένα (1), ο καταχωρητής με το μήκος μηδενίζεται και πηγαίνουμε στην αρχική κατάσταση (Init). Με αυτόν τον τρόπο δεν χάνεται κάποιος κύκλος μεταξύ των πακέτων. Επίσης, σε αυτό το σημείο η State Machine βγάζει έξοδο έναν κωδικό για κάθε πακέτο. Για λόγους ευκολίας ο κωδικός αυτός είναι ένας ο αριθμός του πακέτου, δηλαδή 1 για το πρώτο πακέτο, 2 για το δεύτερο πακέτο κ.ο.κ.

Στο Σχήμα 4.13 φαίνεται η σχηματική αναπαράσταση της State Machine, ενώ παρακάτω παραθέτουμε τον ψευδοκώδικα αυτής.

Case(INIT)

$if(Counter == 2)$

$length1 = Input$

$Counter++$

$Mode \leftarrow INIT$

$elseif(Counter == 3)$

$length2 = Input$

$length = length1 \& length2$

$Counter++$

$Mode \leftarrow HEAD$

$else$

$Counter++$

$Mode \leftarrow INIT$

Case(HEAD)

$if(Counter < HeadLength)$

$Counter++$

$Mode \leftarrow HEAD$

$else$

$Counter++$

$Mode \leftarrow DATA$

Case(DATA)

$if(Counter < length)$

$Counter++$

$Mode \leftarrow DATA$

$else$

$Counter = 1$

$length = 0$

$Output = Output + 1$

$Mode \leftarrow INIT$

4.6 Αυτόματη παραγωγή κώδικα

Ο αριθμός των κανόνων Snort αγγίζει τους 5000, κάνοντας εύκολα αντιληπτό το γεγονός ότι είναι πολύ δύσκολο για κάποιον να κάτσει να γράψει κώδικα, και μάλιστα κώδικα περιγραφής υλικού, που να καλύπτει και τους 5000 κανόνες. Κάτι που έκανε ακόμα δυσκολότερη αυτή τη δουλειά, είναι το γεγονός ότι μιας και μιλάμε για κανόνες ανίχνευσης εισβολών δικτύου, αυτοί αλλάζουν με σχετικά μεγάλη συχνότητα. Θέλοντας, λοιπόν, να αποφύγουμε το μεγάλο φόρτο δουλειάς, αλλά και να λάβουμε υπ΄ όψιν μας τις διαφορετικές εκδόσεις των κανόνων, δημιουργήσαμε ένα προγραμματάκι το οποίο χρησιμοποιώντας λεκτικούς αναλυτές, εργαλεία ανάλυσης δεδομένων και παραγωγής αναφορών, θα δημιουργεί τα .maxj αρχεία που χρειάζεται ο compiler της Maxeler με σχετικά αυτοματοποιημένο τρόπο.

Η ιδέα ήταν ότι θα έχουμε έναν πρόλογο και έναν επίλογο για το τελικό μας αρχείο. Τα δύο αυτά αρχεία έχουν κομμάτια από το .maxj αρχείο που είχε παραχθεί από τον compiler της Maxeler. Οπότε εμείς ουσιαστικά έπρεπε να "χτίσουμε" το ενδιάμεσο αρχείο. Χρησιμοποιώντας την εντολή Cat, γνωστή για την επεξεργασία αρχείων σε Unix-based συστήματα, δημιουργήσαμε το τελικό αρχείο, βάζοντας

cat prolog middle epilogue≫ *kernel.maxj*

Πρώτο βήμα ήταν η δημιουργία ενός λεκτικού αναλυτή, ώστε να απομονώσουμε τις πληροφορίες που χρειαζόμαστε από τον κάθε κανόνα. Χρησιμοποιώντας τον λεκτικό αναλυτή FLEX αναγνωρίζουμε τις λέξεις κλειδιά Content και sid, ώστε να έχουμε τον μοναδικό κωδικό του κανόνα και τη συμβολοσειρά που αναζητάμε, τα οποία και γράφονται σε ένα αρχείο εξόδου. Ο λεκτικός αναλυτής δέχεται ως είσοδο το αρχείο Community.rules, το οποίο κατεβάσαμε από τη σελίδα του Snort και περιέχει τους κανόνες, με τη μορφή που περιγράφονται στην παράγραφο 2.1.2. Όπως προστάζει το συντακτικό του FLEX δημιουργήσαμε κανονικές εκφράσεις για την ανίχνευση των πληροφοριών μέσα στον κανόνα, ακολουθούμενες από ενέργειες. Ακολουθεί ένα παράδειγμα, με την κανονική έκφραση που γράψαμε για να βρούμε τον μοναδικό κωδικό.

$$\{sid\}\{token\}\{digit\}^* \{return \ \ TK\_D;\}$$

Στη συνέχεια, χρησιμοποιώντας την awk και απλές bash script εντολές δημιουργήσαμε το κείμενο που θέλαμε να βρεθεί μεταξύ προλόγου και επιλόγου.

4.7 CPU

Η CPU στη Maxeler τρέχει εκτελέσιμα αρχεία, τα οποία φορτώνουν .max αρχεία και τα εκτελούν στη διαθέσιμη DFE, δίνοντας εισόδους, παραμέτρους και ορίζοντας τις εξόδους. Υποστηρίζονται οι γλώσσες προγραμματισμού C, Python, Matlab κ.α. Εμείς επιλέξαμε C, λόγω εξοικείωσης.

Είσοδοι στο πρόγραμμα μας είναι δύο αρχεία. Το ένα προσομοιώνει τα πακέτα του δικτύου δεδομένων και το άλλο περιέχει τις ρυθμίσεις δικτύου. Όπως αναφέραμε και στο κεφάλαιο 2.1.2 οι κανόνες δίνουν τη δυνατότητα στον χρήστη να τους παραμετροποιήσει σύμφωνα με τις ανάγκες του. Μία από αυτές τις επιλογές είναι και να ορίσει τις διευθύνσεις IP που τον ενδιαφέρουν, καταχωρώντας τες σε μεταβλητές. Για παράδειγμα, μπορεί να θέλει να ελέγχει τα πακέτα που έρχονται από συγκεκριμένες διευθύνσεις, ή να ελέγχει όλα τα πακέτα που πηγαίνουν στο δίκτυο του σπιτιού, της εταιρείας κλπ. Αυτό είναι κάτι που θέλαμε και έπρεπε να συμπεριλάβουμε στην υλοποίηση μας. Έτσι, το αρχείο με τις ρυθμίσεις δικτύου εξυπηρετεί αυτόν ακριβώς το σκοπό.

Η CPU, λοιπόν, διαβάζει τα δύο αρχεία, τα δίνει ως είσοδο στο υλικό και εμφανίζει στην οθόνη και γράφει την έξοδο από το υλικό σε ένα τρίτο αρχείο, το output.txt

4.8 Ολοκληρωμένο Block Diagram

Στο Σχήμα 4.14 παρουσιάζεται το ολοκληρωμένο σχηματικό διάγραμμα, της σχεδίασης μας, όπως αυτή περιγράφηκε παραπάνω.

Η είσοδος του Kernel μας, είναι ένα Stream δεδομένων, τα οποία ξεχωρίζονται στα δεδομένα της κεφαλίδας του πακέτου και τα ωφέλιμα δεδομένα, την χρήσιμη πληροφορία. Αυτό γίνεται με έναν 5-bit μετρητή, ο οποίος μετρώντας από το 1 έως το 24 ελέγχει το

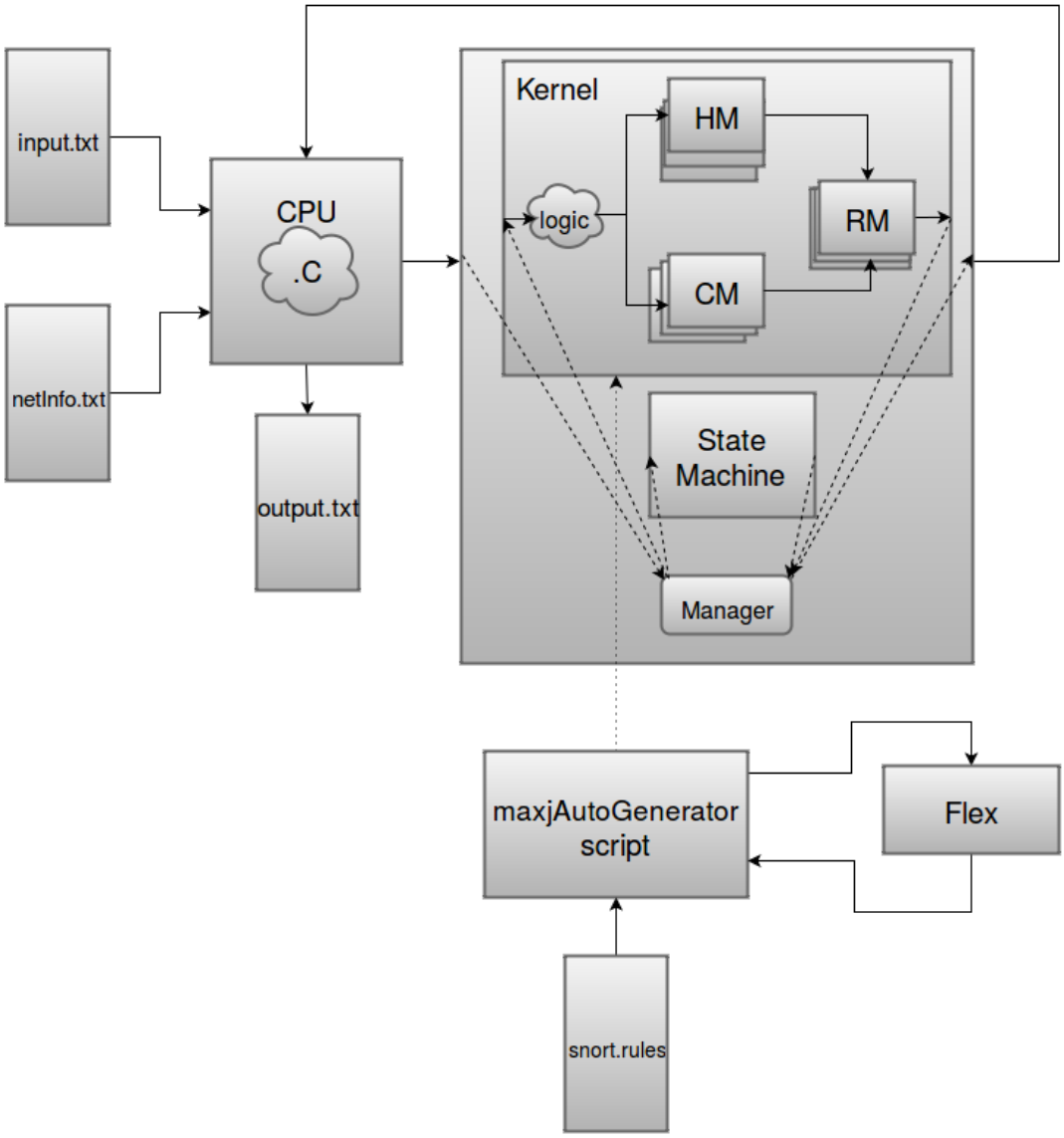
Stream της εισόδου. Είσοδοι, επίσης, είναι και οι πληροφορίες του δικτύου, οι οποίες όπως είδαμε αποθηκεύονται σε 4 καταχωρητές. Η Έξοδος του συστήματος είναι ελεγχόμενη (controlled output). Αν ένας μόνο κανόνας επαληθευτεί, τότε η έξοδος βγάζει τον μοναδικό κωδικό του κανόνα που "χτύπησε". Αν επαληθευτούν πολλοί κανόνες, τότε η έξοδος βγάζει τον πρώτο κανόνα που "χτύπησε" και αυτό γιατί σε αυτή την περίπτωση δεχόμαστε επίθεση, οπότε ποια ενέργεια θα πρέπει να εκτελεστεί πρώτη; Μάλλον θα πρέπει να πάρουμε πιο δραστήρια μέτρα. Αν δεν επαληθευτεί κανένας κανόνας, τότε δεν βγάζει έξοδο. Όπως φαίνεται και στο διάγραμμα, αλλά είδαμε και στο προηγούμενο κεφάλαιο, τη διαχείριση των εισόδων και των εξόδων την κάνει ο manager.

Όμοια είσοδος στη State Machine, είναι το ίδιο Stream δεδομένων, που εισάγεται στον Kernel, και αυτό γιατί θέλουμε, όταν έχουμε έξοδο από τη State Machine να γνωρίζουμε και στον Kernel ότι άλλαξε το πακέτο.

Παρουσιάζουμε παρακάτω τον κώδικα για το Top-level δίνοντας ενδεικτικά κώδικα για έλεγχο δύο κανόνων.

```
1  class SimpleSMKernel extends Kernel {
2    SimpleSMKernel(KernelParameters parameters) {
3      super(parameters);
4
5      DFEVar inp1 = io.input("max", dfeUInt(8));
6      DFEVar IP = io.scalarInput("iph", dfeUInt(64));
7
8      DFEVar IPHome0=IP.cast (dfeUInt((8)));
9      DFEVar IPHome1=IP.cast (dfeUInt((8)));
10     DFEVar IPHome2=IP.cast (dfeUInt((8)));
11     DFEVar IPHome3=IP.cast (dfeUInt((8)));
12   //----------------------> NETWORK CONFIGURATION
13     DFEVar IPNet0=IP.cast (dfeUInt((8)));
14     DFEVar IPNet1=IP.cast (dfeUInt((8)));
15     DFEVar IPNet2=IP.cast (dfeUInt((8)));
16     DFEVar IPNet3=IP.cast (dfeUInt((8)));
17   //----------------------> RULE MATCH (1-5) //in, Variables, sid
18     DFEVar rule=ruleMatch3(inp1, 49,49,49, 6);
19     DFEVar rule1=ruleMatch4(inp1, 98,99,100,101, 5);
20   //----------------------> HEADER MATCH (1-5) Protocol encoding:11->udp,
      06->tcp, 01->icmp, IP: any->0 (header,sid, protocol code, source IP,
```

```
    src Port1,src Port2, dest IP,dest Port1, dest Port2)
21    DFEVar hm=headerMatch(inp1,6,11,IPHome0,IPHome1,IPHome2,IPHome3
      ,25,89,IPNet0,IPNet1, IPNet2, IPNet3,0,0);
22    DFEVar hm1=headerMatch(inp1,5,11,IPHome0,IPHome1,IPHome2,IPHome3
      ,25,89,IPNet0,IPNet1, IPNet2, IPNet3,0,0);
23    //------------------------> FINAL MATCH
24    DFEVar match0 = finalMatch(rule,hm,6);
25    DFEVar match1 = finalMatch(rule1,hm1,5);
26
27    DFEVar y =constant.var( dfeUInt(1), 1);//1
28    DFEVar x =constant.var( dfeUInt(1), 0);//0
29    DFEVar zeroOut =constant.var(dfeUInt(32), 0);
30    //Control for output
31    DFEVar cntrl = dfeBool().newInstance(this);
32       cntrl <==
33                  (match1 === x) |
34                  (match0 === x) ;
35    //Output
36    DFEVar outp =
37                  (match0 === y) ? 6 :
38                  (match1 === y) ? 5 : zeroOut ;
39    //------------------------> STATE MACHINE
40    SMIO mySimpleSM = addStateMachine("SimpleSM", new
      SimpleSMStateMachine(this, width));
41    mySimpleSM.connectInput("max", inp);
42    DFEVar f = mySimpleSM.getOutput("fout");
43    //------------------------> OUTPUT
44    io.output("count1", outp, dfeUInt(32),cntrl);
45    io.output("fout", f, dfeUInt(32));
46  }
47 }
```



Σχήμα 4.14: Ολοκληρωμένο Block Diagram

Κεφάλαιο 5

Αποτελέσματα

5.1 Αποτελέσματα

Σε αυτό το σημείο θα παρουσιάσουμε τα αποτελέσματα της σχεδίασης μας. Παρακάτω θα δούμε τα αποτελέσματα διάφορων μετρήσεων επίδοσης, ώστε να έχουμε συμπεράσματα για τη σχεδίαση μας.

Η πρώτη μέτρηση αφορά την χωρική επίδοση για έλεγχο 48 κανόνων. Αναλυτικά, είχαμε 12 κανόνες που αναζητούσαν συμβολοσειρές μήκους 3 χαρακτήρων, 12 κανόνες που αναζητούσαν συμβολοσειρές μήκους 4 χαρακτήρων, 12 κανόνες που αναζητούσαν συμβολοσειρές μήκους 5 χαρακτήρων, και 12 κανόνες που αναζητούσαν συμβολοσειρές μήκους 6 χαρακτήρων. Σύνολο, λοιπόν, αναζητήσαμε 216 χαρακτήρες με μέσο μήκος συμβολοσειράς 4,5 χαρακτήρες ανά αναζήτηση. Στον Πίνακα 5.1 παρουσιάζονται τα αποτελέσματα της παραπάνω σχεδίασης.

Utilization		
	Total	per Character
Logic	4.76%	0.022%
LUTs	3.51%	0.016%
Primary FFs	4.13%	0.019%
Secondary FFs	1.75%	0.008%
Block Mem	1.36%	0.006%

Πίνακας 5.1: Πίνακας χωρικής επίδοσης για 48 κανόνες

Βάσει των παραπάνω εκτιμάται ότι με τη δεδομένη σχεδίαση μπορούν να "χωρέσουν" τουλάχιστον 1000 κανόνες. Αν λάβουμε υπ' όψιν μας και την επαναχρησιμοποίηση πόρων που κάνει πολύ καλά το σύστημα της Maxeler, μπορούν να υποστηριχθούν πολλοί περισσότεροι.

Σε αυτό το σημείο θα παρουσιάσουμε τα αποτελέσματα της επίδοσης της σχεδίασης μας στο χρόνο. Όμοια με προηγούμενα έχουμε μέτρηση για έλεγχο 48 κανόνων και εισόδους σε διάφορα μεγέθη, όπως φαίνονται στον Πίνακα 5.2. Οι χρόνοι που αναφέρονται παρακάτω αποτελούν μέσο όρο δέκα (10) μετρήσεων, καθώς το υλικό σε κάθε κλήση έδινε διαφορετικά αποτελέσματα με απόκλιση περίπου 0.0005(sec). Τέλος, η τελευταία στήλη του Πίνακα 5.3, αποτελεί τον πραγματικό ρυθμό του συστήματος γιατί έχουμε αφαιρέσει το κόστος σε χρόνο για κλήση του υλικού με σχεδόν μηδενικό αριθμό δεδομένων. Ο χρόνος αυτός είναι 0,03309sec, και αποτελεί μέσο όρο 10 μετρήσεων.

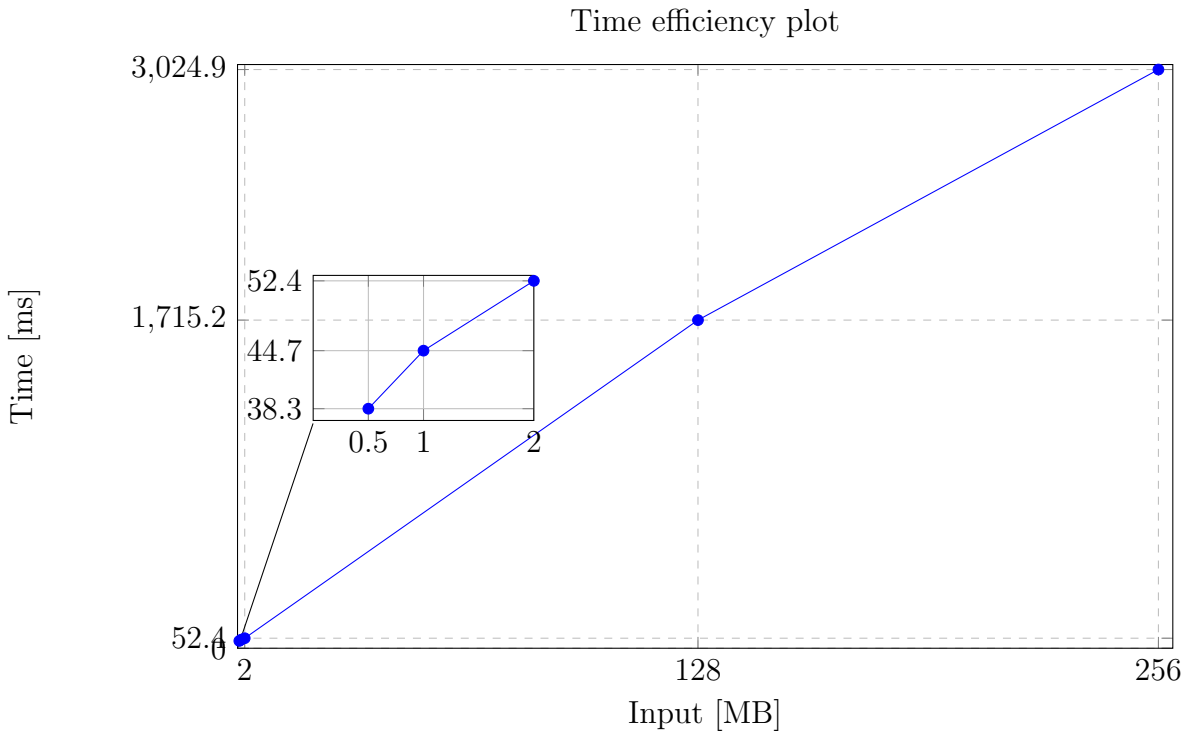
Time efficiency				
Input	Total (sec)	per Rule (ms)	per Character (ms)	rate (MB/sec)
0.5MB	0.0383	0.798	0.177	13.05
1MB	0.0447	0.932	0.207	22.3
2MB	0.0524	1.093	0.243	38.10
128MB	1.7152	35.734	7.941	75.2
256MB	3.0249	63.018	14.004	84.65

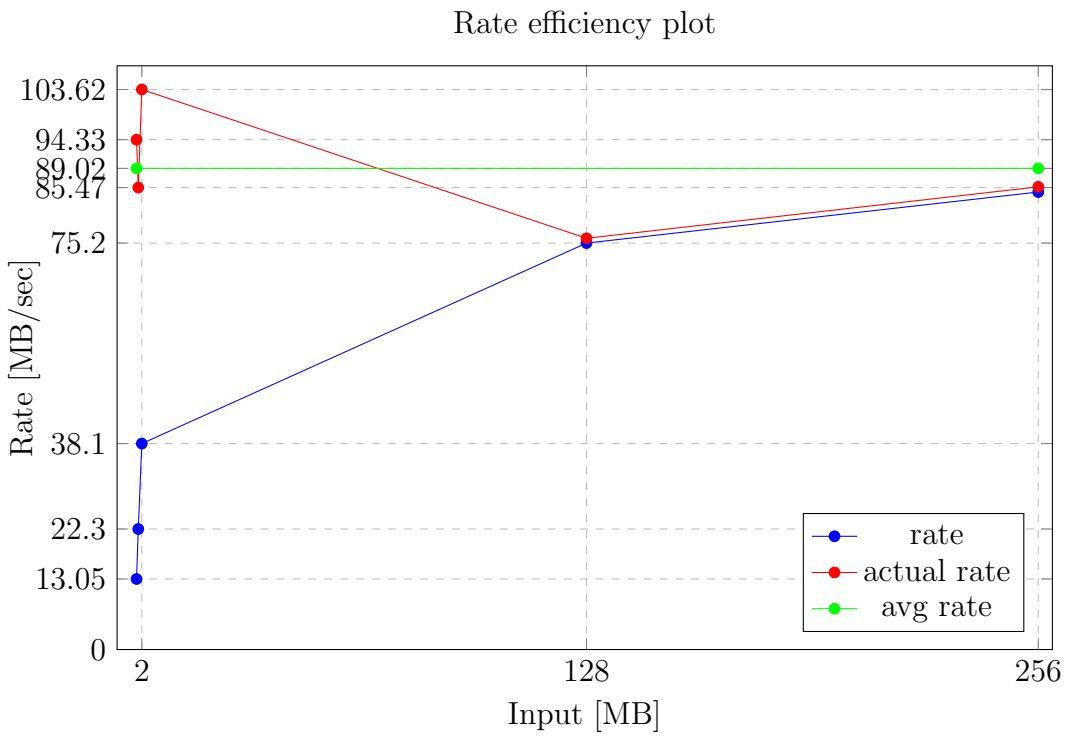
Πίνακας 5.2: Πίνακας χρονικής επίδοσης του συστήματος

Time efficiency and Processing		
Input	Total (sec)	Actual rate (MB/sec)
0.5MB	0.0053	94.33
1MB	0.0117	85.47
2MB	0.0193	103.62
128MB	1.6821	76.09
256MB	2.9901	85.61

Πίνακας 5.3: Πίνακας χρονικής επίδοσης του συστήματος

Όπως προκύπτει από τον παραπάνω πίνακα ο πραγματικός ρυθμός επεξεργασίας του συστήματος είναι σχεδόν σταθερός και κατά μέσο όρο 89.024MB/sec ή 712.192Mbps.





Σχήμα 5.1: Οι γραφικές παραστάσεις του ρυθμού επεξεργασίας όπως προέκυψε πειραματικά (rate), του ρυθμού επεξεργασίας μετά την αφαίρεση του overhead (actual rate) και ο μέσος όρος αυτού, καθώς είδαμε παραμένει σχεδόν σταθερός (avg rate)

5.2 Διδάγματα που αποκομίστηκαν

Κατά τη διάρκεια εκπόνησης της παρούσας διπλωματικής εργασίας, αντιμετωπίστηκαν διάφορες δυσκολίες. Κάποιες από αυτές αντιμετωπίστηκαν επιτυχώς και κάποιες άλλες όχι. Όπως και να έχει, όμως, καθετί που μελετάται προσφέρει γνώση. Έτσι, σε αυτό το σημείο θα παρουσιάσουμε τα διδάγματα που αποκομίσαμε σε όλη την πορεία της δουλειάς μας.

Για την κατανόηση του τρόπου λειτουργίας της Maxeler είναι χρήσιμο, όχι απλά να μελετώνται τα εγχειρίδια που δίνονται, αλλά η μελέτη τους παράλληλα με τα πλούσια παραδείγματα που δίνονται. Η τροποποίηση του δοθέντος κώδικα και ο πειραματισμός κρίνονται απαραίτητα.

Ένα από τα προβλήματα που αντιμετωπίσαμε, ήταν η αδυναμία να κάνουμε σύνθεση

από κάποιους υπολογιστές. Το παραπάνω αποδόθηκε τελικά στο λειτουργικό σύστημα. Για να ολοκληρωθεί με επιτυχία η σύνθεση, θα πρέπει να έχουμε λειτουργικό σύστημα στα αγγλικά. Για παράδειγμα, ήταν αδύνατο να κάνουμε σύνθεση μέσω απομακρυσμένης σύνδεσης στην επιφάνεια εργασίας Linux του Μηχανογραφικού κέντρου του Πολυτεχνείου Κρήτης γιατί έχει ελληνικό λειτουργικό σύστημα. Το παραπάνω συνέβη λόγω κάποιων Warnings που υπήρχαν στον κώδικα μας και το λειτουργικό μετέφραζε σε Προειδοποίηση. Πιθανόν αν δεν υπάρχουν Warnings να μην υπάρχει θέμα ασυμβατότητας λόγω της γλώσσας.

Παρατηρήθηκε ότι δημιουργούνται ασυμβατότητες όταν δημιουργούμε εξ΄ αρχής νέο Project στη Maxeler. Αυτό ξεπερνιέται αν εισάγουμε και τροποποιήσουμε ένα από τα υπάρχοντα.

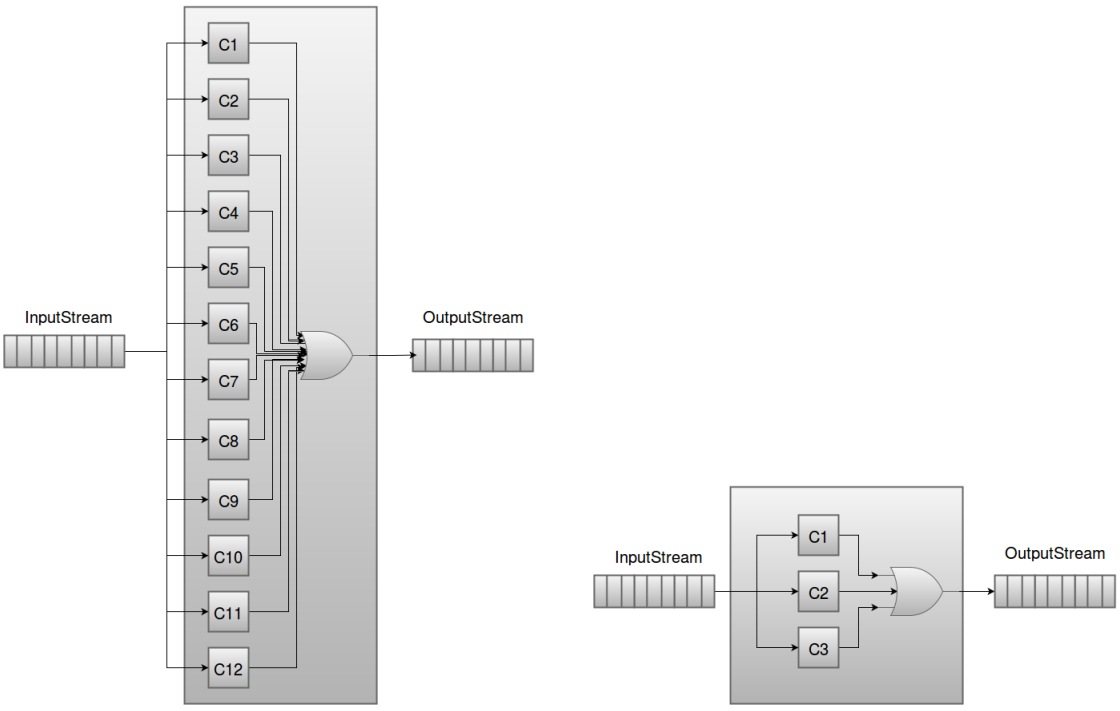
Συστήνεται συχνό clean στα built projects. Επίσης, προτείνεται η δημιουργία εξατομικευμένου manager, διότι ο κώδικας του δοθέντος είναι για απλά συστήματα, με αποτέλεσμα στα πιο σύνθετα να μπερδεύονται οι είσοδοι.

Κατά τη διαδικασία αναζήτησης και καθορισμού της αρχιτεκτονικής του συστήματος μας, μας απασχόλησε ο τρόπος που θα δίνονται οι κανόνες στο υλικό. Μία από τις προσεγγίσεις μας, ήταν να περνιούνται σε μία on-chip μνήμη. Η παραπάνω προσέγγιση απορρίφθηκε διότι είναι κάτι πολύ ακριβό και θα εξηγήσουμε το γιατί. Η Maxeler σε κάθε tick ουσιαστικά ελέγχει ένα στοιχείο της μνήμης με ένα στοιχείο της εισόδου (ή ένα εύρος στοιχείων). Δική μας απαίτηση, όμως, ήταν κάθε στοιχείο της εισόδου να ελέγχεται για κάθε στοιχείο της μνήμης. Αυτή ήταν και η λειτουργική μας απαίτηση, μιας και θέλαμε κάθε πακέτο να ελέγχεται για το σύνολο των κανόνων και όχι για έναν από αυτούς. Η λύση που βρήκαμε σε αυτό, ήταν να πολλαπλασιάσουμε την είσοδο τόσες φορές όσοι είναι ή κανόνες, ή την μνήμη (δηλαδή τους κανόνες) όσο το μέγεθος των πακέτων. Καταλαβαίνουμε, λοιπόν, ότι το παραπάνω θα ήταν πολύ μεγάλη σπατάλη πόρων αφού οι απαιτήσεις σε υλικό θα ήταν πολλαπλάσιες αυτού που θα θέλαμε. Το πρόβλημα διογκώνεται αν λάβουμε υπ΄όψιν μας ότι έχουμε να κάνουμε με την κίνηση ενός δικτύου.

Χρήσιμη διδαχή ήταν επίσης, ότι για ίδιες λειτουργίες, δεν υπάρχει διαφορά στη χρήση του χώρου, η δημιουργία ενός Kernel και πολλαπλών cores, ή αντίστροφα. Μία από τις δοκιμές μας ήταν η δημιουργία ενός Kernel και δύο cores που αναζητούσαν δύο διαφορετικές συμβολοσειρές. Τα αποτελέσματα που λάβαμε ήταν ίδια με αυτά της υλοποίησης όπου είχαμε δύο Kernels να αναζητούν καθένας μία συμβολοσειρά. Σημειώνεται, όμως, ότι οι

Kernels δεν μπορούν να επικοινωνούν μεταξύ τους. Αν για παράδειγμα το αποτέλεσμα της επεξεργασίας που κάνει ο ένας Kernel πρέπει να χρησιμοποιηθεί στην επεξεργασία του άλλου, αυτό είναι αδύνατον. Αυτό ξεπερνιέται αν χρησιμοποιήσουμε πολλαπλά cores, τα οποία μπορούν να επικοινωνούν μεταξύ τους.

Παρόμοια δοκιμή κάναμε για την αναζήτηση των αλφαριθμητικών. Δημιουργήσαμε δύο σχεδιάσεις και υλοποιήσεις. Στην πρώτη είχαμε πολλαπλά cores κάθενα από τα οποία έκανε αναζήτηση κάποιου αλφαριθμητικού. Συγκεκριμένα είχαμε τρία cores για αναζήτηση τεσσάρων διαφορετικών συμβολοσειρών μήκους δύο χαρακτήρων, τρία cores για αναζήτηση τεσσάρων διαφορετικών συμβολοσειρών μήκους τριών χαρακτήρων και τρία cores για αναζήτηση τεσσάρων διαφορετικών συμβολοσειρών μήκους τεσσάρων χαρακτήρων. Συνολικά λοιπόν, δώδεκα cores των οποίων οι έξοδοι κατέληγαν σε μία πύλη OR. Στην δεύτερη περίπτωση δημιουργήσαμε ένα core για αναζήτηση τεσσάρων διαφορετικών συμβολοσειρών μήκους δύο χαρακτήρων, ένα core για αναζήτηση τεσσάρων διαφορετικών συμβολοσειρών μήκους τριών χαρακτήρων και ένα core για αναζήτηση τεσσάρων διαφορετικών συμβολοσειρών μήκους τεσσάρων χαρακτήρων. Κάθε core περιείχε και μια πύλη OR. Συνολικά, λοιπόν, τρία cores των οποίων οι έξοδοι κατέληγαν σε μία πύλη OR. Στο Σχήμα 5.3 παρουσιάζονται τα σχηματικά διαγράμματα των παραπάνω σχεδιάσεων.



Πρώτη σχεδίαση Δεύτερη σχεδίαση

Σχήμα 5.2: Σχηματικά Διαγράμματα δοκιμαστικών σχεδιάσεων

Στον Πίνακα 5.4 παρατηρούμε ότι οι δύο σχεδιάσεις χρησιμοποιούν σχεδόν τους ίδιους πόρους. Έτσι αποφασίσαμε να προχωρήσουμε με τη δεύτερη σχεδίαση, όπως περιγράφεται και παραπάνω.

Utilization			
	Design 1	Design 2	Difference
Logic	2.44%	2.42%	+0.02%
LUTs	2.04%	2.04%	0%
Primary FFs	1.85%	1.83%	+0.02%
Secondary FFs	0.51%	0.51%	0%
Block Mem	1.08%	1.08%	0%

Πίνακας 5.4: Συγκριτικός πίνακας χωρικής επίδοσης διαφορετικών σχεδιάσεων, 1

Θέλοντας να επιβεβαιώσουμε το παραπάνω συμπέρασμα, αποφασίσαμε να δούμε πως θα άλλαζαν τα ποσοστά χρήσης των πόρων, αν οι χαρακτήρες που αναζητούμε έχουν απόσταση μεταξύ τους, κάτι το οποίο απαιτείται από τους κανόνες. Έτσι, πειράζοντας το Offset αλλάξαμε και τις δύο παραπάνω σχεδιάσεις, ώστε να αναζητούμε συμβολοσειρές

μήκους δύο χαρακτήρων με απόσταση μεταξύ τους τριάντα, συμβολοσειρές μήκους τριών χαρακτήρων με απόσταση μεταξύ τους εικοσιπέντε και συμβολοσειρές μήκους τεσσάρων χαρακτήρων με σχεδόν τυχαία απόσταση (0,-10,-32,-60) μεταξύ τους. Τα αποτελέσματα που λάβαμε παρουσιάζονται στον Πίνακα 5.5.

Utilization			
	Design 1	Design 2	Difference
Logic	2.50%	2.59%	-0.09%
LUTs	2.01%	2.00%	+0.01%
Primary FFs	1.89%	1.96%	-0.07%
Secondary FFs	0.52%	0.40%	+0.12%
Block Mem	1.08%	1.08%	0%

Πίνακας 5.5: Συγκριτικός πίνακας χωρικής επίδοσης διαφορετικών σχεδιάσεων, 2

Οι διαφορετικές προσεγγίσεις για την αναζήτηση στα περιεχόμενα του πακέτου, μας έδωσαν χρήσιμα συμπεράσματα. Όπως φαίνεται από τον Πίνακα 5.6, η αύξηση της απόστασης μεταξύ των αναζητούμενων χαρακτήρων, δεν επηρέασε ιδιαίτερα την επίδοση τους συστήματος μας.

Utilization of 2nd Design			
	First approach	Second approach	Difference
Logic	2.42%	2.59%	+0.17%
LUTs	2.04%	2.00%	-0.04%
Primary FFs	1.83%	1.96%	+0.13%
Secondary FFs	0.51%	0.40%	-0.11%
Block Mem	1.08%	1.08%	0%

Πίνακας 5.6: Συγκριτικός πίνακας χωρικής επίδοσης δεύτερης σχεδίασης ανάμεσα στις δύο προσεγγίσεις

Όσον αφορά την εναλλαγή των πακέτων, δοκιμάσαμε διάφορες μεθόδους. Δοκιμάσαμε να ξεχωρίζουμε στην είσοδο τα δεδομένα της κεφαλίδας και της χρήσιμης πληροφορίας του πακέτου και με έναν μετρητή να μετράμε το μήκος του πακέτου. Το πρόβλημα στη συγκεκριμένη προσέγγιση ήταν πως σε κάθε χτύπο του ρολογιού ο καταχωρητής που

κρατούσε το μήκος του πακέτου, έπαιρνε διαφορετική τιμή, με αποτέλεσμα, ο μετρητής να μετράει συνεχώς και να μην έχουμε σωστή συνθήκη στο controlled input.

Μια άλλη προσέγγιση ήταν να δεχόμαστε τα μήκη των πακέτων με scalar input. Οι scalar είσοδοι όμως, χρησιμοποιούνται σαν σταθερές, άρα θα έπρεπε όλα τα πακέτα μας να έχουν το ίδιο μήκος.

Κεφάλαιο 6

Συμπεράσματα-Μελλοντικές Εργασίες

6.1 Συμπεράσματα

Η διαφορά της Ανώτατης εκπαίδευσης από τις προηγούμενες βαθμίδες της εκπαίδευσης, είναι η σύνδεση της με την έρευνα, δηλαδή την παραγωγή νέας γνώσης. Η διπλωματική εργασία αποτελεί την δημιουργική επιστημονική ερευνητική εργασία κάθε φοιτητή και μέσω αυτής αποκτάται η ικανότητα διεξαγωγής πειραμάτων, ανάλυσης των αποτελεσμάτων τους και η εξαγωγή συμπερασμάτων.

Η παρούσα διπλωματική εργασία έδωσε τη δυνατότητα να ερευνήσουμε, να αναλύσουμε και να εξάγουμε χρήσιμα συμπεράσματα για τη χρήση υλικού και υβριδικών υπερυπολογιστών στην προστασία δικτύων και αυτά τα συμπεράσματα θα παρουσιάσουμε σε αυτό το κεφάλαιο.

Όπως είδαμε στο κεφάλαιο 5.1 η επίδοση του συστήματος μας αυξάνεται όσο αυξάνεται και ο όγκος δεδομένων προς επεξεργασία.

Στον Πίνακα 5.1 βλέπουμε ότι η χωρική επίδοση του συστήματος μας παραμένει σχεδόν σταθερή όταν αυξάνουμε τα δεδομένα προς επεξεργασία. Χαρακτηριστικά το κόστος σε χώρο ανά χαρακτήρα, μειώνεται στο μισό όταν διπλασιάζουμε τους κανόνες που ελέγχουμε, και τους χαρακτήρες που αναζητούμε. Αυτό συμβαίνει γιατί η Maxeler κάνει πολύ καλή επαναχρησιμοποίηση των πόρων της. Όταν δηλαδή δημιουργηθεί ένα core στο υλικό, και αυτό αργότερα ζητηθεί ξανά, επαναχρησιμοποιείται αυτό που έχει ήδη δημιουργηθεί.

Όπως φαίνεται και στον Πίνακα 5.2 έχουμε μείωση στο χρόνο που απαιτείται για τον έλεγχο ανά χαρακτήρα και ανά κανόνα όταν αυξάνουμε τα δεδομένα. Επιβεβαιώνεται έτσι το παραπάνω συμπέρασμα, γιατί θα μπορούσε για παράδειγμα η επαναχρησιμοποίηση πόρων να μας κοστίζει πάρα πολύ σε χρόνο, πράγμα που δεν συμβαίνει. Η αύξηση βέβαια της επίδοσης στο χρόνο δεν είναι διπλάσια, όπως είναι στο χώρο, κερδίζουμε όμως, περίπου 66%.

Είναι σαφές ότι η γνώση που αποκτήθηκε θα πρέπει να επιστραφεί στην ίδια την κοινωνία. Το σύστημα που αναπτύξαμε, λοιπόν, θα μπορούσε να αξιοποιηθεί για την προστασία δικτύων σχολείων, νοσοκομείων, εκπαιδευτικών οργανισμών και όχι μονάχα για την προστασία των δεδομένων μεγάλων εταιρειών, όπως είθισται να χρησιμοποιούνται τέτοια συστήματα σήμερα. Είναι βέβαια αντιληπτό ότι για να έχει, για παράδειγμα, ένα σχολείο, τη δυνατότητα να αποκτήσει τον υπερυπολογιστή της εταιρείας Maxeler Technologies θα πρέπει να του παρέχεται και η οικονομική δυνατότητα για αυτό. Έτσι, λύσεις προς αυτήν την κατεύθυνση μπορούν να δοθούν με αλλαγές στην οργάνωση της οικονομίας και κατ΄ επέκταση της κοινωνίας.

6.2 Μελλοντικές Εργασίες

Σχεδόν σε κάθε διπλωματική ή ερευνητική εργασία υπάρχει πληθώρα σκέψεων για μελλοντική επέκταση. Σκέψεις μάλιστα προκύπτουν καθ΄όλη τη διάρκεια εκπόνησης της εργασίας. Είναι, όμως, κοινώς αποδεκτό ότι μονάχα ένα κομμάτι αυτών των σκέψεων μπορούν να διερευνηθούν και να υλοποιηθούν σε συγκεκριμένο χρόνο.

Στην παρούσα διπλωματική εργασία παρουσιάστηκε μία υλοποίηση για ένα κομμάτι του συνόλου των κανόνων Snort. Μελλοντικά η εργασία μπορεί να επεκταθεί καλύπτοντας το σύνολο των κανόνων ή ακόμα και δημιουργία εξατομικευμένων κανόνων. Επίσης, η ομάδα κανόνων που χρησιμοποιήθηκε είχε κάποιες μόνο από τις επιλογές που δίνουν οι κανόνες. Μπορεί, λοιπόν, να γίνει επέκταση του προγράμματος μετάφρασης των κανόνων και αυτόματης παραγωγής κώδικα καθώς και της μονάδας ελέγχου των περιεχομένων του πακέτου, ώστε να ικανοποιούνται όλες οι επιλογές που δίνει το Snort.

Στόχος για μελλοντική δουλειά είναι και η περαιτέρω παραλληλοποίηση του συστήματος μας. Για παράδειγμα, θα μπορούσαν να χρησιμοποιηθούν Kernels παράλληλα, λειτουργία, που μέχρι στιγμής δεν υποστηρίζεται.

Στο υπάρχον σύστημα χρησιμοποιούμε έναν Kernel και πολλαπλά cores. Στην επίδοση του συστήματος, όσον αφορά τον χώρο, όπως είδαμε δεν είχε διαφορά από τη χρήση ενός core και πολλών Kernels. Θα μπορούσαμε, όμως, να μετρήσουμε και την επίδοση όσον αφορά το χρόνο, όπου μπορεί να είχαμε καλύτερα αποτελέσματα.

Όσο αφορά την επίδοση του συστήματος, έχουμε να παρατηρήσουμε ότι θα μπορούσε να βελτιωθεί αν αξιοποιούσαμε πλήρως τις δυνατότητες του υπερυπολογιστή της Maxeler. Είναι ανάγκη να υπάρξει μεγαλύτερη εξοικείωση και τεχνογνωσία για τη συγκεκριμένη πλατφόρμα για να γίνει αυτό.

Γνωρίζουμε επίσης, ότι η Maxeler δίνει τη δυνατότητα σύνδεσης με πραγματικό δίκτυο. Σε επόμενα βήματα, μπορεί να διερευνηθεί η συγκεκριμένη δυνατότητα, ώστε να ληφθούν αποτελέσματα που ανταποκρίνονται σε πραγματικές συνθήκες.

Τέλος, στην εργασία μας χρησιμοποιήθηκε μία από τις τέσσερις Xilinx Virtex 6 FPGAs που προσφέρονται. Η αξιοποίηση και των τεσσάρων, υπολογίζουμε ότι θα οδηγήσει σε αύξηση της επίδοσης του συστήματος μας.

Παράρτημα

```
class SimpleSMKernel extends Kernel {
  SimpleSMKernel(KernelParameters parameters) {
    super(parameters);

    DFEVar inp1 = io.input("max", dfeUInt(8));
    DFEVar IP = io.scalarInput("iph", dfeUInt(64));

    DFEVar IPHome0=IP.cast (dfeUInt((8)));
    DFEVar IPHome1=IP.cast (dfeUInt((8)));
    DFEVar IPHome2=IP.cast (dfeUInt((8)));
    DFEVar IPHome3=IP.cast (dfeUInt((8)));
  //----------------------> NETWORK CONFIGURATION
    DFEVar IPNet0=IP.cast (dfeUInt((8)));
    DFEVar IPNet1=IP.cast (dfeUInt((8)));
    DFEVar IPNet2=IP.cast (dfeUInt((8)));
    DFEVar IPNet3=IP.cast (dfeUInt((8)));
  //----------------------> RULE MATCH (1-5) //in, Variables, sid
    DFEVar rule=ruleMatch3(inp1, 49,49,49, 6);
    DFEVar rule1=ruleMatch4(inp1, 98,99,100,101, 5);
  //----------------------> HEADER MATCH (1-5) Protocol encoding:11->udp,
   06->tcp, 01->icmp, IP: any->0 (header,sid, protocol code, source IP,
   src Port1,src Port2, dest IP,dest Port1, dest Port2)
    DFEVar hm=headerMatch(inp1,6,11,IPHome0,IPHome1,IPHome2,IPHome3
   ,25,89,IPNet0,IPNet1, IPNet2, IPNet3,0,0);
    DFEVar hm1=headerMatch(inp1,5,11,IPHome0,IPHome1,IPHome2,IPHome3
   ,25,89,IPNet0,IPNet1, IPNet2, IPNet3,0,0);
     //----------------------> FINAL MATCH
    DFEVar match0 = finalMatch(rule,hm,6);
    DFEVar match1 = finalMatch(rule1,hm1,5);
```

```
26
27      DFEVar y =constant.var( dfeUInt(1), 1);//1
28      DFEVar x =constant.var( dfeUInt(1), 0);//0
29      DFEVar zeroOut =constant.var(dfeUInt(32), 0);
30      //Control for output
31      DFEVar cntrl = dfeBool().newInstance(this);
32        cntrl <==
33                  (match1 === x) |
34                  (match0 === x) ;
35      //Output
36      DFEVar outp =
37                  (match0 === y) ? 6 :
38                  (match1 === y) ? 5 : zeroOut ;
39      //------------------------> STATE MACHINE
40      SMIO mySimpleSM = addStateMachine("SimpleSM", new
        SimpleSMStateMachine(this, width));
41      mySimpleSM.connectInput("max", inp);
42      DFEVar f = mySimpleSM.getOutput("fout");
43      //------------------------> OUTPUT
44      io.output("count1", outp, dfeUInt(32),cntrl);
45      io.output("fout", f, dfeUInt(32));
46    }
47  }
```

Listing 6.1: Κώδικας Top Level

```
1  class SimpleSMKernel extends Kernel {
2    DFEVar headerMatch(DFEVar in,int sid, int prot, DFEVar srcIP0,DFEVar
       srcIP1,DFEVar srcIP2,DFEVar srcIP3, int srcP1,int srcP2, DFEVar
       DestIP0,DFEVar DestIP1,DFEVar DestIP2,DFEVar DestIP3,int destP1,int
       destP2) {
3
4      DFEVar x =constant.var( dfeUInt(8), 0);//0
5      DFEVar y =constant.var( dfeUInt(8), 1);//1
6      DFEVar o =constant.var( dfeUInt(2), 0);//0
7      DFEVar p =constant.var( dfeUInt(2), 1);//1
8      DFEVar z =constant.var( dfeUInt(2), 2);//2
9      //get protocol
10     DFEVar prH=stream.offset(in, -9);
```

```
11      //11->udp, 06->tcp, 01->icmp
12      DFEVar select = prH===11 ? z
13              : prH===06 ? p
14              : o;
15      //protocol check
16      DFEVar protCheck = prH===prot ? y : x ;
17      //SELECT=0->NU,1->TCP,2->UDP,3->ICMP
18      //Source Port
19       DFEVar srcPH1;
20       srcPH1 = control.mux(select, x, stream.offset(in, -20), stream.
        offset(in, -20)) ;
21       DFEVar srcPH2;
22       srcPH2 = control.mux(select, x, stream.offset(in, -21), stream.
        offset(in, -21)) ;
23      //dEST Port
24       DFEVar destPH1;
25       destPH1= control.mux(select, x, stream.offset(in, -22), stream.
        offset(in, -22)) ;
26       DFEVar destPH2;
27       destPH2= control.mux(select, x, stream.offset(in, -23), stream.
        offset(in, -23)) ;
28      //source port check
29      DFEVar srcP1Check = srcPH1===srcP1 ? y : x ;
30      DFEVar srcP2Check = srcPH2===srcP2 ? y : x ;
31            DFEVar srcPCheck = (srcP1Check===y & srcP2Check===y) ? y : x ;
32            //destination port check
33      DFEVar destP1Check = destPH1===destP1 ? y : x ;
34      DFEVar destP2Check = destPH2===destP2 ? y : x ;
35      DFEVar destPCheck = (destP1Check===y & destP2Check===y) ? y : x ;
36      //Source IP
37      DFEVar srcIPH1=stream.offset(in, -12);
38      DFEVar srcIPH2=stream.offset(in, -13);
39      DFEVar srcIPH3=stream.offset(in, -14);
40      DFEVar srcIPH4=stream.offset(in, -15);
41      //Dest IP
42      DFEVar destIPH1=stream.offset(in, -16);
43      DFEVar destIPH2=stream.offset(in, -17);
44      DFEVar destIPH3=stream.offset(in, -18);
```

```
45      DFEVar destIPH4=stream.offset(in, -19);
46      //Source IP Check
47      DFEVar srcIP1Check = srcIPH1===srcIP0 ? y : x ;
48      DFEVar srcIP2Check = srcIPH2===srcIP1 ? y : x ;
49      DFEVar srcIP3Check = srcIPH3===srcIP2 ? y : x ;
50      DFEVar srcIP4Check = srcIPH4===srcIP3 ? y : x ;
51      DFEVar srcIPCheck = (srcIP1Check===y & srcIP2Check===y & srcIP3Check
      ===y & srcIP4Check===y) ? y : x ;
52      //Dest IP Check
53      DFEVar DestIP1Check = destIPH1 === DestIP0 ? y : x ;
54      DFEVar DestIP2Check = destIPH2 === DestIP1 ? y : x ;
55      DFEVar DestIP3Check = destIPH3 === DestIP2 ? y : x ;
56      DFEVar DestIP4Check = destIPH4 === DestIP3 ? y : x ;
57      DFEVar destIPCheck = (DestIP1Check===y & DestIP2Check===y &
      DestIP3Check===y & DestIP4Check===y) ? y : x ;
58      //final check
59      DFEVar res = (protCheck===y & srcPCheck===y & destPCheck===y &
      srcIPCheck===y & destIPCheck===y) ? sid : x ;
60      // Output
61      return res;
62
63    }
64  }
65  ////////////////////////////////////////////////////////////////
```

Listing 6.2: Κώδικας Header Match

```
1  class SimpleSMKernel extends Kernel {
2    DFEVar ruleMatch3(DFEVar in, int Var0,int Var1,int Var2, int sid) {
3
4      DFEVar z =constant.var( dfeUInt(8), 0);//0
5      DFEVar in0=stream.offset(in, 0);
6      DFEVar in1=stream.offset(in, 1);
7      DFEVar in2=stream.offset(in, 2);
8      // Logic
9       DFEVar result0 = in0===Var0 ? sid : z ;
10      DFEVar result1 = in1===Var1 ? sid : z ;
11      DFEVar result2 = in2===Var2 ? sid : z ;
12     // Output
```

```
13      DFEVar result = (result0===sid) & (result1===sid) & (result2===sid)
    ? sid : z ;
14     //return result;
15     return result;
16   }
17 ////////////////////////////////////////////////////////////////
18   DFEVar ruleMatch4(DFEVar in, int Var0,int Var1,int Var2,int Var3, int
    sid) {
19
20     DFEVar z =constant.var( dfeUInt(8), 0);//0
21     DFEVar in0=stream.offset(in, 0);
22     DFEVar in1=stream.offset(in, -1);
23     DFEVar in2=stream.offset(in, -2);
24     DFEVar in3=stream.offset(in, -3);
25     // Logic
26      DFEVar result0 = in0===Var0 ? sid : z ;
27      DFEVar result1 = in1===Var1 ? sid : z ;
28      DFEVar result2 = in2===Var2 ? sid : z ;
29      DFEVar result3 = in3===Var3 ? sid : z ;
30     // Output
31      DFEVar result = (result0===sid) & (result1===sid)& (result2===sid)&
    (result3===sid) ? sid : z ;
32     //return result;
33     return result;
34   }
35 }
```

Listing 6.3: Ενδεικτικός κώδικας Rule Match

```
1 class SimpleSMKernel extends Kernel {
2   DFEVar finalMatch(DFEVar var1, DFEVar var2, int sid) {
3     DFEVar x =constant.var( dfeUInt(1), 0);//0
4     DFEVar y =constant.var( dfeUInt(1), 1);//1
5     DFEVar result = var1===sid & var2===sid ? y : x ;
6     return result;
7   }
8 }
```

Listing 6.4: Κώδικας Final Match

Βιβλιογραφία

[1] SNORT Network Intrusion Detection System, https://www.snort.org/.

[2] Maxeler Technologies, Dataflow Computing, https://www.maxeler.com/technology/dataflow-computing/.

[3] Maxeler Technologies, Multiscale Dataflow Programming, Version 2013.2.2.

[4] Maxeler Technologies, MaxCompiler Manager Compiler Tutorial, Version 2013.2.2.

[5] Maxeler Technologies, MaxCompiler State Machine Tutorial, Version 2013.2.2.

[6] FLEX: The Fast Lexical Analyzer, http://flex.sourceforge.net/.

[7] Cisco, http://www.cisco.com/c/el$_g r/index.html$.

[8] Netscreen, http://www.juniper.net/support/eol/ns$_h w.html$.

[9] Checkpoint, https://www.checkpoint.com/.

[10] Netfilter, https://www.netfilter.org/.

[11] Nmap, https://nmap.org/.

[12] L7-filter. Application Layer Packet Classifier, http://l7- filter.sourceforge.net.

[13] M. Roesch: "Snort-lightweight intrusion detection for networks." In: *13th USENIX conference on System administration*, Seattle, Washington, 1999, pp. 229-238.

[14] V. Paxson: "Bro: A system for detecting network intruders in real-time." In: *Computer networks*,1999, 31(23): 2435-2463.

[15] Convey Computers-now Micron, https://www.micron.com.

[16] C. R. Clark, C. D. Ulmer, and D. E. Schimmel: "An FPGA-based Network Intrusion Detection System with On-chip Network Interfaces." In: *International Journal of Electronics*, 2006, 93(6): 403-420.

[17] Nigel Jacob and Carla Brodley: "Offloading IDS Computetion to the GPU" In: *22nd Annual Computer Security applications Conference, ACSAC '06*, 2006.

[18] G.Vasiliadis, S. Antonatos, M. Polychronakis, E. P. Markatos and S.Ioannidis: "Gnort: High performance network intrusion detection using graphics processors" In: *Conference: Proceedings 11th International Symposium, Recent Advances in Intrusion Detection, RAID 2008*, Cambridge, MA, USA, September 15-17, 2008.

[19] Vlastimil Košař, Martin Žádník, and Jan Kořenek: "NFA Reduction for Regular Expressions Matching Using FPGA". In: *Field-Programmable Technology (FPT), 2013 International Conference*, pp. 338-341.

[20] Giorgos Vasiliadis, Michalis Polychronakis, and Sotiris Ioannidis: "MIDeA: A Multi-Parallel Intrusion Detection Architecture". In: *CCS '11 Proceedings of the 18th ACM conference on Computer and communications security*, pp. 297-308.

[21] Le Hoang Long, Tran Trung Hieu, Vu Tan Tai, Nguyen Hoa Hung, Tran Ngoc Thinh, and Dinh Duc Anh Vu: "ECEB: Enhanced Constraint Repetition Block for Regular Expression Matching on FPGA". In: *ECTI Transactions on Electrical Eng., Electronics, and Communications Vol.9*, No1 February 2011, pp. 65-74.

[22] Ioannis Sourdis, Dionisios N. Pnevmatikatos, and Stamatis Vassiliadis: "Scalable Multigigabit Pattern Matching for Packet Inspection". In: *IEEE Transactions on very Large Scale Integration (VLSI) Systems*, Vol.16, No2, February 2008, pp. 156-166.

[23] Abhishek Mitra, Walid Najjar, and Laxmi Bhuyan: "Compiling PCRE to FPGA for accelerating SNORT IDS". In: *ANCS '07 Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*, pp. 127-136.

[24] Ioannis Sourdis, Vasilis Dimopoulos, Dionisios Pnevmatikatos, and Stamatis Vassiliadis: "Packet Pre-filtering for Network Intrusion Detection". In: *Architecture for*

Networking and Communications systems, 2006. ANCS 2006. ACM/IEEE Symposium, pp. 183-192.

[25] Ioannis Sourdis, Dionisios Pnevmatikatos, Stephan Wong, and Stamatis Vassiliadis: "A Reconfigurable perfect-Hashing scheme for packet Inspection." In: *International Conference on Field Programmable Logic and Applications*, 2005. pp. 644-647.

[26] Ioannis Sourdis and Dionisios Pnevmatikatos: "Pre-decoded CAMs for Efficient and High-Speed NIDS Pattern Matching". In: *Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium*, pp. 258-267.

[27] Ioannis Sourdis and Dionisios Pnevmatikatos: "Fast, Large-Scale String Match for a 10Gbps FPGA-based Network Intrusion Detection System". In: *13th International Conference*, FPL 2003, Lisbon, Portugal, pp. 880-889.

[28] Βασίλειος Δημόπουλος: "Αξιολόγηση Λογισμικού Ανίχνευσης ικτυακών Εισβολών με Υποβοήθηση από Υλικό". *Διπλωματική Εργασία*, τμήμα Ηλεκτρονικών Μηχανικών και Μηχανικών Υπολογιστών, Πολυτεχνείο Κρήτης, 2004.

[29] Δημήτριος Στέφανος Δασκαλάκης: "Snort DPI on FPGA with GigE". *Διπλωματική Εργασία*, τμήμα Ηλεκτρονικών Μηχανικών και Μηχανικών Υπολογιστών, Πολυτεχνείο Κρήτης, 2012.

[30] Ιωάννης Μακρής: "Παράλληλη κατηγοριοποίηση κεφαλίδας πακέτων με χρήση γραφικού επεξεργαστή". *Διπλωματική Εργασία*, τμήμα Ηλεκτρονικών Μηχανικών και Μηχανικών Υπολογιστών, Πολυτεχνείο Κρήτης, 2010.

[31] Ιωάννης Σούρδης: "Efficient and High-Speed FPGA-based String Matching for Packet Inspection". *Μεταπτυχιακή διατριβή*, τμήμα Ηλεκτρονικών Μηχανικών και Μηχανικών Υπολογιστών, Πολυτεχνείο Κρήτης, 2004.

[32] R.W. Floyd and J.D. Ullman: "The Compilation of Regular Expression into Integrated Circuits," In: *J. ACM*, vol. 29, no. 3, July 1982, pp. 603-622.

[33] R. Sidhu and V.K. Prasanna: "Fast Regular Expression Matching Using FPGAs". In: *Proc. Ninth IEEE Symp. Field-Programmable Custom Computing Machines (FCCM)*, 2001.

[34] C.R. Clark and D.E. Schimmel: "Efficient Reconfigurable Logic Circuit for Matching Complex Network Intrusion Detection Patterns," In: *Proc. 13th Int'l Conf. Field Programmable Logic and Applications* (FPL), 2003.

[35] J. Moscola et al., "Implementation of a Content-Scanning Module for an Internet Firewall". In: *Proc. IEEE Workshop FPGAs for Custom Computing Machines,* Apr. 2003

"The value of a college education is not the learning of many facts,
but the training of the mind to think"
Albert Einstein