Πολυτεχνειο Κρητης

Τμημα Ηλεκτρονικων Μηχανικων και Μηχανικων Υπολογιστων

Εραστηριο Μικροεπεξεργαστων και Υλικου

Διπλωματική Εργασία

Υλοποιηση σε Καρτα Γραφικων, Συστηματος για Αποκρυπτογραφηση Δεδομενων GSM

Νικόλαος Αλεξανδρής

Εξεταστική Επιτροπή

Ιωάννης Παπαευσταθίου (Επιβλέπων)	*Επίκουρος Καθηγητής*
Απόστολος Δόλλας	*Καθηγητής*
Διονύσιος Πνευματικάτος	*Καθηγητής*

Ιανουαριος 2016

Στόχοι

Πρωταρχικός σκοπός αυτής της διπλωματικής εργασίας, είναι να εξερευνήσουμε το πεδίο του παράλληλου προγραμματισμού, με έμφαση τη χρήση σύγχρονων καρτών γραφικών και, συγκεκριμένα, της αρχιτεκτονικής CUDA των καρτών γραφικών της NVIDIA. Ζητούμενο είναι, πέραν της εξοικείωσής μας με τη γενικότερη φιλοσοφία του παράλληλου προγραμματισμού, να κατανοήσουμε το μοντέλο προγραμματισμού της CUDA και να εντρυφήσουμε στις τεχνικές βελτιστοποίησης για την αύξηση της απόδοσης μιας παράλληλης εφαρμογής.

Ως «όχημα» προς αυτή την κατεύθυνση, επιλέγεται η υλοποίηση μιας εφαρμογής κρυπτανάλυσης για τον κρυπτογραφικό αλγόριθμο A5/3 που χρησιμοποιείται στο δίκτυο κινητής τηλεφωνίας GSM γενεάς 2.5G. Επιδίωξή μας αποτελεί η μεταφορά του αλγορίθμου σε κώδικα για παράλληλη εκτέλεση σε κάρτα γραφικών της αρχιτεκτονικής CUDA, η ανάπτυξη των απαιτούμενων λειτουργιών για την διεξαγωγή μιας κρυπτογραφικής επίθεσης σε επίπεδο προσομοίωσης και η βελτιστοποίηση της εφαρμογής για όσο δυνατών ταχύτερους χρόνους εκτέλεσης.

Περίληψη

Στην σύγχρονη ψηφιακή εποχή, η **Κρυπτογραφία** είναι απαραίτητη και χρησιμοποιείται εκτεταμένα σε όλο το φάσμα των ψηφιακών μέσων και εφαρμογών. Το ηλεκτρονικό εμπόριο, οι ψηφιακές επικοινωνίες, η κινητή τηλεφωνία, η ασφάλεια δεδομένων αλλά και προσωπικών διαπιστευτηρίων του χρήστη, καθώς και η ταυτοποίησή του, είναι μερικές από αυτές και υπάρχουν και άλλες πολλές ακόμη. Στην κινητή τηλεφωνία συγκεκριμένα, από τις πιο βασικές εφαρμογές της, είναι η ταυτοποίηση του χρήστη ως έγκυρο μέλος του δικτύου και η διασφάλιση της επικοινωνίας του με άλλους χρήστες, μέσω κρυπτογράφησης (μετατροπή σε μη καταληπτή από τρίτους μορφή) των πληροφοριών που ανταλλάσσει με αυτούς (φωνή, γραπτά μηνύματα, δεδομένα ίντερνετ).

Η **κρυπτανάλυση** από την άλλη, κάνει ακριβώς το αντίθετο. Σκοπός της είναι να δοκιμάζει να παραβιάσει τα κρυπτογραφικά συστήματα ασφαλείας, με σκοπό τον έλεγχο της ανθεκτικότητάς τους. Με αυτόν τον τρόπο μπορούν και αναπτύσσονται εργαλεία με τα οποία μπορούμε να αξιολογούμε τα κρυπτογραφικά συστήματα, ως προς το κατά πόσο αυτά είναι διαβλητά και να εκτιμούμε αν αυτά χρίζουν βελτίωσης ή αντικατάστασης.

Δύο απλοϊκές μέθοδοι κρυπτανάλυσης που μπορούν εν δυνάμει να λύσουν ένα κρυπτογραφικό πρόβλημα, είναι η **επίθεση ωμής βίας** (**brute-force attack**) και οι **πίνακες αναζήτησης** (**lookup table**). Κατά την επίθεση ωμής βίας, ο κρυπταναλυτής κάνει μια εξοντωτική αναζήτηση όλων των πιθανών κλειδιών κρυπτογράφησης, του στοιχείου δηλαδή που ελέγχει ένα σύστημα κρυπτογράφησης, προκειμένου να αποκρυπτογραφήσει ένα κείμενο. Πρόκειται όμως για μια μέθοδο που δεν είναι πρακτικά εφαρμόσιμη, καθότι έχει υψηλή υπολογιστική πολυπλοκότητα, λόγω του δραματικά μεγάλου χώρου των πιθανών κλειδιών. Αντίθετα, στην περίπτωση των πινάκων αναζήτησης, ο κρυπταναλυτής κρυπτογραφεί ένα προεπιλεγμένο κείμενο με όλους τους πιθανούς συνδυασμούς του κλειδιού, και αποθηκεύει σε έναν πίνακα τα ζεύγη κλείδιου κρυπτογράφησης και κρυπτογραφημένου κειμένου. Με αυτόν τον πίνακα, μπορεί μελλοντικά να ανακτήσει το κλειδί που αποκρυπτογραφεί ένα κρυπτογραφημένο κείμενο, σε ελάχιστο χρόνο. Η δεύτερη αυτή προσέγγιση, απαιτεί τεράστιο χώρο για την αποθήκευση των πινάκων, πράγμα που ανεβάζει κατά πολύ το κόστος και έτσι, ούτε αυτή η μέθοδος αποτελεί ένα ρεαλιστικό σενάριο.

Σε μια μελέτη του 1980 [1], ο M. Hellmann προτείνει μια τεχνική **ανταλλαγής χρόνου-μνήμης** (**time-memory trade-off**). Την ονομάζει «πίνακες Hellmann» και είναι ικανή να μειώσει σε μεγάλο ποσοστό τον χώρο που απαιτείται για την αποθήκευση πινάκων αναζήτησης. Το αντάλλαγμα είναι, η φάση της αποκρυπτογράφησης να κοστίσει λίγο παραπάνω σε χρόνο, αλλά εντός ρεαλιστικού πλαισίου.

Το 2003, ο P. Oechslin σε μια δημοσίευσή του [2], προτείνει μια βελτίωση στην τεχνική του M. Hellmann που την ονομάζει **Πίνακες Ουράνιου Τόξου** (**Rainbow Tables**). Με αυτή βελτιώνεται η πιθανότητα επιτυχίας ανεύρεσης κλειδιού κατά την διαδικασία της αποκρυπτογράφησης και ελαχιστοποιούνται κάποια προβλήματα της τεχνικής του M. Hellman.

Στο δίκτυο κινητής τηλεφωνίας GSM γενεάς 2.5G, για την ασφαλή επικοινωνία των χρηστών, χρησιμοποιείται ο κρυπτογραφικός αλγόριθμος **A5/3**, ο οποίος ως βασικό δομικό στοιχείο έχει τον κωδικοποιητή τμημάτων (Block Cipher) **KASUMI**. Ο KASUMI επιλέχθηκε ως βασικό δομικό στοιχείο για την ανάπτυξη των αλγορίθμων **εμπιστευτικότητας (Confidentiality - f8)** και **ακεραιότητας (Integrity - f9)** του συστήματος ασφαλείας για το δίκτυο κινητής τηλεφωνίας 3G (UMTS) και υιοθετήθηκε για χρήση και στον αλγόριθμο A5/3, όταν ο προκάτοχός του A5/1 παραβιάστηκε και κρίθηκε ως μη ενδεδειγμένος προς χρήση.

Σε αυτή τη διπλωματική εργασία, υλοποιούμε μια εφαρμογή, ικανή να υλοποιήσει Πίνακες Ουράνιου Τόξου για χρήση αποκρυπτογράφησης δεδομένων που έχουν κρυπτογραφηθεί με τον αλγόριθμο A5/3. Προκειμένου να επιταχύνουμε την διαδικασία κατασκευής των πινάκων, αλλά και αυτή της αναζήτησης εντός των πινάκων, εκμεταλλευόμαστε την αυξημένη υπολογιστική ισχύ των σύγχρονων καρτών γραφικών, οι οποίες δύνανται να χρησιμοποιηθούν για παράλληλο προγραμματισμό γενικού σκοπού. Συγκεκριμένα, η εφαρμογή μας είναι σχεδιασμένη για να εκτελείται σε κάρτες γραφικών της **NVIDIA** που είναι σχεδιασμένες με βάση την αρχιτεκτονική **CUDA** και υποστηρίζουν το συγκεκριμένο προγραμματιστικό μοντέλο.

Abstract

In this work we designed an application, which uses the time/memory tradeoff technique to create Rainbow Tables. These tables are used to decrypt data of the 2.5G generation of the GSM mobile communication network, encrypted with the A5/3 algorithm. The A5/3 is a stream cipher which uses the KASUMI block cipher in an Output-Feedback-like mode of operation. The application is designed using the CUDA Programming Model, for parallel execution on a modern NVIDIA GPU, build with the CUDA architecture which offers support for General-Purpose GPU-Programming (GP-GPU Programming). Our main goal was to get accustomed with the general method of designing applications to run on parallel, but also, to harvest the computational power of the CUDA architecture and the abilities of the CUDA Programming Model, in order to optimize our application for the maximum possible level of parallelization and execution speed. For the sake of producing fairly comparable results, we developed a second application to be executed on CPUs only, using the OpenMP model for parallel execution on modern multicore CPUs. This second application is used as a measure of comparison.

Ευχαριστίες

Είθισται, εργασίες όπως η συγκεκριμένη να περιέχουν μία ενότητα ευχαριστιών. Δεν θα αποτελέσω εξαίρεση σε αυτόν τον κανόνα. Θα αποτελέσω, μάλλον, χαρακτηριστικό του παράδειγμα.

Πρωτίστως θα ήθελα να ευχαριστήσω τον επιβλέπων καθηγητή μου Ιωάννη Παπαευσταθίου, Επίκουρο Καθηγητή του τμήματος Ηλεκτρονικών Μηχανικών και Μηχανικών Υπολογιστών του Πολυτεχνείου Κρήτης.

Επίσης ευχαριστώ τον Πέτρο, τον Σαράντο, τον Κώστα, τον Βασίλη και τον άλλο Βασίλη, τον Μιχάλη και τον άλλο Μιχάλη και πολλούς άλλους ιδιοκτήτες χώρων αναψυχής, στους οποίους ζήσαμε στιγμές χαλάρωσης, τέρψης, έμπνευσης, συζητήσεων, ζύμωσης, κοινωνικοποίησης και ευφορίας μετά τη συνοδεία (οινο-)πνεύματος, στα διαλείμματα κατά την εκπόνηση αυτής της εργασίας.

Ιδιαίτερα ευχαριστώ όλους τους φίλους μου στα Χανιά και όχι μόνο, που δεν έπαψαν να πιστεύουν σε μένα και να με ενθαρρύνουν μέχρι τελευταία στιγμή, για να αποτελέσουν το καλύτερο ακροατήριο κατά τη διάρκεια της παρουσίασης της Διπλωματικής μου εργασίας. Μια ιδιαίτερη μνεία στον Φώτη για την άριστη τεχνική του στην ανάφλεξη κάρβουνου, στον Χρήστο που έψηνε σουβλάκια για τέσσερις ώρες, αλλά και όλους τους υπόλοιπους που βοήθησαν να να πάρει σάρκα και οστά το εγχείρημα μιας χαρούμενης γιορτής. Να μην ξεχάσω να ευχαριστήσω την Πολίν, για το καταπληκτικό της σχέδιο το οποίο κοσμούσε την αμφίεσή μου κατά την παρουσίαση.

Τέλος, θα ήθελα να ευχαριστήσω τον κύριο Κασετόφωνο, του οποίου οι κασέτες με συνόδευαν μουσικά κατά τις ώρες εργασίας μου, και τον Μιχάλη που μου δάνεισε την ατάκα έναρξης των ευχαριστιών.

Περιεχόμενα

Κατάλογος Σχημάτων

Κατάλογος Πινάκων

Στον πατέρα μου, για την ανοχή του
Στην Μητέρα μου, για την υπομονή της
Στην αδερφή μου, που πάντα έβαζε πλάτη

Κεφάλαιο 1

Κρυπτολογία

Η πρακτική της συγγραφής κειμένων ή/και μηνυμάτων σε ακατάληπτη μορφή (γραφή εν κρυπτό), σύμφωνα με ιστορικές πηγές, φαίνεται να βρίσκει εφαρμογή εδώ και σχεδόν 2500 χρόνια [3, σελ. 4]. Ήδη από την εποχή που οι άνθρωποι έπαψαν να ζουν σε σπήλαια και να οργανώνονται σε φυλές, αρχικά, αλλά και σε κοινωνίες αργότερα, η ανάγκη για απόκρυψη μηνυμάτων και ασφαλή επικοινωνία είναι υπαρκτή. Συνήθως, κάθε τόσο που η ανθρωπότητα εδραίωνε κάποιο επίπεδο πολιτισμού, η ανάγκη και οι μέθοδοι για επικοινωνία εν κρυπτό εμφανίζονταν σχεδόν αυθόρμητα, όπως και η γραφή και η ανάγνωση. Κυβερνήσεις, στρατοί, βασιλείς και διπλωμάτες, κατάσκοποι, εραστές, όλοι είχαν μυστικά και ήταν αποφασισμένοι να τα διαφυλάξουν.

1.1 Εισαγωγή στην Κρυπτολογία

Ο όρος Κρυπτολογία (**Cryptology**) αναφέρεται από κοινού στους επιστημονικούς κλάδους της κρυπτογραφίας (**Cryptography**) και της κρυπτανάλυσης (**Cryptanalysis**), αν και συχνά στην βιβλιογραφία, ο όρος χρησιμοποιείται ταυτόσημα με αυτόν της κρυπτογραφίας.

Κρυπτογραφία

Η **κρυπτογραφία** είναι η επιστήμη που ασχολείται με κρυπτογράφηση (**encryption**), δηλαδή με μεθόδους μετατροπής ενός αρχικού κειμένου (**plaintext**) σε ένα άλλο με μη καταληπτή μορφή, το οποίο από εδώ και στο εξής θα ονομάζουμε κρυπτογραφημένο κείμενο (**ciphertext**). Η αντίστροφη διαδικασία, δηλαδή η μετατροπή ενός μη καταληπτού κειμένου στη αρχική, αναγνώσιμη μορφή του, ονομάζεται αποκρυπτογράφηση (**decryption**). Καί οι δυο αυτές διαδικασίες, συνήθως, ελέγχονται από ένα κρυπτογραφικό κλειδί (**cryptographic Key or Key**) [4].

Για την επίτευξη των παραπάνω στόχων, η κρυπτογραφία μελετά και αναπτύσσει μεθόδους κρυπτογράφησης ή κωδικοποιητές (**Encryption Methods or Ciphers**) και κρυπτογραφικά συστήματα (**Cryptographic Systems**). Αν και, όπως θα δούμε στην παράγραφο 1.3, στα πρώιμα χρόνια της, η κρυπτογραφία περιοριζόταν στην μετατροπή λεκτικών κειμένων σε ακατάληπτη μορφή και μόνο σε λεξικογραφικό επίπεδο, στην σύγχρονη ψηφιακή εποχή, η εφαρμογή της εκτείνεται σε περαιτέρω

ανάγκες όπως εμπιστευτικότητα (**confidentiality**), ακεραιότητα (**integrity**), επικύρωση (**authentication**) και άλλες. Έχει αποδεσμευτεί από τον λεξικογραφικό περιορισμό και βασίζεται πλέον σε ντετερμινιστικούς μαθηματικούς κλάδους όπως η Θεωρία της Πληροφορίας, η Θεωρία Αριθμών, η Στατιστική, η Υπολογιστική Πολυπλοκότητα και άλλες, ενώ χειρίζεται τα δεδομένα σε δυαδικό επίπεδο. Στην καθημερινή μας χρήση, βρίσκει εφαρμογή σε πάρα πολλούς τομείς (δίκτυα υπολογιστών, ηλεκτρονικό εμπόριο, κινητές τηλεπικοινωνίες, ασύρματα δίκτυα κ.α.), αλλά δρα σε ένα χαμηλό επίπεδο των συστημάτων και έτσι, η ύπαρξή της δεν είναι άμεσα αντιληπτή από τον χρήστη, γι΄ αυτό και συχνά λέγεται ότι έχει περάσει σε ένα αόρατο επίπεδο.

Κρυπτανάλυση

Η **κρυπτανάλυση** εν αντιθέσει, είναι η επιστήμη που ασχολείται με την μελέτη και ανάπτυξη τεχνικών, που έχουν ως σκοπό την παραβίαση των κρυπτογραφικών μεθόδων και συστημάτων [4].

Η εφαρμογή μιας τεχνικής παραβίασης ενός κρυπτογραφικού συστήματος ή μιας μεθόδου, ονομάζεται κρυπτογραφική επίθεση ή απλά επίθεση (**cryptografic attack or attack**). Σκοπός μιας επίθεσης είναι να καταφέρει να αποκρυπτογραφήσει ένα κρυπτογραφημένο κείμενο ή, ακόμα καλύτερα, να καταφέρει να ανακτήσει το κλειδί που ελέγχει την κρυπτογράφηση, αποκτώντας έτσι τη δυνατότητα να αποκρυπτογραφεί όλα τα κείμενα που έχουν κρυπτογραφηθεί υπό τον έλεγχο του ίδιου κλειδιού. Κατ΄ επέκταση, η κρυπτανάλυση βοηθά στο να αναπτύσσονται τεχνικές και εργαλεία, με τη βοήθεια των οποίων δύναται να ελέγχεται η ανθεκτικότητα των κρυπτογραφικών συστημάτων ανά τους καιρούς και όποτε η ανθεκτικότητα κρίνεται χαμηλή, να ωθεί έτσι τη βελτίωση αυτών.

Έχουν αναπτυχθεί διάφορες τεχνικές επιθέσεων, οι οποίες κατηγοριοποιούνται ανάλογα με τη φύση της τεχνικής, αλλά και τη φύση των δεδομένων που δύναται να έχει στην διάθεσή του ο κρυπταναλυτής. Μια σύντομη αναφορά σε κάποιες από αυτές και στην κατηγοριοποίησή τους γίνεται στην ενότητα 1.5.

1.2 Σύντομη Ιστορική Αναδρομή

Σε αυτή την ενότητα θα αναφερθούμε στις φάσεις από τις οποίες έχει περάσει η κρυπτογραφία ως πρακτική και πως έμελε να εξελιχθεί στην σημερινή της μορφή. Μια αναλυτική ιστορική αναδρομή είναι πέρα από τους σκοπούς αυτής της παραγράφου και έτσι οι αναφορές μας θα είναι πολύ σύντομες και περιληπτικές. Για τον αναγνώστη που ενδιαφέρεται να μάθει περισσότερες λεπτομέρειες, τον προτρέπουμε να ανατρέξει σε πηγές, όπως για παράδειγμα το βιβλίο του John F. Dooley: "A Brief History of Cryptology and Cryptographic Algorithms" [3].

Αν και υπάρχουν ιστορικές αναφορές που μεταφέρουν την χρήση της κρυπτογραφίας ακόμα και στην προ Χριστού εποχή, σαν επιστήμη, με την έννοια των δημοσιευμένων μελετών της, η κρυπτογραφία θεωρείται μια πολύ νεαρή επιστήμη. Αυτό οφείλεται στο γεγονός ότι μέχρι τις αρχές του 1970 περίπου, ή αποτελούσε μια τέχνη που την κατείχαν πολύ λίγοι, ή αποτελούσε αντικείμενο έρευνας που γινόταν κάτω

από άκρα μυστικότητα και υπό τον έλεγχο μυστικών και στρατιωτικών υπηρεσιών και υπηρεσιών ασφαλείας.

Οι πρώτες αναφορές για χρήση κρυπτογραφίας αφορούν κυρίως τους Έλληνες και τους Ρωμαίους και εφαρμόζεται σε καθαρά λεξικογραφικό επίπεδο για την ασφαλή ανταλλαγή γραπτών μηνυμάτων. Με την πτώση βέβαια της Ρωμαϊκής Αυτοκρατορίας, έρχεται και ο σκοταδισμός για την κρυπτογραφία στον Δυτικό κόσμο. Συνεχίζει όμως κατά τη διάρκεια του μεσαίωνα, να ανθίζει στον Αραβικό κόσμο, μαζί με άλλες τέχνες και επιστήμες. Εκεί αναπτύσσονται για πρώτη φορά οι πρώιμες μέθοδοι κρυπτανάλυσης.

Καθώς μπαίνουμε στην περίοδο της Αναγέννησης, αρχίζει και πάλι να εφαρμόζεται η κρυπτογραφία και στον Δυτικό κόσμο, αποτελώντας απαραίτητο εργαλείο για διπλωμάτες και κυβερνητικούς εκπροσώπους. Συνεχίζει να εξελίσσεται, όντας απαραίτητη για την ασφαλή ανταλλαγή μηνυμάτων κατά την διάρκεια των πολέμων που ακολουθούν. Κατά την περίοδο αυτή, οι περισσότερες ανεπτυγμένες χώρες, διατηρούν υπηρεσίες που απασχολούν μαθηματικούς και ειδικούς σε θέματα κρυπτογραφίας, με σκοπό την ανάπτυξη μεθόδων κρυπτογράφησης αλλά και κρυπτανάλυσης μηνυμάτων, που έχουν υποκλαπεί από τον εχθρό . Η γνώση των πληροφοριών που ανταλλάσσουν μεταξύ τους τα στρατεύματα του εχθρού, πολύ συχνά έχει καθορίσει την έκβαση ενός πολέμου στην ιστορία.

Με το τέλος του 2ου Παγκοσμίου Πολέμου σηματοδοτείται κατά κάποιο τρόπο το τέλος της περιόδου της κλασικής κρυπτογραφίας. Ήδη οι μέθοδοι κρυπτανάλυσης που χρησιμοποιούνταν βασίζονταν πλέον κυρίως σε μαθηματικά και όχι σε κλασικές λεξικογραφικές μεθόδους. Αλλά την πραγματική βάση για την μετάβαση από την κλασική εποχή στην ψηφιακή εποχή των υπολογιστών, την έθεσε ο Claude Elwood Shannon σε μια μελέτη που δημοσιεύτηκε το 1949, με τίτλο Communication Theory of Secrecy Systems [5]. Σε αυτή τη μελέτη ο Shannon τοποθετεί το ζήτημα της κρυπτολογίας σε μια καθαρά μαθηματική βάση, αναπτύσσει την απαραίτητη ορολογία και παρέχει έτσι το θεωρητικό υπόβαθρο για την εξέλιξη των κρυπτογραφικών συστημάτων στην σημερινή τους μορφή.

Από εκεί και έπειτα, η κρυπτογραφία σταματάει σιγά σιγά να αποτελεί αντικείμενο έρευνας για καθαρά στρατιωτικούς σκοπούς ή εθνικής ασφάλειας, ελεγχόμενη από τους αντίστοιχους φορείς και εφαρμοζόμενη με άκρα μυστικότητα. Αρχίζει να αποτελεί αντικείμενο μελέτης στον ακαδημαϊκό χώρο, αλλά και εμπορεύσιμο προϊόν για επιχειρήσεις που έχουν την ανάγκη να διασφαλίσουν τα ψηφιακά τους δεδομένα. Από τις αρχές του 1970 και μετά, αρχίζουν πλέον να δημοσιεύονται μελέτες και αποτελέσματα ερευνών από ακαδημαϊκούς και άλλους φορείς.

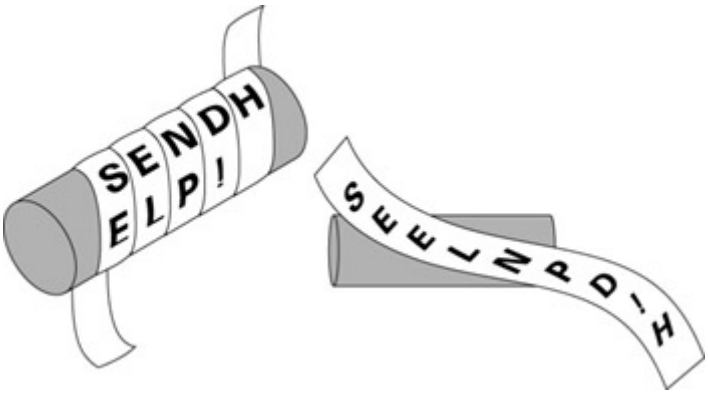
1.3 Κλασικές Τεχνικές Κρυπτογράφησης

Στην ενότητα αυτή θα παρουσιάσουμε τις κλασικές τεχνικές της κρυπτογραφίας, από τις πρώτες ιστορικές αναφορές της χρήσης της μέχρι σήμερα και θα δούμε πως αυτές εξελίχθηκαν, για να φτάσουμε στην σύγχρονη ψηφιακή εποχή.

Κωδικοποιητές Αναδιάταξης (Transposition Ciphers)

Οι κωδικοποιητές αναδιάταξης, όπως υποδηλώνει το όνομά τους, στηρίζονται στην αναδιάταξη των γραμμάτων ενός κειμένου με βάση κάποια συγκεκριμένη αρχή, η οποία πρέπει να είναι γνωστή από όλα τα μέλη που τη χρησιμοποιούν.

Πρόκειται ίσως για την πιο παλιά μέθοδο για την οποία υπάρχουν ιστορικές αναφορές. Την χρησιμοποιούσαν οι Σπαρτιάτες για να ανταλλάσσουν μηνύματα στρατιωτικού σκοπού. Χρησιμοποιούσαν μια σκυτάλη πάνω στην οποία θα τύλιγαν μια λωρίδα από πάπυρο και έγραφαν πάνω σε αυτή το μήνυμα. Μετά ο πάπυρος ξετυλίγονταν από τη σκυτάλη και αποστέλλονταν στον παραλήπτη. Χωρίς τη σκυτάλη, τα γράμματα πάνω στον πάπυρο δεν έβγαζαν κανένα νόημα, καθότι δεν ήταν στη σωστή διάταξη. Ο παραλήπτης θα τύλιγε τον πάπυρο σε μια άλλη σκυτάλη ίδιου διαμετρήματος, ώστε να μπορέσει να διαβάσει το μήνυμα (βλέπε εικόνα 1.1).



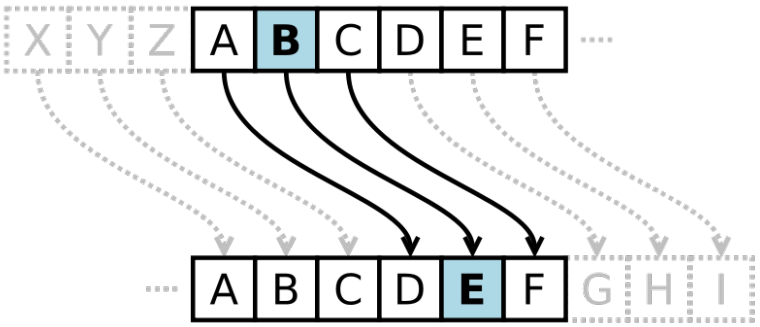
Σχήμα 1.1: Η μέθοδος της Σκυτάλης

Κωδικοποιητές Αντικατάστασης (substitution ciphers)

Σε αυτή την κατηγορία κωδικοποιητών, χρησιμοποιούνταν ένα μοτίβο αντικατάστασης γραμμάτων, το οποίο θα μπορούσε να προκύπτει από το αλφάβητο με κάποια ολίσθηση ως προς τη σειρά των γραμμάτων, ή από μια τυχαία διάταξή τους. Για να κρυπτογραφηθεί το μήνυμα, κάθε γράμμα της αλφαβήτας αντικαθίσταται με το γράμμα του μοτίβου που βρίσκεται στην ίδια θέση με αυτό.

Ιστορικό παράδειγμα ενός κωδικοποιητή αντικατάστασης αποτελεί ο κωδικοποιητής του Καίσαρα (Caesar Cipher), τον οποίο χρησιμοποιούσε ο Ιούλιος Καίσαρας για να ανταλλάσσει μηνύματα με τους αξιωματικούς του, από όπου πήρε και το όνομά του. Ο Ιούλιος Καίσαρας τον χρησιμοποιούσε με μια ολίσθηση τριών θέσεων προς τα αριστερά στο αλφάβητο, όπως φαίνεται στην εικόνα 1.2. Έτσι θα είχαμε το παρακάτω παράδειγμα κρυπτογράφησης με χρήση του κωδικοποιητή του Καίσαρα.

plaintext:	FINALLYIAMGOINGTOGRADUATE
alphabet:	ABCDEFGHIJKLMNOPQRSTUVWXYZ
shift-3:	DEFGHIJKLMNOPQRSTUVWXYZABC
ciphertext:	ILQDOOBLDPJRLQJWRJUDGXDWH



Σχήμα 1.2: Ο Κωδικοποιητής του Καίσαρα

Πρόκειται για έναν κωδικοποιητή, ο οποίος θεωρείται ότι είναι σχετικά εύκολο να αποκρυπτογραφηθεί. Το βασικό εργαλείο κρυπτανάλυσης για τέτοιου τύπου κωδικοποιητές, βασίζεται στην ανάλυση της συχνότητας εμφάνισης των γραμμάτων σε ένα κείμενο και διατυπώθηκε για πρώτη φορά από τον Al Kindi [6], έναν Άραβα πολυμαθή, τον 9ο αιώνα. Με βάση αυτή τη μέθοδο, κάποια γράμματα, στατιστικά, έχουν συχνότερη εμφάνιση σε ένα κείμενο από ότι κάποια άλλα. Κάνοντας μια στατιστική ανάλυση συχνότητας εμφάνισης γραμμάτων και στο κρυπτογραφημένο κείμενο, μπορεί κανείς να αρχίσει να μαντεύει τα αρχικά γράμματα και έτσι να αποκρυπτογραφεί τμηματικά το κείμενο. [3, παρ. 2.3]

Κωδικοποιητές, όπως αυτός του Καίσαρα, ονομάζονται **μονοαλφαβητικοί** κωδικοποιητές (**monoalphabetic ciphers**), διότι χρησιμοποιούν ένα «πειραγμένο» αλφάβητο για την κρυπτογράφηση. Αυτή η πρακτική διατηρούσε τις στατιστικές ιδιότητες του κειμένου και έτσι το έκανε ευάλωτο σε μεθόδους αποκωδικοποίησης, όπως αυτή του Al Kindi. Η λύση σε αυτό το πρόβλημα ήρθε τον 15ο αιώνα από τον Leon Battista Alberti, με την επινόηση των **πολυαλφαβητικών κωδικοποιητών** (**polyalphabetic ciphers**). Με βάση τη μέθοδο του Leon Battista Alberti, η λύση ήταν να χρησιμοποιούνται διαδοχικά περισσότερα από ένα «πειραγμένα» αλφάβητα για την κρυπτογράφηση. Με αυτόν τον τρόπο επιτυγχάνεται μια μεταβολή της κατανομής της συχνότητας εμφάνισης των γραμμάτων, με τέτοιο τρόπο ώστε να μοιάζει ότι όλα τα γράμματα έχουν περίπου την ίδια συχνότητα εμφάνισης. [3, παρ. 3.4]

Παρόλο που η διαδικασία της κρυπτανάλυσης γινόταν πιο δύσκολη με τη χρήση πολυαλφαβητικών κωδικοποιητών, δεν ήταν ανέφικτη. Με το να αρχίσουν να γίνονται και οι πολυαλφαβητικοί κωδικοποιητές ευάλωτοι σε μεθόδους κρυπτανάλυσης, οι μέθοδοι κρυπτογράφησης οδηγήθηκαν προς το επόμενο βήμα της ωρίμανσής τους, τους product ciphers.

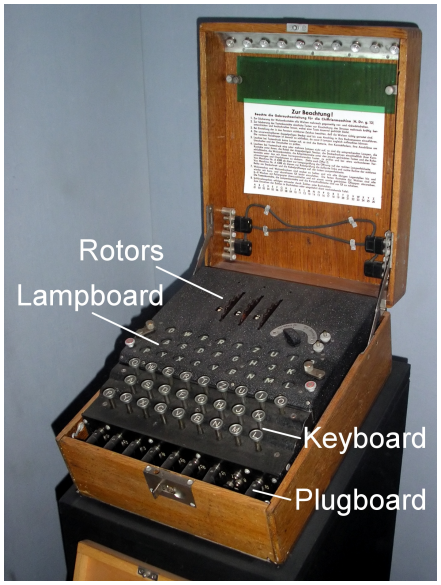
Product Ciphers

Οι **Product Ciphers** είναι κωδικοποιητές που συνδυάζουν τις τεχνικές της αντικατάστασης καί της αναδιάταξης. Ιστορικό παράδειγμα ενός τέτοιου κωδικοποιητή είναι αυτό του ADFGVX που χρησιμοποιήθηκε από τους Γερμανούς κατά τον 1ο Παγκόσμιο Πόλεμο [3, παρ. 5.5]. Οι Product Ciphers θεωρούνται η γέφυρα μετάβασης από τις κλασικές στις σύγχρονες μεθόδους κρυπτογράφησης, καθότι οι περισσότε-

ρες από τις σύγχρονες τεχνικές, χρησιμοποιούν τον συνδυασμό αντικατάστασης και αναδιάταξης.

Ρότορες (Rotor Machines)

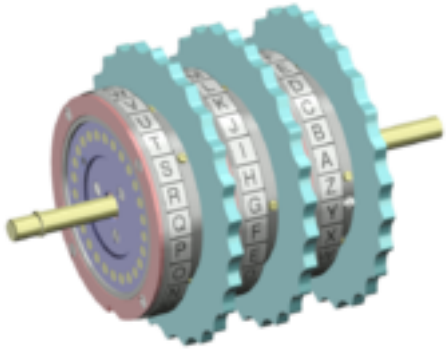
Με την ανάπτυξη του ασυρμάτου ήταν πλέον εφικτό να μεταδίδεται καθημερινά ένας μεγάλος όγκος κρυπτογραφημένων μηνυμάτων. Αυτό κατέστησε επιτακτική την ανάγκη για αυξημένη ασφάλεια, ταχύτητα αλλά και ακρίβεια, τόσο για την κρυπτογράφηση αλλά και την αποκρυπτογράφηση. Η ανάγκη αυτή, σήμανε κατά τη διάρκεια του μεσοπολέμου, την ανάπτυξη ηλεκτρομηχανικών συσκευών για χρήση στην κρυπτογράφηση και αποκρυπτογράφηση, συσκευών γνωστές και ως **Ρότορες** (**Rotor Machines**) [3, κεφ. 8]. Ιστορικό παράδειγμα ενός ρότορα αποτελεί η συσκευή **Enigma** που χρησιμοποιούσαν οι Γερμανοί κατά τον 2ο Παγκόσμιο Πόλεμο (εικόνα 1.3).



Σχήμα 1.3: Η μηχανή (Ρότορας) Enigma

Οι ρότορες ήταν συσκευές οι οποίες αποτελούνταν από συστοιχίες δυο ή και περισσότερων δίσκων, οι οποίοι φέραν στην επιφάνειά τους εγγεγραμμένο το αλφάβητο και στο εσωτερικό της κάθε πλευράς τους, ηλεκτρικές επαφές (βλέπε εικόνα 1.4). Αρχικά στην μηχανή Enigma χρησιμοποιούνταν τρεις δίσκοι, ενώ στη συνέχεια, σε μια βελτιωμένη έκδοση χρησιμοποιήθηκαν τέσσερις. Η συγγραφή των κειμένων γινόταν με τη χρήση ενός κλασικού πληκτρολογίου γραφομηχανής, με το οποίο ήταν εξοπλισμένοι οι ρότορες. Με το πάτημα κάθε κουμπιού, ο πρώτος δίσκος περιστρέφονταν κατά μία θέση. Κάθε φορά που ο πρώτος δίσκος θα έκανε μια πλήρη περιστροφή, ο δεύτερος δίσκος θα περιστρεφόταν κατά μία θέση και ου το κάθε εξής, μέχρις ότου όλοι οι δίσκοι να εκτελέσουν μια πλήρη περιστροφή. Με αυτόν τον τρόπο έχουμε ουσιαστικά ένα πολυαλφαβητικό σύστημα, με ικανά μεγάλη περίοδο, ανάλογα με τον αριθμό των δίσκων που χρησιμοποιούνται. Η σειρά χρήσης των αλφάβητων κρυπτογράφησης, αρχίζει να επαναλαμβάνεται όταν όλοι οι δίσκοι του ρότορα εκτελέσουν

μια πλήρη περιστροφή. Άρα, αν μιλάμε για το λατινικό αλφάβητο που αποτελείτε από 26 γράμματα, ένας ρότορας με τρεις δίσκους μπορεί να παράξει 26 x 26 x 26 = 17.576 διαφορετικά, ανάμεικτα αλφάβητα κρυπτογράφησης, πριν αρχίσει να τα επαναλαμβάνει [3, παρ. 7.2].



Σχήμα 1.4: Διάταξη δίσκων Ρότορα

1.4 Σύγχρονες Μέθοδοι Κρυπτογράφησης

Η σύγχρονη κρυπτογραφία περιλαμβάνει γενικά, κρυπτογραφικούς αλγορίθμους (encryption algorithms), κωδικοποιητές (ciphers) και κρυπτογραφικά συστήματα ή πρωτόκολλα (cryptographic systems or protocols). Χωρίζεται σε δύο κατηγορίες, την **συμμετρική κρυπτογραφία** ή **κρυπτογραφία ιδιωτικού κλειδιού** (**Symetric or Private Key Crytography**) και την **ασύμμετρη** ή **κρυπτογραφία δημοσίου κλειδιού** (**Asymetric or Public Key Cryptography**).

Στην συμμετρική κρυπτογραφία, η διαδικασία της κρυπτογράφησης αλλά και της αποκρυπτογράφησης γίνεται υπό τον έλεγχο του ίδιου κλειδιού και συνήθως χρησιμοποιείται ο ίδιος αλγόριθμος καί στις δύο διαδικασίες. Αντίθετα, στην κρυπτογραφία δημοσίου κλειδιού χρησιμοποιείται ένα ζεύγος κλειδιών, όπου το ένα κλειδί χρησιμοποιείται στην κρυπτογράφηση, ενώ το άλλο στην αποκρυπτογράφηση και οι αλγόριθμοι κρυπτογράφησης και αποκρυπτογράφησης είναι κατά βάση μονόδρομες συναρτήσεις.

Θα αναφέρουμε στη συνέχεια κάποια παραδείγματα κωδικοποιητών για την κάθε κατηγορία, εξηγώντας πως λειτουργούν, περιοριζόμενοι σε αυτά που άπτονται άμεσα της συγκεκριμένης διπλωματικής εργασίας. Για τον αναγνώστη που επιθυμεί να ενημερωθεί για όλο το φάσμα της κρυπτογραφίας, προτείνουμε πηγές όπως τα βιβλία "Handbook of Applied Cryptography" [7] και "Cryptography and Data Security" [4], που καλύπτουν το αντικείμενο σε όλο το εύρος του.

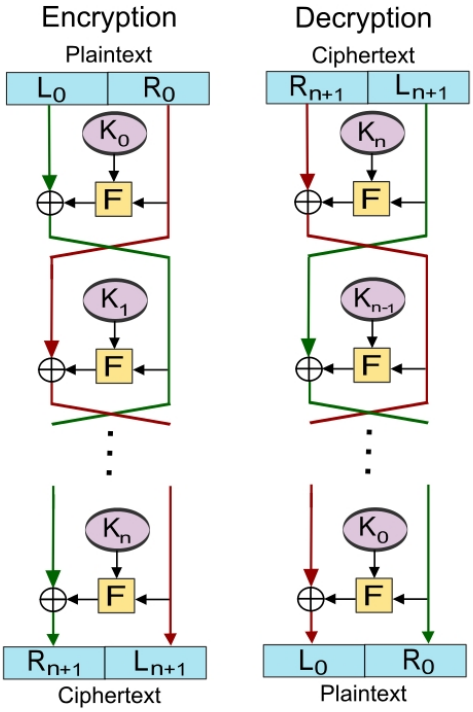
1.4.1 Συμμετρική Κρυπτογραφία (Symetric Cryptography)

Παραδείγματα συμμετρικών κρυπτογραφικών συστημάτων αποτελούν οι **κωδικοποιητές τμημάτων** (**block ciphers**) και οι **κωδικοποιητές ροής** (**stream-ciphers**).

Κωδικοποιητές Τμημάτων (Block Ciphers)

Οι κωδικοποιητές τμημάτων, όπως λέει και το όνομά τους, κρυπτογραφούν το αρχικό κείμενο κατά τμήματα σταθερού μεγέθους, για παράδειγμα 64 bit. Βασίζονται σε μια δομή που περιγράφτηκε για πρώτη φορά από τον Horst Feistel [8] και είναι γενικότερα γνωστή ως δίκτυο Feistel.

Σε ένα δίκτυο Feistel, τα δεδομένα εισόδου που αποτελούν ένα τμήμα (block) από bits, χωρίζονται ακριβώς στη μέση, σε ένα αριστερό και ένα δεξί υποτμήμα. Ο αλγόριθμός της κρυπτογράφησης ενεργεί σε κάθε υποτμήμα ανεξάρτητα, με χρήση μιας συνάρτησης **F**, η οποία εφαρμόζεται επαναληπτικά πολλές φορές. Κάθε τέτοια επανάληψη ονομάζεται **κύκλος (round)** και η συνάρτηση ονομάζεται **συνάρτηση κύκλου (round function)**. Η συνάρτηση F παραμετροποιείται με βάση μια σειρά από επιμέρους κλειδιά, ένα για κάθε κύκλο, τα οποία παράγονται από το κλειδί ελέγχου της κρυπτογράφησης, με βάση έναν **αλγόριθμο παραγωγής επιμέρους κλειδιών (key schedule)**. Με απλά λόγια, πρόκειται για μια δομή που εφαρμόζει την τεχνική της αναδιάταξης, με χρήση πολλαπλών αλφάβητων για την κρυπτογράφηση. Για την αποκρυπτογράφηση, εισάγεται το κρυπτογραφημένο κείμενο στον ίδιο αλγόριθμο, ενώ τα επιμέρους κλειδιά χρησιμοποιούνται με αντίστροφη σειρά. Στην εικόνα 1.5 βλέπουμε ένα τέτοιο δίκτυο, όπου τα K_0...K_n αποτελούν τα επιμέρους κλειδιά για τους εκάστοτε κύκλους 1...n.



Σχήμα 1.5: Δίκτυο Feistel [9]

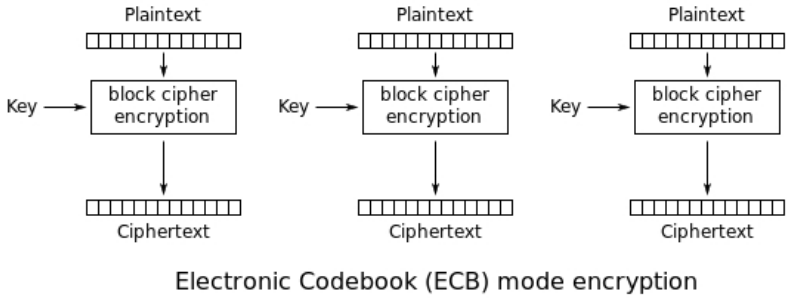
Στην πιο εξελιγμένη τους μορφή, οι κωδικοποιητές τμημάτων, συνδυάζουν την τεχνική της αναδιάταξης με αυτή της αντικατάστασης, με χρήση **πινάκων αντικατάστασης**, γνωστά και ως **S-boxes**. Λόγω αυτού, συνηθίζεται να ονομάζονται και

δίκτυα αντικατάστασης-αναδιάταξης (substitution-permutation networks). Παραδείγματα τέτοιων αποτελούν ο DES [10] και ο AES [11].

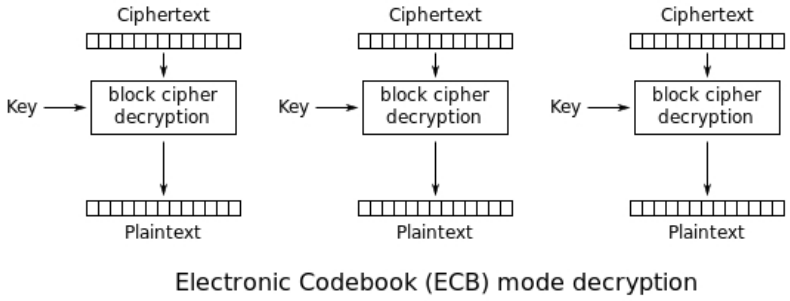
Τρόποι Χρήσης Κωδικοποιητών Τμημάτων (Modes of Operation)

Για την κρυπτογράφηση μεγαλύτερων μηνυμάτων, οι κωδικοποιητές τμημάτων χρησιμοποιούνται με διάφορους τρόπους διασύνδεσης. Οι πιο βασικοί από αυτούς είναι, ο Electronic Code Book (ECB) mode, ο Cipher Block Chaining (CBC) mode, ο Cipher Feedback (CFB) mode και ο Output Feedback (OFB) mode.

Κατά τη χρήση του τρόπου Electronic Code Book (ECB) mode, το μήνυμα προς κρυπτογράφηση χωρίζεται σε τμήματα, μεγέθους ίσου με αυτό της εισόδου του κωδικοποιητή και κάθε τμήμα κρυπτογραφείται ανεξάρτητα από το άλλο. Δεν συνίσταται προς χρήση, καθότι έχει το μειονέκτημα ότι όμοια τμήματα εισόδου παράγουν όμοια τμήματα κρυπτογραφημένου μηνύματος και αυτό τον κάνει ευάλωτο. Ένα διάγραμμα της λειτουργίας του Electronic Code Book (ECB) mode βλέπουμε στις εικόνες 1.6 και 1.7.



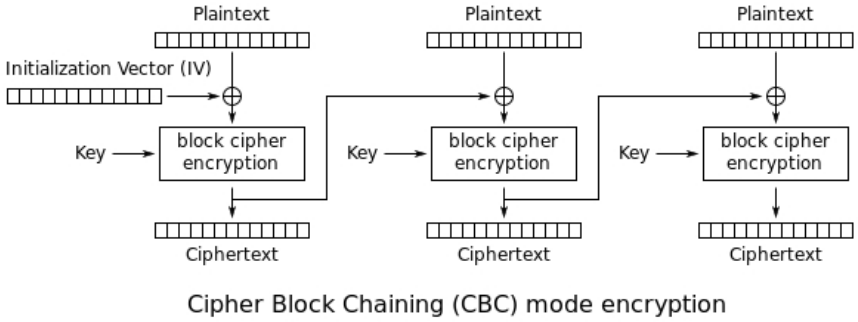
Σχήμα 1.6: Κρυπτογράφηση κατά ECB Mode [12]



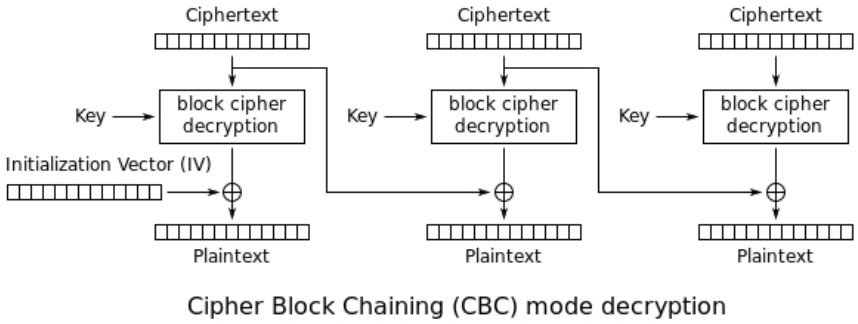
Σχήμα 1.7: Αποκρυπτογράφηση κατά ECB Mode [12]

Στον δε Cipher Block Chaining (CBC) mode τρόπο, στην αρχική είσοδο κάθε τμήματος, εφαρμόζεται μια πράξη XOR με το κρυπτογραφημένο μήνυμα του προηγούμενου τμήματος, πριν κρυπτογραφηθεί. Με αυτόν τον τρόπο, κάθε κρυπτογραφημένο τμήμα εξαρτάται από όλες τις αρχικές εισόδους που έχουν κρυπτογραφηθεί πριν από αυτό. Προκειμένου να είναι κάθε μήνυμα προς κρυπτογράφηση μοναδικό, στο πρώτο

τμήμα εφαρμόζεται ένα διάνυσμα αρχικοποίησης, όπως φαίνεται στην εικόνα 1.8. Η εικόνα 1.9 μας δείχνει την διαδικασία αποκρυπτογράφησης.

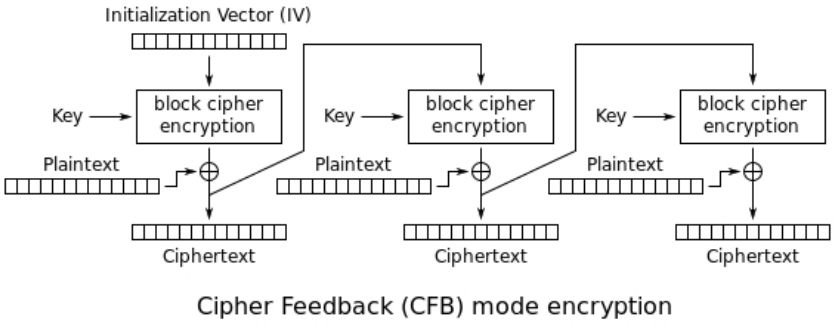


Σχήμα 1.8: Κρυπτογράφηση κατά CBC Mode [12]

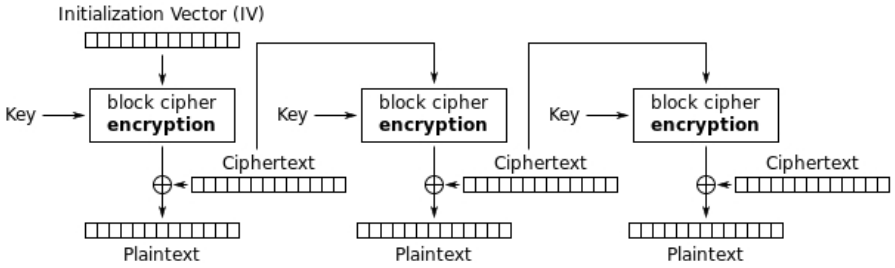


Σχήμα 1.9: Αποκρυπτογράφηση κατά CBC Mode [12]

Παρόμοια με τον Cipher Block Chaining (CBC) mode τρόπο, λειτουργεί και ο Cipher Feedback (CFB) mode τρόπος. Η διαφορά έγκειται στο ότι πρώτα κρυπτογραφείται το διάνυσμα αρχικοποίησης και μετά, στο αποτέλεσμα, εφαρμόζεται μια πράξη XOR με το αρχικό μήνυμα, το αποτέλεσμα της οποίας αποτελεί το κρυπτογραφημένο μήνυμα. Ως είσοδος σε κάθε επόμενο τμήμα, εφαρμόζεται το κρυπτογραφημένο μήνυμα του προηγούμενου τμήματος. Παρατηρούμε την διαφορά στην λειτουργία στις εικόνες 1.10 και 1.11.

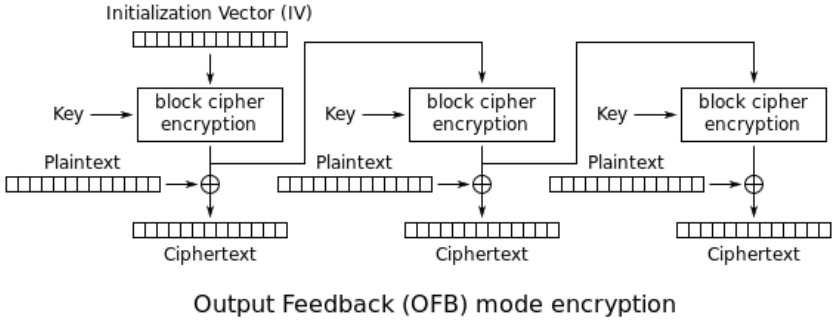


Σχήμα 1.10: Κρυπτογράφηση κατά CFB Mode [12]

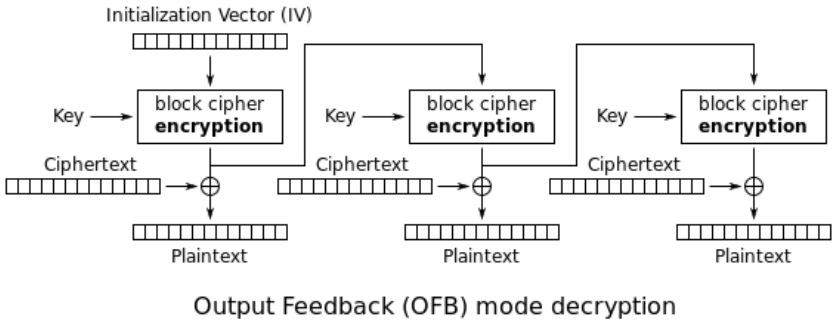


Σχήμα 1.11: Αποκρυπτογράφηση κατά CFB Mode [12]

Τέλος ο Output Feedback (OFB) mode τρόπος, παράγει τμήματα κλειδιών ροής (keystream blocks), τροφοδοτώντας ως είσοδο στο επόμενο τμήμα την έξοδο του προηγούμενου. Στο τέλος, εφαρμόζει μια πράξη XOR ανάμεσα στα τμήματα του αρχικού μηνύματος και στα αντίστοιχα τμήματα των κλειδιών ροής τους. Το αποτέλεσμα αποτελεί το κρυπτογραφημένο μήνυμα. Τα διαγράμματα λειτουργίας φαίνονται στις εικόνες 1.12 και 1.13 αντίστοιχα.



Σχήμα 1.12: Κρυπτογράφηση κατά OFB Mode [12]



Σχήμα 1.13: Αποκρυπτογράφηση κατά OFB Mode [12]

Κωδικοποιητές Ροής (Stream Ciphers)

Οι κωδικοποιητές ροής, σε αντίθεση με τους κωδικοποιητές τμημάτων, δεν κρυπτογραφούν το αρχικό κείμενο κατά τμήματα, αλλά έναν χαρακτήρα ή ένα bit τη φορά. Διακρίνονται σε δυο κατηγορίες, τους **συγχρονιζόμενους (Synchronous Stream Ciphers)** και τους **αυτο-συγχρονιζόμενους (Self-synchronous Stream Ciphers)**. [4, κεφ. 3]

Ένας συγχρονιζόμενος κωδικοποιητής ροής παράγει ένα **κλειδί ροής (keystream)** ανεξάρτητα από το μήνυμα προς κρυπτογράφηση. Σε έναν αυτό-συγχρονιζόμενο κωδικοποιητή ροής, κάθε χαρακτήρας του κλειδιού ροής, παράγεται από έναν σταθερό αριθμό χαρακτήρων κρυπτογραφημένου μηνύματος, που έχει παραχθεί νωρίτερα.

Οι κωδικοποιητές ροής, υλοποιούνται συνήθως με τη χρήση κωδικοποιητών τμημάτων, διασυνδεδεμένοι με τρόπους σαν αυτούς που περιγράψαμε στην υποενότητα 1.4.1. Παράδειγμα συγχρονιζόμενου κωδικοποιητή ροής αποτελεί ο τρόπος Output Feedback (OFB) mode, ενώ παράδειγμα αυτο-συγχρονιζόμενου, ο τρόπος Cipher Feedback (CFB) mode.

1.4.2 Κρυπτογραφία Δημοσίου Κλειδιού (Public Key Cryptography)

Τα προβλήματα που χαρακτηρίζουν την συμβατική, όπως ονομάζεται συχνά στην βιβλιογραφία, συμμετρική κρυπτογραφία, αποτελούν, μεταξύ άλλων, η ασφαλή ανταλλαγή κλειδιών κρυπτογράφησης μέσα από ένα μη ασφαλές κανάλι επικοινωνίας και η ψηφιακή υπογραφή, που αφενός πιστοποιεί τον αποστολέα του μηνύματος και αφετέρου εγγυάται ότι το μήνυμα δεν έχει τροποποιηθεί από κάποιον τρίτο.

Ως μια λύση σε αυτά τα προβλήματα, ο W. Diffie και ο M.E. Hellmann προτείνουν το 1976 μια μέθοδο που την ονομάζουν κρυπτογραφία δημοσίου κλειδιού [13]. Σε αυτήν περιγράφεται ένα σύστημα κρυπτογράφησης που υλοποιείται με τη χρήση ενός ζεύγους διαφορετικών κλειδιών, ένα ιδιωτικό κλειδί που φυλάσσεται με ασφάλεια από τον κάτοχό του και ένα δημόσιο, το οποίο, όπως αντανακλά και το όνομά του, δημοσιοποιείται ελεύθερα προς κάθε ενδιαφερόμενο. Η κρυπτογράφηση γίνεται με τη χρήση του δημοσίου κλειδιού, ενώ η αποκρυπτογράφηση με χρήση του ιδιωτικού κλειδιού. Η δε ανάστροφη χρήση των κλειδιών, υλοποιεί την λειτουργία της ψηφιακής υπογραφής.

Δυο χρόνια αργότερα, οι R.L. Rivest, A. Shamir και L. Adleman [14], καταφέρνουν να υλοποιήσουν ένα τέτοιο σύστημα που θα μπορεί να παράγει δυο διαφορετικά κλειδιά, που να είναι συζευγμένα μεταξύ τους με έναν μαθηματικό τρόπο και να μπορούν να χρησιμοποιηθούν για την μέθοδο που πρότειναν οι W. Diffie και M.E. Hellmann. Ο αλγόριθμος που προτείνουν ονομάζεται RSA, από τα αρχικά των ονομάτων τους και χρησιμοποιείται ακόμη και σήμερα σε πάρα πολλά συστήματα και πρωτόκολλα.

Άλλη μια υποκατηγορία της ασύμμετρης κρυπτογραφίας αποτελούν οι **συναρτήσεις κατακερματισμού** (**Hashing functions**). Αυτές παίρνουν ως είσοδο ένα κείμενο μεταβλητού μεγέθους και παράγουν ως έξοδο ένα κατακερματισμένο κείμενο σταθερού μεγέθους πάντα και αρκετά μικρότερου του αρχικού κειμένου. Λειτουργούν κυρίως χωρίς κλειδί κρυπτογράφησης και είναι μονόδρομες συναρτήσεις,

δεν γίνεται δηλαδή να αντιστραφεί το κατακερματισμένο κείμενο στο αρχικό. Σε αυτό έγκειται και η ασυμμετρία τους. Οι συναρτήσεις κατακερματισμού χρησιμοποιούνται για την υλοποίηση της ψηφιακής υπογραφής, αλλά και για να αποθηκεύονται τα διαπιστευτήρια των χρηστών σε βάσεις δεδομένων, μεταξύ των άλλων χρήσεών τους. Παραδείγματα τέτοιων είναι οι αλγόριθμοι MD5, ο SHA-2 και όλη η οικογένεια αλγορίθμων SHA-x, ο LM και ο NTLM. [15]

1.5 Κατηγορίες Επιθέσεων σε Κωδικοποιητές

Όπως αναφέραμε και στην υποενότητα 1.1, οι μέθοδοι επιθέσεων σε έναν κωδικοποιητή, κατηγοριοποιούνται ανάλογα με τις πληροφορίες και το υλικό που έχει στην διάθεσή του ο κρυπταναλυτής. Τις πιο βασικές κατηγορίες επιθέσεων που εφαρμόζονται σε κωδικοποιητές, συνιστούν η **επίθεση ωμής βίας** (**Brute-Force Attack**), η **επίθεση με κρυπτογραφημένο κείμενο μόνο** (**Ciphertext-Only Attack**), η **επίθεση με γνώση αρχικού κειμένου** (**Known-Plaintext Attack**) και η **επίθεση επιλεγμένου αρχικού κειμένου** (**Chosen-Plaintext Attack**) [7].

Επίθεση Ωμής Βίας (Brute-Force Attack)

Η επίθεση ωμής βίας συνήθως δεν αναφέρεται στην βιβλιογραφία ρητά ως μέθοδος επίθεσης, καθότι δεν αποτελεί εν γένει πρακτικά εφαρμόσιμη μέθοδο. Ουσιαστικά, με βάση αυτή, ο κρυπταναλυτής δοκιμάζει όλα τα πιθανά κλειδιά κρυπτογράφησης προκειμένου να βρει αυτό που αποκρυπτογραφεί το κρυπτογραφημένο κείμενο που τον ενδιαφέρει. Μια διαδικασία που για ένα κλειδί μεγάλου μήκους σημαίνει εν δυνάμει δισεκατομμύρια πιθανούς συνδυασμούς και μια υπολογιστική πολυπλοκότητα για να τα δοκιμάσει όλα, τόσο υψηλή, που τα μέσα για να την εφαρμόσουν ή/και ο απαραίτητος χρόνος, δεν είναι συνήθως υπαρκτά.

Επίθεση με Κρυπτογραφημένο Κείμενο (Ciphertext-Only Attack)

Εδώ ο κρυπταναλυτής, προσπαθεί να ανακτήσει το κλειδί κρυπτογράφησης ή το αρχικό κείμενο, παρατηρώντας μόνο το κρυπτογραφημένο κείμενο. Κωδικοποιητές που είναι ευάλωτοι σε αυτή τη μέθοδο, θεωρούνται γενικά ανασφαλείς.

Επίθεση με Γνώση Αρχικού Κειμένου (Known-Plaintext Attack)

Σε αυτή την κατηγορία, ο κρυπταναλυτής έχει στην διάθεσή του το αρχικό κείμενο και το αντίστοιχο κρυπτογραφημένο κείμενο ή, ίσως και περισσότερα από ένα τέτοια ζεύγη. Προσπαθεί με βάση αυτό το υλικό, να ανακαλύψει ενδεχόμενα επαναλαμβανόμενα μοτίβα που, αν καταφέρει να τα αναπαράξει, μπορεί να αποκομίσει πληροφορίες για τη λειτουργία και την αρχικοποίηση του κωδικοποιητή.

Επίθεση Επιλεγμένου Αρχικού Κειμένου (Chosen-Plaintext Attack)

Τέλος, με αυτή τη μέθοδο επίθεσης, ο κρυπταναλυτής επιλέγει ένα αρχικό κείμενο και παράγει από αυτό, το αντίστοιχο κρυπτογραφημένο κείμενο. Με βάση αυτό προσπαθεί να ανακτήσει οποιαδήποτε απαραίτητη πληροφορία, όπως για παράδειγμα το κλειδί κρυπτογράφησης, προκειμένου μελλοντικά να είναι σε θέση να αποκρυπτογραφήσει παλαιότερα κρυπτογραφημένα κείμενα που έχει ήδη στην κατοχή του.

Κεφάλαιο 2

Ανταλλαγή Χρόνου/Μνήμης (Time-Memory Trade-Off)

Η ανταλλαγή χρόνου/μνήμης, συνιστά μια τεχνική που θέτει έναν συμβιβασμό ανάμεσα στην επίθεση ωμής βίας και στην παραγωγή πινάκων αναζήτησης, για την εφαρμογή μιας κρυπτογραφικής επίθεσης. Ένα συμβιβασμό με την εγγύηση ότι ο απαιτούμενος χώρος αποθήκευσης ενός πίνακα αναζήτησης θα είναι αρκετά μικρός ώστε να δύναται να αποκτηθεί, με το τίμημα ότι, κατά τη φάση της αποκρυπτογράφησης, η αναζήτηση στον πίνακα θα έχει ελαφρώς αυξημένη υπολογιστική πολυπλοκότητα, αλλά σε βαθμό που η επίθεση να είναι εφαρμόσιμη σε ένα ρεαλιστικό πλαίσιο.

2.1 Η Μέθοδος του Hellmann

Η πρώτη φορά που περιγράφεται μια τέτοια μέθοδος με εφαρμογή στην κρυπτογραφία, είναι σε μια δημοσίευση του 1980 με τίτλο "A Cryptanalytic Time-Memory Trade-Off" [1]. Σε αυτή, ο Hellmann αναπτύσσει μια μέθοδο που μπορεί να εφαρμοστεί ως μια επίθεση με επιλεγμένο αρχικό κείμενο και την δοκιμάζει στον κωδικοποιητή τμημάτων DES.

Κατασκευή Πινάκων

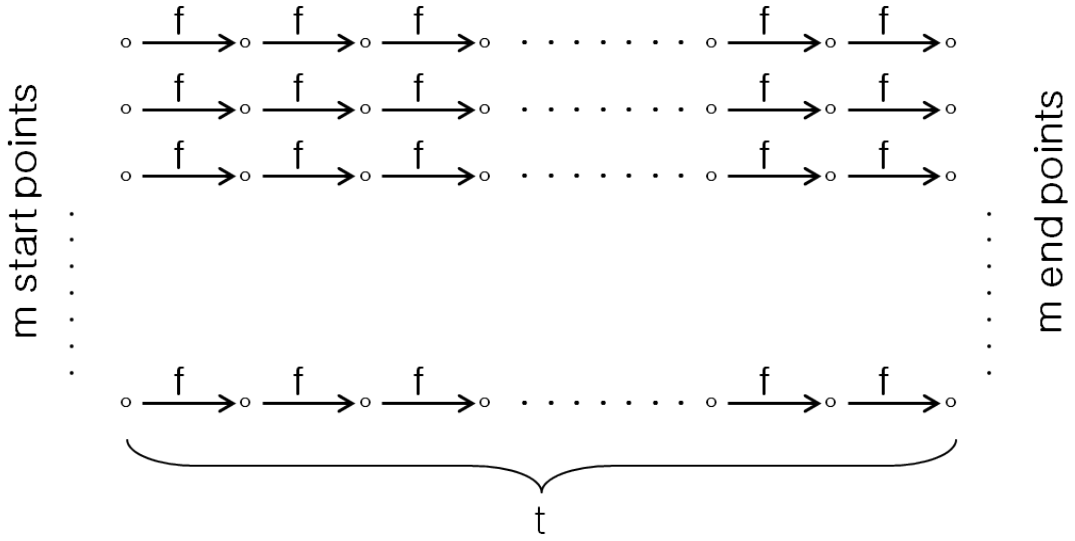
Για την εφαρμογή της μεθόδου, αρχικά επιλέγεται με ομοιόμορφη κατανομή, ένας αριθμός από m διαφορετικά μεταξύ τους κλειδιά, από τον συνολικό χώρο των κλειδιών, N. Αυτά τα ονομάζουμε **αρχικά σημεία** (**starting points or SPs**). Το καθένα από αυτά, για την περίπτωση του DES, έχει μήκος 56bit. Στη συνέχεια επιλέγεται ένα σταθερό αρχικό κείμενο, το οποίο συμβολίζουμε ως P_0. Κατόπιν, κρυπτογραφούμε το επιλεγμένο κείμενο με τον DES, χρησιμοποιώντας ως κλειδιά για τον έλεγχο της κρυπτογράφησης τα αρχικά σημεία. Έστω $C_0 = S_k(P_0)$, η πράξη κρυπτογράφησης του αρχικού κειμένου που παράγει το κρυπτογραφημένο κείμενο C_0. Το αποτέλεσμα είναι m κρυπτογραφημένα κείμενα μήκους 64bit το καθένα.

Σε αυτό το σημείο εισάγεται η ιδέα της **συνάρτησης αναγωγής** (**Reduce Function**) $R(C)$. Η συνάρτηση αυτή επιδρά στο κρυπτογραφημένο κείμενο με έναν απλό και ντετερμινιστικό τρόπο και το μετασχηματίζει σε ένα καινούριο κλειδί μήκους

56bit. Η ακολουθία των πράξεων κρυπτογράφησης και αναγωγής ορίζεται ως $f(K) = R[S_k(P_0)]$. Η κεντρική ιδέα είναι ότι συνάρτηση αναγωγής μπορεί να είναι αρκετά απλή (π.χ. να απορρίπτει τα τελευταία 8bit), αρκεί να παράγει ένα έγκυρο κλειδί, ένα κλειδί δηλαδή που ανήκει στον συνολικό χώρο των κλειδιών N. Ταυτόχρονα όμως, πρέπει, όσες φορές και να επαναληφθεί με τα ίδια δεδομένα, να δίνει πάντα το ίδιο αποτέλεσμα, εξ΄ ου και ντετερμινιστική.

Εφαρμόζοντας τη συνάρτηση αναγωγής στα m κρυπτογραφημένα κείμενα που παράξαμε στο προηγούμενο στάδιο, έχουμε m καινούρια κλειδιά. Αυτά τα χρησιμοποιούμε για να ξανά κρυπτογραφήσουμε το επιλεγμένο αρχικό κείμενο. Επαναλαμβάνουμε αυτή τη διαδικασία συνολικά t φορές. Στο τέλος θα έχουμε παράξει m αλυσίδες μήκους t η κάθε μια, που έχουν ως αρχικό σημείο ένα κλειδί κρυπτογράφησης και ως **τελικό σημείο** (**ending point or EP**) ένα κρυπτογραφημένο κείμενο.

Από αυτές τις αλυσίδες, κρατάμε μόνο τα αρχικά και τα τελικά σημεία, τα ταξινομούμε ως ζεύγη με βάση το τελικό σημείο και τα αποθηκεύουμε ως έναν πίνακα αναζήτησης. Αυτός ο πίνακας μπορεί να χρησιμοποιηθεί για να βρεθεί το κλειδί που αποκρυπτογραφεί ένα κείμενο της επιλογής μας. Όλη η διαδικασία που περιγράψαμε φαίνεται σχηματικά στην εικόνα 2.1.



Σχήμα 2.1: Κατασκευή Πινάκων κατά Hellmann

Θεωρώντας, T τον χρόνο σε πλήθος πράξεων και M την μνήμη σε μέγεθος λέξεων, αυτή η τεχνική δύναται να ανακτήσει το κλειδί κρυπτογράφησης από έναν χώρο N κλειδιών σε χρόνο $T = N^{2/3}$ με χρήση μνήμης $M = N^{2/3}$ λέξεις. Η πιθανότητα να βρεθεί ένα κλειδί σε έναν πίνακα m γραμμών των t κλειδιών ορίζεται ως:

$$P_{table} \geq \frac{1}{N} \sum_{i=1}^{m} \sum_{j=0}^{t-1} \left(1 - \frac{it}{N}\right)^{j+1} \tag{2.1.1}$$

Από την παραπάνω σχέση 2.1.1, προκύπτει ότι, όσο πιο μεγάλος γίνεται ένας πίνακας, τόσο περισσότερο μικραίνει η πιθανότητα επιτυχίας. Για την μέγιστη πιθανότητα

επιτυχίας, οι τιμές για τα m και t φράσσονται με βάση τη σχέση $mt^2 = N$. Η λύση για να καλυφθεί επαρκώς ο συνολικός χώρος των κλειδιών και να αυξηθεί περαιτέρω η πιθανότητα επιτυχίας, είναι να κατασκευάσουμε περισσότερους πίνακες, χρησιμοποιώντας διαφορετική συνάρτηση αναγωγής για τον καθένα. Σε αυτή την περίπτωση, για έναν αριθμό l πινάκων, η πιθανότητα επιτυχίας εύρεσης του κλειδιού ορίζεται ως:

$$P_{success} \geq 1 - \left(\frac{1}{N} \sum_{i=1}^{m} \sum_{j=0}^{t-1} \left(1 - \frac{it}{N} \right)^{j+1} \right)^{l} \tag{2.1.2}$$

Αναζήτηση Κλειδιού σε Πίνακα

Όταν θέλουμε να χρησιμοποιήσουμε τον πίνακα που έχουμε παράξει, για να βρούμε ένα υποψήφιο κλειδί, ακολουθείται η εξής διαδικασία. Αρχικά, για το κρυπτογραφημένο κείμενο, κάνουμε μια αναζήτηση να δούμε αν υπάρχει στα τελικά σημεία του πίνακα. Αν η αναζήτηση είναι επιτυχής, τότε ξέρουμε ότι το κλειδί που αποκρυπτογραφεί το κείμενο, βρίσκεται στη θέση t-1 της αλυσίδας στην οποία ανοίκει το τελικό σημείο. Αρκεί τότε να αναπαράξουμε την αλυσίδα μέχρι το σημείο t-1 προκειμένου να ανακτήσουμε το κλειδί. Υπάρχει πάντα μια μικρή πιθανότητα, το κλειδί που θα βρίσκεται σε αυτή τη θέση να μην είναι το σωστό. Τότε έχουμε την περίπτωση που ονομάζεται **false alarm**.

Αν η αναζήτηση δεν είναι επιτυχής, τότε εφαρμόζουμε στο κρυπτογραφημένο κείμενο τη συνάρτηση αναγωγής και παράγουμε ένα κλειδί. Αυτό το χρησιμοποιούμε για να κρυπτογραφήσουμε το επιλεγμένο κείμενο. Κάνουμε και πάλι μια αναζήτηση στον πίνακα για το νέο κρυπτογραφημένο κείμενο. Αν αυτή τη φορά η αναζήτηση είναι επιτυχής, τότε ξέρουμε ότι το κλειδί που αποκρυπτογραφεί το κείμενο βρίσκεται στην θέση t-2 της αλυσίδας, όπου βρήκαμε το τελικό σημείο. Αν πάλι δεν βρεθεί ταιριαστό τελικό σημείο, επαναλαμβάνουμε την όλη διαδικασία μέχρι και t φορές, όσο το μήκος της αλυσίδας δηλαδή. Αν και πάλι δεν έχουμε επιτυχία, τότε το κλειδί δεν βρίσκεται στον συγκεκριμένο πίνακα.

Προβλήματα της Μεθόδου Hellmann

Ένα από τα προβλήματα που μπορεί να έχει η μέθοδος Hellmann, όπως ήδη αναφέραμε, είναι τα false alarms, όπου ενώ το κρυπτογραφημένο κείμενο που μας ενδιαφέρει μπορεί να υπάρχει στον πίνακα, το αντίστοιχο κλειδί του δεν είναι το σωστό. Η λύση που προτείνεται σε αυτό το πρόβλημα είναι να παράγονται μεγαλύτεροι πίνακες, ή περισσότεροι μικροί.

Πρόβλημα αποτελούν επίσης οι **συγκρούσεις** (**collisions**). Πρόκειται για την περίπτωση όπου δυο ή περισσότερες αλυσίδες, σε κάποιο σημείο τους μετά από μια αναγωγή, καταλήγουν στο ίδιο κρυπτογραφημένο κείμενο. Οπότε έχουμε δυο οι περισσότερες αλυσίδες να ξεκινούν από διαφορετικό αρχικό σημείο και να καταλήγουν στο ίδιο τελικό σημείο. Σε αυτές τις περιπτώσεις επιλέγεται να γίνουν **συγχωνεύσεις** (**merges**), να κρατήσουμε δηλαδή μόνο μια από αυτές τις αλυσίδες. Όπως αντιλαμβανόμαστε διαισθητικά, μια λάθος επιλογή μπορεί τελικά να οδηγήσει σε ένα false alarm. Για να καλύπτουν οι πίνακες επαρκώς τον χώρο των συνολικών κλειδιών

μετά τις συγχωνεύσεις, προτείνεται να υπολογίζονται περισσότερες από τις προβλεπόμενες αλυσίδες ανά πίνακα.

Έκτοτε, έχουν προταθεί διάφορες βελτιώσεις για την μέθοδο του Hellmann προκειμένου να καταπολεμηθούν τα προβλήματα που παρουσιάζει και να βελτιωθεί έτσι η απόδοσή της. Δεν θα μπούμε στην διαδικασία να τις αναφέρουμε όλες, καθότι η αναφορά και ανάλυσή τους ξεφεύγει από τους σκοπούς αυτού του κειμένου. Ενδεικτικά θα σταθούμε σε δύο από αυτές, τις οποίες αναλύουμε στις δύο ενότητες που ακολουθούν, με την δεύτερη από αυτές να είναι αυτή των πινάκων Ουράνιου Τόξου, που αποτελεί και την προσέγγιση που υιοθετούμε στην υλοποίησή μας.

2.2 Διακριτά Σημεία Distinguishing Points

Το 1982, ο Rivest προτείνει μια βελτιστοποίηση στην μέθοδο του Hellmann, που την ονομάζει **Διακριτά Σημεία** (**Distinguishing Points**) [16, σελ. 134]. Με βάση αυτή, αντί οι αλυσίδες να έχουν σταθερό μήκος t, προτείνει να σταματά ο υπολογισμός της αλυσίδας όταν το αποτέλεσμα μιας κρυπτογράφησης ικανοποιεί ένα προκαθορισμένο κριτήριο (π.χ. τα πρώτα 10bit να είναι μηδενικά). Αυτό τότε, αποτελεί ένα διακριτό σημείο και ορίζεται να είναι το τελικό σημείο της αλυσίδας που θα αποθηκευτεί στον πίνακα.

Στην φάση της αναζήτησης σε έναν πίνακα με διακριτά σημεία, εφαρμόζουμε πρώτα τα βήματα υπολογισμού της αλυσίδας στο κρυπτογραφημένο κείμενο που μας ενδιαφέρει, μέχρι να καταλήξουμε σε ένα διακριτό σημείο. Κατόπιν, κάνουμε μια αναζήτηση στον πίνακα για το διακριτό σημείο που παράξαμε.

Δοθέντος ότι όλα τα τελικά σημεία του πίνακα είναι διακριτά σημεία και άρα κάνουμε αναζήτηση μόνο για τέτοια, η μέθοδος αυτή ελαχιστοποιεί τον συνολικό αριθμό των αναζητήσεων και άρα επιταχύνει το στάδιο της αποκρυπτογράφησης, σε σχέση με τη μέθοδο του Hellmann. Επίσης, διευκολύνει την ανίχνευση συγκρούσεων και απλοποιεί τη διαδικασία των συγχωνεύσεων. Έχει όμως το μειονέκτημα ότι οι αλυσίδες έχουν μεταβλητό μήκος. Αυτό οδηγεί σε αυξημένη πιθανότητα να συμβούν false alarms και σε αύξηση της υπολογιστικής πολυπλοκότητας για τον εντοπισμό τους. Ένα άλλο πρόβλημα που μπορεί να προκύψει, είναι μια ή περισσότερες αλυσίδες να μην καταλήξουν ποτέ σε ένα διακριτό σημείο και έτσι ο υπολογισμός τους να εγκλωβιστεί σε έναν ατέρμονο βρόχο. Η λύση για αυτό είναι να ορίσουμε ένα μέγιστο κατώφλι για το μήκος μιας αλυσίδας και όταν κάποια αλυσίδα το υπερβαίνει, να διακόπτεται ο υπολογισμός της.

2.3 Πίνακες Ουράνιου Τόξου (Rainbow Tables)

Το 2003 ο P. Oechslin παρουσιάζει μια πρωτότυπη ως τότε βελτιστοποίηση στην μέθοδο του Hellmann, η οποία μοιράζεται αρκετά από τα προτερήματα της χρήσης διακριτών σημείων, χωρίς όμως να φέρει τα μειονεκτήματά της [2]. Η πρωτοτυπία έγκειται στο ότι, σε κάθε στάδιο αναγωγής των αλυσίδων χρησιμοποιείται μια διαφορετική συνάρτηση αναγωγής. Αν αντιστοιχίσουμε σε κάθε συνάρτηση αναγωγής ένα διαφορετικό χρώμα, τότε, στον πίνακα που παράγεται, κάθε στήλη θα έχει διαφορετικό χρώμα, πράγμα που τον κάνει να μοιάζει με ουράνιο τόξο. Με αυτό το σκεπτικό

P. Oechslin ονομάζει τη μέθοδό του, **Πίνακες Ουράνιου Τόξου** (**Rainbow Tables**).

Το πλεονέκτημα της χρήσης t διαφορετικών συναρτήσεων αναγωγής για μήκος αλυσίδας t, είναι ότι δυο αλυσίδες θα συγχωνευτούν, μόνο όταν η σύγκρουσή λάβει χώρα στην ίδια θέση καί των δυο αλυσίδων. Αυτές οι αλυσίδες τότε θα καταλήξουν στο ίδιο τελικό σημείο, όπως και με τη μέθοδο των διακριτών σημείων, και έτσι είναι εύκολα ανιχνεύσιμες. Αν η σύγκρουση εμφανιστεί σε διαφορετικές θέσεις, η κάθε αλυσίδα συνεχίζει με διαφορετική συνάρτηση αναγωγής και τότε δεν υπάρχει συγχώνευση. Η πιθανότητα να συγχωνευτούν δύο αλυσίδες μετά από μια σύγκρουση είναι $\frac{1}{t}$.

Με το να εφαρμόζεται κάθε συνάρτηση αναγωγής μόνο μια φορά, μηδενίζεται η πιθανότητα ο υπολογισμός μιας αλυσίδας να εγκλωβιστεί σε ατέρμονο βρόχο. Έτσι δεν απαιτείται έλεγχος για μια τέτοια περίπτωση, πράγμα που επιταχύνει τον υπολογισμό της αλυσίδας.

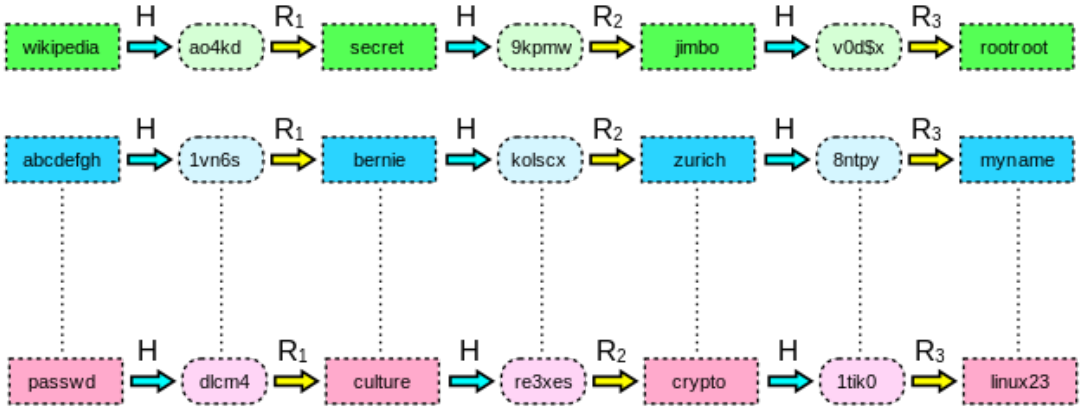
Τέλος, οι αλυσίδες που παράγονται έχουν όλες το ίδιο μήκος και έτσι μειώνεται η πιθανότητα να έχουμε false alarms και αποφεύγεται η επιπλέον δουλειά που θα πρέπει να γίνει για τον εντοπισμό τους.

Η διαδικασία της αναζήτησης λειτουργεί με παρόμοιο τρόπο όπως και στην μέθοδο του Hellmann, με τη διαφορά ότι οι συναρτήσεις αναγωγής εφαρμόζονται με ανάστροφη σειρά, δηλαδή από την τελευταία προς την πρώτη. Ένα διάγραμμα του υπολογισμού των αλυσίδων, για την περίπτωση μιας συνάρτησης κατακερματισμού, βλέπουμε στην εικόνα 2.2, ενώ η εικόνα 2.3 μας δείχνει τη διαδικασία αναζήτησης στον πίνακα.

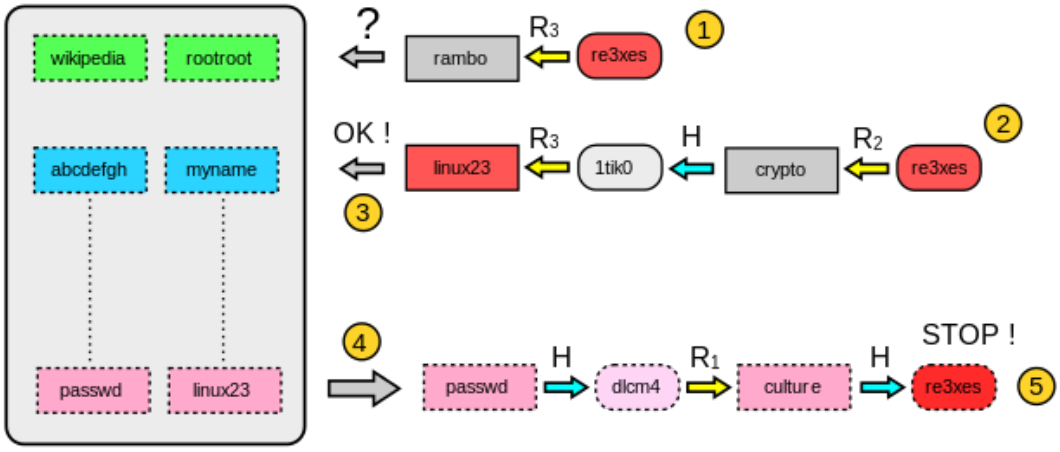
Η πιθανότητα επιτυχίας ενός πίνακα Ουράνιου Τόξου ορίζεται ως:

$$P_{table} = 1 - \prod_{i=1}^{t} \left(1 - \frac{m_i}{N}\right)$$

όπου $m_1 = m$ και $m_{n+1} = N\left(1 - e^{-\frac{m_n}{N}}\right)$. Σύμφωνα με τον Oechslin, η πιθανότητα επιτυχίας ενός πίνακα Ουράνιου Τόξου είναι άμεσα συγκρίσιμη με αυτή ενός πίνακα Hellmann, με την πιθανότητα επιτυχίας t πινάκων Hellmann μεγέθους $m \times t$ ο καθένας, να είναι περίπου ίση με την πιθανότητα επιτυχίας ενός πίνακα Ουράνιου Τόξου μεγέθους $mt \times t$. Ο αριθμός πράξεων για την αναζήτηση σε έναν πίνακα Ουράνιου Τόξου είναι $\frac{t(t-1)}{2}$ και είναι περίπου ο μισός σε σχέση με τους πίνακες Hellmann, όπου για ένα σύνολο t πινάκων μεγέθους $m \times t$ οι πράξεις αναζήτησης ανέρχονται σε t^2.



Σχήμα 2.2: Κατασκευή Πινάκων Ουράνιου Τόξου [17]



Σχήμα 2.3: Αναζήτηση σε Πίνακα Ουράνιου Τόξου [17]

2.4 Ανταλαγή Χρόνου/Μνήμης για Κωδικοποιητές Ροής

Στην περίπτωση που η μέθοδος της ανταλλαγής χρόνου/μνήμης εφαρμοστεί σε κωδικοποιητές ροής, τα πράγματα λειτουργούν λίγο διαφορετικά, ως προς το τι θα επιλέξουμε να είναι ένα τελικό και ένα αρχικό σημείο στον πίνακά μας. Εν γένει, για μια επιτυχής επίθεση σε έναν κωδικοποιητή ροής, δεν είναι απαραίτητα η ανάκτηση του κλειδιού ο στόχος. Συνήθως, ο κρυπταναλυτής, στοχεύει στην ανάκτηση μιας εσωτερικής κατάστασης του κωδικοποιητή, ανεξαρτήτως του πιο είναι το κλειδί, και τότε μπορεί να αναπαράξει τις επόμενες καταστάσεις για την αποκρυπτογράφηση των ακολουθούμενων κειμένων. Όσων αφορά το τι ακριβώς ορίζεται ως μια εσωτερική κατάσταση, αυτό μπορεί να διαφέρει ανάλογα με το πως υλοποιείται ένας κωδικοποιητής ροής, αν και στην βιβλιογραφία αναφέρονται και γενικευμένες μέθοδοι που καλύπτουν περισσότερες από μια κατηγορίες τέτοιων.

Η περίπτωση των κωδικοποιητών ροής έχει επίσης μελετηθεί εκτενώς στην βιβλιογραφία, θα αρκεστούμε παρόλα αυτά να αναφερθούμε σε μια μόνο δημοσίευση

που συμπυκνώνει σε μια αναλυτική προσέγγιση τις περισσότερες από αυτές. Πρόκειται για την μελέτη των Jin Hong και Palash Sarkar [18]. Σύμφωνα με τους Jin Hong και Palash Sarkar, στην περίπτωση των πρωίμων κωδικοποιητών ροής, οι οποίοι υλοποιούνταν συνήθως με τη χρήση LFSR's, ο στόχος είναι η ανάκτηση της εσωτερικής κατάστασης. Επομένως, ως αρχικό σημείο ορίζεται η εσωτερική κατάσταση και ως τελικό σημείο ορίζεται μια ποσότητα του κλειδιού ροής, ίση με το μέγεθος της εσωτερικής κατάστασης. Για νεότερους κωδικοποιητές ροής, που συνήθως υλοποιούνται με χρήση κωδικοποιητών τμημάτων, ο στόχος είναι η ανάκτηση του κλειδιού κρυπτογράφησης. Το αρχικό σημείο τότε ορίζεται να είναι το κλειδί κρυπτογράφησης, ενώ το τελικό σημείο να είναι μια ποσότητα του κλειδιού ροής στο μέγεθος του κλειδιού κρυπτογράφησης. Στις πιο σύγχρονες περιπτώσεις όπου, σε συνδυασμό με το κλειδί κρυπτογράφησης, χρησιμοποιείται και ένα διάνυσμα αρχικοποίησης (IV) του κωδικοποιητή, ο στόχος είναι και πάλι η ανάκτηση του κλειδιού κρυπτογράφησης. Υπάρχει όμως η διαφοροποίηση ότι, το αρχικό σημείο αποτελείται από τον συνδυασμό του κλειδιού και του διανύσματος αρχικοποίησης, ενώ το τελικό σημείο είναι μια ποσότητα του κλειδιού ροής μήκους ίση με αυτόν τον συνδυασμό.

Η πολυπλοκότητα για την εφαρμογή της μεθόδου ανταλλαγής χρόνου/μνήμης, για την περίπτωση κωδικοποιητών ροής που χρησιμοποιούν διάνυσμα αρχικοποίησης, παίρνει τη μορφή $T = M = 2^{\frac{1}{2}(k+v)}$, όπου k είναι το μήκος του κλειδιού και v είναι το μήκος του διανύσματος αρχικοποίησης. Αν μάλιστα ισχύει η σχέση $v < k$, τότε η πολυπλοκότητα είναι μικρότερη από αυτή της εξαντλητικής αναζήτησης που είναι 2^k. Το άμεσο συμπέρασμα που διεξάγεται είναι ότι, αν το διάνυσμα αρχικοποίησης έχει μικρότερο μήκος από το κλειδί, ο κωδικοποιητής ροής είναι ευάλωτος σε επιθέσεις που εφαρμόζουν την τεχνική της ανταλλαγής χρόνου/μνήμης. Όπως θα εξηγήσουμε στο κεφάλαιο 3, αυτό αντανακλάται στην δομή του συστήματος ασφαλείας του δικτύου GSM γενεάς 2.5G.

Κεφάλαιο 3

Το Δίκτυο GSM

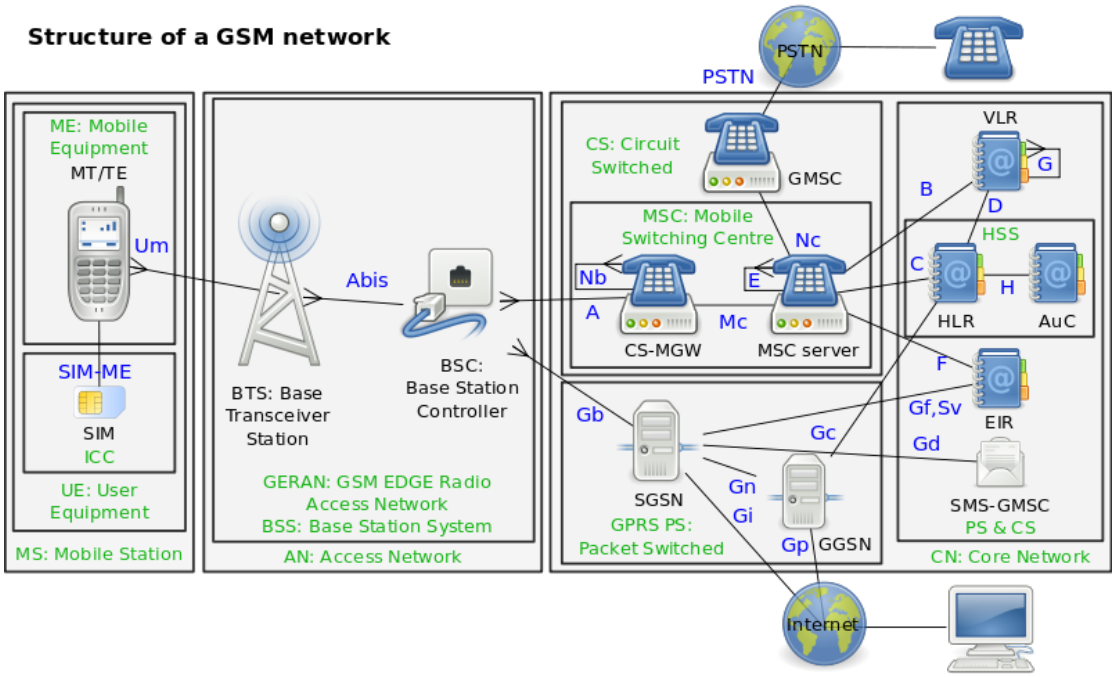
Στα τέλη της δεκαετίας του ΄70 με αρχές ΄80, μια εποχή όπου η ανάπτυξη αναλογικών ασύρματων τηλεπικοινωνιών χαρακτηρίζονταν από έντονο ανταγωνισμό, η θέσπιση των τεχνικών προδιαγραφών και τελικά, η ανάπτυξη του παγκόσμιου ψηφιακού κυψελοτού δικτύου τηλεπικοινωνιών GSM, είναι το αποτέλεσμα μιας επιτυχής συνεργασίας μεταξύ πολλών φορέων [19].

Το 1982 δημιουργείται το Groupe Special Mobile (GSM) committee από την European Conference of Postal and Telecommunications Administrations (CEPT) με σκοπό να σχεδιάσει ένα πανευρωπαϊκό σύστημα κινητών ψηφιακών τηλεπικοινωνιών. Το 1986 ξεκινούν οι διαδικασίες για την δέσμευση του φάσματος συχνοτήτων των 900MHz για το GSM. Το 1988 ολοκληρώνονται οι πλήρεις προδιαγραφές για το δίκτυο GSM. Το 1989, το Groupe Special Mobile (GSM) committee μεταφέρεται στο European Telecommunications Standards Institute (ETSI). Το 1991 διενεργείται το πρώτο τηλεφώνημα μέσω GSM από το δίκτυο της Radiolinja στην Φινλανδία. Μέχρι το 1996 υπάρχουν πλέον 167 δίκτυα σε 94 διαφορετικές χώρες παγκοσμίως, ενώ ο αριθμός των χρηστών προσεγγίζει τα 50 εκατομμύρια [20]. Με την πάροδο των χρόνων, οι τεχνολογίες βελτιώνονται και προστίθενται περισσότερες υπηρεσίες στα δίκτυα, όπως ανταλλαγή γραπτών μηνυμάτων (SMS), υπηρεσία Fax και υπηρεσία ανταλλαγής δεδομένων μέσω μεταγωγής πακέτων (GPRS).

Σήμερα, περίπου 14 χρόνια μετά την έναρξη λειτουργίας του πρώτου εμπορικού GSM δικτύου, απολαμβάνουμε πλέον τις εξελιγμένες υπηρεσίες του δικτύου τέταρτης γενεάς (4G), ενώ οι εργασίες για την ανάπτυξη του δικτύου πέμπτης γενεάς (5G) είναι ήδη σε εξέλιξη.

3.1 Αρχιτεκτονική του Δικτύου GSM

Στην εικόνα του σχήματος 3.1 βλέπουμε τη δομή ενός δικτύου GSM. Ένα δίκτυο GSM διαιρείται σε τρία υποσυστήματα, που αλληλεπιδρούν μεταξύ τους. Αυτά είναι ο **κινητός σταθμός - MS (Mobile Station)**, το **υποσύστημα σταθμού βάσης - BSS (Base Station Subsystem)** και το **υποσύστημα δικτύου - NSS (Network SubSystem)** [21]. Στη συνέχεια θα δώσουμε μια σύντομη περιγραφή των υποσυστημάτων αυτών και των τμημάτων που τα απαρτίζουν.



Σχήμα 3.1: Δομή ενός Δικτύου GSM

Κινητός Σταθμός - MS (Mobile Station)

Ένας MS αποτελείται από την **φορητή συσκευή - ME (Mobile Equipment)** και την **κάρτα SIM (Subscriber Identity Module)**. Στην κάρτα SIM, μεταξύ άλλων, περιέχονται οι αλγόριθμοι για την επικύρωση του χρήστη και για την δημιουργία ενός κλειδιού συνεδρίας K_c που χρησιμοποιείται για την κρυπτογράφηση των δεδομένων μετάδοσης (A3 και A8 αντίστοιχα), το μυστικό κλειδί επικύρωσης K_i και ο κωδικός IMSI (International Mobile Subscriber Identity).

Υποσύστημα Σταθμού Βάσης - BSS (Base Station Subsystem)

Η δουλειά ενός BSS είναι να ελέγχει τις ασύρματες ζεύξεις και να παρέχει συνδεσιμότητα μεταξύ των MS και του δικτύου. Αποτελείται από τον **πομποδέκτη του σταθμού βάσης - BTS (Base Transceiver Station)** και τον **ελεγκτή σταθμού βάσης - BSC (Base Station Controler)**. Ο BTS αναλαμβάνει την αποστολή και τη λήψη σημάτων από και προς τους MS καθώς και την πολυπλεξία τους, τον έλεγχο ισχύος, την κωδικοποίηση και αποκωδικοποίηση φωνής, αλλά και την κρυπτογράφηση όλων των σημάτων που αυτός μεταδίδει. Η βασική λειτουργία του BSC είναι η διαχείριση των κλήσεων. Έχει συνήθως υπό τον έλεγχό του έναν αριθμό από BTSs και φροντίζει για τη μεταβίβαση (handoff) του ελέγχου όταν ένα MS μετακινηθεί από μια κυψέλη σε μια άλλη.

Υποσύστημα Δικτύου - NSS (Network SubSystem)

Τα βασικά στοιχεία που απαρτίζουν έναν NSS είναι το **κέντρο μεταγωγής κινητής τηλεφωνίας - MSC (Mobile Switching Center)** και η **πύλη κέντρου μεταγωγής κινητής τηλεφωνίας - GMSC (Gateway Mobile Switching Center)**. Για την υποστήριξη των λειτουργιών του, περιέχει ακόμα το **μητρώο οικείων συνδρομητών - HLR (Home Location Register)**, το **μητρώο επισκεπτών - VLR (Visitor Location Register)**, το **κέντρο επικύρωσης - AuC (Authentication Center)** και το **μητρώο αναγνώρισης εξοπλισμού - EIR (Equipment Identification Register)**. Ο NSS αναλαμβάνει την εγκαθίδρυση επικοινωνίας μεταξύ μιας φορητής συσκευής και ενός άλλου MSC, ενώ πρόσθετα, διαχειρίζεται την αποστολή γραπτών μηνυμάτων SMS. Έχει συνήθως υπό τον έλεγχό του έναν αριθμό από περισσότερους BSSs. Ακολουθεί μια επεξήγηση των τμημάτων που τον απαρτίζουν.

- **MSC**
  Έχει υπό τον έλεγχό του περισσότερους BSCs, δρομολογεί τις εισερχόμενες και εξερχόμενες κλήσεις, διαχειρίζεται τις λειτουργίες εγγραφής, επικύρωσης χρήστη, πληροφόρηση για την τοποθεσία του χρήστη και την μεταβίβαση

- **GMSC**
  Παρέχει συνδεσιμότητα μεταξύ του δικτύου κινητής τηλεφωνίας και των δικτύων σταθερής τηλεφωνίας PSTN και ISDN

- **HLR**
  Η βάση δεδομένων που περιέχει τα στοιχεία των οικείων συνδρομητών (αριθμός τηλεφώνου, τρέχουσα θέση, μυστικό κλειδί K_i, IMSI)

- **VLR**
  Μια βάση δεδομένων σχεδιασμένη για να ελαχιστοποιεί τις αναζητήσεις στην HLR. Χρησιμοποιείται στις περιπτώσεις όπου ο συνδρομητής απομακρύνεται από το οικείο δίκτυο.

- **AuC**
  Μια ασφαλής βάση δεδομένων που περιέχει τα απαραίτητα στοιχεία για την ταυτοποίηση και επικύρωση του συνδρομητή. Επιτελεί την επικύρωση του χρήστη και παράγει τα κλειδιά συνεδρίας K_c

- **EIR**
  Μια βάση δεδομένων που διατηρεί τρεις λίστες, μια για φορητές συσκευές στις οποίες επιτρέπεται η επικοινωνία μέσω του δικτύου, μια με αυτές στις οποίες δεν επιτρέπεται και μια με φορητές συσκευές που υπόκεινται σε παρακολούθηση για οποιονδήποτε λόγο

3.2 Κανάλι Μετάδοσης GSM

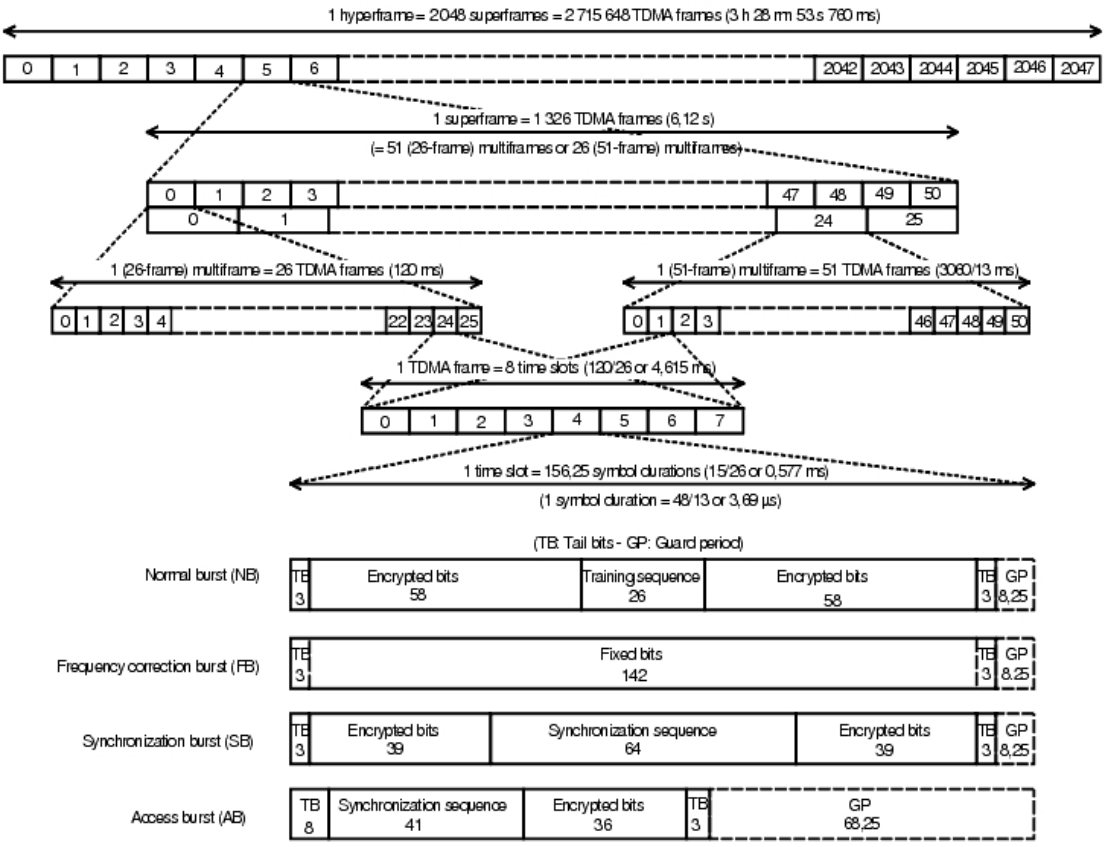
Στην παράγραφο αυτή θα δώσουμε μια σύντομη περιγραφή του καναλιού μετάδοσης του GSM. Θα περιοριστούμε στην αναφορά εκείνων τον στοιχείων που αφορούν άμεσα το κρυπτογραφικό κομμάτι και τον αλγόριθμο A5/3.

Το GSM, για την πολλαπλή πρόσβαση στο κανάλι, χρησιμοποιεί σύστημα πολυπλεξίας με διαίρεση στο επίπεδο του χρόνου (Time Division Multiple Access - TDMA). Έτσι, το κανάλι χωρίζεται σε χρονοθυρίδες και η μετάδοση των δεδομένων γίνεται μέσα σε αυτές υπό τη μορφή ριπών (bursts). Κάθε ριπή έχει χρονική διάρκεια 0.5769 ms, όσο και η κάθε χρονοθυρίδα [22]. Οκτώ διαδοχικές χρονοθυρίδες απαριθμούν ένα πλαίσιο (frame) που έχει χρονική διάρκεια 4.615 ms (8 * 0.5769 ms).

Τα πλαίσια αποτελούν τμήμα μιας μεγαλύτερης δομής που ονομάζεται πολύ-πλαίσιο (multiframe). Για τα κανάλια ελέγχου, ένα πολυ-πλαίσιο απαριθμεί 51 πλαίσια, ενώ για τα κανάλια μετάδοσης, 26 πλαίσια. Με όμοιο τρόπο, σε ένα κανάλι ελέγχου, 51 πολυ-πλαίσια δομούν ένα σούπερ-πλαίσιο και σε ένα κανάλι μετάδοσης, 26 πολυ-πλαίσια επίσης δομούν ένα σούπερ-πλαίσιο (superframe). 2048 σούπερ-πλαίσια αποτελούν ένα υπερ-πλαίσιο (hyperframe).

Κάθε πλαίσιο είναι αριθμημένο σύμφωνα με τη θέση του μέσα σε ένα υπερ-πλαίσιο, με την αρίθμηση να ξεκινά από το μηδέν και να φτάνει ως το 2,715,647. Ο αριθμός πλαισίου χρησιμοποιείται για την παραγωγή της μεταβλητής count, μια μεταβλητή που αποτελεί είσοδο στον αλγόριθμο κρυπτογράφησης. Επομένως, η παραμετροποίηση του αλγόριθμου κρυπτογράφησης, αλλάζει κάθε φορά που αλλάζει και ο αριθμός πλαισίου, δηλαδή κάθε 4.615 ms. Οι αριθμοί πλαισίου επαναλαμβάνονται κάθε 3 ώρες, 28 λεπτά, 53 δευτερόλεπτα και 760 ms.

Στην εικόνα του σχήματος 3.2 βλέπουμε την δομή των πλαισίων, όπως την περιγράψαμε παραπάνω.



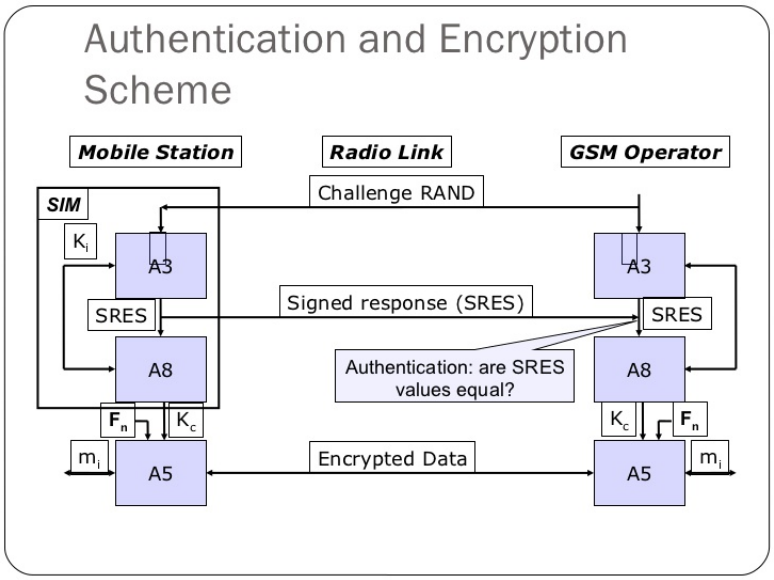
Σχήμα 3.2: Δομή των Πλαισίων στο GSM

3.3 Σύστημα Ασφαλείας του GSM

Οι δύο βασικότερες λειτουργίες του συστήματος ασφαλείας στο GSM δίκτυο, είναι η **επικύρωση** (**authentication**) των χρηστών και η **εμπιστευτικότητα** (**confidentiality**) στην επικοινωνία μεταξύ τους.

Επικύρωση

Κάθε φορά που ένας χρήστης θέλει να εκτελέσει μια κλήση, ενεργοποιείται η φάση της επικύρωσης. Σε αυτή, ο AuC στέλνει στην συσκευή μια τυχαία πρόκληση ονομαζόμενη **RAND**, μεγέθους 128-bit. Η κάρτα SIM της συσκευής, με χρήση του αλγορίθμου A3 και υπό τον έλεγχο του μυστικού κλειδιού K_i τα οποία περιέχει, κρυπτογραφεί την RAND και παράγει μια απάντηση ονομαζόμενη **SRES** μεγέθους 32-bit, την οποία και αποστέλλει στον AuC. Ο AuC, έχοντας κάνει με τη σειρά του τα ίδια βήματα, συγκρίνει την SRES που έλαβε με αυτήν που παρήγαγε ο ίδιος. Μόνο αν οι SRES είναι ταυτόσημες, ο χρήστης ταυτοποιείται και του δίνεται άδεια να χρησιμοποιήσει το δίκτυο, διαφορετικά απαγορεύεται η πρόσβαση του χρήστη στο δίκτυο και αποστέλλεται στην συσκευή μια κατάλληλη ενημέρωση για αυτό. Η διαδικασία αυτή, όπως την περιγράψαμε, απεικονίζεται στο σχήμα 3.3 που ακολουθεί.

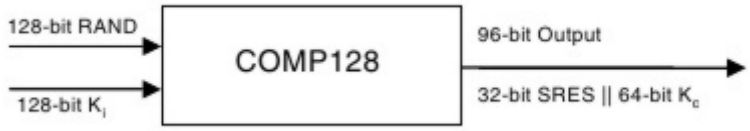


Σχήμα 3.3: Επικύρωση Χρήστη και Παραγωγή K_c στο GSM

Κατά τη φάση της επικύρωσης, ταυτόχρονα με την παραγωγή της SRES όπως επίσης φαίνεται στο παραπάνω σχήμα 3.3, παράγεται και ένα κρυπτογραφικό κλειδί για την τρέχουσα συνεδρία με χρήση του αλγορίθμου A8, το K_c με μέγεθος 64-bit. Το K_c χρησιμοποιείται στην λειτουργία της εμπιστευτικότητας ως το κλειδί που θα ελέγχει την κρυπτογράφηση των δεδομένων που αποστέλλει ο χρήστης, όπως φωνή και SMS, αλλά και δεδομένα GPRS.

Οι αλγόριθμοι A3 και A8, υλοποιούνται συνήθως ως ένας ενιαίος κωδικοποιητής, που ονομάζεται COMP128. Ο COMP128 είναι στην ουσία μια μονόδρομη συνάρτηση

κατακερματισμού, η οποία όπως ήδη αναφέραμε, βρίσκεται και εκτελείται εντός της κάρτας SIM. Παίρνει ως είσοδο την RAND και το μυστικό κλειδί K_i, καθένα από τα οποία έχει μέγεθος 128-bit και επιστρέφει ως έξοδο μια ποσότητα από 96-bit. Τα πρώτα 32-bit αυτής αποτελούν την SRES, ενώ τα τελευταία 64-bit αποτελούν το κλειδί συνεδρίας K_c. Στο σχήμα 3.4 βλέπουμε ένα διάγραμμα του COMP128 [21, σελ. 12].



Σχήμα 3.4: Ο Κωδικοποιητής COMP128

Εμπιστευτικότητα

Η λειτουργία της εμπιστευτικότητας, όπως έχουμε ήδη αναφέρει, συνίσταται στην κρυπτογράφηση των δεδομένων που μεταδίδει ο χρήστης. Για τον σκοπό αυτό, στο GSM χρησιμοποιείται η γενιά αλγορίθμων A5, στην οποία ανοίκουν οι αλγόριθμοι:

- **A5/0**
  Αποτελεί κρυπτογραφικό αλγόριθμο κατ΄ ευφημισμό μόνο. Χρησιμοποιείται για να δηλώσει ότι τα δεδομένα που μεταδίδονται δεν κρυπτογραφούνται. Αυτό συμβαίνει κυρίως σε χώρες όπου ισχύουν οι περιορισμοί του ITAR (International Traffic in Arms Regulation).

- **A5/1**
  Ο πρώτος κρυπτογραφικός αλγόριθμος που χρησιμοποιήθηκε στο GSM (2G) στην Ευρώπη.

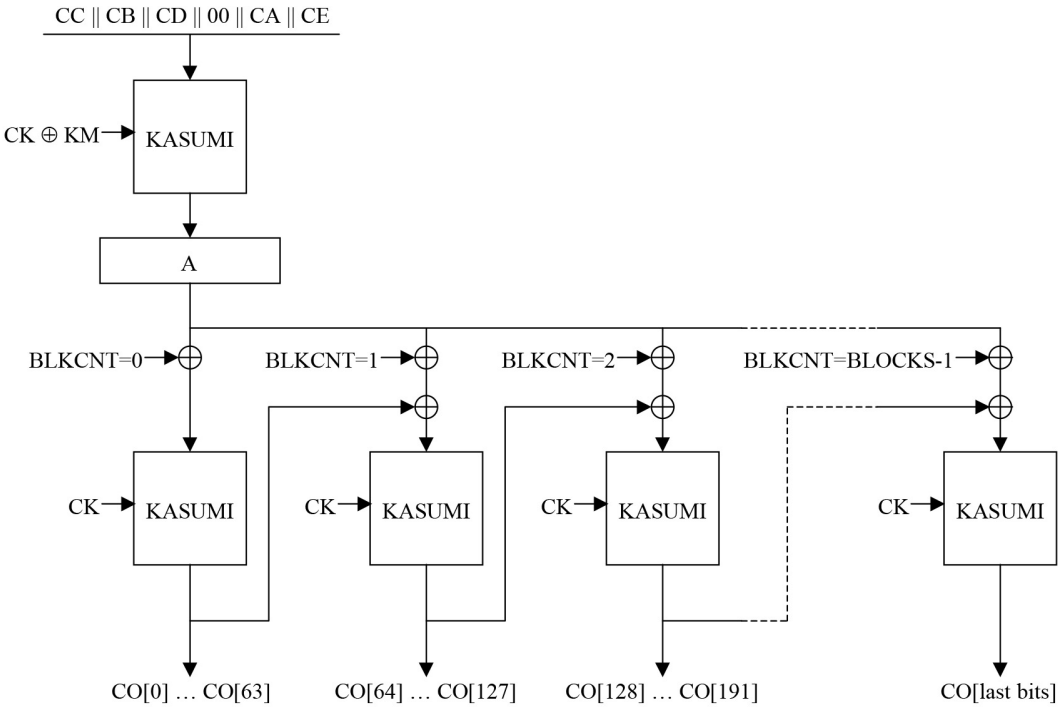
- **A5/2**
  Μια ασθενέστερη εκδοχή του A5/1, που σχεδιάστηκε για να εξαχθεί στις ΗΠΑ.

- **A5/3**
  Η ισχυρότερη εκδοχή του A5. Σχεδιάστηκε κατά τη φάση ανάπτυξης των προδιαγραφών για το UMTS (3G), υπό την αιγίδα της 3GPP, με σκοπό να αντικαταστήσει τον A5/1 στην δεύτερη φάση εξέλιξης του GSM (2.5G).

Στην εν λόγω διπλωματική εργασία, η εφαρμογή που υλοποιείται, στοχεύει στον A5/3. Θα περιγράψουμε αναλυτικά την λειτουργία του A5/3 στις ενότητες που ακολουθούν. Οι υπόλοιπες εκδοχές του A5 δεν θα μας απασχολήσουν περαιτέρω σε αυτή την εργασία, οπότε και δεν θα αναφερθούμε στην λειτουργία τους. Για τον αναγνώστη που ενδιαφέρεται για την λειτουργία τους, προτείνουμε να ανατρέξει στην βιβλιογραφία και συγκεκριμένα στις πηγές [21] και [22].

3.4 Ο Αλγόριθμος A5/3

Ο αλγόριθμος A5/3 είναι ένας κωδικοποιητής ροής (stream cipher), ο οποίος χρησιμοποιεί τον κωδικοποιητή τμημάτων KASUMI ως βασικό δομικό στοιχείο, σε μια συνδεσμολογία τύπου OFB - Output Feedback mode για την παραγωγή κλειδιών ροής (keystreams) προς κρυπτογράφηση και αποκρυπτογράφηση των δεδομένων (βλέπε σχήμα 3.5) [23]. Ακολουθεί η αναλυτική περιγραφή του A5/3. Ο KASUMI περιγράφεται αναλυτικά στην ενότητα 3.5.

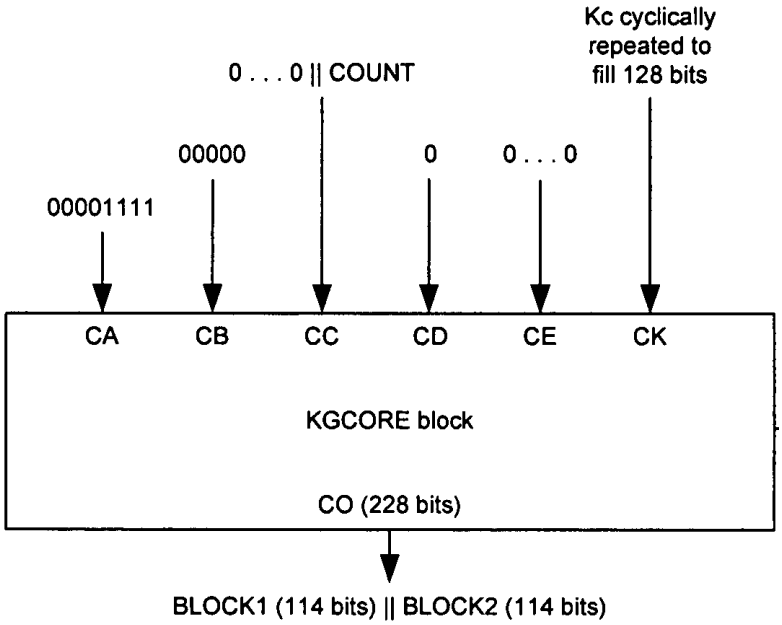


Σχήμα 3.5: Η Συνάρτηση KGCore

Ο A5/3 ορίζεται υπό μια γενικευμένη συνάρτηση που ονομάζεται KGCore, η οποία δέχεται ως εισόδους τις παραμέτρους CA, CB, CC, CD, CE και CK και δίνει ως έξοδο CO το κλειδί ροής (βλέπε σχήμα 3.6). Πρόσθετη είσοδο στην KGCore που δεν εμφανίζεται στα σχήματα, αποτελεί η παράμετρος CL, η οποία δηλώνει το μέγεθος του κλειδιού ροής που θέλουμε να παράξουμε. Στην περίπτωση δεδομένων φωνής, που θα μας απασχολήσει στα πλαίσια αυτής της διπλωματικής εργασίας, το μέγεθος του κλειδιού ροής είναι 228-bits, όπου τα πρώτα 114-bits χρησιμοποιούνται για την κρυπτογράφηση και αποκρυπτογράφηση των δεδομένων που αποστέλλονται προς το σταθμό βάσης (uplink), ενώ τα επόμενα 114-bits, ομοίως, για δεδομένα που αποστέλλονται προς τον κινητό σταθμό (downlink). Ο πίνακας 3.1 μας δίνει αντίστοιχα το μέγεθος των παραμέτρων της KGCore.

Πίνακας 3.1: Μέγεθος Εισόδων της KGCore

Παράμετρος	**Μέγεθος**
CA	8 bits CA[0]...CA[7]
CB	5 bits CB[0]...CB[4]
CC	32 bits CC[0]...CC[31]
CD	1 bit CD[0]
CE	16 bits CE[0]...CE[15]
CK	128 bits CK[0]...CK[127]
CL	Ακέραιος απο $1...2^{19}$



Σχήμα 3.6: Είσοδοι και Έξοδος της Συνάρτησης KGCore

Η τιμή της CA καθορίζεται από το είδος των δεδομένων που μεταδίδονται.
Για δεδομένα φωνής έχουμε CA[0]...CA[7] = 00001111.
Για δεδομένα ECSD έχουμε CA[0]...CA[7] = 11110000.
Τέλος, για δεδομένα GPRS έχουμε CA[0]...CA[7] = 11111111.

Η τιμή της CC, για δεδομένα φωνής και ECSD, παίρνει μηδενικές τιμές για τα πρώτα 10 bits, ενώ η τιμή των τελευταίων 22 bits καθορίζεται από τον αριθμό του πλαισίου μετάδοσης, μια παράμετρος που ονομάζεται COUNT. Για την περίπτωση δεδομένων GPRS, καί τα 32 bits της CC καθορίζονται από τον αριθμό του πλαισίου μετάδοσης. Τότε η παράμετρος ονομάζεται INPUT.

Η CD, για δεδομένα φωνής και ECSD, παίρνει μηδενική τιμή, ενώ για δεδομένα GPRS παίρνει τιμή ανάλογα με την κατεύθυνση μετάδοσης (uplink/downlink).

Η CK φέρει το κλειδί συνεδρίας K_c. Στην περίπτωση του GSM που αυτό έχει μέγεδος 64 bits, το κλειδί επαναλαμβάνεται κυκλικά ώστε να επιμηκυνθεί στο μέγεθος των 128 bits, που είναι και το μέγεθος της παραμέτρου CK. Οπότε και έχουμε

CK[0]...CK[127] = $K_c||K_c$.

Οι παράμετροι CB και CE, για όλες τις περιπτώσεις, παίρνουν μηδενική τιμή.

Αρχικοποίηση

Στο πρώτο στάδιο του αλγορίθμου, παράγεται ένα **διάνυσμα αρχικοποίησης** (**Initialisation Vector - IV**) το οποίο αποθηκεύεται στον καταχωρητή A, μεγέθους 64 bit, όσο και το μέγεθος του τμήματος που δέχεται ως είσοδο και παράγει ως έξοδο ο KASUMI (βλέπε σχ. 3.5). Για να γίνει αυτό, θέτει στον μετατροπέα κλειδιού **KM** την τιμή 0x55555555555555555555555555555555 και εφαρμόζει την πράξη $KM = CK \oplus KM$. Το αποτέλεσμα αποτελεί ένα τροποποιημένο κλειδί που θα χρησιμοποιηθεί για την κρυπτογράφηση της εισόδου σε ένα τμήμα του KASUMI. Η είσοδος στον KASUMI ορίζεται να είναι η ποσότητα CC||CB||CD||00||CA||CE και η έξοδος αποτελεί το διάνυσμα αρχικοποίησης που αποθηκεύεται στον καταχωρητή A. Τέλος, η παράμετρος BLKCNT που μετρά τον αριθμό των τμημάτων που θα παραχθούν για το κλειδί ροής, ορίζεται σε μηδενική τιμή (βλέπε σχήμα 3.5).

Παραγωγή Κλειδιού Ροής

Εφόσον ολοκληρωθεί το στάδιο της αρχικοποίησης, ο αλγόριθμος περνάει στο στάδιο παραγωγής του κλειδιού ροής. Το μήκος του κλειδιού που θα παραχθεί καθορίζεται από την τιμή της παραμέτρου CL και παράγεται ανά τμήματα μεγέθους 64 bits. Στην περίπτωση που η τιμή της CL δεν είναι πολλαπλάσιο των 64 bits, τα bits που περισσεύουν απλά «πετιούνται».

Για να ξεκινήσει η διαδικασία παραγωγής του κλειδιού ροής, ορίζεται ο αριθμός των συνολικών τμημάτων (BLOCKS) που πρέπει να παραχθούν στην τιμή CL/64 στρογγυλοποιημένη προς τα πάνω στον πλησιέστερο ακέραιο αριθμό. Εν συνεχεία, για κάθε τμήμα (**KSB**) που πρέπει να παραχθεί και για $1 \leq n \leq KSB$, εκτελείται η πράξη

$$KSB_n = KASUMI[A \oplus BLKCNT \oplus KSB_{n-1}]_{CK}$$

με το BLKCNT να παίρνει τιμή ίση με n κάθε φορά. Όταν ολοκληρωθεί η διαδικασία αυτή, τα επιμέρους bits των τμημάτων, με τη σειρά που αυτά παράχθηκαν, αποτελούν την έξοδο CO του αλγορίθμου, από όπου θα διατηρηθούν μόνο τα πρώτα CL bits. Αυτά χωρίζονται κατόπιν σε δυο ισομεγέθη blocks, τα BLOCK1 και BLOCK2, μεγέθους 114 bits το καθένα.

Για την κρυπτογράφηση και αποκρυπτογράφηση των δεδομένων που μεταδίδονται προς στον σταθμό βάσης (uplink) χρησιμοποιείται το BLOCK1, ενώ αντίστοιχα, για δεδομένα που μεταδίδονται προς τον κινητό σταθμό (downlink) χρησιμοποιείται το BLOCK2. Η διαδικασία της κρυπτογράφησης και αποκρυπτογράφησης εκτελείται με την εφαρμογή μιας πράξης XOR ανάμεσα στα δεδομένα και το αντίστοιχο BLOCK, ανά bit.

3.5 Ο Κωδικοποιητής Τμημάτων KASUMI (KASUMI Block Cipher)

Ο KASUMI είναι ένας κωδικοποιητής τμημάτων που σχεδιάστηκε κατά τις διεργασίες ανάπτυξης του δικτύου κινητής τηλεφωνίας τρίτης γενιάς (UMTS) από την 3GPP, προκειμένου να αποτελέσει την καρδιά των αλγορίθμων για την εμπιστευτικότητα (confidentiality - f8) και την ακεραιότητα (integrity - f9) στο UMTS δίκτυο [24]. Η σχεδίασή του βασίστηκε στον κωδικοποιητή τμημάτων MYSTY 1 [25]. Αργότερα υιοθετήθηκε για χρήση και στον αλγόριθμο A5/3.

Ο KASUMI είναι σχεδιασμένος με βάση τη δομή Feistel και παράγει μια έξοδο μεγέθους 64 bits από μια είσοδο 64 bits υπό τον έλεγχο ενός κλειδιού μήκους 128 bits σε μια σειρά από οκτώ κύκλους (rounds). Η λειτουργία του βασίζεται σε τρεις υποσυναρτήσεις, τις **FL, FO** και **FI** και κάθε μία από αυτές χρησιμοποιείται σε συνδυασμό με μια σειρά από υποκλειδιά, τα **KL, KO** και **KI** αντίστοιχα. Οι συναρτήσεις FL και FO έχουν ως είσοδο μια ποσότητα μεγέθους 32 bits. Η συνάρτηση FO ορίζεται ως ένα δίκτυο που εφαρμόζει τρεις φορές τη συνάρτηση FI. Η συνάρτηση FI έχει ως είσοδο μια ποσότητα μεγέθους 16 bits. Σε κάθε εφαρμογή της η FI, εφαρμόζει δυο **λειτουργίες αντικατάστασης** που ονομάζονται **S9** και άλλες δυο που ονομάζονται **S7**. Κατά αυτόν τον τρόπο, ο KASUMI έχει μια φωλιασμένη δομή τριών επιπέδων, όμοια με αυτή του MYSTY 1 [26, εν. 7.4.1]. Μια εικόνα αυτής της δομής βλέπουμε στο σχήμα 3.7.

3.5.1 Το Εξωτερικό Δίκτυο

Ο KASUMI επενεργεί στα δεδομένα διαιρώντας την είσοδο σε δυο ποσότητες των 32 bits, τις L_0 και R_0, όπου

$$INPUT = L_0 \| R_0 \tag{3.5.1}$$

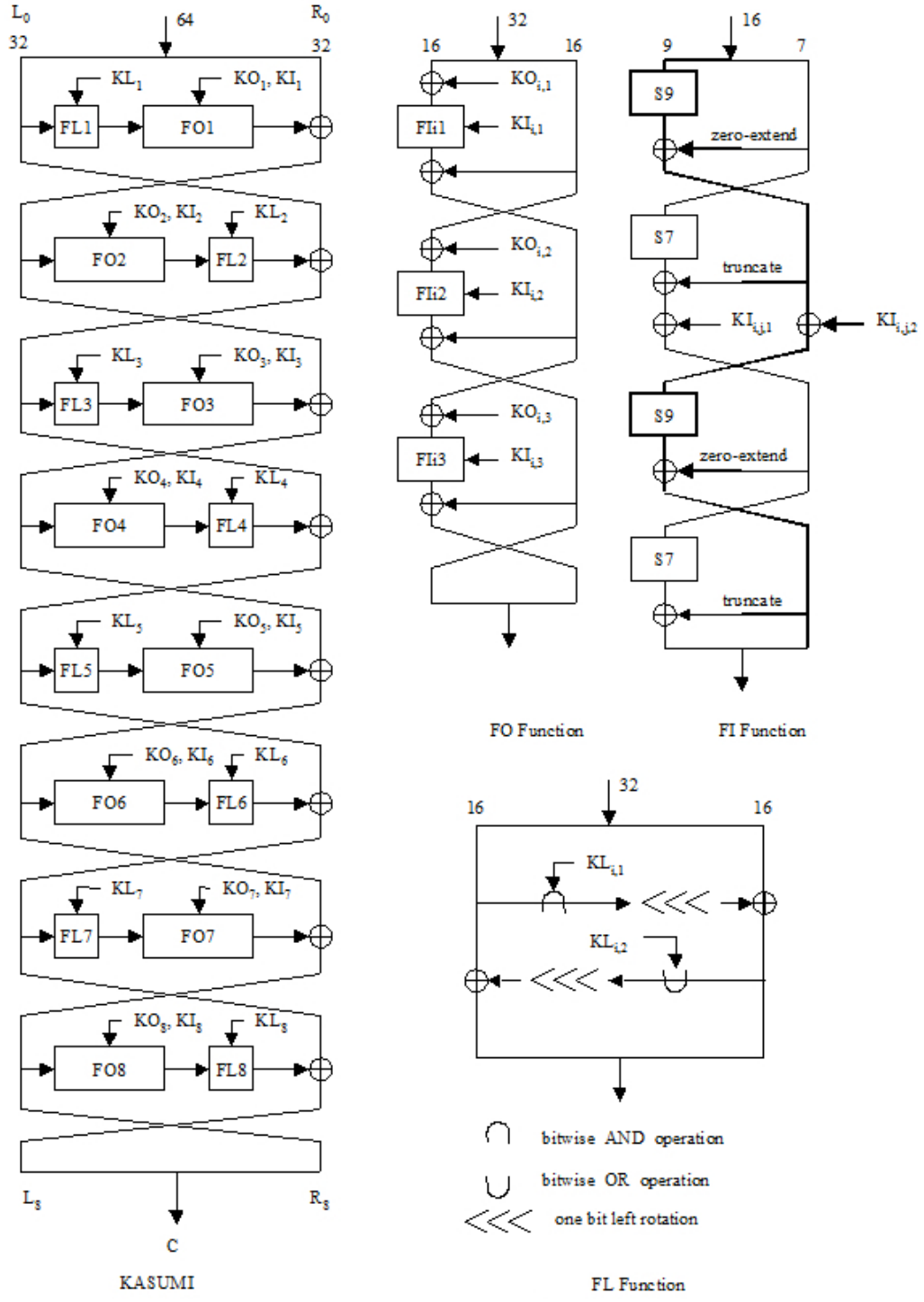
Για κάθε κύκλο i, με $1 \leq i \leq 8$ ορίζονται οι εξής πράξεις:

$$R_i = L_{i-1} \tag{3.5.2}$$

$$L_i = R_{i-1} \oplus f_i\left(L_{i-1}, RK_i\right) \tag{3.5.3}$$

όπου η ποσότητα $L_{i-1}||R_{i-1}$ αποτελεί το τμήμα εισόδου, η ποσότητα $L_i||R_i$ αποτελεί το τμήμα εξόδου και η παράμετρος RK_i το κλειδί του i-οστου κύκλου, ορισμένο ως μια τριπλέτα από υποκλειδιά (KL_i, KO_i, KI_i). Τα υποκλειδιά παράγονται από το κλειδί CK με χρήση ενός αλγόριθμου παραγωγής υποκλειδιών, την λειτουργία του οποίου θα εξηγήσουμε στην υποενότητα 3.5.5. Η τελική έξοδος του KASUMI μετά και τον όγδοο κύκλο ορίζεται ως

$$OUTPUT = L_8 || R_8 \tag{3.5.4}$$



Σχήμα 3.7: Ο Κωδικοποιητής Τμημάτων KASUMI

Στην εξίσωση (3.5.3), η συνάρτηση f για μονούς κύκλους, δηλαδή για $i = 1, 3, 5, 7$ ορίζεται ως

$$f_i\left(I, RK_i\right) = FO_i\left(FL_i\left(I, KL_i\right), KO_i, KL_i\right) \tag{3.5.5}$$

ενώ για ζυγούς κύκλους, δηλαδή για $i = 2, 4, 6, 8$ ορίζεται ως

$$f_i\left(I, RK_i\right) = FL_i\left(FO_i\left(I, KO_i, KI_i\right), KL_i\right) \tag{3.5.6}$$

Συνεπώς, σε κάθε κύκλο, η f_i παίρνει ως είσοδο I μια ποσότητα μεγέθους 32 bits και παράγει ως έξοδο O επίσης μια ποσότητα μεγέθους 32 bits. Χρησιμοποιεί δυο υποσυναρτήσεις, την FL_i σε συνδυασμό με τα υποκλειδιά KL_i και την FO_i σε συνδυασμό με τα υποκλειδιά KO_i και KI_i.

3.5.2 Οι Συναρτήσεις FL

Μια συνάρτηση FL_i παίρνει ως εισόδους ένα τμήμα δεδομένων I μεγέθους 32 bits και ένα υποκλειδί KL_i μεγέθους επίσης 32 bits. Το υποκλειδί KL_i διασπάται σε δυο ποσότητες των 16 bits, τα $KL_{i,1}$ και $KL_{i,2}$, όπου

$$KL_i = KL_{i,1}||KL_{i,2} \tag{3.5.7}$$

Τα δεομένα εισόδου I διασπόνται επίσης σε δυο ποσότητες των 16 bits, τις L και R, με

$$I = L||R \tag{3.5.8}$$

Σε κάθε χρήση της, μια FL εφαρμόζει τις εξής πράξεις:

- $\mathbf{ROL}\left(\mathbf{D}\right)$ κυκλική ολίσθηση προς τα αριστερά κατά ένα bit ενός τμήματος D
- $\mathbf{D_1} \cup \mathbf{D_2}$ η λογική πράξη OR ανά bit ανάμεσα σε δυο τμήματα D_1 και D_2
- $\mathbf{D_1} \cap \mathbf{D_2}$ η λογική πράξη AND ανά bit ανάμεσα σε δυο τμήματα D_1 και D_2

Με βάση τα παραπάνω, η έξοδος μεγέθους 32 bits για κάθε FL ορίζεται ως $L'||R'$ όπου:

$$L' = L \oplus ROL\left(R' \cup KL_{i,2}\right) \tag{3.5.9}$$

$$R' = R \oplus ROL\left(L \cap KL_{i,1}\right) \tag{3.5.10}$$

3.5.3 Οι Συναρτήσεις FO

Μια συνάρτηση FO_i παίρνει ως εισόδους ένα τμήμα δεδομένων I μεγέθους 32 bits και δυο υποκλειδιά μεγέθους 48 bits το καθένα, τα KO_i και KI_i. Το τμήμα δεδομένων I διασπάται σε δυο υποτμήματα των 16 bits, τα L_0 και R_0 όπου $I = L_0||R_0$, ενώ τα υποκλειδιά διασπώνται το καθένα σε τρία υποτμήματα των 16 bits ως ακολούθως:

$$KO_i = KO_{i,1}||KO_{i,2}||KO_{i,3} \text{ και } KI_i = KI_{i,1}||KI_{i,2}||KI_{i,3}$$

Τότε, για κάθε ακέραιο j με $1 \leq j \leq 3$ για κάθε j-οστη εφαρμογή της συνάρτησης FO_i ορίζονται οι πράξεις:

$$R_j = FI_{i,j}\left(L_{j-1} \oplus KO_{i,j}, KI_{i,j}\right) \oplus R_{j-1} \tag{3.5.11}$$

$$L_j = R_{j-1} \tag{3.5.12}$$

Την έξοδο της συνάρτησης FO_i αποτελεί το τμήμα μεγέθους 32 bits $L_3||R_3$.

3.5.4 Οι Συναρτήσεις FI

Οι συναρτήσεις $FI_{i,j}$ παίρνουν ως εισόδους ένα τμήμα δεδομένων I μεγέθους 16 bits και ένα υποκλειδί $KI_{i,j}$ επίσης μεγέθους 16 bits. Το τμήμα δεδομένων I διασπάται σε δυο άνισα υποτμήματα, ένα εξ αριστερών μεγέθους 9 bits, το L_0 και ένα εκ δεξιών μεγέθους 7 bits, το R_0. Με όμοιο τρόπο, το υποκλειδί $KI_{i,j}$ διασπάτε και αυτό σε δυο άνισα υποτμήματα, ένα μεγέθους 7 bits, το $KI_{i,j,1}$ και ένα μεγέθους 9 bits, το $KI_{i,j,2}$, όπου $KI_{i,j} = KI_{i,j,1}||KI_{i,j,2}$.

Κάθε συνάρτηση $FI_{i,j}$ χρησιμοποιεί δυο κουτιά αντικατάστασης (S-Boxes), τα S7 και S9. Το S7 αντικαθιστά μια είσοδο μεγέθους 7 bits με μια άλλη ποσότητα των 7 bits και το S9 αντικαθιστά μια είσοδο μεγέθους 9 bits με μια άλλη ποσότητα των 9 bits. Η ακριβής λειτουργία των S-Boxes περιγράφεται στο τεχνικό δελτίο της 3GPP για τον KASUMI [24]. Εμείς στην εφαρμογή μας τους χρησιμοποιούμε υπό τη μορφή πινάκων αντικατάστασης, τους οποίους διαθέτει η 3GPP στο εν λόγω τεχνικό δελτίο.

Τέλος, μια συνάρτηση $FI_{i,j}$ χρησιμοποιεί τις ακόλουθες πράξεις:

- **ZE (D)** επιμηκύνει μια είσοδο D των 7 bits σε 9 bits προσθέτοντας δυο μηδενικά bit στο αριστερό άκρο της D
- **TR (D)** συρρικνώνει μια είσοδο D των 9 bits σε 7 bits αφαιρώντας δυο bit από το αριστερό άκρο της D

Με βάση τα παραπάνω, η λειτουργία της συνάρτησης $FI_{i,j}$ ορίζεται από μια σειρά πράξεων οι οποίες είναι οι:

$$\begin{array}{ll} L_1 = R_0 & R_1 = S9[L_0] \oplus ZE\,(R_0) \\ L_2 = R_1 \oplus KI_{i,j,2} & R_2 = S7[L_1] \oplus TR\,(R_1) \oplus KI_{i,j,1} \\ L_3 = R_2 & R_3 = S9[L_2] \oplus ZE\,(R_2) \\ L_4 = S7[L_3] \oplus TR\,(R_3) & R_4 = R_3 \end{array}$$

Την έξοδο μιας συνάρτησης $FI_{i,j}$ αποτελεί το υποτμήμα μεγέθους 16 bits $L_4||R_4$.

3.5.5 Παραγωγή Υποκλειδιών (Key-Schedule)

Σε κάθε κύκλο του, ο KASUMI χρησιμοποιεί ένα διαφορετικό κλειδί μεγέθους 128 bits τα οποία παράγονται από το κλειδί CK και χρησιμοποιούνται για την παραγωγή των υποκλειδιών. Η όλη διαδικασία περιγράφεται στη συνέχεια.

Ορίζεται ένα κλειδί K ίδιο με το κλειδί CK. Το κλειδί K διασπάται σε οκτώ υποτμήματα των 16 bits ως εξής:

$$K = K_1||K_2||K_3||K_4||K_5||K_6||K_7||K_8$$

Ακόμη ορίζονται 8 νέα υποτμήματα των 16 bits, τα $K'_1...K'_8$. Κάθε ένα από τα K'_j με $1 \leq j \leq 8$ παράγεται από τα $\mathrm{K_j}$ και μια σειρά από σταθερές $\mathrm{C_j}$, όπως αυτές απεικονίζονται στον πίνακα 3.2, με βάση τη συνάρτηση

$$K'_j = K_j \oplus C_j$$

.

Πίνακας 3.2: Σταθερές C_j

C_1	0x0123
C_2	0x4567
C_3	0x89AB
C_4	0xCDEF
C_5	0xFEDC
C_6	0xBA98
C_7	0x7654
C_8	0x3210

Κατόπιν, τα υποκλειδιά KL, KO, KI παράγονται σύμφωνα με τον πίνακα 3.3, όπου η πράξη $D \ll n$ ορίζει την κυκλική ολίσθηση προς αριστερά του υποτμήματος D κατά n bits.

Πίνακας 3.3: Υπολογισμός Υποκλειδιών

	1	2	3	4	5	6	7	8
$KL_{i,1}$	$K_1 \ll 1$	$K_2 \ll 1$	$K_3 \ll 1$	$K_4 \ll 1$	$K_5 \ll 1$	$K_6 \ll 1$	$K_7 \ll 1$	$K_8 \ll 1$
$KL_{i,2}$	K'_3	K'_4	K'_5	K'_6	K'_7	K'_8	K'_1	K'_2
$KO_{i,1}$	$K_2 \ll 5$	$K_3 \ll 5$	$K_4 \ll 5$	$K_5 \ll 5$	$K_6 \ll 5$	$K_7 \ll 5$	$K_8 \ll 5$	$K_1 \ll 5$
$KO_{i,2}$	$K_6 \ll 8$	$K_7 \ll 8$	$K_8 \ll 8$	$K_1 \ll 8$	$K_2 \ll 8$	$K_3 \ll 8$	$K_4 \ll 8$	$K_5 \ll 8$
$KO_{i,3}$	$K_7 \ll 13$	$K_8 \ll 13$	$K_1 \ll 13$	$K_2 \ll 13$	$K_3 \ll 13$	$K_4 \ll 13$	$K_5 \ll 13$	$K_6 \ll 13$
$KI_{i,1}$	K'_5	K'_6	K'_7	K'_8	K'_1	K'_2	K'_3	K'_4
$KI_{i,2}$	K'_4	K'_5	K'_6	K'_7	K'_8	K'_1	$K'24$	K'_3
$KI_{i,3}$	K'_8	K'_1	K'_2	K'_3	K'_4	K'_5	K'_6	K'_7

Κεφάλαιο 4

CUDA

Τον Νοέμβριο του 2006 η NVIDIA λανσάρει την πρώτη κάρτα γραφικών που υποστηρίζει το DirectX 10 της Microsoft, την GeForce 8800 GTX, η οποία αποτελεί και την πρώτη κάρτα γραφικών που είναι σχεδιασμένη με την αρχιτεκτονική CUDA. Μια αρχιτεκτονική που στοχεύει στο να απλοποιήσει την χρήση των καρτών γραφικών για προγραμματισμό γενικού σκοπού (General Purpose - GPU Computing). [27, σελ.6]

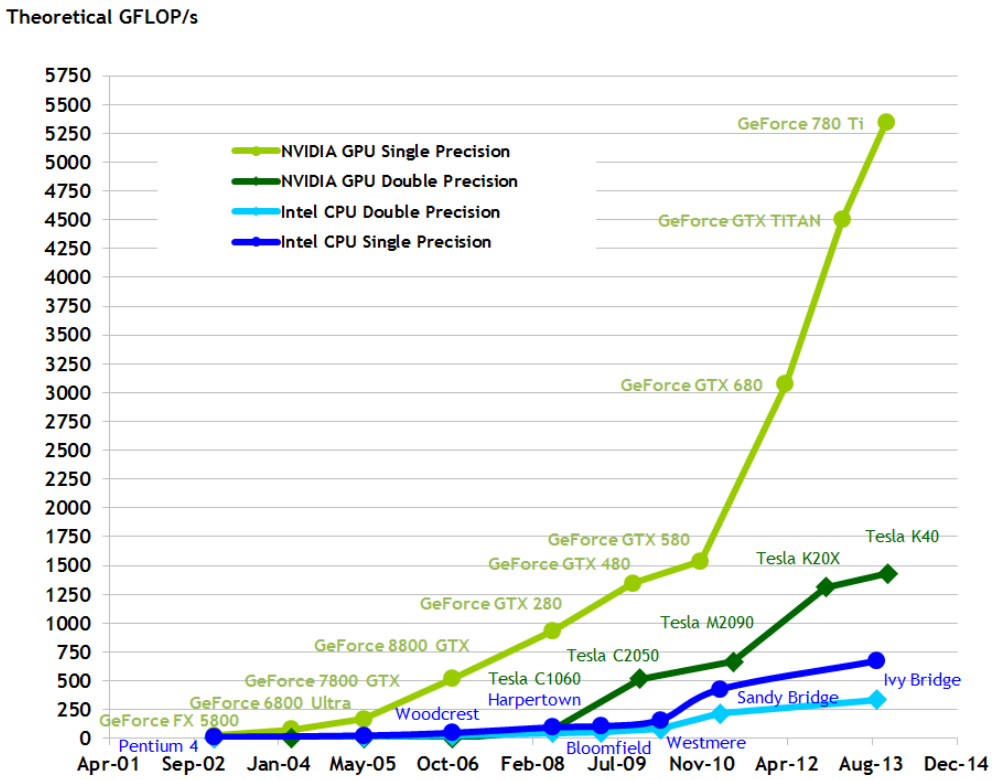
4.1 Εισαγωγή στην CUDA

Η CUDA (Compute Unified Device Architecture) είναι μια παράλληλη αρχιτεκτονική καρτών γραφικών σχεδιασμένη από την NVIDIA. Ταυτόχρονα αποτελεί ένα προγραμματιστικό μοντέλο, που με κατάλληλες **εφαρμογές διεπαφής (Application Interface - API)** και έναν **μεταγλωττιστή (Compiler)**, επιτρέπει στον προγραμματιστή να σχεδιάζει εφαρμογές γενικού σκοπού που να μπορούν να εκτελούνται εντός της κάρτας γραφικών εν παραλλήλω (**GP-GPU Programming**), χρησιμοποιώντας γλώσσες προγραμματισμού υψηλού επιπέδου όπως C, C++ και Fortran. Όλες οι εφαρμογές διεπαφής, μαζί με τον μεταγλωττιστή και τα απαραίτητα εργαλεία για τον προγραμματισμό, είναι δωρεάν διαθέσιμα από την NVIDIA μέσω ενός πακέτου εγκατάστασης, το **CUDA Toolkit**, σε εκδόσεις συμβατές με τα περισσότερα σύγχρονα λειτουργικά συστήματα, μέσω της σελίδας της [28].

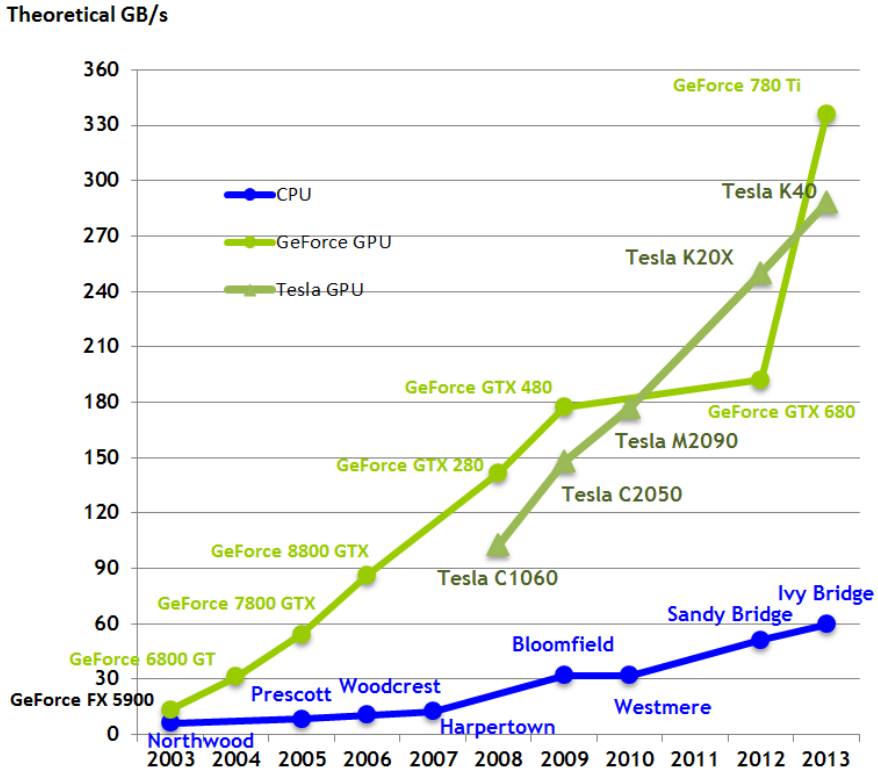
Οι συνεχείς απαιτήσεις της αγοράς για απόδοση υψηλής ποιότητας τρισδιάστατων γραφικών σε πραγματικό χρόνο (βασικό χαρακτηριστικό της βιομηχανίας παιχνιδιών ηλεκτρονικών υπολογιστών για ρεαλιστική απεικόνιση), αποτέλεσαν μοχλό ώστε ο **γραφικός επεξεργαστής (Programmable Graphic Processor - GPU)** να εξελιχθεί σε έναν πολυπύρηνο, πολυνηματικό επεξεργαστή υψηλού παραλληλισμού. Πρόσθετα χαρακτηριστικά του αποτελούν η δραματικά υψηλότερη υπολογιστική του ισχύς και το υψηλότερο εύρος μνήμης του (Memory Bandwidth), έναντι των κοινών πολυπύρηνων επεξεργαστών (Multicore CPU's). Χαρακτηριστικά που πηγάζουν από το γεγονός ότι είναι κατασκευασμένος για παράλληλη επεξεργασία μεγάλου όγκου δεδομένων και έτσι, η ανάγκη ύπαρξης δομών ελέγχου ροής κ.α. εντός του ολοκληρωμένου, που αυξάνουν την πολυπλοκότητα κατασκευής και μειώνουν την υπολογιστική απόδοση, είναι πολύ μικρότερη. Αυτό μας δείχνει η εικόνα του σχήματος 4.1.[29]

Σχήμα 4.1: Η GPU αφιερώνει περισσότερα τρανζίστορ για επεξεργασία δεδομένων[29]

Το διάγραμμα του σχήματος 4.2 μας δείχνει την αύξηση της υπολογιστικής ισχύος των καρτών γραφικών ανά γενεά σε σύγκριση με τις αντίστοιχες ανά περίοδο γενεές επεξεργαστών. Στο διάγραμμα του σχήματος 4.3, σε μια παρόμοια σύγκριση, βλέπουμε την αύξηση του εύρους μνήμης για κάρτες γραφικών και επεξεργαστές αντίστοιχα.



Σχήμα 4.2: Υπολογιστική Ισχύς για CPU και GPU [29]

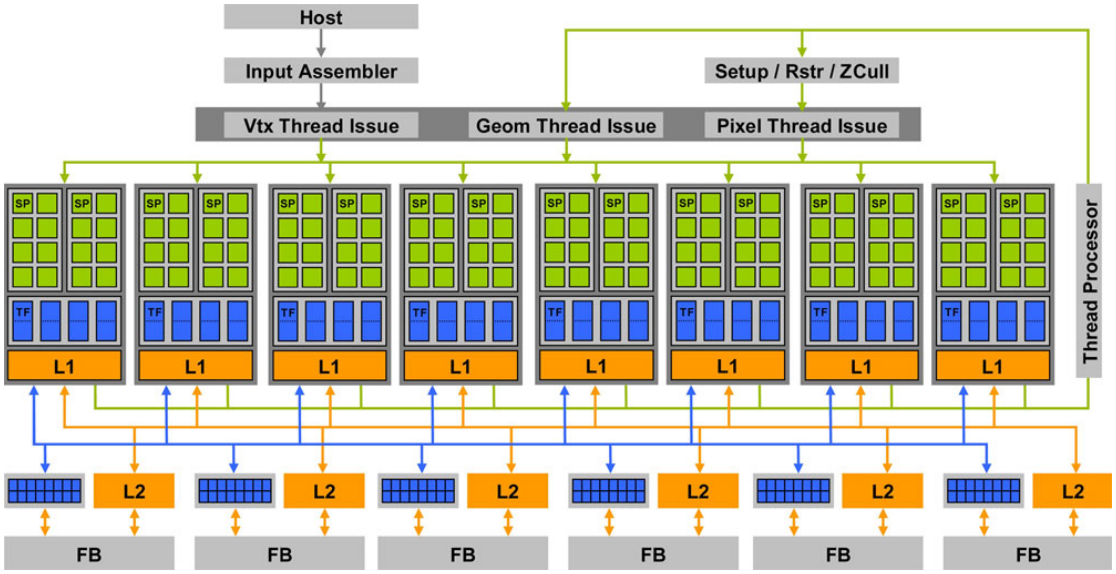


Σχήμα 4.3: Εύρος Μνήμης για CPU και GPU [29]

4.2 Η Αρχιτεκτονική

Όπως αναφέρουμε και στην εισαγωγή του κεφαλαίου 4, η Geforce 8800 GTX ήταν η πρώτη κάρτα γραφικών σχεδιασμένη από την NVIDIA με βάση την αρχιτεκτονική CUDA. Αποτέλεσε την πρώτη γενεά αρχιτεκτονικής CUDA και φέρει την κωδική ονομασία **Tesla**. Ακολούθησε η γενεά **Fermi** με εκτενείς βελτιώσεις στην ιεραρχία μνήμης καθώς και αύξηση του αριθμού των μικροεπεξεργαστών, του αριθμού των νημάτων που μπορεί να εκτελέσει ταυτόχρονα, της υπολογιστικής ισχύος, του μεγέθους μνήμης αλλά και του εύρους και της ταχύτητας του ρολογιού της μνήμης. Την αρχιτεκτονική Fermi διαδέχτηκε αργότερα η αρχιτεκτονική **Kepler**. Τη χρονική στιγμή της συγγραφής αυτού του κειμένου, η τρέχουσα αρχιτεκτονική είναι η **Maxwell**.

Το σχήμα 4.4 περιλαμβάνει το διάγραμμα της αρχιτεκτονικής της Geforce 8800 GT, σχεδιασμένη και αυτή με βάση την αρχιτεκτονική Tesla. Όπως μπορούμε να δούμε, είναι δομημένη ως μια κλιμακωτή συστοιχία από πολυεπεξεργαστές ροής (**Streaming Multiprocessors - SMs**), καθένας από τους οποίους απαρτίζει έναν αριθμό από μικροεπεξεργαστές ροής (**Streaming Processors**) ή αλλιώς, πυρήνες (**Cores**).[30, Κεφάλαιο 3] Κάθε SM είναι κατάλληλα σχεδιασμένος ώστε να μπορεί να εκτελεί εκατοντάδες νήματα (**threads**) ταυτόχρονα. Θα αναφερθούμε αναλυτικά σε αυτό στην ενότητα 4.3, όπου θα μιλήσουμε για το προγραμματιστικό μοντέλο.

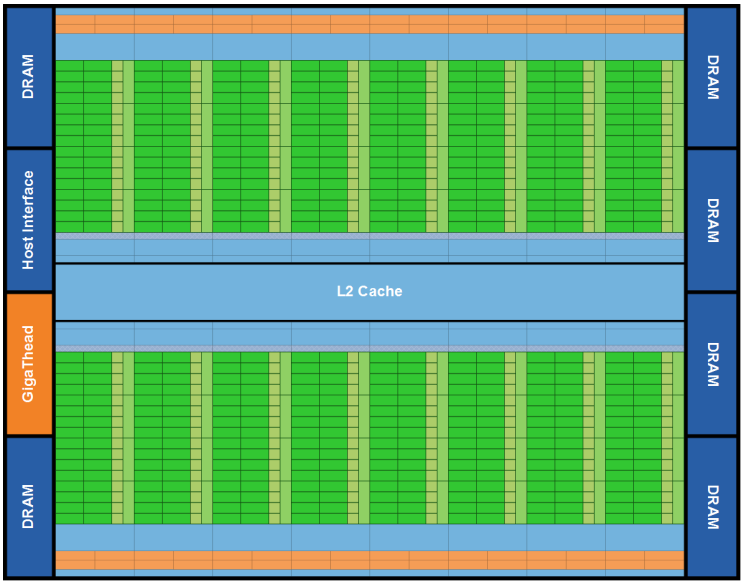


Σχήμα 4.4: Διάγραμμα της GeForce 8800 GT

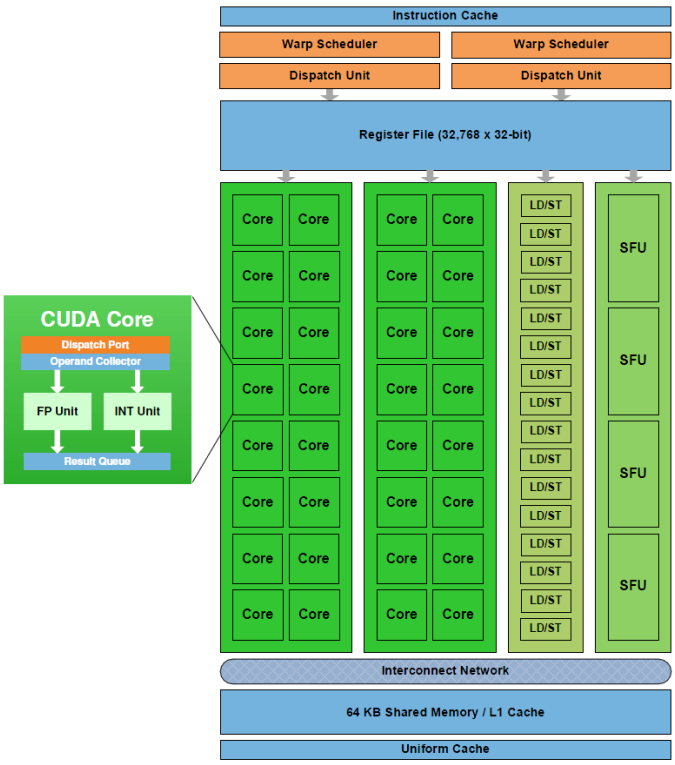
Στην παρούσα φάση που συντάσσεται αυτό το κείμενο, στο Εργαστήριο Μικροεπεξεργαστών και Υλικού του Πολυτεχνείου Κρήτης, διαθέτουμε έναν υπολογιστή εξοπλισμένο με μια κάρτα γραφικών GeForce GTX 580 της NVIDIA, σχεδιασμένη με βάση την αρχιτεκτονική Fermi, στην οποία και έχει εκπονηθεί η εν λόγω διπλωματική εργασία. Στο σχήμα 4.5 απεικονίζεται το διάγραμμα της αρχιτεκτονικής Fermi. Όπως βλέπουμε, και εδώ η σχεδίαση ακολουθεί την κλιμακωτή δομή από συστοιχίες των SMs. Αυτός ήταν και ο στόχος της NVIDIA άλλωστε: μια σχεδίαση με επαναλαμβανόμενες διασυνδεδεμένες βασικές δομές, που όσο η τεχνολογία βελτιώνεται, να αυξάνεται ο αριθμός των βασικών δομών που μπορούν να χωρέσουν μέσα στον γραφικό επεξεργαστή. Αυτό δίνει τη δυνατότητα, εφαρμογές που έχουν γραφτεί για παλαιότερες γενεές αρχιτεκτονικής, να είναι συμβατές με τις νεώτερες γενεές, αλλά και να επωφελούνται από τον αυξημένο αριθμό από SMs, καθώς θα μπορούν να εκτελούνται επιτυγχάνοντας καλύτερες αποδόσεις και χρόνους εκτέλεσης, χωρίς περαιτέρω τροποποιήσεις στον κώδικα.

Μία Fermi GPU διαθέτει 512 Cores, μοιρασμένους σε 16 SMs των 32 Cores και μπορεί να υποστηρίξει έως 6GB GDDR5 DRAM μνήμη. Στο σχήμα 4.6 παρατηρούμε σε μικρογραφία την δομή ενός SM μιας Fermi GPU. Κάθε SP ενός SM περιέχει μια **arithmetic logic unit (ALU)** και μια **floating point unit (FPU)**, ενώ για κάθε SM διατίθενται τέσσερις **special function units (SFU)** οι οποίες αναλαμβάνουν πολύπλοκους υπολογισμούς, όπως τριγωνομετρικοί αριθμοί, υπολογισμός ρίζας και άλλους παρόμοιους.

Προκειμένου να μπορεί να υπάρχει διάκριση των ιδιοτήτων της GPU από γενεά σε γενεά, ορίζεται το **Compute Capability (x,y)**, ως ένας αριθμός που δηλώνει μια μεγάλη έκδοση (κομμάτι x) και μια μικρή έκδοση (κομμάτι y). GPUs που φέρουν την ίδια μεγάλη έκδοση, ανήκουν στην ίδια γενεά αρχιτεκτονικής. Η μικρή έκδοση χαρακτηρίζει βελτιώσεις σε μια αρχιτεκτονική, που μπορεί με τη σειρά τους να φέρουν και πρόσθετες ιδιότητες. Η GeForce GTX 580 που έχουμε στην διάθεσή μας είναι Compute Capability 2.0.



Σχήμα 4.5: Διάγραμμα Αρχιτεκτονικής Fermi [31]

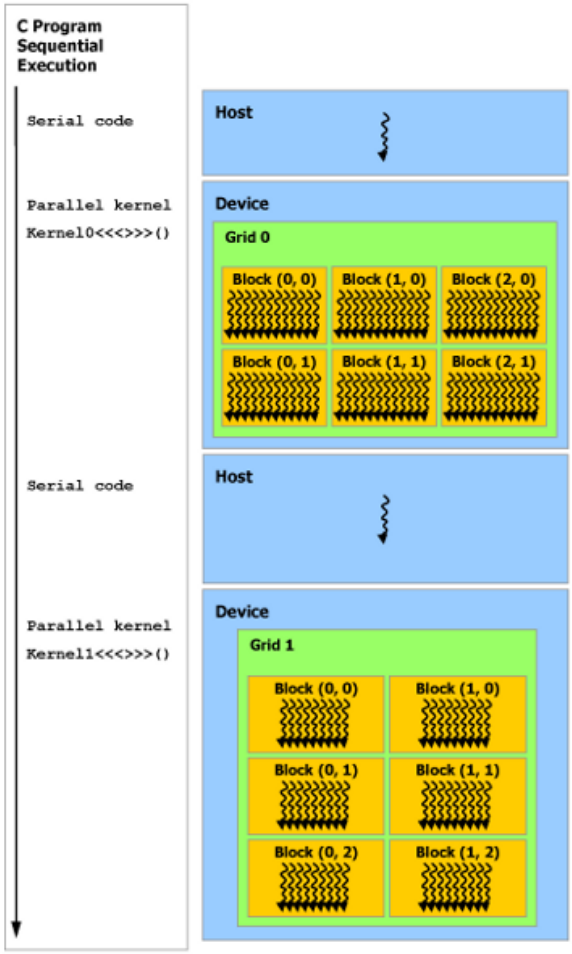


Σχήμα 4.6: Διάγραμμα SM Fermi [31]

4.3 Το Προγραμματιστικό Μοντέλο

Αν δούμε την GPU ως μια πλατφόρμα παράλληλου προγραμματισμού, βλέπουμε αμέσως ότι δεν πρόκειται για μια αυτόνομη μονάδα. Θεωρείται δεδομένο ότι πρόκειται για ένα υποσύστημα, το ονομάζουμε **Device**, που είναι προσαρτημένο σε έναν

ηλεκτρονικό υπολογιστή, τον ονομάζουμε **Host**, και μέσω μιας εφαρμογής που εκτελείται από τον Host, αναθέτουμε στο Device εκείνα τα κομμάτια της εφαρμογής που μας ενδιαφέρει να εκτελεστούν εν παραλλήλω από αυτό. Ενόσω το Device είναι απασχολημένο εκτελώντας μέρη της εφαρμογής μας εν παραλλήλω, ο Host δύναται να συνεχίζει να εκτελεί άλλα ανεξάρτητα τμήματα της εφαρμογής σειριακά, περιμένοντας για τα αποτελέσματα από το Device. Γι΄ αυτό λέγεται ότι πρόκειται για ένα ετερογενές προγραμματιζόμενο σύστημα, όπως βλέπουμε στην εικόνα του σχήματος 4.7.



Σχήμα 4.7: CUDA - Ένα ετερογενές προγραματιζόμενο σύστημα [29]

Όπως αναφέραμε και στην ενότητα 4.1, προκειμένου να μπορούμε να προγραμματίζουμε αυτό το σύστημα, η NVIDIA παρέχει δωρεάν τα απαραίτητα APIs, προκειμένου να σχεδιάζουμε εφαρμογές με τη χρήση των γλωσσών προγραμματισμού C/C++ και Fortran. Επίσης, από τρίτους παρόχους, υπάρχουν διαθέσιμοι wrappers, οι οποίοι μπορούν να μετατρέψουν σε CUDA, κώδικα γραμμένο σε άλλες γλώσσες προγραμματισμού, όπως για παράδειγμα Matlab, Python, java, C# και άλλες.

Χάριν απλοποίησης θα αναλύσουμε μόνο την περίπτωση της γλώσσας προγραμματισμού C, με τη χρήση τη οποίας έχει υλοποιηθεί και η εν λόγω διπλωματική εργασία. Η CUDA επεκτείνει το συντακτικό της C με ένα σετ από δεσμευμένες λέξεις, οι οποίες χρησιμοποιούμενες ως προθέματα στον κώδικα, επιτρέπουν στον

προγραμματιστή να καθορίσει ποια τμήματα αυτού προβλέπονται για εκτέλεση στο Device. Ακόμη, προσφέρει μια σειρά από βιβλιοθήκες συναρτήσεων, με τις οποίες ο προγραμματιστής μπορεί να ορίσει ενέργειες που προβλέπονται για εκτέλεση στο Device. Τέλος, διατίθεται ο nvcc compiler, ο οποίος παράγει από τον κώδικά μας ένα εκτελέσιμο για GPU.

Η Δομή του Κώδικα

Ο κώδικας κάθε εφαρμογής CUDA χωρίζεται σε δυο μέρη, το κομμάτι που εκτελείται στον Host (δηλ. στον CPU) αφ΄ εξής επονομαζόμενο **Host Code**, και το κομμάτι που εκτελείται στο Device, αφ΄ εξής επονομαζόμενο **Device Code**. Κάθε αυτόνομο κομμάτι κώδικα, που υλοποιεί μια συγκεκριμένη εργασία, που θέλουμε να εκτελεστεί στο Device, ονομάζεται **Kernel**. Ένας kernel ορίζεται ως μια συνάρτηση στο κομμάτι του Device Code, με χρήση του προθέματος __global__. Το κομμάτι του Host Code περιλαμβάνει, συνοπτικά, την συνάρτηση main(), από όπου και ξεκινά η εκτέλεση της εφαρμογής, τα τμήματα του κώδικα που θα εκτελεστούν στον Host και κλήσεις προς kernels. Το δε κομμάτι του Device Code περιλαμβάνει τουλάχιστον έναν ή και περισσότερους kernels και ότι άλλες βοηθητικές συναρτήσεις και παράμετροι χρειάζονται για την εκτέλεση αυτών. Συναρτήσεις που τοποθετούνται στο κομμάτι του Device Code, ορίζονται σε καθολικό επίπεδο με χρήση του προθέματος __device__. Για τον ορισμό των παραμέτρων στο κομμάτι του Device Code θα αναφερθούμε αναλυτικά στην ενότητα 4.4.

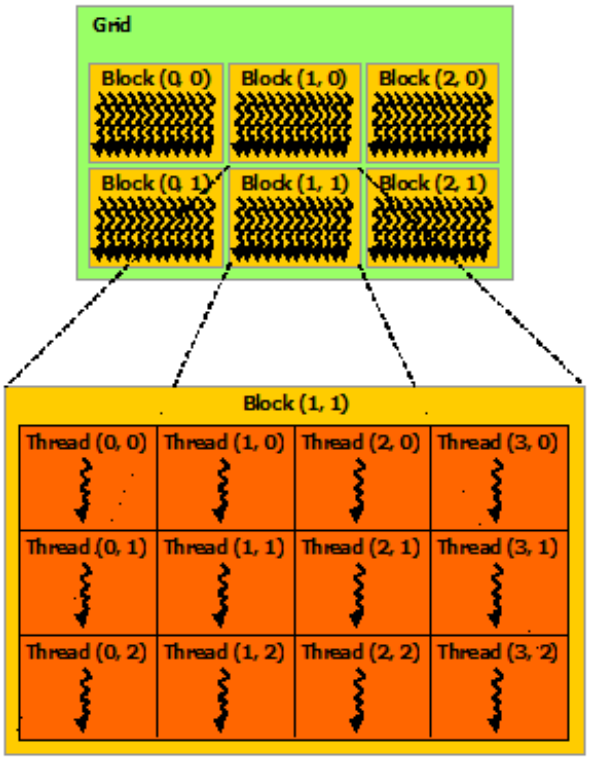
Η συνηθισμένη ροή εκτέλεσης μιας εφαρμογής είναι, ξεκινώντας από τον Host, η δέσμευση απαραίτητου χώρου μνήμης στο Device και η μεταφορά των απαραίτητων δεδομένων σε αυτό (βλ. ενότητα 4.4), η κλήση σε έναν ή περισσότερους kernels διαδοχικά, που θα επεξεργαστούν εν παραλλήλω τα δεδομένα αυτά και, με το πέρας της εκτέλεσης του kernel, η μεταφορά των αποτελεσμάτων από το Device πίσω στο Host.

Μοντέλο Παραλληλισμού

Η CUDA χρησιμοποιεί ένα μοντέλο παραλληλισμού που το ονομάζει **SIMT (Single Instruction Multiple Threads)**. Αυτό προτάσσει την εκτέλεση της ίδιας εντολής ή του ίδιου σετ εντολών, από πολλά, παράλληλα, ανεξάρτητα μεταξύ τους νήματα (**Threads**). Είναι παρόμοιο με το μοντέλο **SIMD (Single Instruction Multiple Data)** που συναντάται στις τεχνολογίες SSE, MMX και AVX των σύγχρονων επεξεργαστών [30, σελ. 6], με τη διαφορά ότι κάθε thread διαθέτει τα προσωπικά του δεδομένα, τους προσωπικούς του καταχωρητές (Registers) και τον προσωπικό του μετρητή εντολών (instuction counter), που του επιτρέπουν να εκτελείται και να διακλαδίζεται αυτόνομα και ανεξάρτητα από τα υπόλοιπα threads. Πράγμα μη εφικτό στο μοντέλο SIMD, όπου συνήθως, ένα διάνυσμα από δεδομένα, μοιράζεται από κοινού τα παραπάνω αναφερθέντα στοιχεία, ανά πυρήνα σε έναν επεξεργαστή.[29]

Κάθε kernel επομένως, αποτελεί ένα σετ εντολών, που θα εκτελεστεί εν παραλλήλω από έναν μεγάλο αριθμό από threads. Τα threads που δημιουργούνται με την κλήση ενός kernel, ομαδοποιούνται σε ένα πλέγμα (**Grid**), το οποίο υποδιαιρείται σε **Blocks** από **threads**. Εντός του κάθε block, τα threads υποδιαιρούνται περαιτέρω

σε ομάδες των 32 που ονομάζονται **Warps** (βλ. εικόνα σχήματος 4.8).



Σχήμα 4.8: Διάταξη των threads [29]

Τα blocks μοιράζονται προς εκτέλεση στους διαθέσιμους SMs, με μέγιστο αριθμό τα 8 blocks ανά SM (ανεξαρτήτως αρχιτεκτονικής), δοθέντος ότι υπάρχουν αρκετοί διαθέσιμοι πόροι για να τα υποστηρίξουν (καταχωρητές, επαρκείς τοπική μνήμη κτλ.). Αν οι πόροι δεν επαρκούν, τότε αυτομάτως, στον κάθε SM θα μοιραστούν λιγότερα Blocks, με τέτοιο τρόπο ώστε αυτά να μπορούν να εκτελεστούν. Στην αρχιτεκτονική Fermi, με τους 16 συνολικά SMs, γίνεται θεωρητικά να εκτελεστούν ταυτόχρονα έως και 128 blocks των 1024 threads το καθένα. Βέβαια, στην πραγματικότητα, αυτός ο αριθμός δεν είναι σχεδόν ποτέ εφικτός, καθότι οι πόροι του κάθε SM, όπως θα αναλύσουμε στην ενότητα 4.4, είναι περιορισμένοι, και έτσι τα threads ανά block αλλά και τα blocks ανά SM είναι συνήθως μικρότερα των θεωρητικών τιμών. Μια συνοπτική εικόνα των διαθέσιμων πόρων ανά compute capability μας δίνει ο πίνακας 4.1. Την ευθύνη για τον σωστό ορισμό του μεγίστου αριθμού των threads που μπορούν να υποστηριχτούν ανά εφαρμογή και πως αυτά μπορούν να υποδιαιρεθούν κατά τον ιδανικότερο τρόπο σε blocks την φέρει ο προγραμματιστής.

Πίνακας 4.1: Διαθέσιμοι Πόροι σε έναν SM ανά Compute Capability [32]

Resources	**1.0**	**1.1**	**1.2**	**1.3**	**2.0**
Number of 32-bit registers per SM	8K	8K	16K	16K	32K
Max. amount of shared memory per SM	16KB	16KB	16KB	16KB	48KB
Amount of local memory per thread	16KB	16KB	16KB	16KB	512KB
Constant memory size	64KB	64KB	64KB	64KB	64KB

Κάθε block εντός του Grid ταυτοποιείται μοναδικά με χρήση της προκαθορισμένης μεταβλητής **blockIdx**, ενώ κάθε thread εντός του κάθε block ταυτοποιείται μονάδικα με την προκαθορισμένη μεταβλητή **threadIdx**. Ο αριθμός των Blocks μέσα σε ένα Grid καθορίζεται από την προκαθορισμένη μεταβλητή **gridDim** και ο αριθμός των threads ανά block καθορίζεται από την προκαθορισμένη μεταβλητή **blockDim**. Οι μεταβλητές gridDim, blockDim, threadIdx και blockIdx είναι δομές των τριών διαστάσεων, που σημαίνει ότι μπορούμε να ορίζουμε blocks της μιας, των δυο ή και των τριών διαστάσεων από threads, καθώς και Grids της μιας οι των δυο διαστάσεων από blocks.

Χρονοπρογραμματισμός των Threads

Όπως μπορούμε να δούμε στην εικόνα του σχήματος 4.6, κάθε SM στην αρχιτεκτονική Fermi διαθέτει δυο χρονοπρογραμματιστές Warp (**Warp schedulers**). Έτσι, σε κάθε SM, μπορούν να εκτελούνται ταυτόχρονα δυο Warp, σε κάθε χρονική στιγμή.

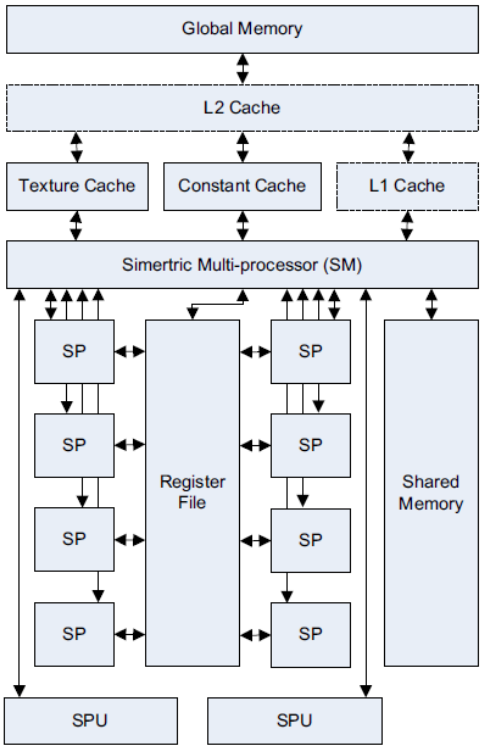
Ο κάθε χρονοπρογραμματιστής διαλέγει ένα Warp που είναι έτοιμο προς εκτέλεση και αναθέτει μια εντολή σε μια συστοιχία των 16 SPs, σε 16 μονάδες φόρτωσης/αποθήκευσης, ή σε τέσσερις SFUs. Δοθέντος ότι κάποιες εντολές απαιτούν περισσότερους κύκλους ρολογιού από κάποιες άλλες, όσο ένα Warp περιμένει να ολοκληρωθεί η εκτέλεση μιας «αργής» εντολής, επιλέγεται ένα άλλο Warp που είναι έτοιμο προς εκτέλεση και δεν περιμένει. Με αυτή την τεχνική επιτυγχάνεται η απόκρυψη καθυστέρησης που δημιουργείται από εντολές που κοστίζουν περισσότερους κύκλους ρολογιού (Latency Hiding). Και αυτό λειτουργεί διότι, η εναλλαγή από το ένα Warp στο άλλο έχει ουσιαστικά μηδενική καθυστέρηση εφόσον, όπως έχουμε αναφέρει ήδη, το κάθε thread έχει τα δικά του δεδομένα, τους δικούς του καταχωρητές και τον δικό του μετρητή εντολών.

4.4 Ιεραρχία Μνήμης

Το διάγραμμα του σχήματος 4.9 μας δείχνει τις διάφορες κατηγορίες μνήμης που περιέχει μια GPU και πως αυτές συνδέονται μεταξύ τους. Να πούμε ότι το διάγραμμα αυτό δεν αποτελεί την απεικόνιση της φυσικής δομής των μνημών, όπως αυτές είναι τοποθετημένες μέσα στο πυρίτιο, αλλά περισσότερο ένα μοντέλο συνδεσιμότητας των διαφόρων τύπων μνημών με τα υπόλοιπα στοιχεία της GPU.

Αναλυτικά, οι διάφοροι τύποι μνημών που διαθέτει μια GPU είναι οι:

- Καταχωρητές ή **Registers**
- Κοινή Μνήμη ή **Shared Memory**
- Καθολική Μνήμη ή **Global Memory**
- Σταθερή Μνήμη ή **Constant Memory**
- Τοπική Μνήμη ή **Local Memory**



Σχήμα 4.9: Ιεραρχία Μνήμης ενός SM [30]

Διαθέτει επίσης μια **L1 Cache** και μια **L2 Cache**. Η Local Memory, η οποία δεν φαίνεται στο διάγραμμα, αποτελεί μια νοητή μνήμη, η φυσική θέση της οποίας βρίσκεται στην Global Memory και θα μιλήσουμε αργότερα για το πότε και πώς χρησιμοποιείτε [32].

Όλα τα είδη μνημών μιας GPU φέρουν συγκεκριμένες ιδιότητες, όπως ταχύτητα προσπέλασης, μέγεθος, εμβέλεια και χρόνος ζωής. Το να γνωρίζουμε πως ακριβώς έχουν αυτές οι ιδιότητες για τον κάθε τύπο μνήμης, μας βοηθάει να διαλέξουμε το ποια και πότε θα χρησιμοποιήσουμε, προκειμένου η εφαρμογή μας να εκτελείται με την καλύτερη δυνατή απόδοση. Ο πίνακας 4.2 συνοψίζει τις ιδιότητες όλων των τύπων μνήμης. Συνεχίζουμε αναλύοντας τους παραπάνω τύπους μνημών και πώς ακριβώς τους χαρακτηρίζουν οι ιδιότητές τους.

Πίνακας 4.2: Ιδιότητες Μνημών [32]

Memory	Located	Cached	Access	Scope	Lifetime
Register	cache	n/a	Host:none, Kernel:R/W	thread	thread
Local	Device	1.x:no, 2.x:yes	Host:none, Kernel:R/W	thread	thread
Shared	cache	n/a	Host:none, Kernel:R/W	block	block
Global	device	1.x:no, 2.x:yes	Host:R/W, Kernel:R/W	application	application
Constant	device	Yes	Host:R/W, Kernel:R	application	application

Καταχωρητές (Registers)

Οι καταχωρητές βρίσκονται εντός των SMs και κάθε SP διαθέτει τους δικούς του προσωπικούς καταχωρητές. Η ταχύτητα προσπέλασης ενός καταχωρητή είναι πάρα

πολύ μεγάλη, αλλά ο διαθέσιμος αριθμός τους ανά block είναι περιορισμένος.

Κάθε απλή μεταβλητή που δηλώνεται εντός ενός Kernel και δεν φέρει κάποιο συγκεκριμένο πρόθεμα, αυτόματα τοποθετείται σε κάποιον καταχωρητή. Το ίδιο συμβαίνει και με πίνακες για τους οποίους το μέγεθος, η δομή και η δεικτοδότηση είναι γνωστά κατά τη μεταγλώττιση της εφαρμογής.

Η εμβέλεια των καταχωρητών έχει το εύρος ενός thread, που σημαίνει ότι κάθε thread έχει τις μεταβλητές του αποθηκευμένες σε δικούς του, προσωπικούς καταχωρητές. Συνεπώς, και η διάρκεια ζωής τους είναι η διάρκεια ζωής του εκάστοτε thread.

Οι καταχωρητές είναι προσπελάσιμοι για εγγραφή και ανάγνωση μόνο έσωθεν του Kernel και για οποιαδήποτε προσπέλαση δεν απαιτείται συγχρονισμός.

Κοινή Μνήμη (Shared Memory)

Κάθε SM διαθέτει ένα περιορισμένο μέγεθος κοινής μνήμης, προσπελάσιμης από όλα τα threads που εδρεύουν εντός του ίδιου block. Αυτό καθορίζει την διάρκεια ζωής της να είναι ίδια με αυτή του εκάστοτε block. Επίσης, κάνει την ταχύτητα προσπέλασής της να είναι πάρα πολύ μεγάλη, περίπου 100 φορές πιο γρήγορη από αυτή της καθολικής μνήμης.

Όμοια με τους καταχωρητές, είναι και αυτή προσπελάσιμη προς εγγραφή και ανάγνωση μόνο εντός ενός kernel, άρα και η εμβέλειά της είναι αυτή τού kernel. Αντίθετα όμως σε σχέση με τους καταχωρητές, εδώ απαιτείται συγχρονισμός στην περίπτωση όπου, διαφορετικά threads του ίδιου block, επιθυμούν να προσπελάσουν τα ίδια δεδομένα εντός της.

Προκειμένου να τοποθετηθεί μια μεταβλητή στην κοινή μνήμη, πρέπει να τη δηλώσουμε εντός του kernel με χρήση του προθέματος __shared__.

Καθολική Μνήμη (Global Memory)

Η καθολική μνήμη βρίσκεται εντός της κάρτας γραφικών, αλλά όχι εντός του γραφικού επεξεργαστή. Τοποθετείται πριν το δίαυλο διασύνδεσης της κάρτας γραφικών με τον ηλεκτρονικό υπολογιστή και έχει αρκετό μέγεθος, έως και 6GB. Λέγεται έτσι γιατί είναι η μόνη που είναι προσβάσιμη για εγγραφή καί ανάγνωση, τόσο από τον CPU όσο και από την GPU. Πρακτικά είναι ο μόνος τρόπος με τον οποίο μπορούν να μεταφερθούν δεδομένα στην κάρτα γραφικών. [30, Κεφάλαιο 6]

Γενικά, η προσπέλαση της καθολικής μνήμης είναι πάρα πολύ αργή. Αυτό δεν την καθιστά προτιμητέα για συχνή προσπέλαση από εντός του kernel, καθότι κάθε προσπέλαση σε αυτή κοστίζει πολλαπλούς κύκλους ρολογιού και αυτό ρίχνει δραματικά την απόδοση της εφαρμογής μας. Είναι προσπελάσιμη από όλα τα threads ενός kernel και γενικά απαιτείται προσοχή στον συγχρονισμό, διότι δεν υπάρχει μηχανισμός ελέγχου των επιμέρους threads. Συγχρονισμός στην προσπέλασή της επιτυγχάνεται ουσιαστικά, διασπώντας το πρόβλημα σε επιμέρους kernels και συγχρονίζοντας τις κλήσεις σε αυτούς, μέσα από το Host Code. Η διάρκεια ζωής της είναι ίδια με αυτή της εφαρμογής.

Για να τοποθετήσουμε μια μεταβλητή στην καθολική μνήμη, την δηλώνουμε στο κομμάτι του Device Code και σε καθολικό επίπεδο (δηλ. εκτός του kernel) με χρήση

του προθέματος __device__. Εναλλακτικά, για μεταφορά δεδομένων στην κάρτα γραφικών, δεσμεύουμε χώρο στην καθολική μνήμη από το Host Code με χρήση της συνάρτησης cudaMalloc() και μεταφέρουμε τα δεδομένα στον χώρο αυτό με χρήση της συνάρτησης cudaMemcpy(). Με την ίδια συνάρτηση επιτυγχάνεται και η αντίστροφη διαδικασία, μεταφορά δεδομένων δηλ. από την κάρτα γραφικών πίσω στον Host. Στην περίπτωση που έχουμε δεσμεύσει μνήμη με χρήση της συνάρτησης cudaMalloc(), πριν το τέλος της εφαρμογής, απαιτείται αποδέσμευση του χώρου με χρήση της συνάρτησης cudaFree().

Σταθερή Μνήμη (Constant Memory)

Η σταθερή μνήμη είναι και αυτή μια νοητή μνήμη, όπως και η τοπική μνήμη, η φυσική θέση της οποίας βρίσκεται στον ίδιο χώρο με την καθολική μνήμη. Είναι προσπελάσιμη για εγγραφή και ανάγνωση από το κομμάτι του Host Code μέσω των συναρτήσεων cudaMemcpyToSymbol() και cudaMemcpyFromSymbol() αντίστοιχα, ενώ διατίθεται για μόνο ανάγνωση σε όλα τα threads όλων των kernels. Μεταβλητές οι οποίες θέλουμε να τοποθετηθούν στην σταθερή μνήμη, ορίζονται στο καθολικό κομμάτι του Device Code (δηλ. εκτός του kernel) με χρήση του προθέματος __constant__.

Αντίθετα με την καθολική μνήμη, μόνο ένα μικρό μέρος μπορεί να οριστεί για χρήση ως σταθερή μνήμη και αυτό ανέρχεται στα 64KB για όλες τις αρχιτεκτονικές. Η ταχύτητα προσπέλασής της είναι ίδια με αυτή της καθολικής μνήμης, δηλ. αρκετά αργή, αλλά είναι cached. Δοθέντος ότι είναι διαθέσιμη στα threads μόνο για ανάγνωση, είναι εγγυημένο από τον μηχανισμό ότι το περιεχόμενο της cache δεν θα αλλάξει ποτέ κατά τη διάρκεια εκτέλεσης της εφαρμογής και έτσι, μέσω της cache, μπορούμε να έχουμε πολύ μεγαλύτερη ταχύτητα προσπέλασης σε σχέση με αυτή της καθολικής μνήμης.

Όμοια με την καθολική μνήμη, η σταθερή μνήμη έχει διάρκεια ζωής, αυτή της εφαρμογής και είναι επίσης προσπελάσιμη, όπως ήδη αναφέραμε, μόνο για ανάγνωση, από όλα τα threads όλων των kernels.

Τοπική Μνήμη (Local Memory)

Όπως αναφέραμε και στην ενότητα 4.4, η τοπική μνήμη βρίσκεται ουσιαστικά στην καθολική μνήμη, επομένως η ταχύτητα προσπέλασης της είναι πολύ μικρή. Σε αυτή θα τοποθετούν εκείνες οι μεταβλητές από αυτές που έχουμε ορίσει εντός του kernel, οι οποίες δεν χωράνε να τοποθετηθούν στον περιορισμένο χώρο των καταχωρητών. Πρόκειται για ένα φαινόμενο στο οποίο αναφερόμαστε ως υπερχείλιση καταχωρητών.

Η εμβέλεια προσπέλασής της περιορίζεται στο εύρος ενός thread και η διάρκεια ζωής της είναι αυτή του thread, όπως ακριβώς ισχύει και για τους καταχωρητές.

Δεν υπάρχει τρόπος να ορίσουμε συγκεκριμένες μεταβλητές να τοποθετηθούν εντός της τοπικής μνήμης. Αυτή είναι μια διαδικασία που θα την αναλάβει αυτόβουλα ο compiler για περιπτώσεις

- πινάκων των οποίων η δεικτοδότηση δεν είναι δυνατόν να καθοριστεί κατά τη φάση της μεταγλώττισης

- μεγάλων δομών και πινάκων που θα καταλάμβαναν πολύ χώρο στους καταχωρητές
- οποιωνδήποτε άλλων μεταβλητών δεν χωράνε στους καταχωρητές λόγω υπερχείλισης

Κεφάλαιο 5

Επιθέσεις στον KASUMI - A5/3 και Εφαρμογές Πινάκων Ουράνιου Τόξου

5.1 Επιθέσεις στον KASUMI - A5/3

Από το 2003 που δημοσιεύτηκε η σχεδίαση του A5/3, έχουν γίνει διάφορες μελέτες επιθέσεων, τόσο στον A5/3 αλλά και στον KASUMI. Αν και οι περισσότερες από αυτές δείχνουν ότι σε θεωρητικό επίπεδο, μια επίθεση στον κρυπτογραφικό αλγόριθμο του GSM είναι πρακτικά εφαρμόσιμη από τη σκοπιά της υπολογιστικής πολυπλοκότητας, ωστόσο κατά τη γνώση μας, δεν έχει υπάρξει ακόμα κάποια υλοποίηση, ικανή να παραβιάσει τον κρυπτογραφικό αλγόριθμο A5/3 στην πράξη.

Από τις πιο πρώιμες μελέτες είναι αυτή των Biham, Dunkelman και Keller [33], που χρησιμοποιεί έναν συνδυασμό των Boomerang και Rectangle attacks με χρήση συσχετιζόμενων κλειδιών, για την ανάκτηση του κλειδιού κρυπτογράφησης του KASUMI. Η μέθοδός τους απαιτεί $2^{54.6}$ επιλεγμένα αρχικά κείμενα, κρυπτογραφημένα με χρήση τεσσάρων συσχετιζόμενων κλειδιών, έχει χρονική πολυπλοκότητα της τάξεως των $2^{76.1}$ πράξεων κρυπτογράφησης και εφαρμόζεται στο εύρος καί των οκτώ επαναλήψεων του KASUMI.

Σε μια άλλη μελέτη του 2008 [34], οι Barkan, Biham και Keller, παρουσιάζουν μια επίθεση πραγματικού χρόνου σε επίπεδο κρυπτογραφικού πρωτοκόλλου, τύπου man-in-the-middle attack. Η μέθοδος τους στοχεύει τον ασθενέστερο αλγόριθμο A5/2 και είναι εφαρμόσιμη εφόσον η συσκευή τον υποστηρίζει, ανεξάρτητα από το αν τον υποστηρίζει και ο πάροχος. Δοθέντος ότι όλη η οικογένεια των A5 αλγορίθμων χρησιμοποιεί το ίδιο κλειδί κρυπτογράφησης, ανακτώντας αυτό με τον παραπάνω τρόπο επίθεσης, γίνεται εφικτή η αποκρυπτογράφηση δεδομένων που έχουν κρυπτογραφηθεί είτε με τον A5/1 είτε με τον A5/3.

Από τις πιο ενδιαφέρουσες δουλειές είναι αυτή των Dunkelman, Keller και Shamir [35]. Περιγράφει μια μέθοδο που την ονομάζουν sandwich attack, με την οποία δύναται να ανακτηθεί το κλειδί κρυπτογράφησης του KASUMI με χρήση τεσσάρων συσχετιζόμενων κλειδιών, 2^{26} δεδομένα, 2^{30} bytes μνήμης και 2^{32} χρονική πολυπλοκότητα.

Πιο πρόσφατα, οι Jia, Li, Rechberger, Chen και Wang [36], κάνουν κάποιες

παρατηρήσεις στις συναρτήσεις FL και FO του KASUMI και σε συνδυασμό με μια αδυναμία στον αλγόριθμο παραγωγής υποκλειδιών προτείνουν μια επίθεση που εφαρμόζεται σε επτά από τους οκτώ κύκλους του KASUMI. Η μέθοδός τους, στην χειρότερη περίπτωση, απαιτεί 2^{62} γνωστά αρχικά κείμενα και $2^{115.8}$ κρυπτογραφήσεις.

Στην συνέχεια, οι Wang, Dong, Jia και Zhao [37], βελτιώνουν την παραπάνω δουλειά συνδυάζοντάς τη με μια τεχνική έγχυσης λάθους (Fault Injection Attack [38]) στον αλγόριθμο παραγωγής υποκλειδιών. Με έγχυση ενός μόνο λάθους υποκλειδιού μεγέθους 16 bits, επιτυγχάνεται ανάκτηση του κλειδιού κρυπτογράφησης του KASUMI με χρονική πολυπλοκότητα 2^{32} κρυπτογραφήσεις. Η προσομοίωση της μεθόδου σε έναν ηλεκτρονικό υπολογιστή, ανακτά το κλειδί σε μόλις μερικά λεπτά.

Τέλος, είναι σκόπιμο να αναφερθούμε και στην διπλωματική εργασία του Π. Παπαντωνάκη από το οικείο μας τμήμα των Ηλεκτρονικών Μηχανικών και Μηχανικών Υπολογιστών του Πολυτεχνείου Κρήτης [39]. Ο κ. Παπαντωνάκης στην διπλωματική του εργασία, υλοποίησε σε αναδιατασσόμενη λογική (FPGA), μια εφαρμογή για την κατασκευή των πινάκων ουράνιου τόξου, για χρήση στην αποκρυπτογράφηση δεδομένων, κρυπτογραφημένα με τον A5/3.

5.2 Εφαρμογές Πινάκων Ουράνιου Τόξου

Ένα πρότζεκτ στο οποίο αξίζει περισσότερο από όλα να αναφερθούμε, είναι αυτό του Karsten Nohl και της ομάδας του [40]. Πρόκειται για μια δουλειά που στοχεύει στην αποκρυπτογράφηση δεδομένων που έχουν κρυπτογραφηθεί με τον αλγόριθμο A5/1. Με την χρήση πινάκων Ουράνιου Τόξου και καρτών γραφικών για την παραλληλοποίηση του υπολογισμού τους, αλλά και πρόσθετων εφαρμογών λογισμικού ανοικτού κώδικα, ο Nohl και η ομάδα του, κατασκεύασαν ένα σετ από 40 πίνακες ουράνιου τόξου συνολικού όγκου 2TB, με το οποίο είναι σε θέση να ανακτήσουν το μυστικό κλειδί με επιτυχία 90%. Η διαδικασία αποκρυπτογράφησης με χρήση δυο καρτών γραφικών επιτυγχάνεται σε περίπου πέντε δευτερόλεπτα. Μαζί τους συνεργάστηκαν οι Glendrange, Hove και Hvideberg στα πλαίσια της μεταπτυχιακής τους εργασίας με τίτλο "Decoding GSM" [22].

Κατά την έρευνά μας, ανακαλύψαμε και μια σειρά εφαρμογών που χρησιμοποιούν πίνακες Ουράνιου Τόξου για την αποκρυπτογράφηση δεδομένων. Η πρώτη από αυτές είναι το Ophcrack [41] και έχει υλοποιηθεί από τον P. Oechslin, που είναι και ο εμπνευστής της μεθόδου αυτής. Το Ophcrack μπορεί να χρησιμοποιηθεί για να ανακτήσει κωδικούς διαπιστευτηρίων που έχουν κωδικοποιηθεί με χρήση των συναρτήσεων κατακερματισμού LM και NTLM. Η εφαρμογή είναι δωρεάν διαθέσιμη από την ιστοσελίδα της, μαζί μια μια σειρά από πίνακες, αλλά εκτελείται μόνο σε CPU.

Άλλη μια εφαρμογή που χρησιμοποιεί Πίνακες ουράνιου Τόξου είναι η RainbowCrack [42]. Αντίθετα με το Ophcrack, το RainbowCrack εκμεταλλεύεται την ύπαρξη καρτών γραφικών Nvidia/AMD στο σύστημα για παραλληλοποίηση των υπολογισμών. Χρησιμοποιείται και αυτό για την ανάκτηση κωδικών από συναρτήσεις κατακερματισμού και υποστηρίζει μια ευρύτερη γκάμα τέτοιων, σε σχέση με το Ophcrack. Διαθέσιμες στην ιστοσελίδα τους, εκτός από την εφαρμογή για την ανάκτηση των κωδικών, είναι επίσης και η εφαρμογή για την κατασκευή των πινάκων αλλά και για την ταξινόμησή τους.

Ανακαλύψαμε, τέλος, την εφαρμογή του Cryptohaze [43], το GPU Rainbow Cracker που ακολουθεί παρόμοια φιλοσοφία με το RainbowCrack. Με την εφαρμογή του Cryptohaze ασχοληθήκαμε εκτενέστερα στη φάση της υλοποίησης της εφαρμογής μας και θα αναφερθούμε λεπτομερώς σε αυτή στην ενότητα 6.6.

Κεφάλαιο 6

Υλοποίηση

Στα κεφάλαια που ακολουθούν περιγράφουμε αναλυτικά τα βήματα που ακολουθήθηκαν για την ανάπτυξη της εφαρμογής μας, από τις πρώτες πειραματικές εκδόσεις, μη παράλληλης εκτέλεσης σε γλώσσα προγραμματισμού **C**, έως την τελική βελτιστοποιημένη έκδοση για παράλληλη εκτέλεση σε **CUDA**, με όλα τα ενδιάμεσα στάδια βελτιστοποίησης.

6.1 Εξοπλισμός

Κατά τη διαδικασία ανάπτυξης της εφαρμογής, χρησιμοποιήθηκαν διάφορες κάρτες γραφικών της NVIDIA και διάφορες εκδόσεις του CUDA Toolkit, όλα σε συνδυασμό με ηλεκτρονικούς υπολογιστές που φέρουν λειτουργικό σύστημα linux.

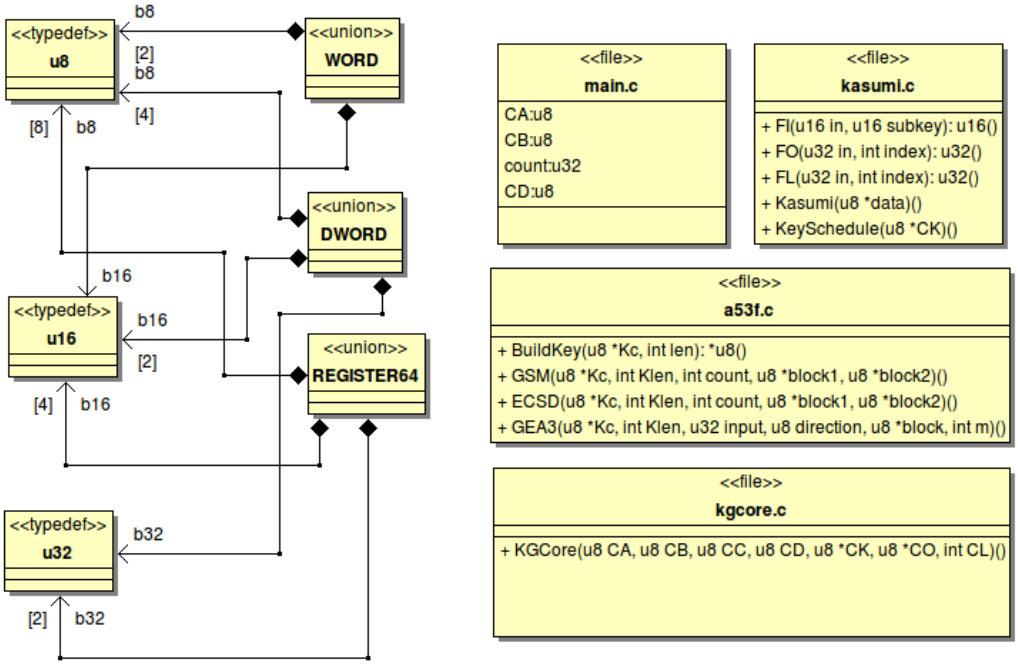
Συγκεκριμένα, οι πρώιμες εκδόσεις σε γλώσσα προγραμματισμού C καθώς και η πρώτη έκδοση σε CUDA για παράλληλη εκτέλεση, αναπτύχθηκαν σε έναν υπολογιστή με λειτουργικό σύστημα linux της διανομής ubuntu έκδοσης 10.04, με επεξεργαστή intel Core2Duo E8400 @ 3 GHz, 2 GB μνήμη RAM, κάρτα γραφικών GeForce 9800GTX με compute capability 1.1 και CUDA Toolkit έκδοσης αρχικά 3.2 και στην πορεία, 4.0 και 5.0. Η εκπόνηση της βελτιστοποίησης της έκδοσης σε CUDA έλαβε χώρα σε ένα laptop με λειτουργικό σύστημα linux της διανομής ubuntu έκδοσης 12.04, με επεξεργαστή intel i3-M330 @ 2.13 GHz, 4 GB μνήμη RAM, κάρτα γραφικών GeForce GT 325M με compute capability 1.2 (κατασκευασμένη με την τεχνολογία Optimus για την υποστήριξη υβριδικών γραφικών) και CUDA Toolkit έκδοσης 6.5. Η μεταγλώττιση του τελικού πηγαίου κώδικα και η εκτέλεση των πειραμάτων για κάθε στάδιο βελτιστοποίησης έγιναν σε έναν υπολογιστή με λειτουργικό σύστημα linux της διανομής CentOS έκδοσης 7.1.1503, με επεξεργαστή intel i7-3770 @ 3.40 GHz, 16 GB μνήμη RAM, κάρτα γραφικών GeForce GTX 580 με compute capability 2.0 και CUDA Toolkit έκδοσης 7.5.

6.2 Προσομοίοση του A5/3 και του KASUMI σε Γλώσσα Προγραμματισμού C

Πρώτη μας δουλειά ήταν να βρούμε μια υλοποίηση του KASUMI και της KGCore σε γλώσσα προγραμματισμού C ή, αν χρειαστεί, να υλοποιηθούν από το μηδέν. Στο

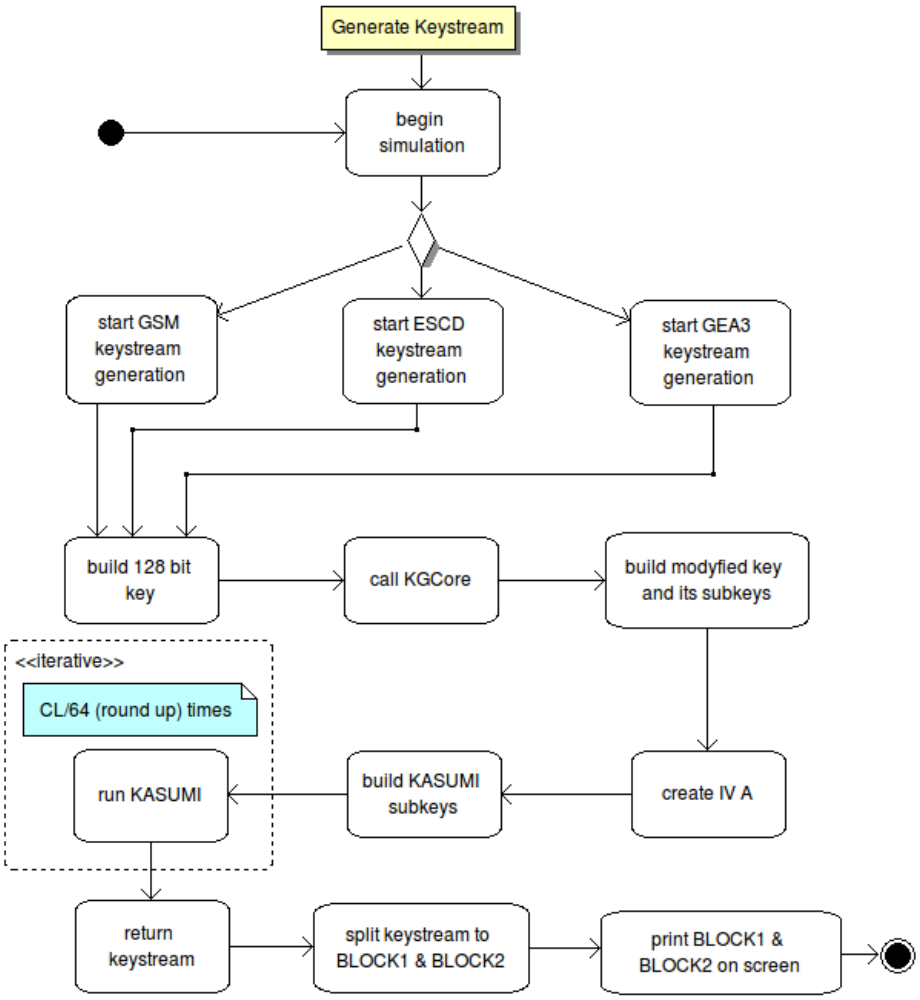
τεχνικό δελτίο που περιγράφει τη λειτουργία του A5/3 [23], υπάρχει διαθέσιμη σε γλώσσα προγραμματισμού C η υλοποίηση του KASUMI και της KGCore, καθώς και οι συναρτήσεις για την επιμήκυνση του κλειδιού συνεδρίας K_C σε 128 bits και αυτές για την παραγωγή κλειδιών ροής. Οι συναρτήσεις αυτές υλοποιήθηκαν στα πλαίσια της ανάπτυξης του συστήματος κρυπτογράφησης από την 3GPP, για χρήση προσομοίωσης και ελέγχου της ορθής λειτουργίας των αλγορίθμων σε συνδυασμό με τα δεδομένα δοκιμών που αναφέρονται στο τεχνικό δελτίο [44].

Χρησιμοποιώντας τον διαθέσιμο κώδικα από το τεχνικό δελτίο της 3GPP, αναπτύξαμε την βασική συνάρτηση main() που «δένει» μαζί όλες τις επιμέρους συναρτήσεις για την παραγωγή κλειδιών ροής. Πέραν του αρχείου που περιέχει την main(), το main.c, ο πηγαίος κώδικας απαρτίζεται από άλλα τρία αρχεία, ως ακολούθως. Το a53f.c που περιέχει την συνάρτηση επιμήκυνσης του κλειδιού συνεδρίας K_C, BuildKey(), καθώς και τις συναρτήσεις παραγωγής κλειδιών ροής GSM(), ECSD() και GEA3() για τους αντίστοιχους τύπους δεδομένων. Το αρχείο kgcore.c που περιέχει την συνάρτηση KGCore(). Το αρχείο kasumi.c που περιέχει την συνάρτηση παραγωγής των επιμέρους κλειδιών KL_i, KO_i και KI_i, επονομαζόμενη KeySchedule(), την βασική συνάρτηση Kasumi() και τις υποσυναρτήσεις FI(), FO() και FL(). Εντός της συνάρτησης FI() περιέχονται τα κουτιά αντικατάστασης S7[ ] και S9[ ] με την μορφή πινάκων αναζήτησης. Καί τα τρία αυτά αρχεία συνοδεύονται από τα αντίστοιχα header files τύπου .h, τα οποία φέρουν τους ορισμούς των πρωτοτύπων των συναρτήσεων που είναι υλοποιημένες στα αρχεία του πηγαίου κώδικα τύπου .c και, όπου χρειάζεται, τους ορισμούς κάποιων καθολικών μεταβλητών. Η δομή αυτών των αρχείων, όπως την περιγράψαμε, απεικονίζεται στο σχήμα 6.1.



Σχήμα 6.1: Δομή Αρχείων Έκδοσης Προσομοίωσης

Η εκτέλεση της εφαρμογής ξεκινά με μια κλήση σε μια από της συναρτήσεις παραγωγής κλειδιών ροής. Κατόπιν, μέσα από τη συνάρτηση παραγωγής κλειδιών ροής, καλείται η συνάρτηση επιμήκυνσης του κλειδιού συνεδρίας και στη συνέχεια η συνάρτηση KGCore(). Η KGCore() με τη σειρά της καλεί την KeySchedule() για την κατασκευή των επιμέρους κλειδιών, κατασκευάζει το διάνυσμα αρχικοποίησης A και ολοκληρώνει την εκτέλεσή της με τον απαιτούμενο αριθμό κλήσεων της συνάρτησης Kasumi() για την παραγωγή του ζητηθέν κλειδιού ροής. Η συνάρτηση Kasumi() υλοποιεί τους οκτώ κύκλους του KASUMI ως τέσσερις επαναλήψεις σε έναν βρόχο for(), συνδυάζοντας στην κάθε επανάληψη από έναν μονό και τον ακολουθούμενο ζυγό κύκλο του KASUMI, όπου καλούνται οι συναρτήσεις FO() και FL() με την αντίστοιχη σειρά τους. Μέσα από κάθε κλήση της FO(), καλείται η FI() για την χρήση των κουτιών αντικατάστασης S7[ ] και S9[ ]. Την αλληλουχία αυτή της εκτέλεσης βλέπουμε στο σχήμα 6.2.



Σχήμα 6.2: Διάγραμμα Εκτέλεσης Έκδοσης Προσομοίωσης

Για την επαλήθευση της ορθής λειτουργίας αυτής της υλοποίησης, χρησιμοποιήσαμε τα δεδομένα δοκιμών από το τεχνικό δελτίο [44] της 3GPP, με βάση τα οποία και επιβεβαιώθηκε ότι η εφαρμογή μας λειτουργεί σωστά και δίνει τα αναμενόμενα αποτελέσματα. Ακολουθεί ο πίνακας 6.1, ο οποίος περιέχει ενδεικτικά κάποια κλειδιά συνεδρίας και τις επιμέρους παραμέτρους ως εισόδους και τα αναμενόμενα κλειδιά ροής ως εξόδους, για δεδομένα GSM. Στον πίνακα δεν αναφέρονται οι παράμετροι CB CD και CE, οι οποίες έχουν μηδενική τιμή σε όλες τις περιπτώσεις, καθώς και η παράμετρος CK η οποία αποτελείται από την επιμήκυνση του κλειδιού συνεδρίας ως $K_C||K_C$.

Πίνακας 6.1: Αποτελέσματα Δοκιμών

Inputs				**Output**
K_C	**CA**	**CC**	**CL**	**BLOCK1, BLOCK2**
0x2BD6459F 82C5BC00	0x0F	0x24F20F	228	0x889EEAAF9ED1BA1ABBD8436232E440, 0x5CA3406AA244CF69CF047AADA2DF40
0x952C4910 4881FF48	0x0F	0x061272	228	0xFB4D5FBCEE13A33389285686E9A5C0, 0x25090378E0540457C57E367662E440
0xEFA8B222 9E720C2A	0x0F	0x33FD3F	228	0x0E4015755A336469C3DD8680E30340, 0x6F10669E2B4E18B042431A28E47F80

6.3 Ιδιαιτερότητες του Κώδικα της 3GPP

Στο σημείο αυτό, θεωρούμε σκόπιμο να αναφερθούμε σε κάποιες ιδιαιτερότητες του κώδικα που δανειστήκαμε από τα τεχνικά δελτία της 3GPP, μιας και αυτές θα μας απασχολήσουν αργότερα κατά τη φάση της βελτιστοποίησης της έκδοσης παράλληλης εκτέλεσης σε CUDA.

Έχουμε αναφέρει ήδη το γεγονός ότι τα κουτιά αντικατάστασης (SBoxes) βρίσκονται εντός της συνάρτησης FI(). Αυτά ορίζονται εκεί ως τοπικές μεταβλητές τύπου πίνακα (arrays), που φέρει σε κάθε θέση του μια ποσότητα μεγέθους 16 bits χωρίς μαθηματικό πρόσημο (unsigned). Για τις ανάγκες εκτέλεσης μιας προσομοίωσης σε έναν κώδικα που εκτελείται σειριακά και για παραγωγή ενός κλειδιού ροής κάθε φορά, αυτό είναι μια λογική επιλογή, καθότι τα κουτιά αντικατάστασης χρησιμοποιούνται μόνο εντός της συνάρτησης FI(), οπότε και έχει νόημα να αποτελούν τοπικές μεταβλητές. Θα δούμε όμως κατά τη φάση της βελτιστοποίησης ότι, για τον κώδικα που εκτελείται εν παραλλήλω και για παραγωγή πολλαπλών κλειδιών ροής ταυτόχρονα, η προσέγγιση αυτή σημαίνει πολλαπλά αντίγραφα των ίδιων στατικών μεταβλητών που χρησιμεύουν σε ανάγνωση μόνο. Πολλαπλές ίδιες μεταβλητές σημαίνει εν γένει σπατάλη μνήμης, ένας πόρος πολύτιμος όταν προγραμματίζουμε σε CUDA, μιας και αυτή της κάρτας γραφικών είναι σχετικά περιορισμένη όπως δείχνει και ο πίνακας 4.1. Για τον λόγο αυτό, στον ορισμό των κουτιών αντικατάστασης στον κώδικα σε CUDA, ακολουθήσαμε μια διαφορετική προσέγγιση.

Επόμενη ιδιαιτερότητα στην όποια θέλουμε να σταθούμε, αποτελεί ο ορισμός των μεταβλητών στις οποίες αποθηκεύονται τα επιμέρους κλειδιά KL_i, KO_i και KI_i. Αυτά ορίζονται εντός του αρχείου kasumi.c ως καθολικές μεταβλητές (global) τύπου πίνακα, που σε κάθε θέση του φέρει μια ποσότητα των 16 bits και πάλι χωρίς μαθη-

ματικό πρόσημο. Η λογική του να είναι ορισμένα ως καθολικές μεταβλητές, πηγάζει από την αναγκαιότητα, αυτά να είναι προσπελάσιμα από όλες τις συναρτήσεις, τουλάχιστον εντός του ίδιου αρχείου του πηγαίου κώδικα, μιας και η kasumi() αλλά και όλες οι υποσυναρτήσεις της τα χρησιμοποιούν. Το πρόβλημα που μπορεί να προκύψει με αυτή την υλοποίηση στην περίπτωση του κώδικα παράλληλης εκτέλεσης, είναι, το κάθε νήμα που παράγει ένα κλειδί ροής σε συνδυασμό με ένα διαφορετικό κλειδί συνεδρίας από τα άλλα νήματα, να «πανωγράφει» τις μεταβλητές των επιμέρους κλειδιών με τα δικά του δεδομένα σε αυθαίρετο χρόνο, με αποτέλεσμα την χρονική στιγμή που θα ζητηθούν τα επιμέρους κλειδιά από κάποια συνάρτηση, η ακεραιότητα των τιμών τους να μην μπορεί να εγγυηθεί. Συνεπώς, στην περίπτωση του παράλληλου κώδικα, πρέπει να διασφαλίσουμε ότι κάθε νήμα θα έχει στην διάθεσή του τις δικές του, ιδιωτικές μεταβλητές για τα επιμέρους κλειδιά.

Τελευταίο στοιχείο στο οποίο θέλουμε να εφιστήσουμε την προσοχή του αναγνώστη, είναι ο τρόπος με τον οποίο η Kasumi() και οι υποσυναρτήσεις FL(), FO() και FI() διαχειρίζονται τα δεδομένα εισόδου. Τα δεδομένα εισόδου εδώ, απαρτίζει το τμήμα των 64 bits που πρόκειται να κρυπτογραφηθεί από τον KASUMI. Θυμίζουμε από την περιγραφή της λειτουργίας του KASUMI, ότι το τμήμα εισόδου υποδιαιρείται σε δυο τμήματα των 32 bits και κάθε τέτοιο υποτμήμα, με την εφαρμογή κάθε υποσυνάρτησης σε αυτό, υποδιαιρείται εκ νέου σε δυο τμήματα των 16 bits. Ακόμη και στην περίπτωση της FI() που υποδιαιρεί το τμήμα εισόδου σε δυο άνισες ποσότητες των επτά και εννέα bits αντίστοιχα, σε επίπεδο υλοποίησης σε κώδικα, οι άνισες αυτές ποσότητες αποθηκεύονται σε μεταβλητές μεγέθους 16 bits. Σε επίπεδο κώδικα λοιπόν, στην φάση της διάσπασης του τμήματος εισόδου, η Kasumi() εκχωρεί τα υποτμήματα σε δυο τοπικές μεταβλητές μεγέθους 32 bits, μια για το αριστερό και μια για το δεξί υποτμήμα. Όμοια, η κάθε υποσυνάρτηση μετά τη διάσπαση, εκχωρεί τα νέα υποτμήματα σε δυο τοπικές μεταβλητές μεγέθους 16 bits, ως αριστερό και δεξί υποτμήμα αντίστοιχα. Με το πέρας των υπολογισμών, η κάθε υποσυνάρτηση ενοποιεί τα υποτμήματα σε μια μεταβλητή αναλόγου μεγέθους, την τιμή της οποίας και επιστρέφει στην συνάρτηση που την κάλεσε. Η δε συνάρτηση Kasumi(), χρησιμοποιεί έναν δείκτη για την ενοποίηση και επιστροφή των υποτμημάτων που φέρουν το τελικό αποτέλεσμα των υπολογισμών.

Το πρόβλημα που ανακύπτει με όλες αυτές τις διασπάσεις και ενοποιήσεις, πηγάζει από την κατασκευαστική φύση του υλικού που χρησιμοποιείται. Συγκεκριμένα, ο επεξεργαστής είναι κατασκευασμένος με βάση το big-endian μοντέλο, ενώ η κάρτα γραφικών με βάση το little-endian μοντέλο. Δεδομένου ότι, όλες οι πράξεις του KASUMI αποτελούν δυαδικές πράξεις που εφαρμόζονται σε επίπεδο bit, είναι σημαντικό κατά τις διασπάσεις και τις εκχωρήσεις των δεδομένων σε νέες μεταβλητές, να είναι εγγυημένη η σωστή τους τοποθέτηση στην μνήμη του συστήματος όπου εκτελείται ο κώδικας, ανεξάρτητα από το endian μοντέλο με το οποίο αυτό είναι κατασκευασμένο (big-endian vs little-endian). Για την διασφάλιση της σωστής τοποθέτησης στην μνήμη του συστήματος, οι εκχώρηση των τιμών στις μεταβλητές γίνεται σε πολλαπλά στάδια ανά ποσότητες των 8 bits, σε συνδυασμό με τη χρήση τριών κατάλληλων ενώσεων (unions), μια μεγέθους 16 bits, μια μεγέθους 32 bits και μια μεγέθους 64 bits. Να θυμίσουμε ότι στις γλώσσες προγραμματισμού C/C++, μία ένωση αποτελεί ένα σύνθετο τύπο δεδομένων, ο οποίος απαρτίζει μια συλλογή από άλλους απλούς ή/και σύνθετους τύπους δεδομένων. Κάτι σαν δομή δηλαδή, με την σημαντική όμως

διαφορά ότι, όλα τα δεδομένα που απαριθμεί μοιράζονται την ίδια θέση μνήμης στο σύστημα. Αυτό μας δίνει την ευελιξία να μπορούμε να προσπελάσουμε τα δεδομένα σε μια συγκεκριμένη θέση μνήμης με χρήση διαφορετικού τύπου δεδομένων κάθε φορά, ανάλογα την αναγκαιότητα. Έτσι για παράδειγμα μπορούμε να διαχειριζόμαστε τα δεδομένα εισόδου/εξόδου μας, άλλοτε ως ποσότητες των 8 bits, ή των 16 bits, ή των 32 bits, κατά το δοκούν.

Όπως αντιλαμβανόμαστε, επειδή στην εφαρμογή μας, επενεργούμε στα δεδομένα σε επίπεδο bit και όχι σε επίπεδο τιμής (bitwise operations vs operations by value), απαιτείται προσοχή στον τρόπο με τον οποίο εκχωρούνται τα δεδομένα στην μνήμη κάθε φορά.

6.4 Κατασκευή Πινάκων σε C (Σειριακή Εκτέλεση)

Το αμέσως επόμενο βήμα που ακολουθήσαμε, ήταν να εμπλουτίσουμε την έκδοση προσομοίωσης κατασκευής κλειδιών ροής, με της απαραίτητες λειτουργίες ώστε να κατασκευάζει πίνακες Ουράνιου Τόξου και να τους αποθηκεύει. Για να απλοποιήσουμε την διαδικασία της υλοποίησης, εστιάσαμε στην κατασκευή πινάκων μόνο για δεδομένα GSM (φωνή). Κάνοντας αυτήν την επιλογή, μπορούμε να απαλείψουμε εντελώς τις συναρτήσεις GSM(), ECSD() και GEA3(), καλώντας κατευθείαν την KGCore() για την παραγωγή κλειδιών ροής, μειώνοντας ταυτόχρονα το βάθος κλήσεων συναρτήσεων κατά ένα επίπεδο.

Αρχικά έπρεπε να επινοήσουμε έναν τρόπο ώστε να διαχειριζόμαστε τα αρχικά και τελικά σημεία κάθε αλυσίδας του πίνακα. Για τον σκοπό αυτό επιλέξαμε να ορίσουμε μια δομή που την ονομάζουμε reg256. Η δομή αυτή αποτελείται από δύο διανύσματα, ένα για το αρχικό και ένα για το τελικό σημείο της αλυσίδας. Τα διανύσματα αυτά ορίζονται ως πίνακες δεκαέξι στοιχείων που το καθένα αποτελεί μια ποσότητα των 16 bits χωρίς μαθηματικό πρόσημο. Συνεπώς, κάθε διάνυσμα έχει συνολικό μήκος 256 bits. Ο λόγος που επιλέξαμε το κάθε στοιχείο του πίνακα των διανυσμάτων να είναι μια ποσότητα των 16 bits έχει να κάνει με το ζήτημα του endianness που περιγράψαμε νωρίτερα. Μελετώντας την λειτουργία της συνάρτησης Kasumi() και των υποσυναρτήσεων αυτής, βλέπουμε ότι, οι όποιες πράξεις, εφαρμόζονται σε ποσότητες των 16 bits. Καταχωρώντας λοιπόν τα δεδομένα μας ανά ποσότητες των 16 bits, διασφαλίζουμε ότι δεν θα υπάρχει πρόβλημα στις πράξεις λόγο του διαφορετικού endian μοντέλου μεταξύ του επεξεργαστή και της κάρτας γραφικών.

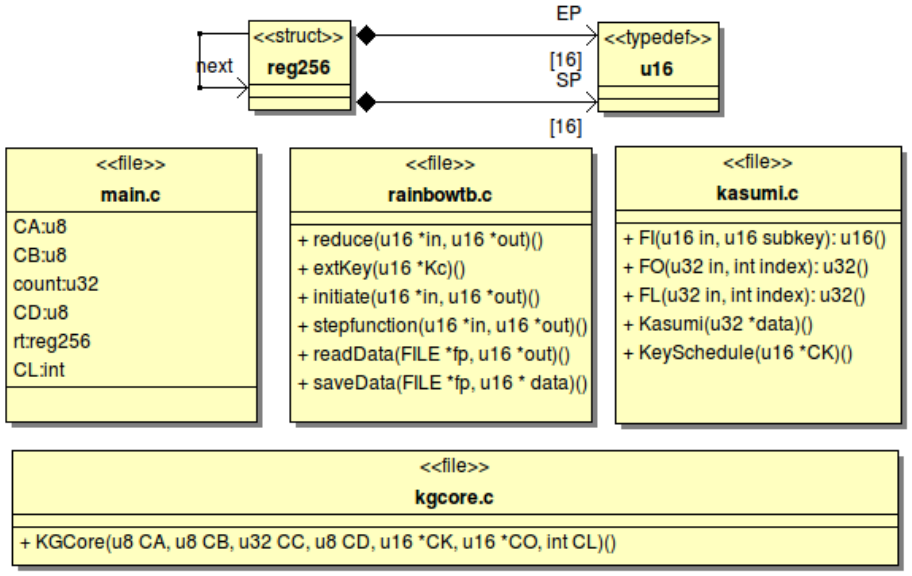
Τα αρχικά σημεία, τα οποία βρίσκονται αποθηκευμένα σε ένα αρχείο, διαβάζονται από το αρχείο με χρήση της συνάρτησης readData(). Για την αποθήκευσή τους στην μνήμη του συστήματος προς χρήση, δημιουργούμε μια συνδεδεμένη λίστα από δομές τύπου reg256, με τόσους κρίκους όσα και τα αρχικά σημεία που περιέχει το αρχείο. Κάθε αρχικό σημείο, αποθηκεύεται κατόπιν στον αντίστοιχο κρίκο της λίστας, στο προβλεπόμενο για αυτό διάνυσμα.

Για την επαναληπτική διαδικασία του υπολογισμού των αλυσίδων του πίνακα, υλοποιήσαμε δύο συναρτήσεις, την initiate() και την stepfunction(). Η initiate() αρχικοποιεί την διαδικασία υπολογισμού αλυσίδων, επεκτείνοντας αρχικά το κλειδί συνεδρίας σε 128 bits, καλώντας εν συνεχεία την KGCore() για την παραγωγή ενός

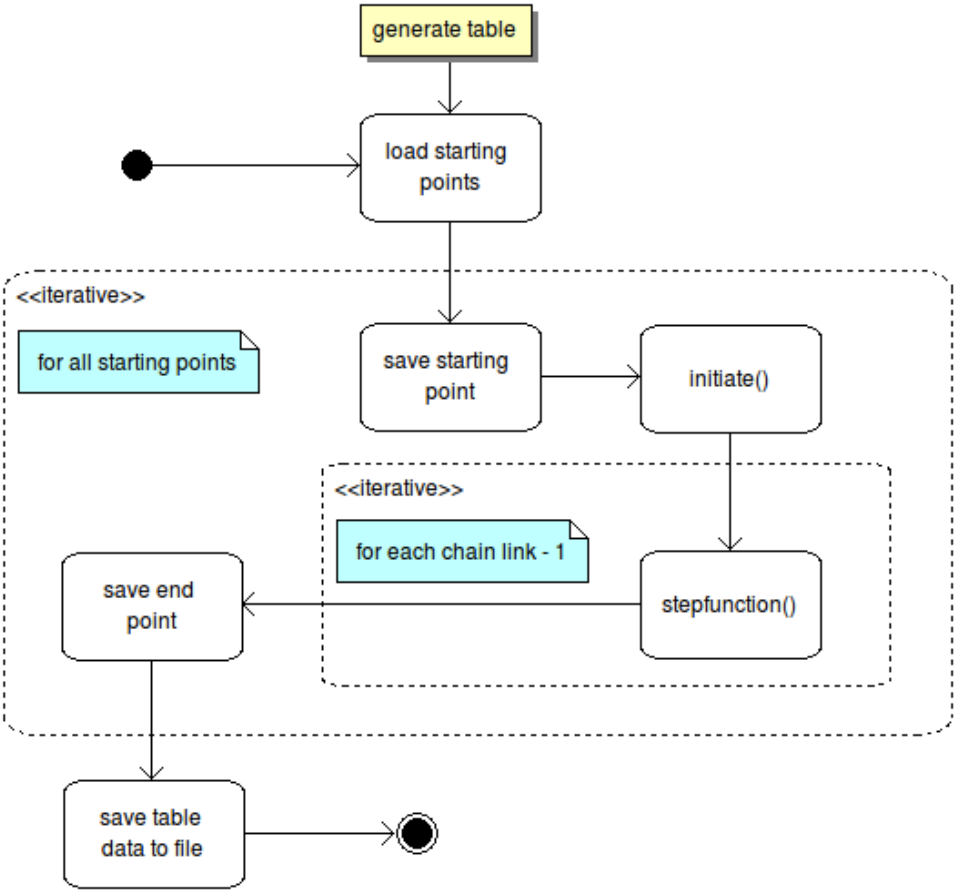
κλειδιού ροής και, αφού αυτό επιστραφεί, διασπάται στα αντίστοιχα BLOCK1 και BLOCK2. Η stepfunction() εφαρμόζει την συνάρτηση αναγωγής (reduce()) στο κρυπτογραφημένο διάνυσμα για να παράξει ένα νέο κλειδί συνεδρίας και στην συνέχεια επαναλαμβάνει μια κλήση στην συνάρτηση initiate(). Έτσι, ο υπολογισμός μιας αλυσίδας μήκους N, ακολουθεί τη σειρά, μια κλήση στην initiate() και N-1 κλήσεις στην stepfunction(). Τα ενδιάμεσα αποτελέσματα κατά τον υπολογισμό της αλυσίδας αποθηκεύονται στο διάνυσμα του αρχικού σημείου, ενώ το τελικό αποτέλεσμα αποθηκεύεται ως τελικό σημείο στο προβλεπόμενο για αυτό διάνυσμα της δομής reg256. Η όλη διαδικασία επαναλαμβάνεται για όλα τα υπόλοιπα αρχικά σημεία. Αφού ολοκληρωθεί ο υπολογισμός όλων των αλυσίδων, οι συνδυασμοί αρχικού και τελικού σημείου από κάθε αλυσίδα, αποθηκεύονται σε ένα νέο αρχείο. Την αποθήκευση επιτελεί η συνάρτηση saveData.

Όσων αφορά την συνάρτηση αναγωγής, στην παρούσα φάση της υλοποίησης, χάριν ευκολίας, εφαρμόσαμε την τεχνική του Hellmann. Δηλαδή, κάθε στήλη του πίνακα εφαρμόζει την ίδια συνάρτηση αναγωγής, η οποία κρατά τα πρώτα 64 bits ως νέο κλειδί και «πετά» τα υπόλοιπα. Άρα πρακτικά, σε αυτό το σημείο, η εφαρμογή μας κατασκευάζει πίνακες Hellmann και όχι πίνακες Ουράνιου Τόξου. Η αναπροσαρμογή της συνάρτησης αναγωγής, με κατάλληλο τρόπο για την κατασκευή πινάκων Ουράνιου Τόξου, έγινε σε μελλοντικό στάδιο, στο οποίο και θα αναφερθούμε εκτενώς στην ενότητα 6.5, όπου περιγράφουμε την έκδοση για εκτέλεση σε CUDA.

Το σχήμα 6.3 μας δείχνει την δομή των αρχείων του πηγαίου κώδικα για την κατασκευή πινάκων, ενώ η ροή της εκτέλεσης της εφαρμογής για την κατασκευή πινάκων απεικονίζεται στο σχήμα 6.4.



Σχήμα 6.3: Κατασκευή Πινάκων σε C - Δομή Πηγαίου Κώδικα



Σχήμα 6.4: Κατασκευή Πινάκων σε C - Ροή Εκτέλεσης

6.5 Μετφορά της Εφαρμογής από C σε CUDA

Σε αυτό το στάδιο, η εφαρμογή μας είναι έτοιμη για μεταφορά σε CUDA. Η γενική φιλοσοφία για το μοντέλο παραλληλισμού που επιλέξαμε να ακολουθήσουμε, είναι κάθε νήμα που εκτελείται στην κάρτα γραφικών να αναλαμβάνει τον υπολογισμό μιας αλυσίδας. Όσων αφορά τη μορφή των πινάκων, επιλέγουμε κάθε πίνακας να αποθηκεύει ζεύγη από ένα κλειδί των 64 bits ως αρχικό σημείο και ένα κλειδί ροής των 114 bits ως τελικό σημείο. Από αυτή την επιλογή προκύπτει ότι, απαιτείται μια διαφορετική ομάδα πινάκων για κάθε κατεύθυνση μετάδοσης. Επίσης, για κάθε πίνακα που θα κατασκευάζεται θα επιλέγεται ένας διαφορετικός count. Με αυτόν τον τρόπο συνδυάζουμε το διάνυσμα αρχικοποίησης με το αρχικό σημείο.

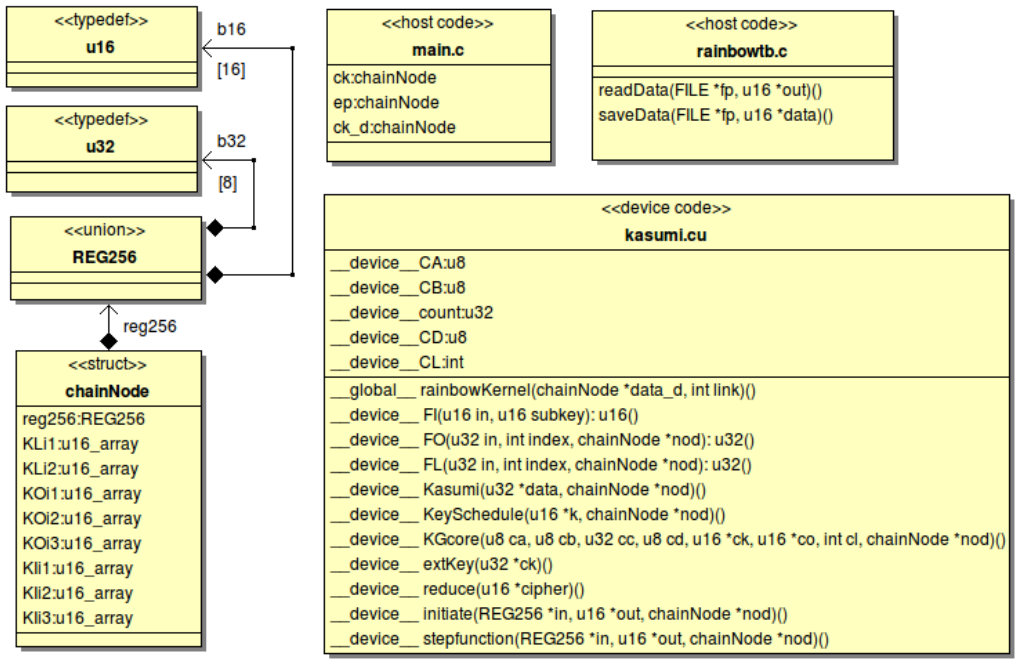
Προσαρμογές

Για την ορθή λειτουργία της εφαρμογής είναι απαραίτητο να γίνουν κάποιες τροποποιήσεις. Η βασικότερη από αυτές, όπως έχουμε ήδη αναφέρει στην ενότητα 6.3, είναι να φροντίσουμε κάθε νήμα να έχει τις δικές του ιδιωτικές μεταβλητές για τα επιμέρους κλειδιά που απαιτούνται για την εφαρμογή του KASUMI. Για την επίτευξη αυτού του στόχου, καταργούμε την δομή reg256 που διαχειρίζεται τα αρχικά και τελικά σημεία του πίνακα. Δημιουργούμε μια καινούρια δομή η οποία, νοητά, αποτελεί έναν κρίκο της αλυσίδας που υπολογίζεται για την κατασκευή του πίνακα και έτσι την ονομάζουμε chainNode. Δοθέντος ότι σε κάθε κρίκο υπολογισμού μιας αλυσίδας, ο KASUMI ελέγχεται από ένα καινούριο κλειδί και απαιτείται και ένα διάνυσμα για την διατήρηση των ενδιάμεσων αποτελεσμάτων κατά τον υπολογισμό της αλυσίδας, η δομή αυτή περιέχει τις μεταβλητές για τα επιμέρους κλειδιά του KASUMI, ορισμένες ως arrays οκτώ στοιχείων των 16 bits το καθένα, και ένα διάνυσμα μήκους 256 bits. Το διάνυσμα αυτό ορίζεται ως μία ένωση, που μας δίνει τη δυνατότητα να επενεργούμε σε αυτό είτε ανά τμήματα των 16 bits είτε ανά τμήματα των 32 bits και την ονομάζουμε REG256.

Η χρήση μιας συνδεδεμένης λίστας για την αποθήκευση των αρχικών και τελικών σημείων του πίνακα, αν και επιτελεί το σκοπό της, αποδείχτηκε αρκετά δύσχρηστη μέθοδος, καθότι για κάθε αρχικό σημείο που φορτώνεται από το αρχείο που τα περιέχει, απαιτείται ξεχωριστά δέσμευση μνήμης στο σύστημα για κάθε κόμβο της λίστας. Έτσι, επιλέξαμε να αντικαταστήσουμε την συνδεδεμένη λίστα με έναν δείκτη σε τύπο κρίκου αλυσίδας (chainNode), για τον οποίο να δεσμεύεται εξαρχής και εξολοκλήρου η απαιτούμενη μνήμη για όλα τα αρχικά σημεία. Τον δείκτη αυτόν, μπορούμε να τον διαχειριζόμαστε ταυτόχρονα και ως μια μεταβλητή τύπου πίνακα, μια ιδιότητα που πηγάζει από την γλώσσα προγραμματισμού C εξορισμού της.

Με βάση το μοντέλο προγραμματισμού CUDA και τη δομή του κώδικα, όπως περιγράφεται στην ενότητα 4.3, είναι απαραίτητος και ο διαχωρισμός του πηγαίου κώδικα της εφαρμογής μας σε Host Code και Device Code αντίστοιχα. Έτσι, ως συναρτήσεις που καλούνται από το device, ορίζονται όλες οι συναρτήσεις που χρησιμοποιούνται για την παραγωγή των κλειδιών ροής, καθώς και οι συναρτήσεις αναγωγής, αλλά καί οι initiate() και stepfunction(). Ο Kernel, που τον ονομάζουμε rainbowKernel(), υλοποιεί τον υπολογισμό των αλυσίδων για την κατασκευή των πινάκων και χρησιμοποιεί όλες τις παραπάνω συναρτήσεις. Όλο το κομμάτι του Device Code περιέχεται στο αρχείο kasumi.cu ενώ το κομμάτι του Host Code αποτελείται από τα αρχεία main.c και το rainbowtb.c, το οποίο κατά βάση περιέχει τις συναρτήσεις readData() και saveData() για την φόρτωση των αρχικών σημείων και την αποθήκευση του πίνακα αντίστοιχα. Για την επέκταση του κλειδιού συνεδρίας Kc σε 128 bits, αντικαθιστούμε την συνάρτηση buildKey(), που ήταν αρκετά πολύπλοκη ώστε να υποστηρίζει επέκταση κλειδιών διαφόρων μεγεθών, με μια πιο απλή, την extKey(), η οποία υλοποιεί την επέκταση του κλειδιού μόνο με δύο πράξεις αντιγραφής, θεωρώντας δεδομένο ότι το κλειδί συνεδρίας θα έχει πάντα σταθερό μήκος ίσο με 64 bits. Τέλος, οι μεταβλητές CA, CB, count (CC), CD και CL, ορίζονται ως καθολικές μεταβλητές στο κομμάτι του Device Code, μιάς και εκεί χρησιμοποιούνται και, ως εκ τούτου, τοποθετούνται στην καθολική μνήμη της κάρτας γραφικών. Ακολουθεί το σχήμα 6.5 με το διάγραμμα της δομής του πηγαίου κώδικα, όπως περιγράφεται σε

αυτή την παράγραφο.



Σχήμα 6.5: Κατασκευή Πινάκων σε CUDA - Δομή Κώδικα

Ροή Εκτέλεσης

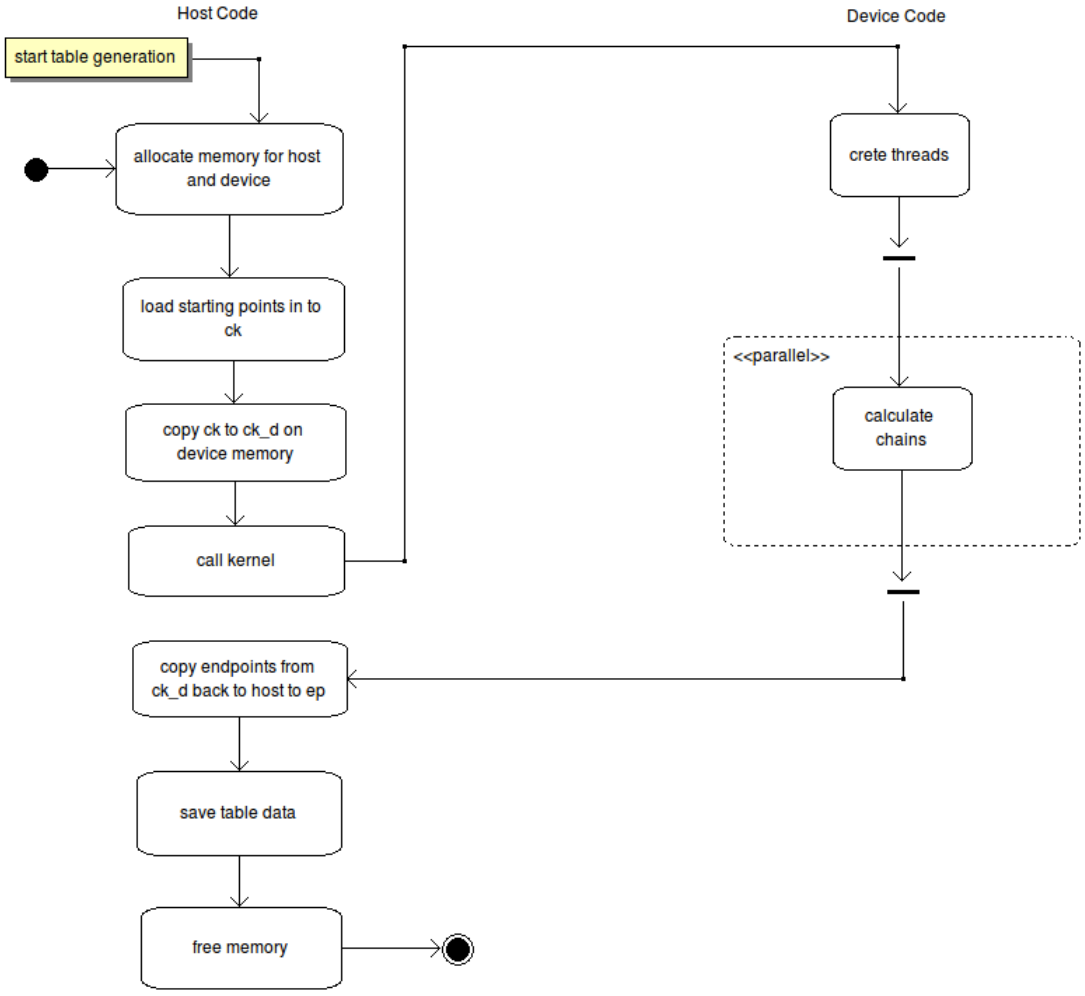
Η εφαρμογή για την κατασκευή πινάκων σε CUDA ξεκινά με την δέσμευση μνήμης για τα αρχικά και τελικά σημεία στον host και για τα αρχικά σημεία στο device. Έτσι, για τον host, ορίζουμε έναν δείκτη τύπου chainNode για τα αρχικά και έναν για τα τελικά σημεία, τους ck και ep αντίστοιχα. Με όμοιο τρόπο, για τα αρχικά σημεία στο device, ορίζουμε τον δείκτη τύπου chainNode ck_d.

Αφού δεσμευτεί η μνήμη για τους παραπάνω δείκτες, η εφαρμογή διαβάζει από το εξωτερικό αρχείο τα αρχικά σημεία και τα αποθηκεύει στο διάνυσμα τύπου REG256 των αντίστοιχων δομών chainNode ανά ποσότητες των 16 bits.

Εν συνεχεία, τα αρχικά σημεία μεταφέρονται στην καθολική μνήμη της κάρτας γραφικών και καλείται ο kernel προς εκτέλεση, όπου ο έλεγχος μεταφέρεται από τον Host Code στον Device Code. Ο kernel δημιουργεί από ένα νήμα για κάθε αρχικό σημείο και κάθε νήμα υπολογίζει μια αλυσίδα μήκους link κρίκων. Ο υπολογισμός μιας αλυσίδας γίνεται με τον ίδιο ακριβώς τρόπο που γίνεται και στην σειριακή έκδοση σε C. Να σημειώσουμε εδώ, ότι στην παρούσα έκδοση, ο kernel δεν μεταφέρει καθόλου τα δεδομένα από την καθολική μνήμη στους καταχωρητές, πράγμα φυσικά που κάνει την εφαρμογή μας πάρα πολύ αργή.

Όταν όλα τα νήματα ολοκληρώσουν τους υπολογισμούς τους, ο έλεγχος επιστρέφει και πάλι στον Host Code και τα τελικά σημεία μεταφέρονται από την κάρτα γραφικών στον δείκτη ep. Αφού στην συνέχεια αποθηκευτεί ο πίνακας σε ένα αρχείο, η εφαρμογή ολοκληρώνει την εκτέλεσή της, αποδεσμεύοντας την μνήμη που είχε δεσμευτεί αρχικά.

Ένα διάγραμμα της ροής εκτέλεσης μας δείχνει το σχήμα 6.6.



Σχήμα 6.6: Κατασκευή Πινάκων σε CUDA - Ροή Εκτέλεσης

6.6 Ο Κώδικας του Cryptohaze

Με την ολοκλήρωση του πρωτότυπου της εφαρμογής κατασκευής πινάκων σε CUDA, ξεκινήσαμε μια έρευνα στο διαδίκτυο, για την αναζήτηση παρόμοιων εφαρμογών, προκειμένου να έχουμε ένα μέτρο σύγκρισης για την εφαρμογή μας. Μεταξύ άλλων, βρήκαμε και το project του Cryptohaze.com [43]. Ο κώδικας του Cryptohaze είναι διαθέσιμος ως ανοικτό λογισμικό και υποστηρίζει την κατασκευή πινάκων Ουράνιου Τόξου για μια σειρά από συναρτήσεις κατακερματισμού, συγκεκριμένα τις MD4, MD5, NTLM και SHA1. Υποστηρίζει ακόμη και την χρήση των πινάκων για αποκρυπτογράφηση δεδομένων, καθώς και κάποιες πρόσθετες λειτουργίες όπως την συνένωση και την δεικτοδότηση πινάκων. Έχει αναπτυχθεί με βάση το αντικειμενοστραφές μοντέλο προγραμματισμού και έτσι χρησιμοποιεί κυρίως την γλώσσα

προγραμματισμού C++. Πρόκειται για έναν αρκετά αποδοτικό κώδικα, με βάση τις αναφορές του δημιουργού του, αλλά η επιλογή του αντικειμενοστραφούς μοντέλου τον καθιστά αρκετά περίπλοκο και είναι διαρθρωμένος σε έναν μεγάλο αριθμό αρχείων.

Δοθείσας της ολοκληρωμένης λειτουργικότητας του κώδικα του Cryptohaze, αποπειραθήκαμε να ενσωματώσουμε σε αυτόν την λειτουργία για την κατασκευή πινάκων για τον αλγόριθμο A5/3. Για το όλο εγχείρημα αφιερώσαμε ένα σεβαστό κομμάτι χρόνου λόγο της έκτασης του κώδικα, καθότι χρειαζόταν να επέμβουμε με τροποποιήσεις στο μεγαλύτερο ποσοστό των αρχείων. Δυστυχώς, όλη αυτή η προσπάθεια αποδείχτηκε μάταιη, διότι ο κώδικας του Cryptohaze είναι δομημένος ώστε να επενεργεί στα δεδομένα ανά τμήματα των 32 bits σε όλο το εύρος των λειτουργιών του και αυτό, στην περίπτωσή μας του A5/3 που επενεργεί στα δεδομένα ανά τμήματα των 16 bits, λόγω του διαφορετικού endian μοντέλου ανάμεσα στον επεξεργαστή και την κάρτα γραφικών, διέφθειρε τα αποτελέσματα των πράξεων. Το ισοδύναμο χρόνου που θα χρειαζόταν να καταναλωθεί για την επίλυση αυτού του προβλήματος, δεν δικαιολογούσε την περαιτέρω ενασχόλησή μας με τον κώδικα του Cryptohaze. Η ολοκλήρωση και βελτίωση του πρωτότυπου κώδικα που ήδη είχαμε αναπτύξει, μπορούσε να πραγματοποιηθεί σε ισοδύναμο αν όχι σε λιγότερο χρόνο και θα είχε μια πιο καθαρή δομή, προσαρμοσμένη αυστηρά στις ανάγκες της συγκεκριμένης δουλειάς.

Ωστόσο, η ενασχόλησή μας με τον κώδικα του Cryptohaze μας έδωσε διάφορες ιδέες για βελτιωτικές τροποποιήσεις του κώδικά μας, από λειτουργική σκοπιά, αρκετές από τις οποίες και εφαρμόσαμε και τις αναλύουμε στην ενότητα 6.7 που ακολουθεί.

6.7 Λειτουργικές Βελτιώσεις του Κώδικα CUDA

Από τις πρώτες και σημαντικότερες λειτουργικές βελτιώσεις που έπρεπε να εφαρμοστούν στην εφαρμογή μας, ήταν η μεταφορά των αρχικών σημείων για τον υπολογισμό των αλυσίδων, από την καθολική μνήμη της κάρτας γραφικών στους καταχωρητές του εκάστοτε νήματος. Θυμίζουμε ότι, όπως έχουμε περιγράψει στην ενότητα 4.4, οι καταχωρητές ορίζονται ως μεταβλητές εντός του kernel. Έτσι, εντός της συνάρτησης rainbowKernel() που αποτελεί το σώμα του kernel για τον υπολογισμό των αλυσίδων, ορίζουμε ένα στιγμιότυπο της δομής chainNode που το ονομάζουμε newChainNode και το οποίο λειτουργεί ως καταχωρητής για την αποθήκευση των αρχικών σημείων στο κάθε νήμα. Την μεταφορά των δεδομένων από την καθολική μνήμη στους καταχωρητές, αναλαμβάνει μια νέα συνάρτηση που ονομάζουμε LoadRegistersFromGlobalMemory().

Κρίνοντας ότι, κατά των υπολογισμό των αλυσίδων εντός του kernel, είναι πιο λειτουργικό για το κλειδί συνεδρίας και το κλειδί ροής να αποθηκεύονται σε ξεχωριστά διανύσματα, αλλάξαμε την δομή chainNode ώστε να περιέχει ένα διάνυσμα secretKey μήκους 128 bits και ένα διάνυσμα cipher μήκους 256 bits αντίστοιχα. Κάθε ένα από αυτά τα διανύσματα, υλοποιείται ως μια ένωση που δίνει τη δυνατότητα να προσπελαστούν τα δεδομένα τους είτε ανά τμήματα των 16 bits είτε ανά τμήματα των 32 bits, και πάλι για την διασφάλιση της σωστής τοποθέτησης των δεδομένων στην μνήμη του συστήματος λόγω του διαφορετικού endian μοντέλου.

Το μοτίβο χρήσης των συναρτήσεων initiate() και stepfunction() για τον υπολογισμό των αλυσίδων έμοιαζε να προσθέτει στον κώδικα μια πολυπλοκότητα άνευ

αναγκαιότητας. Για τον λόγο αυτό, καταργήσαμε αυτές τις δυο συναρτήσεις και ο υπολογισμός των αλυσίδων υλοποιείται πλέον με βάση ενός μοναδικού βρόχου for() που εκτελεί επαναληπτικά τις λειτουργίες επέκτασης κλειδιού συνεδρίας σε 128 bits, παραγωγής κλειδιού ροής και αναγωγής του κλειδιού ροής σε ένα καινούριο κλειδί συνεδρίας, για το επιθυμητό μήκος αλυσίδας.

Στο σημείο αυτό, ήταν επίσης σκόπιμο να αναπροσαρμόσουμε κατάλληλα την συνάρτηση αναγωγής, ώστε η εφαρμογή μας να παράγει πίνακες Ουράνιου Τόξου αντί για Πίνακες Hellmann που παρήγαγε ως τώρα. Έτσι, με βάση μια επεξήγηση που υπάρχει στο forum του project Free Rainbow Tables [45], αλλά όπως πηγάζει και από τον ορισμό της συνάρτησης αναγωγής που αναφέρεται στις ενότητες 2.1 και 2.3, μια κατάλληλη, ντετερμινιστική συνάρτηση, διαφορετική για κάθε στήλη του πίνακα, μπορεί να έχει τη μορφή:

[Result] =
([First 64 bit of hash] + [Table Index] + [Position in chain]) % [KeySpace].
Η χρήση του δείκτη σε έναν πίνακα (Table Index), μια παράμετρος που επιλέγεται κατά βούληση για την αρίθμηση μεμονωμένων πινάκων που ανοίκουν στην ίδια συλλογή και την οποία υιοθετούμε στην εφαρμογή μας, χρησιμεύει στην διαφοροποίηση των συναρτήσεων αναγωγής από πίνακα σε πίνακα. Η δε χρήση της θέσης στην αλυσίδα (Position in Chain), εγγυάται τη διαφοροποίηση της συνάρτησης αναγωγής για κάθε στήλη του πίνακα. Ο χώρος των πιθανών κλειδιών (KeySpace) στην δική μας περίπτωση, ορίζεται στην τιμή 2^{64} και την θέση του hash παίρνει το κλειδί ροής. Σύμφωνα με αυτή τη φιλοσοφία, αναπροσαρμόσαμε τη συνάρτηση αναγωγής που χρησιμοποιούμε στην εφαρμογή μας.

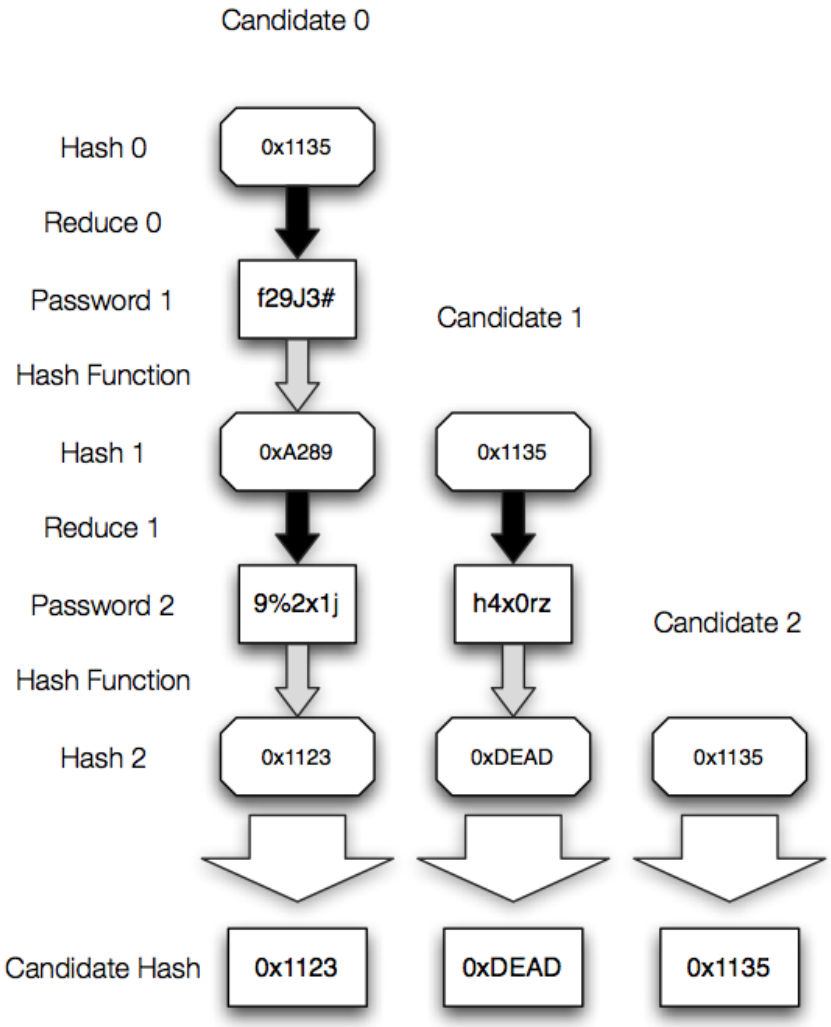
Τέλος, προκειμένου να αποφευχθεί η χρονοβόρα και πολύπλοκη διαδικασία, του να δημιουργούμε αρχεία που θα περιέχουν τα αρχικά σημεία, αναπτύξαμε έναν μηχανισμό παραγωγής τυχαίων αρχικών σημείων, που καλείται με την έναρξη της εφαρμογής κατασκευής πινάκων. Ο μηχανισμός αυτός, υλοποιήθηκε ως ένας maximum length LFSR εύρους 32 bits. Η επιλογή των κατάλληλων taps για τη maximum length λειτουργία έγινε σύμφωνα με έναν πίνακα που περιέχεται σε ένα τεχνικό δελτίο της XILINX [46]. Για να διασφαλίσουμε ότι σε κάθε πίνακα που κατασκευάζεται, παράγονται διαφορετικά αρχικά σημεία, ο LFSR για κάθε διαφορετικό πίνακα που κατασκευάζεται, αρχικοποιείται σε διαφορετική τιμή. Επίσης, για να ήμαστε σίγουροι ότι όλα τα αρχικά σημεία που θα παραχθούν από τον LFSR είναι μοναδικά, η εφαρμογή διαθέτει μια δικλίδα ασφαλείας που μας ενημερώνει για την περίπτωση που ο LFSR επιστρέψει στην αρχική του κατάσταση με κατάλληλο μήνυμα και διακόπτει την εφαρμογή.

6.8 Αποκρυπτογράφηση Δεδομένων

Πριν να προχωρήσουμε στην διαδικασία της βελτιστοποίησης της εφαρμογής για την αύξηση των επιδόσεων, δώσαμε προτεραιότητα στην ανάπτυξη του δεύτερου σκέλους της εφαρμογής μας, αυτό της χρήσης των πινάκων για την αποκρυπτογράφηση δεδομένων και την ανάκτηση του κλειδιού συνεδρίας, αν και εφόσον αυτό υπάρχει στον πίνακα. Η διαδικασία της χρήσης των πινάκων για την αποκρυπτογράφηση και ανάκτηση κλειδιού ακολουθεί μια σειρά από στάδια, όπως αυτά περιγράφονται στην

συνέχεια.

Το πρώτο στάδιο στην αποκρυπτογράφηση δεδομένων, όπως έχουμε εξηγήσει και στο θεωρητικό σκέλος στην ενότητα 2.1, είναι να ελέγξουμε αν, για το εν λόγω κρυπτογραφημένο κείμενο που μας ενδιαφέρει να αποκρυπτογραφήσουμε, το κλειδί ροής που το παρήγαγε εμφανίζεται ως τελικό σημείο στον πίνακά μας. Έτσι, δοθέντος ότι για την κρυπτογράφηση εφαρμόζουμε $keystream \oplus plaintext = ciphertext$, για να ανακτήσουμε το κλειδί ροής από το κρυπτογραφημένο κείμενο, εφαρμόζουμε $ciphertext \oplus plaintext = keystream$, όπου το $plaintext$ αποτελεί το επιλεγμένο κείμενο για την εφαρμογή της επίθεσης. Αν υπάρξει ταύτιση κατά την αναζήτηση στον πίνακα για το ανακτηθέν κλειδί ροής, αναπαράγουμε την αλυσίδα που αντιπροσωπεύει το αντίστοιχο αρχικό σημείο, ως το τελικό της σημείο και ανακτούμε το κλειδί που βρίσκεται μια θέση πριν την εφαρμογή της τελευταίας συνάρτησης αναγωγής. Διαφορετικά, αν δεν υπάρξει ταύτιση, εφαρμόζουμε ένα-ένα τα βήματα υπολογισμού μιας αλυσίδας στο ανακτιθέν κλειδί ροής και ελέγχουμε περιοδικά αν κάποιο από τα ενδιάμεσα αποτελέσματα εμφανίζεται ως τελικό σημείο στον πίνακα.

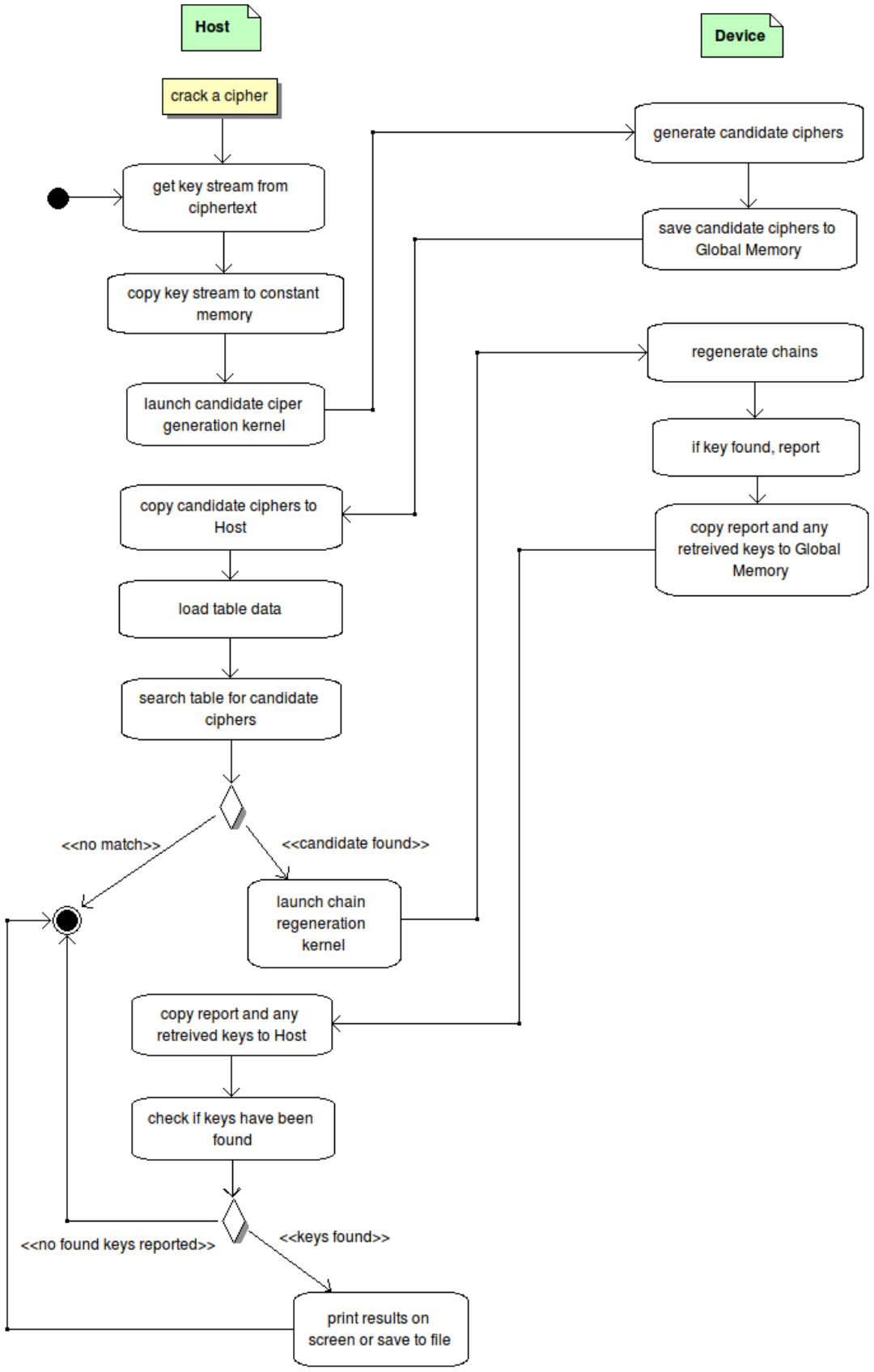


Σχήμα 6.7: Παραγωγή Candidate Ciphers [47]

Η παραπάνω διαδικασία της αναζήτησης, σε επίπεδο κώδικα, υλοποιείται με τον εξής τρόπο. Πρώτα θα πρέπει να παράγξουμε από το ανακτηθέν κλειδί ροής, μια λίστα με όλα τα πιθανά τελικά σημεία που μπορεί να περιλαμβάνει ο πίνακας, δοθέντος ότι το εν λόγο κλειδί ροής, κατά τον υπολογισμό της αλυσίδας, θα μπορούσε να έχει εμφανιστεί σε κάθε πιθανή θέση αυτής. Αυτή η λίστα θα περιέχει τόσα πιθανά τελικά σημεία, όσα και το μήκος της αλυσίδας, ή αλλιώς, όσες και οι συναρτήσεις αναγωγής που χρησιμοποιήθηκαν, συμπεριλαμβανομένου και του εν λόγω κλειδιού ροής. Ονομάζουμε την λίστα αυτή **Candidate Ciphers** και η παραγωγή της εκτελείται εν παραλλήλω στην κάρτα γραφικών, με την κλήση ενός kernel που τον ονομάζουμε CandidateCiphersGenerationKernel(). Η παραλληλοποίηση ακολουθεί και εδώ το μοτίβο της κατασκευής πινάκων, όπου κάθε νήμα αναλαμβάνει τον υπολογισμό μιας αλυσίδας. Ένα παράδειγμα της παραγωγής των candidate ciphers με τυχαία δεδομένα, όπως εφαρμόζεται για συναρτήσεις κατακερματισμού, βλέπουμε στο σχήμα 6.7.

Στο επόμενο στάδιο, με την ολοκλήρωση της παραγωγής των Candidate Ciphers, ο έλεγχος επιστρέφει στον Host, όπου διενεργείται μια αναζήτηση για τον έλεγχο ταύτισης κάποιου από τα Candidate Ciphers με κάποιο από τα τελικά σημεία του πίνακα. Για κάθε ταύτιση που θα υπάρξει, δημιουργείται μια λίστα που περιέχει τα συσχετιζόμενα αρχικά σημεία του πίνακα. Όταν ολοκληρωθεί η αναζήτηση, ο έλεγχος περνάει και πάλι στο Device, για την εκτέλεση του τελικού σταδίου όπου, με την χρήση ενός kernel που τον ονομάζουμε RegenerateChainsKernel(), υπολογίζονται εκ νέου οι επιλεγμένες αλυσίδες από τη λίστα των αρχικών σημείων για την ανάκτηση του κλειδιού. Με την ολοκλήρωση του RegenerateChainsKernel(), ο έλεγχος επιστρέφει στον Host, όπου τα αποτελέσματα της ανάκτησης κλειδιού, είτε τυπώνονται στην οθόνη είτε αποθηκεύονται σε ένα αρχείο κατ επιλογή. Στο διάγραμμα του σχήματος 6.8 βλέπουμε τη ροή εκτέλεσης για το σκέλος της αποκρυπτογράφησης.

Οι πιθανές εκβάσεις κατά το σκέλος της αποκρυπτογράφησης έχουν ως εξής. Υπάρχει το ενδεχόμενο να μην υπάρξει ταύτιση, οπότε ο πίνακας που χρησιμοποιούμε δεν περιέχει το επιθυμητό κλειδί και τα δεδομένα δεν μπορούν να αποκρυπτογραφηθούν. Άλλο ενδεχόμενο είναι να υπάρξει ταύτιση, αλλά να μην βρεθεί το αντίστοιχο κλειδί. Αυτή είναι η περίπτωση ενός false alarm όπως έχουμε εξηγήσει στην ενότητα 2.1. Τέλος, υπάρχει το επιθυμητό ενδεχόμενο, από μια ταύτιση να ανακτηθεί στο σωστό κλειδί. Να σημειώσουμε εδώ, ότι υπάρχει επίσης η πιθανότητα να έχουμε περισσότερες από μία ταυτίσεις κατά την αναζήτηση. Σε αυτή την περίπτωση στον πίνακά μας υπάρχει collision της περίπτωσης όπου, το ίδιο κρυπτογραφημένο κείμενο εμφανίζεται σε διαφορετικές αλυσίδες, αλλά όχι στην ίδια στήλη του πίνακα, για αυτό το λόγο και δεν έχει υπάρξει συγχώνευση των αλυσίδων. Από τα διαφορετικά κλειδιά που θα ανακτηθούν στην περίπτωση του collision, το πολύ ένα μπορεί να είναι έγκυρο.



Σχήμα 6.8: Αποκρυπτογράφηση Δεδομένων - Ροή Εκτέλεσης

Κεφάλαιο 7

Βελτιστοποίηση και Επιδόσεις

Στο σημείο αυτό η εφαρμογή μας βρίσκεται στο στάδιο μιας λειτουργικής έκδοσης για παράλληλη εκτέλεση σε CUDA και υποστηρίζει καί τις δύο επιθυμητές λειτουργίες, αυτή της κατασκευής πινάκων Ουράνιου Τόξου, το offline κομμάτι προϋπολογισμών όπως χαρακτηρίζεται στην βιβλιογραφία, αλλά και αυτή της χρήσης των πινάκων για την αποκρυπτογράφηση δεδομένων και την ανάκτηση του κλειδιού συνεδρίας, το λεγόμενο online ή real time κομμάτι βάση βιβλιογραφίας. Εφεξής, η προτεραιότητα δίδεται στην αξιολόγηση και ανάλυση της εφαρμογής σε σχέση με τις επιδόσεις και τους χρόνους εκτέλεσης που μπορούν να επιτευχθούν στην αρχιτεκτονική CUDA. Στο κεφάλαιο αυτό θα αναφερθούμε στα βήματα που ακολουθήσαμε για τη βελτιστοποίηση της εφαρμογής μας με σκοπό την αύξηση των επιδόσεων αυτής, ακολουθώντας τις προτεινόμενες πρακτικές από τη βιβλιογραφία.

7.1 Αξιοποίηση Ιδιοτήτων Μνημών της CUDA

Έχουμε ήδη αναλύσει στην ενότητα 4.4 τα πλεονεκτήματα και μειονεκτήματα των διαφόρων τύπων μνήμης που διαθέτει μια κάρτα γραφικών κατασκευασμένη με βάση την αρχιτεκτονική CUDA. Στο στάδιο αυτό, γνωρίζουμε ότι οι παράμετροι εισόδου για την συνάρτηση KGCore() (CA, CB, count, CD, CL), δοθέντος ότι είναι τοποθετημένες στην καθολική μνήμη, προκαλούν μεγάλη καθυστέρηση στην εκτέλεση της εφαρμογής μας. Η KGCore() καλείται για κάθε κρίκο της κάθε αλυσίδας που υπολογίζεται και σε κάθε κλήση της έχουμε για κάθε παράμετρο μια προσπέλαση της καθολικής μνήμης, μια μνήμη που, όπως έχουμε εξηγήσει είναι πάρα πολύ αργή.

Μια λύση σε αυτό το πρόβλημα, αποτελεί η επιλογή μιας άλλης μνήμης προς τοποθέτηση αυτών των παραμέτρων. Λαμβάνοντας υπόψιν το γεγονός ότι, αυτές οι παράμετροι έχουν σταθερές τιμές καθ΄ όλη τη διάρκεια εκτέλεσης της εφαρμογής, δοκιμάζουμε να τις τοποθετήσουμε στην σταθερή μνήμη και να επωφεληθούμε από την ιδιότητα της cache. Δοκιμάζουμε επίσης, να τοποθετήσουμε στην σταθερή μνήμη και τα κουτιά αντικατάστασης S-Boxes, ώστε να μην αποτελούν πλέον τοπικές μεταβλητές της υποσυνάρτησης FI() και να έχουμε έτσι τον έλεγχο για το σε ποια μνήμη τοποθετούνται, αποφεύγοντας ταυτόχρονα την δημιουργία πολλαπλών αντιγράφων τους σε κάθε κλήση της FI().

Η εφαρμογή εκτελείται για την κατασκευή 65536 αλυσίδων μήκους 100000 κρίκων η κάθε μία, διαρθρωμένες σε 512 blocks των 128 threads. Για αυτό το μέγεθος,

ο χρόνος εκτέλεσης που παίρνουμε είναι 4 λεπτά και 8 δευτερόλεπτα. Ενδεικτικά αναφέρουμε ότι ο χρόνος εκτέλεσης πριν τη χρήση της σταθερής μνήμης και για ίδιο μέγεθος πίνακα, ήταν στις τέσσερις ώρες, σαράντα πέντε λεπτά και δέκα εννέα δευτερόλεπτα. Επομένως, η χρήση της σταθερής μνήμης δίνει μια επιτάχυνση της τάξεως του 59x.

Για την αποκρυπτογράφηση ενός κρυπτογραφημένου κειμένου, η εφαρμογή εκτελείται για σύνολο 100000 threads διαρθρωμένα σε 782 blocks των 128 threads. Ο χρόνος εκτέλεσης της εφαρμογής για την παραγωγή των υποψήφιων τελικών σημείων και την αναζήτηση στον πίνακα, καθώς και για την αναπαραγωγή των απαραίτητων αλυσίδων για την ανάκτηση του κλειδιού, ανέρχεται στα τρία λεπτά και δέκα έξι δευτερόλεπτα. Η επιτάχυνση που επιτυγχάνεται σε σχέση με το πρωτότυπο είναι της τάξεως του 67x.

Δοκιμάσαμε εναλλακτικά την τοποθέτηση των παραμέτρων εισόδου της KGCore() στους καταχωρητές και χρησιμοποιήσαμε δείκτες για την προσπέλασή τους. Η πρακτική αυτή δεν βελτίωσε καθόλου τον χρόνο εκτέλεσης. Έτσι αποφασίσαμε να κρατήσουμε τις παραμέτρους εισόδου της KGCore() τοποθετημένες στην σταθερή μνήμη, εξασφαλίζοντας περισσότερους διαθέσιμους καταχωρητές για άλλες χρήσεις, αλλά και να διατηρήσουμε και την προσπέλασή τους με χρήση δεικτών, που από προγραμματιστική άποψη τη θεωρούμε καλύτερη ως πρακτική.

7.2 Επαναπροσδιορισμός Μεταβλητών

Αν και η δομή chainNode βοήθησε αρχικά στο να διαθέτει το κάθε νήμα τις δικές του, ιδιωτικές μεταβλητές για τα επιμέρους κλειδιά του KASUMI, το να μεταφέρονται αυτές από και προς τον Host, αποτελεί έναν πλεονασμό, καθότι δεν έχουν καμία χρησιμότητα στο κομμάτι του host code. Για τον λόγο αυτό, ακυρώνουμε την χρήση της δομής chainNode. Για την διατήρηση των επιμέρους κλειδιών του KASUMI στους καταχωρητές, ορίζουμε μια νέα δομή που περιέχει αυστηρά και μόνο αυτά, την οποία ονομάζουμε subKeys και κάθε νήμα δημιουργεί ένα στιγμιότυπο αυτής προς χρήση. Το στιγμιότυπο αυτό, δίδεται ως είσοδος σε όποια συνάρτηση απαιτείται με τη χρήση δεικτών.

Με την ακύρωση της δομής chainNode χρειαζόμαστε έναν καινούριο σετ καταχωρητών για τον υπολογισμό των αλυσίδων. Για τη χρήση αυτή ορίζουμε δύο νέες ενώσεις, την register128 και register256, που αποτελούν διανύσματα μήκους 128 bits και 256 bits αντίστοιχα. Σε ότι αφορά την προσπέλαση των δεδομένων, τα διανύσματα αυτά ακολουθούν την ίδια φιλοσοφία με τις προηγούμενες ενώσεις για την αποφυγή προβλημάτων σε σχέση με το θέμα του endianness. Τώρα, κάθε νήμα έχει ένα στιγμιότυπο της register128 για το κλειδί συνεδρίας (newKey) και ένα στιγμιότυπο της register256 για το κλειδί ροής/κρυπτογραφημένο κείμενο (newCipher).

Η αποδόμηση αυτή μας έδωσε μία επιτάχυνση της τάξης των δέκα περίπου δευτερολέπτων, τόσο κατά την κατασκευή πινάκων αλλά και για το κομμάτι της αποκρυπτογράφησης, που λογικά προκύπτει από το γεγονός ότι πλέον μεταφέρονται λιγότερα δεδομένα από και προς την κάρτα γραφικών, αλλά όσο αφορά τον χρόνο εκτέλεσης του kernel, ίσως και να μην δίνει ιδιαίτερη βελτίωση. Σε γενικές γραμμές, αυτή η αλλαγή οδήγησε σε μια επιτάχυνση της τάξεως του 1x.

7.3 Αποδόμηση Βρόχων

Οποιαδήποτε εντολή ελέγχου ροής, όπως οι εντολές if, for, while κτλ., μπορούν να οδηγήσουν νήματα που ανοίκουν στο ίδιο warp να αποκλίνουν από το κοινό μονοπάτι εκτέλεσης και να ακολουθήσουν ένα διαφορετικό. Στην περίπτωση που συμβεί κάτι τέτοιο, τα διαφορετικά μονοπάτια δεν εκτελούνται εν παραλλήλω, καθότι όλα τα νήματα σε ένα warp μοιράζονται έναν κοινό program counter, και έτσι πρέπει να εκτελεστούν σειριακά. Αποτέλεσμα είναι να αυξάνεται ο συνολικός αριθμός των εντολών που έχει να εκτελέσει αυτό το warp και επομένως και ο χρόνος εκτέλεσης της εφαρμογής.

Σε μερικές περιπτώσεις ο μεταγλωττιστής της Nvidia μπορεί να αποφασίσει να αποδομήσει έναν βρόχο ελέγχου ροής, προκειμένου να αποφευχθεί η απόκλιση από το κοινό μονοπάτι εκτέλεσης. Στις περιπτώσεις όμως που δεν θα τα καταφέρει, συνίσταται την αποδόμηση αυτή να την αναλάβει ο προγραμματιστής, τροποποιώντας κατάλληλα τον κώδικα.

Η συνάρτηση KGCore() χρησιμοποιεί έναν βρόχο while για την παραγωγή του κλειδιού ροής, εντός του οποίου καλείται η συνάρτηση Kasumi() όσες φορές χρειαστεί προκειμένου να παραχθεί ένα κλειδί ροής του απαιτούμενου μήκους CL. Εντός του βρόχου while, κάθε φορά που μια κλήση στην Kasumi() επιστρέφει ένα τμήμα του κλειδιού ροής, αυτό αποθηκεύεται στο προβλεπόμενο διάνυσμα εξόδου με χρήση ενός βρόχου for. Έχουμε λοιπόν έναν σύνθετο βρόχο τύπου, βρόχος μέσα σε βρόχο. Παραθέτουμε το εν λόγω κομμάτι από τον πηγαίο κώδικα:

```
while(cl > 0)
{
        //First we calculate the next 64-bits of keystream
        //XOR in A and BLKCNT to last value
        temp[0] ^= A[0];
        temp[1] ^= A[1];
        temp[1] ^= blkcnt;


        //KASUMI it to produce the next block of keystream
        Kasumi(temp, nod);


        //Set <n> to the number of bytes of input data
        //we have to modify.
        if(cl >= 64)
        n=2;
        else
        n = (cl+31)/32;


        //Copy out the keystream
        for(i=0; i<n; ++i)
```

```
        {
        *co++ = temp[i]>>16;
        *co++ = temp[i];
        }

        cl -= 64; //Done another 64 bits
        ++blkcnt; //increment BLKCNT
}
```

Αποδομήσαμε τον παραπάνω σύνθετο βρόχο και χρησιμοποιούμε πλέον ένα έλεγχο if μετά από κάθε κλήση στην Kasumi() για να ελέγχουμε το πότε έχει ολοκληρωθεί η παραγωγή του κλειδιού ροής. Παραθέτουμε τον πηγαίο κώδικα με τον αποδομημένο βρόχο:

```
//Now run the key stream generator

// First key stream block
temp[0] ^= A[0];
temp[1] ^= A[1];
temp[1] ^= blkcnt;

Kasumi(temp, nod);

*co++ = temp[0]>>16;
*co++ = temp[0];
*co++ = temp[1]>>16;
*co++ = temp[1];

cl -= 64; //Done another 64 bits
++blkcnt; //increment BLKCNT

if(cl <= 0)
return;

// Second key stream block
temp[0] ^= A[0];
temp[1] ^= A[1];
temp[1] ^= blkcnt;

Kasumi(temp, nod);

*co++ = temp[0]>>16;
*co++ = temp[0];
*co++ = temp[1]>>16;
*co++ = temp[1];
```

```
cl -= 64; //Done another 64 bits
++blkcnt; //increment BLKCNT

if(cl <= 0)
return;

// Third key stream block
temp[0] ^= A[0];
temp[1] ^= A[1];
temp[1] ^= blkcnt;

Kasumi(temp, nod);

*co++ = temp[0]>>16;
*co++ = temp[0];
*co++ = temp[1]>>16;
*co++ = temp[1];

cl -= 64; //Done another 64 bits
++blkcnt; //increment BLKCNT

if(cl <= 0)
return;

// Fourth key stream block
temp[0] ^= A[0];
temp[1] ^= A[1];
temp[1] ^= blkcnt;

Kasumi(temp, nod);

*co++ = temp[0]>>16;
*co++ = temp[0];
*co++ = temp[1]>>16;
*co++ = temp[1];

cl -= 64; //Done another 64 bits
++blkcnt; //increment BLKCNT

if(cl <= 0)
return;
```

Η αποδόμηση αυτή έδωσε μια βελτίωση στον χρόνο εκτέλεσης, που από τα τέσσερα λεπτά μειώθηκε στα δύο λεπτά και είκοσι ένα δευτερόλεπτα για την κατασκευή πινάκων, μια επιτάχυνση της τάξεως του 1.7x. Για την αποκρυπτογράφηση, πήραμε χρόνο εκτέλεσης ίσο με ένα λεπτό και πενήντα δύο δευτερόλεπτα και μια επιτάχυνση της τάξεως του 1.7x επίσης.

Δοκιμάσαμε στη συνέχεια να εφαρμόσουμε την στρατηγική αποδόμησης βρόχων και σε άλλες συναρτήσεις, όπως για παράδειγμα στον βρόχο της συνάρτησης Kasumi() που υλοποιεί τους οκτώ κύκλους του KASUMI, αλλά καμία από τις επόμενες αποδομήσεις δεν οδήγησε σε επιπλέον επιτάχυνση. Ωστόσο, διατηρήσαμε κάποιες από αυτές τις αποδομήσεις, με το σκεπτικό ότι μπορεί να εξυπηρετήσουν σε άλλους τρόπους βελτιστοποίησης στη συνέχεια. Ενδεικτικά παραθέτουμε τον πηγαίο κώδικα για τον βρόχο do...while της συνάρτησης Kasumi() πριν και μετά την αποδόμηση.

```
//        Before Loop Unrolling
//-------------------------------------

n = 0;
do{
        temp = FL( left , n, nod);
        temp = FO( temp, n++, nod);
        right ^= temp;
        temp = FO( right , n, nod);
        temp = FL( temp, n++, nod);
        left ^= temp;
}while(n <= 7);
```

```
//        After Loop Unrolling
//-------------------------------------

temp = FL( left , 0, nod);
temp = FO( temp, 0, nod);
right ^= temp;
temp = FO( right , 1, nod);
temp = FL( temp, 1, nod);
left ^= temp;

temp = FL( left , 2, nod);
temp = FO( temp, 2, nod);
right ^= temp;
temp = FO( right , 3, nod);
temp = FL( temp, 3, nod);
left ^= temp;

temp = FL( left , 4, nod);
temp = FO( temp, 4, nod);
right ^= temp;
temp = FO( right , 5, nod);
temp = FL( temp, 5, nod);
left ^= temp;
```

```
temp = FL( left , 6, nod);
temp = FO( temp, 6, nod);
right ^= temp;
temp = FO( right , 7, nod);
temp = FL( temp, 7, nod);
left ^= temp;
```

7.4 Ελαχιστοποίηση Βαθους Κλήσης Συναρτήσεων

Όπως έχουμε εξηγήσει στο κεφάλαιο 6, σύμφωνα με τη δομή του κώδικα, οι kernels, τόσο για την κατασκευή των πινάκων όσο και για την αποκρυπτογράφηση δεδομένων, καλούν την συνάρτηση KGCore() για την κατασκευή του διανύσματος αρχικοποίησης, η οποία στην συνέχεια καλεί την συνάρτηση Kasumi() για την παραγωγή των κλειδιών ροής, ενώ η Kasumi() με την σειρά της καλεί τις υποσυναρτήσεις της. Επομένως, έχουμε ένα βάθος κλήσεων συναρτήσεων τριών επιπέδων.

Στην απόπειρά μας να εξερευνήσουμε αν αυτή η δομή επηρεάζει την απόδοση της εφαρμογής μας, δοκιμάσαμε να μειώσουμε αυτό το βάθος, απαλείφοντας αρχικά ένα επίπεδο, αυτό των υποσυναρτήσεων του KASUMI. Έτσι, οι υποσυναρτήσεις του KASUMI ορίστηκαν ως macros, μια μορφή ορίσματος που επιβάλει στον μεταγλωττιστή, να αντικαταστήσει κάθε κλήση της συνάρτησης με το σώμα της. Κατά αυτόν τον τρόπο αποφεύγεται η όποια καθυστέρηση προκαλείται κατά την μεταφορά του ελέγχου ροής από το τρέχων σημείο του κώδικα στην συνάρτηση που καλείται. Ενδεικτικά παραθέτουμε το κομμάτι του πηγαίου κώδικα για την συνάρτηση FI() όπως αυτή είναι ορισμένη σαν macro.

```
#define FI(i, n, s, k){\
        n = (i>>7);\
        s = (i&0x7f);\
        n = S9[n] ^ s;\
        s = S7[s] ^ (n & 0x7F);\
        s ^= (k>>9);\
        n ^= (k&0x1FF);\
        n = (S9[n] ^ s);\
        s = (S7[s] ^ (n & 0x7F));\
        i = (s<<9) + n;\
}
```

Ο ορισμός των υποσυναρτήσεων του KASUMI ως macros, γίνεται πιο εύκολος όταν τις αναγκάσουμε να μην επιστρέφουν τιμές. Οπότε, τις τροποποιούμε ώστε η παράμετρος εξόδου να δίδεται ως είσοδος προς αποθήκευση του αποτελέσματος, αντί να επιστρέφονται ως τιμή από την εκάστοτε συνάρτηση. Λογική απόρροια της επιλογής αυτής, είναι να χρήζει και η συνάρτηση Kasumi() κατάλληλης τροποποίησης. Με την ευκαιρία αυτή, για την χρήση των επιμέρους κλειδιών του KASUMI, αντί

να δίδεται ως είσοδος ολόκληρη η δομή τους, δίδονται μόνο τα απαραίτητα για τον εκάστοτε κύκλο κλειδιά. Ακολουθεί το κομμάτι του πηγαίου κώδικα της τροποποιημένης συνάρτησης Kasumi().

```
__device__ inline void Kasumi(register64 &data, subKeys *nod)
{
uint32_t left, right, temp, temp2;
uint16_t l_16, r_16, s9box, s7box, a, b;

/* Start by getting the data into two 32-bit words */
left = data.b32[0];
right = data.b32[1];

temp2 = left;
FL( temp2, l_16, r_16, a, b, nod->KLi1[0], nod->KLi2[0]);
temp = temp2;
FO( temp, l_16, r_16, s9box, s7box, nod->KOi1[0], nod->KOi2[0],
nod->KOi3[0], nod->KIi1[0], nod->KIi2[0], nod->KIi3[0]);
right ^= temp;
temp = right;
FO( temp, l_16, r_16, s9box, s7box, nod->KOi1[1], nod->KOi2[1],
nod->KOi3[1], nod->KIi1[1], nod->KIi2[1], nod->KIi3[1]);
FL( temp, l_16, r_16, a, b, nod->KLi1[1], nod->KLi2[1]);
left ^= temp;

temp2 = left;
FL( temp2, l_16, r_16, a, b, nod->KLi1[2], nod->KLi2[2]);
temp = temp2;
FO( temp, l_16, r_16, s9box, s7box, nod->KOi1[2], nod->KOi2[2],
nod->KOi3[2], nod->KIi1[2], nod->KIi2[2], nod->KIi3[2]);
right ^= temp;
temp = right;
FO( temp, l_16, r_16, s9box, s7box, nod->KOi1[3], nod->KOi2[3],
nod->KOi3[3], nod->KIi1[3], nod->KIi2[3], nod->KIi3[3]);
FL( temp, l_16, r_16, a, b, nod->KLi1[3], nod->KLi2[3]);
left ^= temp;

temp2 = left;
FL( temp2, l_16, r_16, a, b, nod->KLi1[4], nod->KLi2[4]);
temp = temp2;
FO( temp, l_16, r_16, s9box, s7box, nod->KOi1[4], nod->KOi2[4],
nod->KOi3[4], nod->KIi1[4], nod->KIi2[4], nod->KIi3[4]);
right ^= temp;
temp = right;
FO( temp, l_16, r_16, s9box, s7box, nod->KOi1[5], nod->KOi2[5],
nod->KOi3[5], nod->KIi1[5], nod->KIi2[5], nod->KIi3[5]);
```

```
FL( temp, l_16, r_16, a, b, nod->KLi1[5], nod->KLi2[5]);
left ^= temp;

temp2 = left;
FL( temp2, l_16, r_16, a, b, nod->KLi1[6], nod->KLi2[6]);
temp = temp2;
FO( temp, l_16, r_16, s9box, s7box, nod->KOi1[6], nod->KOi2[6],
nod->KOi3[6], nod->KIi1[6], nod->KIi2[6], nod->KIi3[6]);
right ^= temp;
temp = right;
FO( temp, l_16, r_16, s9box, s7box, nod->KOi1[7], nod->KOi2[7],
nod->KOi3[7], nod->KIi1[7], nod->KIi2[7], nod->KIi3[7]);
FL( temp, l_16, r_16, a, b, nod->KLi1[7], nod->KLi2[7]);
left ^= temp;

//return the correct endian result
data.b32[0] = left;                    data.b32[1] = right;
```

Με την εφαρμογή αυτών των αλλαγών, επιτυγχάνουμε τη μείωση του βάθους κλήσεων των συναρτήσεων από τρία σε δύο επίπεδα. Σε σχέση όμως με την απόδοση, δεν παρατηρήσαμε καμία βελτίωση. Επομένως, το βάθος στο οποίο εκτείνονται οι κλήσεις των συναρτήσεων, φαίνεται να μην επηρεάζει την απόδοση της εφαρμογής, διατηρήσαμε ωστόσο αυτές τις αλλαγές.

7.5 Χρήση Κοινής Μνήμης για τα SBoxes

Παρόλο που, στο σύνολό τους, οι παράμετροι που τοποθετούνται στην σταθερή μνήμη, δεν υπερβαίνουν το μέγιστο επιτρεπτό όριο των 64KB που παρέχει μια κάρτα με compute capability 2.0, η επιλογή αυτής για την περίπτωση των κουτιών αντικατάστασης μπορεί να μην αποτελεί την κατάλληλη επιλογή. Δοθέντος ότι αυτά είναι ορισμένα ως πίνακες, κάθε φορά που χρησιμοποιούνται, ζητείται ένα συγκεκριμένο στοιχείο του πίνακα. Αυτό μας οδηγεί στην υπόθεση ότι, ενδέχεται να μην τοποθετούνται ολόκληροι οι πίνακες στην cache, αλλά μόνο συγκεκριμένα στοιχεία από αυτούς, που έχουν ζητηθεί τουλάχιστον μια φορά. Ως αποτέλεσμα, οι πίνακες καταλήγουν να τοποθετούνται στην cache τμηματικά και κατόπιν πολλαπλών κλήσεων. Πιθανό είναι ακόμα, λόγω των πολλαπλών κλήσεων το περιεχόμενο της cache να αλλάζει πολύ συχνά, και έτσι ουσιαστικά να αυτοακυρώνεται ο σκοπός της. Τέλος, δεν αποκλείουμε το ενδεχόμενο, ο μεταγλωττιστής να επιλέγει να μην τοποθετήσει τα κουτιά αντικατάστασης στην σταθερή μνήμη λόγω μεγέθους, με αποτέλεσμα αυτά να καταλήγουν αυτομάτως στην τοπική μνήμη.

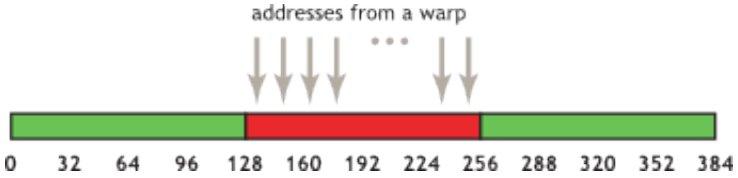
Βασιζόμενοι σε αυτή την υπόθεση, τοποθετήσαμε τα κουτιά αντικατάστασης στην κοινή μνήμη. Το γεγονός ότι χρησιμοποιούνται για ανάγνωση μόνο, μας απαλλάσσει και από την διαδικασία για τον συγχρονισμό των νημάτων. Πρακτικά, αφού οι τιμές τους δεν αλλάζουν ποτέ, μπορούμε να επιτρέπουμε σε όλα τα νήματα να διαβάζουν από την κοινή μνήμη σε αυθαίρετο χρόνο. Στην χειρότερη περίπτωση, θα υπάρξουν κάποιες καθυστερήσεις στην προτεραιότητα ανάγνωσης από τα συγκρουόμενα νήματα

που θα προσπαθήσουν να διαβάσουν το ίδιο στοιχείο ταυτόχρονα, αλλά δεν πρόκειται να έχουμε διαφθορά στα αποτελέσματα.

Η πρακτική αυτή, έδωσε τη μεγαλύτερη μέχρι στιγμής βελτίωση στην απόδοση της εφαρμογής μας, μετά την επιλογή χρήσης της σταθερής μνήμης. Ο χρόνος εκτέλεσης μειώθηκε από τα δύο λεπτά και τριάντα τρία δευτερόλεπτα σε είκοσι επτά δευτερόλεπτα για την κατασκευή ενός πίνακα των 65536 αλυσίδων με μήκος 100000 κρίκων η κάθε μια. Αυτό αντιστοιχεί σε μία επιτάχυνση της τάξεως του 5.7x. Ο χρόνος για την αποκρυπτογράφηση μειώθηκε στα είκοσι τέσσερα δευτερόλεπτα, μια επιτάχυνση της τάξεως του 4.7x.

7.6 Coalesced Memory Access

Οι προσπελάσεις της καθολικής μνήμης για εγγραφή και ανάγνωση, αποτελούν τον σημαντικότερο παράγοντα που μπορεί να επηρεάσει την απόδοση μιας εφαρμογής. Αυτό συμβαίνει διότι, όπως έχουμε εξηγήσει και στην ενότητα 4.4, αλλά το είδαμε και να συμβαίνει με την πρωτότυπη έκδοση της εφαρμογής μας, η προσπέλαση της καθολικής μνήμης κοστίζει πολύ χρόνο. Με κάθε προσπέλαση στην καθολική μνήμη, είναι δυνατή η μεταφορά μέχρι και 128 byte δεδομένων που είναι τοποθετημένα σε συνεχόμενες θέσεις μνήμης, μέσω της L1 cache, σε επίπεδο warp. Με άλλα λόγια, για κάθε 32 νήματα που αποτελούν ένα warp, μια προσπέλαση μνήμης για ένα συνεχόμενο εύρος των 128 byte, όσο και της γραμμής της L1 cache, δύναται να εξυπηρετηθεί με μια μόνο κλήση, αρκεί τα δεδομένα που ζητούμε να είναι ευθυγραμμισμένα σε αυτό το εύρος. Θέλουμε δηλαδή, διαδοχικά νήματα να προσπελαύνουν διαδοχικές θέσης της καθολικής μνήμης. Χρησιμοποιώντας αυτό το μοτίβο προσπέλασης (Coalesced Memory Access), μπορούμε να μειώσουμε δραματικά τον απαιτούμενο αριθμό προσπελάσεων της καθολικής μνήμης και, κατ΄ επέκταση να αυξήσουμε την απόδοση της εφαρμογής μας. Το σχήμα που ακολουθεί μας δείχνει αυτό το μοτίβο προσπέλασης της καθολικής μνήμης.



Σχήμα 7.1: Coalesced Memory Access

Στη εφαρμογή μας, τα δεδομένα μεταφέρονται από τον Host στο Device σε ποσότητες των 16 bit. Από τη στιγμή που θα τοποθετηθούν στην καθολική μνήμη της κάρτας γραφικών, μπορούμε να τα προσπελαύνουμε για εγγραφή και ανάγνωση, σε ποσότητες οποιουδήποτε μεγέθους μας βολεύει, χωρίς να ανησυχούμε για το endian μοντέλο. Έτσι, η μεταφορά των αρχικών σημείων από την καθολική μνήμη στους καταχωρητές, γίνεται σε ποσότητες των 32 bit, προκειμένου να ελαχιστοποιήσουμε τις προσβάσεις στην καθολική μνήμη.

Δοκιμάζοντας να προσπελάσουμε την καθολική μνήμη με τον coalesced τρόπο δεν παρατηρήσαμε καμία βελτίωση στον χρόνο εκτέλεσης. Αυτό μπορεί να έχει τις

εξής εξηγήσεις. Είτε τα δεδομένα μας είναι ήδη ικανοποιητικά ευθυγραμμισμένα και δεν πρόκειται, έτσι κι αλλιώς, να πετύχουμε περαιτέρω απόδοση, είτε θα αρχίσουμε να έχουμε βελτίωση στην απόδοση για έναν πολύ μεγαλύτερο αριθμό νημάτων, της τάξεως που να προσεγγίζει το εκατομμύριο. Προς επαλήθευση της δεύτερης εξήγησης, δοκιμάσαμε την κατασκευή πινάκων μέχρι και 262144 αλυσίδων, που αντιστοιχεί σε ίδιο αριθμό νημάτων, αλλά ούτε και σε αυτό το μέγεθος παρατηρήσαμε κάποια βελτίωση. Δεν επιδιώξαμε να δοκιμάσουμε μεγαλύτερα μεγέθη καθότι στην πράξη, για την συγκεκριμένη εφαρμογή, δεν πρόκειται να κατασκευαστούν πίνακες μεγαλύτεροι των 150000 αλυσίδων το πολύ. Αυτό προκύπτει από τη θεωρία για την μέθοδο ανταλλαγής χρόνου/μνήμης, βάση της οποίας, οι πολύ μεγάλοι πίνακες διατρέχουν κίνδυνο για αυξημένες συγκρούσεις και συγχωνεύσεις, κάτι που τους κάνει μη αποδοτικούς, οπότε η κατασκευή μεγάλων πινάκων στερείται χρησιμότητας. Αντίθετα, έχει νόημα η κατασκευή περισσότερων μικρών πινάκων.

7.7 Σύγκριση Εκτέλεσης σε CPU

Προκειμένου να μπορούμε να έχουμε ένα συγκριτικό δείγμα της απόδοσης που μπορούμε να πετύχουμε με χρήση της αρχιτεκτονικής CUDA για παράλληλη εκτέλεση, έναντι της εκτέλεσης σε CPU μόνο, χρειάστηκε να έχουμε και μια έκδοση της εφαρμογής μας η οποία να εκτελείται μόνο σε CPU. Για να είναι και κάπως τίμια αυτή η σύγκριση, θα ήταν θεμιτό η έκδοση αυτή να μπορεί να εκμεταλλεύεται την ύπαρξη πολλαπλών πυρήνων στους σύγχρονους επεξεργαστές, προς ανάπτυξη κάποιου βαθμού παραλληλισμού.

Η πρώτη μας σκέψη για την δημιουργία μιας τέτοιας έκδοσης, ήταν να μεταγλωττίσουμε τον πηγαίο κώδικα της CUDA απευθείας σε ένα εκτελέσιμο μόνο για CPU. Δυστυχώς όμως, ο μεταγλωττιστής της CUDA, έπαψε να υποστηρίζει αυτή την επιλογή από την έκδοση 4.0 και έπειτα. Αν και υπάρχουν άλλες εναλλακτικές, όπως περιγράφει ο Rob Faber σε ένα άρθρο του στο ιστολόγιο του Dr.Dobbs [48], αποφασίσαμε τελικά να ακολουθήσουμε μια άλλη πρακτική.

Ο πηγαίος κώδικας της εφαρμογής μας εκτείνεται σε ένα αρκετά διαχειρίσιμο μέγεθος, γι΄ αυτό αποφασίσαμε να τον τροποποιήσουμε, αφαιρώντας όλες τις λειτουργίες και το συντακτικό για CUDA και αντικαθιστώντας τους kernels με βρόχους for, έτσι ώστε να λειτουργεί μόνο για CPU. Για το τίμιο της υπόθεσης, χρησιμοποιούμε το μοντέλο OpenMP [49], ένα μοντέλο παράλληλου προγραμματισμού που υποστηρίζεται από πολλές πλατφόρμες και γλώσσες προγραμματισμού και από τον gcc compiler επίσης. Με χρήση του OpenMP μπορούμε να εκμεταλλευτούμε τους πολλαπλούς πυρήνες ενός σύγχρονου επεξεργαστή για παρράλληλη εκτέλεση της εφαρμογής μας. Έτσι, στον επεξεργαστή intel i7-3770 του υπολογιστή του εργαστηρίου μας, η εφαρμογή για CPU μόνο, μπορεί θεωρητικά να εκτελεί οκτώ νήματα ταυτόχρονα.

Ο χρόνος εκτέλεσης της εφαρμογής μόνο σε CPU, για την κατασκευή ενός πίνακα μεγέθους 65536 αλυσίδων μήκους 100000 κρίκων η κάθε μία, στον υπολογιστή του εργαστηρίου μας, είναι σαράντα τέσσερα λεπτά και σαράντα ένα δευτερόλεπτα, ή αλλιώς, 2681 δευτερόλεπτα. Σε σύγκριση με τα είκοσι επτά δευτερόλεπτα που διαρκεί η εκτέλεση της βελτιστοποιημένης εφαρμογής εν παραλλήλω στην κάρτα γραφικών, η επιτάχυνση που παίρνουμε είναι της τάξεως του 99x.

7.8 Συγκεντρωτικά Αποτελέσματα

Στην συνέχεια παραθέτουμε μερικούς πίνακες που συγκεντρώνουν τα αποτελέσματα της βελτιστοποίησης. Ο πίνακας 7.1 μας δείχνει τους χρόνους εκτέλεσης για την κατασκευή πινάκων στην κάρτα γραφικών και τις σχετικές επιταχύνσεις που επιτυγχάνονται από στάδιο σε στάδιο, αλλά και την συνολική επιτάχυνση της βελτιστοποιημένης, τελικής έκδοσης σε σχέση με το πρωτότυπο.

Τα αντίστοιχα δεδομένα για το κομμάτι της αποκρυπτογράφησης παρουσιάζονται στον πίνακα 7.1.

Τέλος, υπάρχει και ο πίνακας 7.3 που μας δείχνει τους χρόνους εκτέλεσης και τις επιταχύνσεις για την κατασκευή πινάκων σε CPU μόνο.

Πίνακας 7.1: Επιτάχυνση Κατασκευής Πινάκων σε GPU ανά Στάδιο Βελτιστοποίησης

H/W Specs	**GPU:** GeForce GTX 580, 16 SM's, 32 Cores/SM		
# **Συνολικών Threads:** 65536			
# **Threads Ταυτόχρονης Εκτέλεσης:** 65536			
Διάρθρωση των Threads: 512 Blocks, 128 Threads/Block			
Στάδια Βελτιστοποίησης	**Χρόνος Εκτέλεσης (hh:mm:ss)**	**Πρόσθετη Επιτάχυνση από το Προηγούμενο Στάδιο**	**Συνολική Επιτάχυνση σε Σχέση με το Πρωτότυπο**
0: Πρωτότυπο	04:45:19	-	-
1: Χρήση Σταθερής Μνήμης	00:04:08	59x	59x
2: Κατάργηση Δομής chainNode	00:03:57	1.0x	72x
3: Αποδόμηση Βρόχων	00:02:21	1.7x	121x
4: Χρήση Κοινής Μνήμης	00:00:27	5.2x	634x
Συνολική Επιτάχυνση σε Σχέση με το Στάδιο Βελτιστοποίησης #1: 9.2x			

Πίνακας 7.2: Επιτάχυνση Αποκρυπτογράφησης σε GPU ανά Στάδιο Βελτιστοποίησης

H/W Specs	**GPU:** GeForce GTX 580, 16 SM's, 32 Cores/SM		
# **Συνολικών Threads:** 100000			
# **Threads Ταυτόχρονης Εκτέλεσης:** 100000			
Διάρθρωση των Threads: 782 Blocks, 128 Threads/Block			
Στάδια Βελτιστοποίησης	**Χρόνος Εκτέλεσης (hh:mm:ss)**	**Πρόσθετη Επιτάχυνση από το Προηγούμενο Στάδιο**	**Συνολική Επιτάχυνση σε Σχέση με το Πρωτότυπο**
0: Πρωτότυπο	03:38:38	-	-
1: Χρήση Σταθερής Μνήμης	00:03:16	67x	67x
2: Κατάργηση Δομής chainNode	00:03:06	1.0x	70x
3: Αποδόμηση Βρόχων	00:01:52	1.7x	117x
4: Χρήση Κοινής Μνήμης	00:00:24	4.7x	546x
Συνολική Επιτάχυνση σε Σχέση με το Στάδιο Βελτιστοποίησης #1: 8.2x			

Πίνακας 7.3: Χρόνοι Κατασκευής Πινάκων σε CPU

H/W Specs	**CPU:** Intel i-3770 @ 3.40 GHz, **cpu cores:** 4, **threads/core:** 2	
# **Συνολικών Threads προς Εκτέλεση:** 65536		
# **Threads Ταυτόχρονης Εκτέλεσης**	Χρόνος Εκτέλεσης (hh:mm:ss)	Επιτάχυνση
1	01:22:13	-
8	00:44:41	1.8x
Συνολική Επιτάχυνση CUDA Έναντι CPU: 99x		

Κεφάλαιο 8

Συμπεράσματα

8.1 Τεχνικά Προβλήματα που Αντιμετωπίσαμε

Στο σημείο αυτό θα θέλαμε να κάνουμε μια συγκεντρωτική αναφορά στα διάφορα τεχνικά προβλήματα που συναντήσαμε και πως αυτά αντιμετωπίστηκαν, ως μια παρακαταθήκη της όποιας γνώσης αποκομίσαμε, με την πεποίθηση ότι ίσως βοηθήσει τον αναγνώστη που θα ενδιαφερθεί να ασχοληθεί με το αντικείμενο του προγραμματισμού σε CUDA, να αντιμετωπίσει παρόμοια προβλήματα που θα συναντήσει.

Endianness

Το πιο ουσιώδες ζήτημα που κληθήκαμε να αντιμετωπίσουμε, ήταν αυτό της litte endian αρχιτεκτονικής των καρτών γραφικών CUDA. Αν και αυτό φυσικά αναφέρεται στα τεχνικά δελτία της NVIDIA, δεν είναι κάτι στο οποίο θα δώσει κανείς ιδιαίτερη σημασία. Δοθέντος ότι προγραμματίζουμε σε μια γλώσσα προγραμματισμού υψηλού επιπέδου (C/C++) και αν λάβουμε υπόψιν ότι συνήθως δουλεύουμε με αριθμητικές τιμές, ο μεταγλωττιστής είναι σε θέση να χειριστεί το ζήτημα του endianness, ακόμα και στην περίπτωση του προγραμματισμού σε CUDA, όπου έχουμε ένα υβριδικό σύστημα που αποτελείται από μια big endian αρχιτεκτονική (επεξεργαστής) και μία little endian αρχιτεκτονική (κάρτα γραφικών). Λόγω όμως της ιδιαίτερης φύσης του κρυπτογραφικού αλγορίθμου με τον οποίο ασχοληθήκαμε, το θέμα του endianness αποτέλεσε πρόβλημα και τελικά, κατά τύχη θα έλεγε κανείς, ανακαλύψαμε αυτή τη διαφορά.

Όταν δουλεύει κανείς με κρυπτογραφικούς αλγορίθμους, σπανίως χρησιμοποιεί αριθμητικές τιμές. Όπως και στην περίπτωσή μας, συνήθως αναφερόμαστε στις μεταβλητές μας και επενεργούμε σε αυτές σε επίπεδο bit και όχι σε επίπεδο τιμής. Και πάλι, όλα λειτουργούν χωρίς πρόβλημα, όταν η endian αρχιτεκτονική είναι σταθερή. Όταν όμως τα δεδομένα μεταφέρονται μεταξύ διαφορετικών endian αρχιτεκτονικών, χρειάζεται μεγάλη προσοχή στο πως πρέπει να γίνει αυτό προκειμένου να μην διαφθείρονται τα αποτελέσματα, και δυστυχώς, δεν υπάρχει κάποια πάγια τακτική για αυτό.

Η λύση στο πρόβλημά μας, ήρθε μετά από πολλές δοκιμές. Έπρεπε να κατα-

λήξουμε σε ένα σταθερό, ελάχιστο εύρος bit στο οποίο εφαρμόζονται οι κρίσιμες για την ακεραιότητα των δεδομένων πράξεις, όπως οι ολισθήσεις και οι αναδιατάξεις. Αυτό προέκυψε από τον τρόπο με τον οποίο λειτουργεί ο KASUMI, να είναι τα 16 bits. Κατ΄ επέκταση, έπρεπε και τα δεδομένα να εκχωρούνται ανά ποσότητες των 16 bits σε αντίστοιχου τύπου μεταβλητές. Όσο τηρούμε αυτό το μοτίβο, διασφαλίζεται η ακεραιότητα των αποτελεσμάτων μας και με τη χρήση κατάλληλων ενώσεων, για απλές πράξεις όπως αντιγραφές/συγκρίσεις ή αριθμητικές πράξεις σε επίπεδο bit, μπορούμε να αναφερόμαστε στα δεδομένα και σε μεγαλύτερο εύρος bit με ασφάλεια.

Separate Compilation

Στον προγραμματισμό γενικά, θεωρείται πάγια τακτική η διάσπαση του κώδικα σε περισσότερα αρχεία. Με αυτόν τον τρόπο, φροντίζουμε ο κώδικάς μας να έχει μια διάρθρωση που βοηθάει στη εποπτεία και την επαναχρησιμοποίηση τμημάτων αυτού. Σε μια τέτοια διάρθρωση κώδικα, τα διάφορα αρχεία μεταγλωττίζονται χωριστά παράγοντας object files, και με χρήση ενός linker, τα τελευταία ενώνονται για να παράξουν το τελικό εκτελέσιμο.

Η παραπάνω τεχνική παραγωγής ενός εκτελέσιμου, επονομαζόμενη separate compilation, άρχισε να υποστηρίζεται από το Cuda Toolkit μόλις από την έκδοση 5.0 και έπειτα. Πριν από αυτή, ο μεταγλωττιστής της Cuda, απαιτούσε όλος ο κώδικας να βρίσκεται σε ένα και μοναδικό αρχείο.

Όσο η ανάπτυξη της εφαρμογής μας βρισκόταν σε πρώιμο στάδιο, προκειμένου να μπορούμε να εφαρμόσουμε μια σχετική διάσπαση του κώδικά μας σε περισσότερα του ενός αρχεία, εφαρμόσαμε κάποια «τρικ» για τις παλαιότερες εκδόσεις του Cuda Toolkit. Στο κομμάτι του Device Code, φροντίσαμε όλες οι συναρτήσεις να είναι ορισμένες με χρήση της λέξης-κλειδί inline. Η χρήση του inline στην γλώσσα C/C++, αποτελεί μια σύσταση στον μεταγλωττιστή, να αντικαταστήσει την κλήση μιας συνάρτησης με το σώμα αυτής. Αν και η απόφαση για το αν θα γίνει τελικά η αντικατάσταση επαφίεται στον μεταγλωττιστή, η σύσταση αυτή αναγκάζει τον μεταγλωττιστή να ανοίξει το αρχείο όπου βρίσκεται η συνάρτηση, να διαβάσει το σώμα της και να το συμπεριλάβει στο κομμάτι που είναι προς μεταγλώττιση. Με αυτό το τρικ, κάνουμε τον μεταγλωττιστή της CUDA να «πιστεύει» ότι ο κώδικας βρίσκεται όλος σε ένα αρχείο. Για να επιτυγχάνεται, τέλος η κλήση των kernels από το κομμάτι του Host Code, ορίζονται εξωτερικές συναρτήσεις με χρήση της λέξης-κλειδί extern για να τους καλούν.

Δυναμική Εκχώριση Μνήμης στην GPU

Στον προγραμματισμό, επιλέγουμε να δουλέψουμε με δείκτες, πλην άλλων λόγων, στην περίπτωση που δεν είναι από την αρχή ξεκάθαρο πόση μνήμη θα χρειαστούμε για τα δεδομένα μας, και αυτό θα γίνει γνωστό κατά την διάρκεια εκτέλεσης της εφαρμογής. Σε αυτήν την περίπτωση, απαιτείται δυναμική εκχώρηση μνήμης, είτε με χρήση της malloc() στην γλώσσα C, είτε με χρήση new/delete στην γλώσσα C++.

Η δυναμική εκχώρηση μνήμης στο Device μέσω του Device Code, υποστηρίζεται στην CUDA από αρχιτεκτονικές με compute capability 2.0 και άνω. Όσο η εκπόνηση της εφαρμογής μας διεξάγονταν σε παλαιότερες αρχιτεκτονικές, προκειμένου

να μπορούμε να απολαμβάνουμε την ευελιξία της χρήσης δεικτών, χωρίς να χρειάζεται να κάνουμε δυναμική εκχώρηση μνήμης, χρησιμοποιήσαμε μια δυνατότητα που προσφέρει ο μεταγλωττιστής της C++, αυτή της αποδιευθυνσιοδότησης (&).

Όταν εκχωρούνται απλές μεταβλητές σε μία συνάρτηση στη C/C++, δημιουργούνται τοπικά αντίγραφα αυτών, στα οποία εκχωρούνται οι τιμές τους. Όταν η συνάρτηση ολοκληρώσει την εργασία της, τα αντίγραφα αυτά διαγράφονται και οι αρχικές μεταβλητές διατηρούν τις αρχικές τους τιμές. Στην C++ συγκεκριμένα, αν κατά την εκχώρηση των μεταβλητών χρησιμοποιήσουμε ως πρόθεμα σε αυτές το &, τότε η μεταβλητή αποτιμάται με βάση την διεύθυνση μνήμης και όχι με βάση την τιμή της. Τότε, οι όποιες αλλαγές, εγγράφονται απευθείας στην συγκεκριμένη διεύθυνση μνήμης, μέσα από την καλούμενη συνάρτηση, επομένως οι νέες τιμές μεταφέρονται αυτόματα στις μεταβλητές και έξω από αυτή.

Με βάση την παραπάνω τεχνική, καταφέραμε να χρησιμοποιούμε τις δομές και ενώσεις ως δείκτες, ορίζοντάς τες ως απλά στιγμιότυπα και όχι με χρήση δέσμευσης μνήμης σε δείκτες, όπως συνηθίζεται να γίνεται.

8.2 Σύγκριση με άλλες Υλοποιήσεις

Μελετώντας τα συγκεντρωτικά αποτελέσματα των πινάκων που παραθέτουμε στην ενότητα 7.8, όπως αυτά προκύπτουν από τη διαδικασία της βελτιστοποίησης που αναπτύξαμε στο προηγούμενο κεφάλαιο, το άμεσο συμπέρασμα που διεξάγεται, είναι ότι, σε σχέση με τη χρήση ενός επεξεργαστή, η χρήση μιας κάρτας γραφικών για παράλληλη εκτέλεση της εφαρμογής μας, μας δίνει μια επιτάχυνση της τάξεως του 99x και σε απόλυτο χρόνο εκτέλεσης για την παραγωγή ενός πίνακα των 65536 αλυσίδων μήκους 100000 κρίκων έκαστη, τα είκοσι επτά δευτερόλεπτα. Σημαντικό πλεονέκτημα για αυτό το κέρδος στην ταχύτητα εκτέλεσης, αποτελεί το γεγονός ότι το πρόβλημα που κληθήκαμε να επιλύσουμε, αφενός παραλληλοποιείται σε πολύ μεγάλο βαθμό και αφετέρου, οι ιδιότητες της κάρτας γραφικών επέτρεψαν στη φάση της βελτιστοποίησης να απαλειφθούν αρκετές από τις πολυπλοκότητές του.

Μέχρι και τη χρονική στιγμή της συγγραφής αυτού του κειμένου, κατά τη γνώση μας, δεν έχει γίνει δημοσίως γνωστή κάποια άλλη εφαρμογή για παράλληλη εκτέλεση σε κάρτα γραφικών ή άλλη πλατφόρμα παραλληλισμού που να κατασκευάζει πίνακες Ουράνιου Τόξου για τον αλγόριθμο κρυπτογράφησης A5/3. Τα πιο κοντινά δείγματα από άλλες υλοποιήσεις, προς σύγκριση με τα δικά μας αποτελέσματα, είναι η δουλειά του κ. Παπαντωνάκη από το οικείο μας τμήμα [39] και η δουλειά του Karsten Nohl και της ομάδας του για τον αλγόριθμο A5/1 [40].

Η υλοποίηση του κ. Παπαντωνάκη σε μια Virtex5 XC5VLX330T για τον αλγόριθμο A5/3, δύναται να κατασκευάσει έναν πίνακα μεγέθους 65536 αλυσίδων μήκους 131072 κρίκων η κάθε μία σε χρόνο 8,71 ημερών [39, σελ. 51]. Βέβαια, εδώ οφείλουμε να αναφέρουμε ότι, το Clock Rate της Virtex5 XC5VLX330T για την υλοποίηση του κ. Παπαντωνάκη ανέρχεται στα 91.896 MHz ενώ το θεωρητικό μέγιστο Clock Rate της GTX 580 που διαθέτουμε στο εργαστήριο, ορίζεται από τον κατασκευαστή στα 1590 MHz.

Για την περίπτωση του Karsten Nohl και της ομάδας του, όπως προκύπτει από την περιγραφή των Glendrange, Hove και Hvideberg που συνεργάστηκαν μαζί του

[22, σελ. 80], η εφαρμογή που χρησιμοποιήθηκε για την κατασκευή των πινάκων, το Kraken, έχει υλοποιηθεί από τον έτερο συνεργάτη τους Frank A. Stevenson. Είναι μια εφαρμογή για παράλληλη εκτέλεση σε κάρτες γραφικών της ATI, και με αυτή κατάφεραν να κατασκευάσουν ένα σετ πινάκων με υπολογισμό $8,662 * 10^9$ αλυσίδων συνολικά σε περίπου πενήντα έξι ώρες. Η εφαρμογή εκτελούνταν σε έναν υπολογιστή με δύο κάρτες γραφικών HD 5970 της ATI και δύο σκληρούς δίσκους, εκτελώντας ταυτόχρονα δύο στιγμιότυπα της εφαρμογής, ένα σε κάθε κάρτα γραφικών. Το θεωρητικό μέγιστο Clock Rate της εν λόγω κάρτας με βάση τον κατασκευαστή είναι στα 725 MHz. Το μήκος της κάθε αλυσίδας δεν αναφέρεται. Σε σύγκριση με την απόδοση της Virtex5 XC5VLX330T, η εφαρμογή που υλοποιήσαμε παρουσιάζει πολύ καλύτερη απόδοση. Από την άλλη, σε σύγκριση με αυτή του Stevenson, χρήζει ακόμη σημαντικής βελτίωσης, αν αυτό είναι εφικτό. Για αντίστοιχο, συνολικό μέγεθος πινάκων, όπως αυτοί της ομάδας του Nohl, με χρήση μόνο μιας κάρτας γραφικών, η δική μας εφαρμογή θα χρειαζόταν περίπου σαράντα δύο ημέρες, υποθέτοντας ότι οι αλυσίδες καί στις δύο περιπτώσεις έχουν το ίδιο μήκος που ανέρχεται σε 100000 κρίκους.

Τέλος, υπάρχει και η εφαρμογή του Cryptohaze [43], στην οποία αναφερθήκαμε και νωρίτερα. Όπως αναφέρει στην σελίδα του, για την κατασκευή ενός πίνακα Ουράνιου Τόξου για την αποκρυπτογράφηση δεδομένων που έχουν κρυπτογραφηθεί με τη συνάρτηση κατακερματισμού MD5, δύναται να υπολογίζει περίπου 430 εκατομμύρια κρίκους ανά δευτερόλεπτο σε μια κάρτα GTX295 NVIDIA. Ή αλλιώς, 4300 αλυσίδες μήκους 100000 κρίκων ανά δευτερόλεπτο. Η αντίστοιχη μέτρηση για την δική μας εφαρμογή είναι περίπου 2400 αλυσίδες ανά δευτερόλεπτο. Βέβαια, να τονίσουμε εδώ, ότι η συνάρτηση κατακερματισμού MD5, έχει μακράν μικρότερη υπολογιστική πολυπλοκότητα σε σχέση με τον κωδικοποιητή τμημάτων KASUMI. Η MD5 δεν απαιτεί τη χρήση κλειδιού για την κρυπτογράφηση, οπότε εκπίπτει και η κατασκευή επιμέρους κλειδιών, δεν χρησιμοποιεί κουτιά αντικατάστασης και γενικά υλοποιείται με πολύ απλές πράξεις. Θυμίζουμε άλλωστε, ότι οι συναρτήσεις κατακερματισμού γενικά, είναι σχεδιασμένες με τέτοιο τρόπο ώστε να είναι πάρα πολύ γρήγορες.

8.3 Γενικά Συμπεράσματα

Όσων αφορά τους γενικότερους στόχους που θέσαμε στην αρχή αυτού του κειμένου, θεωρούμε ότι αυτοί εκπληρώθηκαν και με το παραπάνω. Εξοικειωθήκαμε με την φιλοσοφία του παράλληλου προγραμματισμού, αλλά και με την αρχιτεκτονική της CUDA και του μοντέλου προγραμματισμού της, καθώς και τις δυνατότητες που αυτή προσφέρει. Ταυτόχρονα, λόγω της φύσεως του προβλήματος που επιλέξαμε να λύσουμε, εμβαθύναμε σε ζητήματα που άπτονται της επιστήμης της κρυπτογραφίας.

Για το αν τελικά η εφαρμογή που υλοποιήσαμε, μπορεί να αποτελέσει και στην πράξη μια πραγματοποιήσιμη επίθεση για τον A5/3 σε ρεαλιστικά πλαίσια, υπολείπεται ακόμη πολύ δουλειά σε περαιτέρω επίπεδα προκειμένου να διεξαχθεί ένα ξεκάθαρο συμπέρασμα. Ως έχουν όμως τα πράγματα αυτή τη στιγμή, για την παραγωγή ενός ικανού αριθμού πινάκων σε ρεαλιστικό χρόνο, θα απαιτούνταν η συλλογική τους κατασκευή από περισσότερους υπολογιστές, εξοπλισμένοι με κάρτες γραφικών, ταυ-

τόχρονα. Μένει όμως ακόμη να διερευνηθεί το ποια θα πρέπει να είναι η δομή των πινάκων που θα μας δώσει την καλύτερη σχέση μεταξύ όγκου πινάκων και πιθανότητα επιτυχίας της επίθεσης.

Σε σχέση με το βαθμό βελτιστοποίησης που πετύχαμε, θα λέγαμε ότι φτάσαμε σε ένα αρκετά ικανοποιητικό επίπεδο, αν και πάντα μπορεί να υπάρχει χώρος για καλύτερα αποτελέσματα, όπως θα συζητήσουμε και στο επόμενο κεφάλαιο.

Εν κατακλείδι, οι κάρτες γραφικών της NVIDIA αλλά και γενικότερα, αδιαμφισβήτητα αποτελούν ένα εργαλείο χαμηλού κόστους και προσβάσιμο στο ευρύ κοινό, με το οποίο μπορεί κανείς επιτύχει υψηλές αποδόσεις για τις εφαρμογές του. Θα μπορούσαν κάλλιστα, για ένα πλήθος εφαρμογών, να αντικαταστήσουν υπερυπολογιστές για χρήση παράλληλου προγραμματισμού, οι οποίοι έχουν μακράν μεγαλύτερο κόστος αγοράς. Επίσης, φέρουν το πλεονέκτημα ότι, εφαρμογές που έχουν υλοποιηθεί σε παλαιότερες αρχιτεκτονικές είναι άμεσα μεταφέρσιμες σε νεότερες αρχιτεκτονικές, συχνά χωρίς να χρειάζεται να τροποποιηθούν, και να βελτιώνονται έτσι οι αποδόσεις τους, ευθέως ανάλογα με τις ιδιότητες που προσφέρουν οι νεότερες αρχιτεκτονικές.

Κεφάλαιο 9

Μελλοντική Εργασία

Όπως έχει σκιαγραφηθεί σε όλη την έκταση αυτού του κειμένου, η παρούσα διπλωματική εργασία είναι ένα εγχείρημα που συνδυάζει τον τομέα της κρυπτανάλυσης, με έμφαση στο σύστημα ασφαλείας της κινητής τηλεφωνίας του δικτύου GSM (2.5G), με τον τομέα του παράλληλου προγραμματισμού. Ως εκ τούτου, θέτει βάσεις για μελλοντική εργασία προς περισσότερες από μία κατευθύνσεις.

Μένοντας κοντά στους στόχους που θέσαμε, δηλαδή την εκμετάλλευση κατά τον καλύτερο δυνατό τρόπο των δυνατοτήτων που μας προσφέρει μια κάρτα γραφικών για παραλληλοποίση της εφαρμογής μας, μια κατεύθυνση θα ήταν να εξετάσουμε το ενδεχόμενο της επιπλέον αύξησης της απόδοσης της εφαρμογής μας. Στην δική μας υλοποίηση, χρησιμοποιήσαμε ως βάση τον κώδικα από τα τεχνικά δελτία της 3GPP για τις συναρτήσεις του Kasumi() και την KGCore(), και επιλέξαμε η παραλληλοποίηση να γίνεται στο επίπεδο του υπολογισμού των αλυσίδων. Στο μεσοδιάστημα, υπάρχουν διαθέσιμοι κώδικες σε CUDA που υλοποιούν κάποιους γνωστούς κωδικοποιητές τμημάτων, όπως για παράδειγμα ο AES, που χαρακτηρίζονται από παρόμοια δομή και πολυπλοκότητα με τον KASUMI. Θα είχε ενδιαφέρον να μελετήσουμε κάποιες από αυτές τις υλοποιήσεις, οι οποίες ίσως και να δώσουν ιδέες για παραλληλισμό σε κάποιο άλλο επίπεδο, όπου με ολοκληρωτικό επανασχεδιασμό της υλοποίησης του KASUMI και της KGCore(), να μπορέσουμε να πετύχουμε ακόμη καλύτερη απόδοση.

Αντικείμενο μελλοντικής εργασίας αποτελεί και η δοκιμή ρεαλιστικών σεναρίων κατασκευής πινάκων και χρήσης αυτών για την αποκρυπτογράφηση δεδομένων. Ένα ρεαλιστικό σενάριο προϋποθέτει την χρήση πολλών πινάκων, που οδηγεί στην ανάγκη για μεγάλο αποθηκευτικό χώρο, πόροι που εμείς δεν είχαμε στην διάθεσή μας κατά την διεξαγωγή του πειραματικού σταδίου εκτέλεσης της εφαρμογής. Σε ένα ρεαλιστικό σενάριο, όπου απαιτείται μαζική παραγωγή πολλών πινάκων ταυτόχρονα, αλλά και μαζική αποκρυπτογράφηση δεδομένων, θα μπορούσαν να βοηθήσουν σημαντικά στην αύξηση της απόδοσης, κάποιες ιδιότητες που προσφέρουν οι νεότερες αρχιτεκτονικές CUDA, όπως τα streams και το zero copy. Με τη χρήση των streams μπορούμε να διεξάγουμε μεταφορά δεδομένων από και προς την κάρτα γραφικών, ενώ ταυτόχρονα αυτή εκτελεί κάποιον kernel. Κατ΄ αυτόν τον τρόπο μπορούμε να καλύψουμε την καθυστέρηση που δημιουργείται όσο ο Host περιμένει τη ολοκλήρωση του kernel, προκείμενου να μπορέσει να μεταφέρει εκ νέου δεδομένα στο Device. Το zero copy από την άλλη, έχει την ιδιότητα να καθιστά προσπελάσιμα από το Device, δεδομένα

που βρίσκονται τοποθετημένα στον Host, χωρίς αυτά να χρειάζεται να μεταφερθούν στην καθολική μνήμη, μέσω κατάλληλων δεικτών. Δεν μπορούμε να ήμαστε βέβαιοι για το αν η χρήση του zero copy μπορεί να δώσει κάποια βελτίωση στην εφαρμογή μας, αλλά σίγουρα αξίζει να πειραματιστούμε με αυτή την ιδιότητα.

Παραμένοντας στο παράδειγμα ενός ρεαλιστικού σεναρίου, αλλά οδηγούμενοι προς μια άλλη κατεύθυνση, καθοριστικό ρόλο για την συνολική απόδοση του εγχειρήματος της κρυπτανάλυσης, θα παίξει η δομή με την οποία θα πρέπει να κατασκευαστούν οι πίνακες, ώστε να έχουν το μικρότερο δυνατό όγκο σε συνδυασμό με τη μεγαλύτερη πιθανότητα επιτυχίας. Εδώ θα πρέπει να μελετηθούν περισσότερες αναφορές στο κομμάτι της τεχνικής ανταλλαγής χρόνου/μνήμης με έμφαση την εφαρμογή της σε κωδικοποιητές ροής και να αναλύσουμε τη φύση του αλγορίθμου A5/3, τα δεδομένα από τα οποία εξαρτάται το διάνυσμα αρχικοποίησης του KASUMI, αλλά και τη φύση του ίδιου του GSM δικτύου, προκειμένου να καταλήξουμε σε ένα αποδοτικό φορμά πίνακα.

Αν, τελικά, θα θέλαμε να στοχεύσουμε προς τον απόλυτο σκοπό ενός τέτοιου εγχειρήματος, δηλαδή την ανάπτυξη ενός συστήματος κρυπτανάλυσης που να δύναται να εφαρμοστεί σε πραγματικά δεδομένα, απαραίτητη εργασία, σε πρώτο στάδιο, αποτελεί η συλλογή πραγματικών δεδομένων και η ανάλυσή τους, ώστε να αποκτηθεί γνώση για τη δομή και το περιεχόμενό τους. Σε δεύτερο στάδιο, ακολουθεί η διαδικασία παραγωγής ή συλλογής ζευγών γνωστών αρχικών κειμένων και κρυπτογραφημένου κειμένου από το δίκτυο. Καί τα δύο αυτά στάδια θα παίξουν επίσης ρόλο στην δομή που θα πρέπει να έχουν οι πίνακες.

Κλείνοντας, ως μελλοντική εργασία, αφήνουμε και το ενδεχόμενο να τροποποιηθεί κατάλληλα η εφαρμογή μας, ώστε να μπορεί να εφαρμοστεί και για τον αλγόριθμο εμπιστευτικότητας f8 που χρησιμοποιείται για την κρυπτογράφηση δεδομένων στο δίκτυο κινητής τηλεφωνίας τρίτης γενεάς (UMTS), μιας και χρησιμοποιεί και αυτός τον KASUMI ως βασικό δομικό στοιχείο για την παραγωγή κλειδιών ροής.

Bibliography

[1] M. Hellman, "A cryptanalytic time-memory trade-off," *IEEE Transactions on Information Theory*, vol. 26, no. 4, pp. 401– 406, Jul. 1980.

[2] P. Oechslin, "Making a Faster Cryptanalytic Time-Memory Trade-Off," in *Advances in Cryptology - CRYPTO 2003*, G. Goos, J. Hartmanis, J. Leeuwen, and D. Boneh, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, vol. 2729, pp. 617–630. [Online]. Available: http://www.springerlink.com/content/u9gxwd29p2tnx3wl/

[3] J. F. Dooley, *A Brief History of Cryptology and Cryptographic Algorithms*, ser. SpringerBriefs in Computer Science. Cham: Springer International Publishing, 2013. [Online]. Available: http://link.springer.com/10.1007/978-3-319-01628-3

[4] D. E. Robling Denning, *Cryptography and data security*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1982.

[5] C. Shannon, "Communication theory of secrecy systems," *Bell system technical journal*, vol. 28, no. 4, pp. 656–715, 1949.

[6] "Frequency analysis," Jul. 2015, page Version ID: 673573975. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Frequency_analysis&oldid=673573975

[7] A. J. Menezes, A. J. Menezes, and Menezes, *Handbook of Applied Cryptography*, 0002nd ed. Crc Press, Oct. 1996.

[8] H. Feistel, *Cryptography and Computer Privacy*. Scientific American, 1973. [Online]. Available: https://books.google.gr/books?id=Y0BenQEACAAJ

[9] "Feistel cipher," Mar. 2015, page Version ID: 651206460. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Feistel_cipher&oldid=651206460

[10] FIPS, *Data Encryption Standard (DES)*, 1999. [Online]. Available: http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf

[11] ——, *Advanced Encryption Standard (AES)*, 2001. [Online]. Available: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

[12] "Block cipher mode of operation," Sep. 2015, page Version ID: 679070873. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Block_cipher_mode_of_operation&oldid=679070873

[13] W. Diffie and M. Hellman, "New directions in cryptography," *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, Nov. 1976.

[14] R. L. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public-key Cryptosystems," *Commun. ACM*, vol. 21, no. 2, pp. 120–126, Feb. 1978. [Online]. Available: http://doi.acm.org/10.1145/359340.359342

[15] "Cryptographic hash function," Aug. 2015, page Version ID: 678312162. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Cryptographic_hash_function&oldid=678312162

[16] J. Hong, K. C. Jeong, E. Y. Kwon, I.-S. Lee, and D. Ma, "Variants of the Distinguished Point Method for Cryptanalytic Time Memory Trade-Offs," in *Information Security Practice and Experience*, L. Chen, Y. Mu, and W. Susilo, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, vol. 4991, pp. 131–145. [Online]. Available: http://www.springerlink.com/content/m555p63741648731/

[17] "Rainbow table," Aug. 2015, page Version ID: 678367407. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Rainbow_table&oldid=678367407

[18] J. Hong and P. Sarkar, "New Applications of Time Memory Data Tradeoffs," in *Advances in Cryptology - ASIACRYPT 2005*, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, and B. Roy, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, vol. 3788, pp. 353–372. [Online]. Available: http://www.springerlink.com/content/p61p3l5g12161t52/

[19] "The beginnings." [Online]. Available: http://www.gsmhistory.com/the-beginnings/

[20] "History." [Online]. Available: http://www.gsma.com/aboutus/history

[21] M. Y. Rhee, *Mobile Communication Systems and Security*, 1st ed. John Wiley & Sons, Apr. 2009.

[22] M. Glendrange, K. Hove, and E. Hvideberg, *Decoding GSM*. Institutt for telematikk, 2010, we have participated in the creation of almost two terabytes of tables aimed at cracking A5/1, the most common ciphering algorithm used in GSM. Given 114-bit of known plaintext, we are able to reco ... [Online]. Available: http://ntnu.diva-portal.org/smash/record.jsf?searchId=1&pid=diva2:355716

[23] "3gpp specification: 55.216; Document 1: A5/3 and GEA3 specifications." [Online]. Available: http://www.3gpp.org/DynaReport/55216.htm

[24] "3gpp specification: 35.202; Document 2: Kasumi specification." [Online]. Available: http://www.3gpp.org/DynaReport/35202.htm

[25] M. Matsui, "New block encryption algorithm MISTY," in *Fast Software Encryption*, ser. Lecture Notes in Computer Science, E. Biham, Ed. Springer Berlin Heidelberg, Jan. 1997, no. 1267, pp. 54–68. [Online]. Available: http://link.springer.com/chapter/10.1007/BFb0052334

[26] V. Niemi, *UMTS Security*, 1st ed. Chichester, England: Wiley, Dec. 2003.

[27] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*, 1st ed. Addison-Wesley Professional, 2010.

[28] "CUDA Toolkit." [Online]. Available: https://developer.nvidia.com/cuda-toolkit

[29] "Programming Guide :: CUDA Toolkit Documentation." [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#introduction

[30] S. Cook, *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*, 1st ed. Amsterdam ; Boston: Morgan Kaufmann, Nov. 2012.

[31] J. v. Oosten, "CUDA Thread Execution Model." [Online]. Available: http://www.3dgep.com/cuda-thread-execution-model/

[32] ——, "CUDA Memory Model." [Online]. Available: http://www.3dgep.com/cuda-memory-model/

[33] E. Biham, O. Dunkelman, and N. Keller, "A Related-Key Rectangle Attack on the Full KASUMI," in *Proceedings of ASIACRYPT 2005, LNCS 3788*. Springer-Verlag, 2005, pp. 443–461.

[34] E. Barkan, E. Biham, and N. Keller, "Instant ciphertext-only cryptanalysis of GSM encrypted communication," *Journal of Cryptology*, vol. 21, no. 3, pp. 392–429, 2008. [Online]. Available: http://link.springer.com/article/10.1007/s00145-007-9001-y

[35] O. Dunkelman, N. Keller, and A. Shamir, *A Practical-Time Attack on the A5/3 Cryptosystem Used in Third Generation GSM Telephony," Cryptology ePrint Archive: Report 2010/013*, 2010.

[36] K. Jia, L. Li, C. Rechberger, J. Chen, and X. Wang, "Improved Cryptanalysis of the Block Cipher KASUMI," in *Selected Areas in Cryptography*, ser. Lecture Notes in Computer Science, L. R. Knudsen and H. Wu, Eds. Springer Berlin Heidelberg, Aug. 2012, no. 7707, pp. 222–233. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-642-35999-6_15

[37] Z. Wang, X. Dong, K. Jia, and J. Zhao, "Differential Fault Attack on KASUMI Cipher Used in GSM Telephony," *Mathematical Problems in Engineering*, vol. 2014, p. e251853, Jun. 2014. [Online]. Available: http://www.hindawi.com/journals/mpe/2014/251853/abs/

[38] A. Barenghi, L. Breveglieri, I. Koren, and D. Naccache, "Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures," in *Proceedings of the IEEE*, 2012, pp. 3056–3076.

[39] P. Papantonakis. [Online]. Available: http://dias.library.tuc.gr/view/12635

[40] K. Nohl, "Decrypting GSM phone calls | Security Research Labs," 2010. [Online]. Available: https://srlabs.de/decrypting_gsm/

[41] "Ophcrack." [Online]. Available: http://ophcrack.sourceforge.net/

[42] "RainbowCrack - Crack Hashes with Rainbow Tables." [Online]. Available: http://project-rainbowcrack.com/index.htm

[43] "Cryptohaze.com GPU Rainbow Cracker." [Online]. Available: http://cryptohaze.com/gpurainbowcracker.php

[44] "3gpp specification: 55.217; Specification of the A5/3 encryption algorithms for GSM and ECSD, and the GEA3 encryption algorithm for GPRS; Document 2: Implementors' test data." [Online]. Available: http://www.3gpp.org/DynaReport/55217.htm

[45] "Free Rainbow Tables | Forum • View topic - Reduction function explaination." [Online]. Available: https://www.freerainbowtables.com/phpBB3/viewtopic.php?f=4&t=4669

[46] P. Alfke, "XAPP052 - Efficient Shift Registers, LFSR Counters, and Long Pseudo-Random Sequence Generators (7/96) - xapp052.pdf," Jul. 1996. [Online]. Available: http://www.xilinx.com/support/documentation/application_notes/xapp052.pdf

[47] "Rainbow Tables - Cryptohaze Project Wiki." [Online]. Available: http://www.cryptohaze.com/wiki/index.php/Rainbow_Tables#Candidate_Hash_Generation

[48] R. Farber, "Running CUDA Code Natively on x86 Processors." [Online]. Available: http://www.drdobbs.com/parallel/running-cuda-code-natively-on-x86-proces/231500166

[49] "OpenMP.org." [Online]. Available: http://openmp.org/wp/