ΠΑΝΕΠΙΣΤΗΜΙΟ
ΘΕΣΣΑΛΙΑΣ

Σχολή Θετικών Επιστημών

Τμήμα Πληροφορικής και Τηλεπικοινωνιών

Σχεδίαση και Προσομοίωση μίας Ασφαλούς Αρχιτεκτονικής Επεξεργαστή αξιοποιώντας Προηγμένη Κρυπτογράφηση

Θεόδωρος Αγγούρης

Πτυχιακή Εργασία

Υπεύθυνος

Δρ. Γεώργιος Δημητρίου

Επίκουρος Καθηγητής

Ιούνιος, 2025

ΠΑΝΕΠΙΣΤΗΜΙΟ
ΘΕΣΣΑΛΙΑΣ

School of Science

Department of Informatics & Telecommunications

Design and Simulation of a Secure Processor Architecture using Advanced Encryption

Theodoros Angouris

Final Thesis

Advisor

Dr. Georgios Dimitriou

Assistant Professor

June, 2025

«Με ατομική μου ευθύνη και γνωρίζοντας τις κυρώσεις [1], που προβλέπονται από της διατάξεις της παρ. 6 του άρθρου 22 του Ν. 1599/1986, δηλώνω ότι:

*1. Δεν παραθέτω κομμάτια βιβλίων ή άρθρων ή εργασιών άλλων αυτολεξεί **χωρίς να τα περικλείω σε εισαγωγικά** και χωρίς να αναφέρω το συγγραφέα, τη χρονολογία, τη σελίδα. Η αυτολεξεί παράθεση χωρίς εισαγωγικά χωρίς αναφορά στην πηγή, είναι λογοκλοπή. Πέραν της αυτολεξεί παράθεσης, λογοκλοπή θεωρείται και η παράφραση εδαφίων από έργα άλλων, συμπεριλαμβανομένων και έργων συμφοιτητών μου, καθώς και η παράθεση στοιχείων που άλλοι συνέλεξαν ή επεξεργάσθηκαν, χωρίς αναφορά στην πηγή. Αναφέρω πάντοτε με πληρότητα την πηγή κάτω από τον πίνακα ή σχέδιο, όπως στα παραθέματα.*

*2. Δέχομαι ότι η αυτολεξεί **παράθεση χωρίς εισαγωγικά**, ακόμα κι αν συνοδεύεται από αναφορά στην πηγή σε κάποιο άλλο σημείο του κειμένου ή στο τέλος του, είναι αντιγραφή. Η αναφορά στην πηγή στο τέλος π.χ. μιας παραγράφου ή μιας σελίδας, δεν δικαιολογεί συρραφή εδαφίων έργου άλλου συγγραφέα, έστω και παραφρασμένων, και παρουσίασή τους ως δική μου εργασία.*

3. Δέχομαι ότι υπάρχει επίσης περιορισμός στο μέγεθος και στη συχνότητα των παραθεμάτων που μπορώ να εντάξω στην εργασία μου εντός εισαγωγικών. Κάθε μεγάλο παράθεμα (π.χ. σε πίνακα ή πλαίσιο, κλπ), προϋποθέτει ειδικές ρυθμίσεις, και όταν δημοσιεύεται προϋποθέτει την άδεια του συγγραφέα ή του εκδότη. Το ίδιο και οι πίνακες και τα σχέδια

4. Δέχομαι όλες τις συνέπειες σε περίπτωση λογοκλοπής ή αντιγραφής.

Ημερομηνία: 9/7/2025

Ο Δηλών

(1) «Όποιος εν γνώσει του δηλώνει ψευδή γεγονότα ή αρνείται ή αποκρύπτει τα αληθινά με έγγραφη υπεύθυνη δήλωση
του άρθρου 8 παρ. 4 Ν. 1599/1986 τιμωρείται με φυλάκιση τουλάχιστον τριών μηνών. Εάν ο υπαίτιος αυτών των πράξεων
σκόπευε να προσπορίσει στον εαυτόν του ή σε άλλον περιουσιακό όφελος βλάπτοντας τρίτον ή σκόπευε να βλάψει άλλον, τιμωρείται με κάθειρξη μέχρι 10 ετών.»

Περίληψη

Σε αυτή την εργασία παρουσιάζεται και προτείνεται μία καινοτόμα αρχιτεκτονική για την επίλυση του ζητήματος των cold boot επιθέσεων σε σύγχρονους επεξεργαστές, βασισμένη στον επεξεργαστή του MIPS και της RISC αρχιτεκτονικής, με την χρήση του AES-GCM αλγορίθμου και το νέο πρότυπο FIPS 203 για προστασία από Κβαντικούς Υπολογιστές. Η αρχιτεκτονική σχεδιάζεται και αναπτύσσεται από το μηδέν, προσομοιώνεται και δοκιμάζεται με την χρήση της γλώσσας προγραμματισμού C++ και με επιτυχία κρυπτογραφεί τα δεδομένα που είναι αποθηκευμένα στην Κύρια Μνήμη του Ηλεκτρονικού Υπολογιστή και τελικά αποδεικνύεται μία πολύ καλή και ευκολά εφαρμόσιμη λύση για τις σύγχρονες αρχιτεκτονικές υπολογιστών γενικής χρήσης και ενσωματωμένων συστημάτων, με χαμηλή επίδραση στο κόστος χρόνου και παραγωγής.

Abstract

In this work, a new and innovative solution is presented to secure modern processor architectures from the cold boot attack, based on the MIPS processor and the RISC architecture using the AES-GCM algorithm and the new FIPS 203 standard to counter Quantum Computers. The architecture is designed and programmed from scratch, is simulated and is tested using the C++ programming language and successfully encrypts data stored in the Main Memory of Electronic Computers and finally gets proven that it is a robust and easy to adapt solution for new General Purpose Computer Architectures and for integrated systems, with low impact in the cost of time and production.

Περιεχόμενα

Εισαγωγή

Οι Ηλεκτρονικοί Υπολογιστές (Η/Υ) έχουν γίνει αναπόσπαστο κομμάτι της ζωής μας εδώ και δεκαετίες, είτε στη μορφή ενός σταθερού υπολογιστή (Desktop) ή φορητού υπολογιστή (Laptop), είτε το κινητό μας τηλέφωνο (Smartphone) είτε ακόμη και το έξυπνο ρολόι (Smartwatch) που φοράμε. Πια, εν έτη 2025 και στην Ελλάδα, δεν νοείται νοικοκυριό όπου να μην υπάρχει τουλάχιστον ένας Η/Υ καθώς και μία γραμμή σύνδεσης με το διαδίκτυο. Ο μέσος χρήστης χρησιμοποιεί αυτές τις δύο τεχνολογίες μαζί για επικοινωνία, ψυχαγωγία, ενημέρωση, πληρωμή λογαριασμών, παραγγελία τροφίμων, κράτηση θέσεων κλπ. Επίσης, πολλοί εργάζονται από το σπίτι, τηλεργασία (Work from Home), και προσφέρουν ακόμη πιο εύκολα τις υπηρεσίες τους μέσω διαδικτύου γλιτώνοντας χρόνο από τη διαδρομή προς και από το γραφείο.

Αν και όταν χρησιμοποιούμε την λέξη υπολογιστής εννοούμε συνήθως το μεγάλο, τετράγωνο κουτί, το desktop ή το laptop μας, στην πραγματικότητα το smartphone (ή και το απλό παλιό και καλό κινητό με κουμπιά), το smartwatch, η έξυπνη τηλεόραση, ο δρομολογητής (Router) μας, η έξυπνη λυχνία (λάμπα) είναι όλα ουσιαστικά το ίδιο πράγμα. Έχουν ένα κοινό κομμάτι, τον πραγματικό υπολογιστή, την Κεντρική Μονάδα Επεξεργασίας (ΚΜΕ/CPU), κοινώς τον επεξεργαστή, όπου λειτουργεί με τον ίδιο τρόπο σε όλα και έχει τον ίδιο σκοπό, μία εργασία, να υπολογίσει κάτι, να βγάλει ένα αποτέλεσμα μεταξύ 2 ή και περισσότερων τελούμενων, να ενεργοποιήσει τη λάμπα εφ' όσον έχει σκοτάδι (διαβάζει την τιμή του αισθητήρα), να κινήσει τον παίκτη στο παιχνίδι αριστερά εφ' όσον πατήθηκε το πλήκτρο "a", να στείλει μία παραγγελία κλπ. Όλες αυτές οι συσκευές ενορχηστρώνονται από μία ΚΜΕ, και συνήθως έχουν και το ίδιο μοντέλο. Πιο συγκεκριμένα, μερικές από αυτές τις συσκευές π.χ. smartwatch, τις ονομάζουμε IoT (Internet of Things/Διαδίκτυο των πραγμάτων) γιατί ουσιαστικά συνδέονται με ένα δίκτυο (π.χ. το διαδίκτυο) υπολογίζουν διάφορα πράγματα και συνήθως χρησιμοποιούν αισθητήρες, αλλά στην πραγματικότητα έχουν την ίδια ΚΜΕ όπου τα τελούμενα εισόδου της πια είναι οι τιμές εξόδου (που δείχνει) ο αισθητήρας. Με τις τιμές αυτές γίνεται η επεξεργασία (ο υπολογισμός) για να επιτευχθεί ένα αποτέλεσμα, συνήθως μία εντολή, π.χ. κλείσε το φως στην άλλη άκρη του κόσμου. Στην πραγματικότητα, για να πάμε και ακόμη ένα χαμηλότερο επίπεδο, οι υπολογισμοί είναι συνήθως προσθέσεις και πολλαπλασιασμοί και μάλιστα συνήθως υπάρχουν και άλλα επεξεργαστικά στοιχεία όπου συνοδεύουν την ΚΜΕ.

Συμπεραίνουμε λοιπόν ότι με πολύ απλά λόγια, **υπολογιστής** είναι η **ΚΜΕ (Επεξεργαστής)** και χρησιμοποιείται παντού, ακόμη και σε πολύ μικρά κυκλώματα, και *η εργασία του είναι μία, η επεξεργασία κάποιων τελούμενων με τρόπο τέτοιο ώστε να επιτευχθεί ένα χρήσιμο αποτέλεσμα (Από την άποψη των ηλεκτρονικών κυκλωμάτων, ρεύμα ή όχι ρεύμα)*, όπου ύστερα μπορεί να *μετατραπεί* σε κάτι ουσιώδες στον άνθρωπο, Διεπαφή Ανθρώπου-Υπολογιστή (HCI, Human Computer Interaction). Πιο συγκεκριμένα, το βασικό υλικό της ΚΜΕ είναι ο μικροεπεξεργαστής, εκεί γίνονται οι υπολογισμοί, απλά η ΚΜΕ έχει και κάποια ακόμη υλικά απαραίτητα για τους υπολογιστές του σήμερα.

Για να είμαστε λίγο πιο ακριβείς, στην πραγματικότητα οι ΚΜΕ όπου χρησιμοποιούνται στους υπολογιστές Γενικής Χρήσης (General Purpose) όπως ο σταθερός υπολογιστής και το laptop μας ονομάζονται Επεξεργαστές Γενικής Χρήσης (General Purpose Processors/GPPs) και πια υπάρχουν 2 βασικές εταιρίες όπου τους παράγουν, Intel και AMD. Ενώ για τις υπόλοιπες χρήσεις, π.χ. IoT, ελεγκτές, χρησιμοποιούνται λιγότερο πολύπλοκοι επεξεργαστές όπου όμως είναι πολύ καλοί στο να κάνουν κάποιες, αλλά συγκεκριμένες εργασίες, π.χ.

ψηφιακή επεξεργασία σήματος και παράγονται συνήθως από την STMicroelectronics, Atmel, Texas Instruments κ.α.

Παρ' όλα αυτά, οι επεξεργαστές όπου χρησιμοποιούνται στα smartphone (π.χ. Qualcomm, ARM) ή βρίσκονται ενσωματωμένοι σε μικρότερα υπολογιστικά συστήματα όπου μπορούν να χρησιμοποιηθούν για πάρα πολλούς σκοπούς (Ρομποτική, Τηλεπικοινωνίες, σύνθετα σενάρια IoT) όπως το Raspberry Pi (Broadcom), NVIDIA Jetson Nano (NVIDIA), εγκεφάλους αυτοκινήτων (Bosch) κ.α., μοιάζουν αρκετά με αυτούς της Intel/AMD γιατί είναι και αυτοί Γενικού Σκοπού. Μπορούν πολύ εύκολα να τρέξουν (και υποστηρίζονται επισήμως) κλασσικά Λειτουργικά Συστήματα (Operating Systems/OS), όπως Unix/Linux, αλλά ακόμη και τα Windows της Microsoft, εκεί όμως βρίσκεται η λεπτομέρεια, στο εάν υποστηρίζεται κάποιο λογισμικό. Αυτό εξαρτάται από ποιες Αρχιτεκτονικές Συνόλου Εντολών (Instruction Set Architecture/ISA) χρησιμοποιούν π.χ. x86, x86-64, AMD64, IA-64, 6502, ARM/A32, RISC-V, AVR.

Συμπεραίνουμε λοιπόν, ότι όπου και να κοιτάξουμε, όποια συσκευή και να διαλέξουμε, θα έχει ενσωματωμένη μία ΚΜΕ αλλά και διάφορους, υποστηρικτικής σημασίας (μικρο)επεξεργαστές. Μπορούμε πια να κατανοήσουμε τις βασικές διαφορές μεταξύ τους για το ποια είναι η πιο κατάλληλη να χρησιμοποιήσουμε, που όμως έχουν κοινή τον βασικό τους σκοπό, στο πιο χαμηλό επίπεδο ό,τι ISA και να υποστηρίζουν, οι βασικές λειτουργίες είναι πρόσθεση και ο πολλαπλασιασμός, και οι ΚΜΕ βρίσκονται παντού. Επίσης, καταλαβαίνουμε ότι εφ' όσον οι ΚΜΕ μοιάζουν μεταξύ τους αρκετά, σημαίνει ότι οι αρχιτεκτονικές τους είναι παρόμοιες και μοιράζονται πολλά κοινά.

Το γεγονός ότι χρησιμοποιούνται παντού, π.χ. Κλασσικοί Η/Υ, Ρομποτικά Συστήματα, Τραπεζικά μηχανήματα, διακομιστές (servers), υπολογιστές όπου υποστηρίζουν κρίσιμες υπηρεσίες, σημαίνει ότι έχουν μεγάλη αξία για την κοινωνία. Προχωρήσαμε πια στην Ψηφιακή Κοινωνία και όπως μετατρέψαμε διάφορους πόρους σε ψηφιακούς και έτσι η έννοια της Ασφάλειας πήρε κι άλλες διαστάσεις. Οι διαχειριστές των εν λόγω συστημάτων πρέπει να εξασφαλίζουν την Εμπιστευτικότητα, Ακεραιότητα και Διαθεσιμότητα (Confidentiality, Integrity and Availability/CIA) της Πληροφορίας καθώς και είναι υποχρεωμένοι από Ελεγκτικούς Μηχανισμούς σύμφωνα με τη Νομοθεσία, αλλά και οδηγίες όπως π.χ. στην Ευρωπαϊκή Ένωση/ΕΕ DORA και NIS2, να προστατεύουν τις υπηρεσίες αυτές. Όπως πάντα υπήρχαν επιτήδειοι στην κοινωνία μας, έτσι και τώρα όπου έχουμε εισέλθει στη νέα αυτή εποχή, το σενάριο άλλαξε, και πια οι Η/Υ γίνονται οι πιο κερδοφόροι στόχοι καθώς τα δεδομένα όπου είναι αποθηκευμένα μέσα σε αυτούς είναι πολύτιμα και με τη νέα μορφή εργασίας (Τηλεργασία), σημαίνει ότι απόρρητη πληροφορία μεταφέρεται και αποθηκεύεται σε πολλούς Η/Υ ταυτόχρονα (Server και Η/Υ εργαζομένου).

Συνήθως, στόχος γίνεται το λογισμικό το οποίο τρέχει πάνω στο υλικό του Η/Υ αλλά υπάρχουν και επιθέσεις σχεδιασμένες να εκμεταλλευτούν το ίδιο το υλικό και τις ευπάθειες της ίδιας της αρχιτεκτονικής και οργάνωσης μίας ΚΜΕ σε περίπτωση φυσικής πρόσβασης στο σύστημα/στόχος. Είναι εξίσου σημαντική η θωράκιση της συσκευής όπου δεδομένα κρίσιμης σημασίας επεξεργάζονται σε αυτή και δυστυχώς δεν υπάρχει πολύ ευαισθητοποίηση για το ζήτημα αυτό.

Η εργασία αυτή επικεντρώνεται αφ' ενός στην σχεδίαση και προσομοίωση μίας RISC (Reduced Instruction Set Computing) αρχιτεκτονικής ΚΜΕ βασισμένη στον MIPS (Microprocessor without Interlocked Pipelined Stages) επεξεργαστή και αφ' ετέρου στην απόδειξη, μέσω πειράματος, μίας καινοτόμου λύσης για την αύξηση της Ασφάλειας του υλικού των Η/Υ γενικού σκοπού (που προφανώς μπορεί να υιοθετηθεί ακόμη πιο εύκολα σε άλλου επεξεργαστών), ενσωματώνοντάς την με την προσομοίωση και τελικά την ανάλυση του αποτελέσματος της συγκεκριμένης προσέγγισης καθώς και πιθανά τρωτά σημεία.

Το Πρόβλημα

Στη σύγχρονη Κοινωνία της Πληροφορίας, Ψηφιακή Κοινωνία, όπου και να κοιτάξουμε γύρω μας, θα δούμε κάθε είδους συσκευή, είτε το smartwatch μας, είτε το smartphone μας, είτε διάφορες IoT (Internet of Things) συσκευές όπου ελέγχουν λυχνίες, ψυγεία, γκαραζόπορτες, όπου ουσιαστικά είναι ένας Ηλεκτρονικός Υπολογιστής (Η/Υ) και περιέχουν μία Κεντρική Μονάδα Επεξεργασίας (ΚΜΕ) για την επεξεργασία των δεδομένων όπου βρίσκονται αποθηκευμένα στη Μνήμη της συσκευής.

Όπως βρίσκουμε Η/Υ μέσα στα νοικοκυριά, έτσι ακριβώς θα τους βρούμε και σε βιομηχανικές εγκαταστάσεις, εταιρείες, νοσοκομεία, ερευνητικά εργαστήρια κ.α. Μπορεί να αφορά μία συσκευή ελέγχου ρομποτικού βραχίονα, μια συσκευή ελέγχου πίεσης ή ένα διακομιστή αποθήκευσης απόρρητων πληροφοριών (προϊόν σε έρευνα ή αποτελέσματα πειράματος).

Επιτήδειοι λοιπόν ψάχνουν (και βρίσκουν) τρόπους να αποκτήσουν πρόσβαση σε αυτές τις συσκευές, είτε για να εξάγουν όποια πληροφορία καταφέρουν να έχουν πρόσβαση, είτε ακόμη να στείλουν τις δικές τους εντολές για να ελέγξουν τη συσκευή με τρόπο τέτοιο όπου μπορεί να έχει καταστροφικές συνέπειες. Αν και συνήθως, η πρόσβαση είναι απομακρυσμένη (δηλαδή ο κακόβουλος χρήστης δεν έχει φυσική πρόσβαση στη συσκευή και συνήθως βρίσκεται χιλιόμετρα μακριά) και εκμεταλλεύονται ευπάθειες στο λογισμικό όπου τρέχει πάνω στο υλικό (Hardware) και μάλιστα είναι συνδεδεμένο με το διαδίκτυο, μερικές φορές, οι επιτήδειοι καταφέρνουν να έχουν φυσική πρόσβαση στη συσκευή, είτε ακόμη ο στόχος τους είναι η ίδια τους η συσκευή όπου αγόρασαν, που όμως δεν θα έπρεπε να έχουν πρόσβαση στο λογισμικό, και χρησιμοποιούν διάφορες μεθόδους (Reverse Engineering) για να καταφέρουν να εξαγάγουν τα δεδομένα (π.χ. το ειδικό λογισμικό της εταιρείας μίας παιχνιδομηχανής) για να βρουν ευπάθειες και να τις εκμεταλλευτούν προς όφελος τους (π.χ. να «κλέψουν» σε ένα ηλεκτρονικό παιχνίδι).

Η αλήθεια είναι ότι επιθέσεις υλικού απαιτούν πιο εξιδεικευμένες γνώσεις και ακριβά εργαλεία παρότι μια επίθεση με ένα δωρεάν, και εύκολα προσβάσιμο εργαλείο, για κάποιο λογισμικό όπου θα το βρει οποιοσδήποτε στο διαδίκτυο και με ελάχιστη, ή ακόμη και μηδαμινή γνώση, θα καταφέρει να το χρησιμοποιήσει. Αυτό επίσης σημαίνει ότι η πιθανότητα κάποιος να εκμεταλλευτεί ή και να βρει ευπάθειες στο λογισμικό παραμένει σχετικά μικρή (πολύ μεγάλη συμμετοχή, μικρή εξειδίκευση) μέσα σε άπειρες προσπάθειες, ενώ όμως η πιθανότητα να βρει κάποιος μία ευπάθεια στο ίδιο το υλικό, να ψάξει και να προσπαθήσει ώρες πολλές και να επιτύχει είναι αρκετά μεγάλη (μικρή συμμετοχή, μεγάλη εξειδίκευση), και το ερώτημα είναι, ποιος πραγματικά θέλει να εκμεταλλευτεί μία συσκευή και για ποιον λόγο. Αν αφαιρέσουμε από τη συνάρτηση τους χομπίστες, μάλλον συμπεραίνουμε ότι συνήθως αφορά πολύ εξειδικευμένες ομάδες με πολύ μεγάλο κίνητρο (π.χ. χρήματα).

Όποιος και να είναι ο λόγος, αυτό σημαίνει ότι προφανώς κάθε πρόσωπο ή εταιρεία θέλει να προστατέψει τη λύση/προϊόν της και μάλιστα υπάρχει και πολύ μεγάλη άνοδος σε επαγγελματίες όπου εργάζονται για να αναλύσουν και τεστάρουν τα προϊόντα (Λογισμικό ή Υλικό) πριν βγουν έξω στον καταναλωτή. Υπάρχει όμως και μία συνεχόμενη αλλά και εξελισσόμενη μάχη μεταξύ των δύο πλευρών, ομάδες προάσπισης της ασφάλειας και ομάδες επιτήδειων (αντιμέτωπες και με την κείμενη νομοθεσία), που κυρίως κάνουν το ίδιο πράγμα, ψάχνουν ευπάθειες αλλά με άλλο σκοπό και ηθική. Σε αυτή την εργασία δεν θα αναλυθεί καθόλου το λογισμικό κομμάτι της υπόθεσης και θα επικεντρωθούμε στο υλικό των Η/Υ.

Υπάρχουν διάφορα τρωτά σημεία στις αρχιτεκτονικές των Η/Υ αλλά και οργανώσεις των υλικών στοιχείων των ηλεκτρονικών συσκευών όπως και στην ίδια την αρχιτεκτονική της ΚΜΕ. Συνήθως όμως το μεγαλύτερο πρόβλημα που προσπαθεί να αντιμετωπιστεί είναι η προστασία των δεδομένων όπου βρίσκονται αποθηκευμένα στην μνήμη. Μα πώς θα προστατέψεις τα δεδομένα όπου βρίσκονται μέσα στον ίδιο τον Η/Υ, όσο πιο κοντά γίνεται στην ΚΜΕ για να επεξεργαστούν; Αυτός δεν είναι ο σκοπός έτσι κι αλλιώς; Τα δεδομένα πρέπει να βρίσκονται όσο πιο κοντά γίνεται στην ΚΜΕ και με τρόπο μορφοποιημένο και ταξινομημένο ώστε η προσπέλαση τους να είναι άμεσα και χωρίς καθυστερήσεις. Στην τελική αυτό που χρειαζόμαστε κάθε νέο χρόνο, είναι πιο γρήγορους επεξεργαστές (αν και με αυτό δεν υπονοείται μόνο η ταχύτητα επεξεργασίας, Clock Speed μετρήσιμο σε Hertz). Αν και η Επιστήμη των Υπολογιστών και η Πληροφορική έχει δει πολλές και ραγδαίες εξελίξεις στα χρόνια της, το πρόβλημα παραμένει, ή μάλλον ανανεώνεται, και οι χρήστες ζητάνε πιο γρήγορους Η/Υ ικανοί να προελαύνουν πιο πολλά δεδομένα ανά δευτερόλεπτο. Πώς θα προστατέψεις τα δεδομένα σου που τα θες στην πιο απλή μορφή και όσο πιο έτοιμα για επεξεργασία;

Είναι ένα πολύ μεγάλο ζήτημα και έχουν δοθεί αρκετές λύσεις από πολλές εταιρείες ηλεκτρονικών συσκευών, είτε να οργανώσουν το υλικό έτσι ώστε η φυσική πρόσβαση να περιοριστεί όσο πιο πολύ γίνεται και να χρειαστούν όλο και πιο εξειδικευμένα εργαλεία, την αποφυγή κλασσικών διαύλων επικοινωνίας για αποσφαλμάτωση (Debugging) πριν φύγει η μονάδα από το εργοστάσιο ή για πιθανές μετέπειτα επιδιορθώσεις από ειδικό προσωπικό πιστοποιημένο από την ίδια την εταιρεία, είτε ακόμη με την εισαγωγή διάφορων μονάδων για την κρυπτογράφηση των δεδομένων όπου βρίσκονται όμως σε συγκεκριμένα σημεία, συνήθως όμως στη μη πτητική (non-volatile) μνήμη. Ένα πολύ απλό και ευρέως διαδεδομένο παράδειγμα είναι η τεχνική του Jailbreak, όπου χρήστες προσπαθούν να επιτεθούν στη συσκευή όπου έχουν αγοράσει, π.χ. μία κονσόλα βιντεοπαιχνιδιών για να κατεβάσουν το λογισμικό της, και να το αναλύσουν για να βρουν μεθόδους να το εκμεταλλευτούν. Στα κινητά χρησιμοποιείται ο όρος Root. Προφανώς η διαρροή (Leak) του λογισμικού (Πιο κρίσιμα, του πηγαίου κώδικα/source code) επιφέρει πολύ μεγάλο κίνδυνο για την εταιρεία όπου το δημιούργησε καθώς επένδυσε σε έρευνα και παραγωγή (R&D) και πια το προϊόν τους είναι κοινό μυστικό όπου μπορεί να χρησιμοποιηθεί από τον ανταγωνισμό της. Επίσης, μεγάλο ενδιαφέρον υπάρχει για την προστασία των δεδομένων όπου «κάθονται» στην πτητική, τυχαίας προσπέλασης, μνήμη (Random Access Memory, RAM/ΜΤΠ), το πιο βασικό στοιχείο μετά την ίδια την ΚΜΕ.

Επειδή πιθανά τρωτά σημεία βρίσκονται παντού, όταν κάτι χρειάζεται να έχει πρόσβαση από κάπου, υπάρχουν αρκετές ευπάθειες, διαφόρων επιπέδων κρισιμότητας όπου απαιτούν διαφορετικά εργαλεία, διαφορετικές προσεγγίσεις και συνήθως απαιτείται συνδυασμός εξ αυτών για την επιτυχή ολοκλήρωση μίας επίθεσης, υπάρχουν διάφορες λύσεις για κάθε μία μονάδα υλικού σε έναν Η/Υ, είναι δύσκολο να τα κατηγοριοποιήσουμε σε μία ομάδα και άρα εάν διορθωθεί κάποια (με κόστος ίσως ταχύτητα ή περιορισμένη λειτουργικότητα), δεν σημαίνει ότι τα δεδομένα είναι ασφαλή. Μπορεί να δημιουργήθηκε μία καινούρια ευπάθεια ή απλά να περιορίσαμε κάποιες επιθέσεις. Επίσης, έχει πολύ μεγάλη σημασία από ποια οπτική γωνία αντιμετωπίζεται και ένα πρόβλημα. Αυτό λοιπόν σημαίνει ότι υπάρχουν πολλά πράγματα που πρέπει να ελεγχθούν και ότι όμως, μία απλή λύση, μπορεί να ωφελήσει συνολικά την ασφάλεια μίας συσκευής ακόμη και εάν θεωρείται ότι βελτιώνει μόνο κάτι πολύ συγκεκριμένο. Μάλιστα, σημασία έχει και κατά πόσο αυτό που βελτιώθηκε επηρεάζει (θετικά) το όλο σύστημα.

Αυτή η εργασία έγινε με σκοπό την ανάπτυξη μίας νέας λύσης που αφορά την περεταίρω ασφάλιση των δεδομένων όπου ανταλλάζονται μεταξύ της ΚΜΕ και της μνήμης του Η/Υ μέσω του διαύλου επικοινωνίας τους (Bus). Πιο συγκεκριμένα την κρυπτογράφηση των

δεδομένων όπου αποθηκεύονται στη μνήμη με την χρήση προηγμένων αλγορίθμων, για την κρυπτογράφηση/αποκρυπτογράφηση αλλά και την ανταλλαγή των κλειδιών. Η προσέγγιση αυτή έγινε έχοντας υπ' όψη ένα πολύ βασικό και κρίσιμο θέμα ασφαλείας των Η/Υ εν' ονόματι Cold Boot Attack (Επίθεση Ψυχρής Εκκίνησης) όπου θέτει σε κίνδυνο τα δεδομένα τα οποία βρίσκονται στην ΜΤΠ όσο λειτουργεί ο Η/Υ.

Cold Boot Attack

Το Cold Boot Attack ή η Επίθεση Ψυχρής Εκκίνησης είναι μία μέθοδος παραβίασης ασφαλείας όπου επιτρέπει σε έναν επιτιθέμενο όπου έχει φυσική πρόσβαση σε κάποιον Η/Υ να ανακτήσει τα δεδομένα όπου βρίσκονται αποθηκευμένα στη ΜΤΠ (Memory Blades) κατά τη λειτουργία ή λίγο μετά την απενεργοποίηση του υπολογιστή. Αν και η ΜΤΠ είναι μία πτητική μνήμη, που σημαίνει ότι χάνει τα δεδομένα της σε περίπτωση μη τροφοδότησης με ηλεκτρικό ρεύμα, η απώλεια τον δεδομένων δεν γίνεται άμεσα, ιδιαίτερα δε όταν θερμοκρασία της διατηρείται σε πολύ χαμηλά επίπεδα ή κυριολεκτικά είναι παγωμένη (π.χ. με τη χρήση ψυκτικού σπρέι). Έτσι ουσιαστικά αποθηκεύεται η κατάσταση του Η/Υ της χρονικής στιγμής εκείνης, για αρκετά δευτερόλεπτα ή και λεπτά. Η επίθεση περιλαμβάνει την επανεκκίνηση ή την αποσύνδεση του συστήματος και την άμεση εκκίνηση από εξωτερικό μέσο, ή ακόμα η μονάδα της μνήμης μπορεί να αφαιρεθεί και να τοποθετηθεί σε άλλο σύστημα και να ανακτηθούν τα δεδομένα του Η/Υ, καθώς όλα είναι αποθηκευμένα στην ΜΤΠ, δεδομένα όπου περιέχουν ευαίσθητες στοιχεία, κωδικούς πρόσβασης, κωδικούς BIOS κλπ.

Η επίθεση αυτή μπορεί να εκτελεστεί και να εκμεταλλευτεί με διάφορους τρόπους και τελικά ο επιτιθέμενος, να ανακτήσει τα δεδομένα όπως εξηγήθηκε και παραπάνω, και προφανώς ολόκληρα τα προγράμματα τα οποία έχουν φορτωθεί στη μνήμη. Ο μόνος ουσιαστικός τρόπος θωράκισης από αυτή την επίθεση είναι η κρυπτογράφηση των δεδομένων των οποίων εισέρχονται σε αυτή.

Ένα παράδειγμα

Ας εξετάσουμε ένα απλό παράδειγμα, για να κατανοήσουμε πως επεκτείνεται το πρόβλημα αυτό πρακτικά.

Ας υποθέσουμε ότι υπάρχει μία μεγάλη εταιρεία όπου έχει επενδύσει πολλούς πόρους στην έρευνα και ανάπτυξη (R&D) μίας λύσης που περιλαμβάνει σμήνη αυτόνομων τροχοφόρων ρομποτικών οχημάτων (ρομπότ) για να παραδίδει παραγγελίες από ένα σημείο στο άλλο (delivery), για παράδειγμα από την αποθήκη όπου βρίσκεται το προϊόν στο σπίτι. Προφανώς τα ρομποτικά οχήματα φέρουν τουλάχιστον έναν ηλεκτρονικό σύστημα (ηλεκτρονικός υπολογιστής) όπου ενορχηστρώνει τις απαραίτητες ενέργειες που πρέπει να κάνει το κάθε ρομπότ, όπως αποφυγή εμποδίων, οργάνωση πλάνου διαδρομής, τον έλεγχο των κινητήρων των τροχών, επεξεργασία δεδομένων που εισέρχονται στον Η/Υ από τους on-board (πάνω στο ρομπότ) αισθητήρες, sensor fusion (σύντηξη αισθητήρων, ενοποίηση όλων των δεδομένων που μαζεύονται από τους αισθητήρες για την περισσότερη βεβαιότητα γεγονότων), επικοινωνία με το νέφος για πιθανή αλλαγή πλάνου ή και εντοπισμό θέσης, χρήση AI (Τεχνητής Νοημοσύνης) για τους παραπάνω υπολογισμούς και πολλές άλλες που είναι δύσκολο να διατυπωθούν απλά. Τρέχουν πολλές απαιτητικές εφαρμογές παράλληλα. Το αποτέλεσμα πρέπει να υπολογίζεται σε κλάσματα του δευτερολέπτου με εξαιρετικά υψηλή βεβαιότητα, καθώς είναι κρίσιμο να μην υπάρξει λάθος υπολογισμού σε ένα όχημα που κινείται ανάμεσα σε άλλα οχήματα ή και ανθρώπους, σε πολυσύχναστους δρόμους. Ένα τέτοιο ρομποτικό σύστημα απαιτεί εξοπλισμό ακριβείας, στιβαρά υλικά και πολύ έρευνα πριν βγει στους δρόμους. Το υλικό, η ΚΜΕ όπου φέρει καθώς και όμως, η ποιότητα του λογισμικού (όλες οι διεργασίες για τις προαναφερθείσες λειτουργίες) πρέπει να είναι κλάσης άριστης ποιότητας και δεν υπάρχει χώρος για το παραμικρό λάθος. Τα υπολογιστικά συστήματα (συνήθως σε μορφή ενσωματωμένων συστημάτων), φέρουν ΚΜΕ όπως αυτές που αναφέρθηκαν πιο πάνω και που συνήθως τρέχουν σε Unix/Linux λειτουργικά συστήματα, και ειδικότερα λογισμικό όπως το ROS (Robot Operating System). Ο κώδικας (τα προγράμματα) όμως που τρέχει πάνω σε όλα αυτά, είναι η περιουσία της εταιρείας. Για αυτόν τον λόγο, πρέπει να εξαλειφθεί η περίπτωση κλοπής του λογισμικού.

Φανταστείτε τώρα, ότι ένα τέτοιο ρομπότ γίνεται στόχος από την αντίπαλη εταιρεία όπου θέλει να μάθει τα μυστικά της λειτουργίας του. Αφού το ρομπότ κινείται ελεύθερα και αυτόνομα στους δημόσιους δρόμους, δεν είναι καθόλου δύσκολο, κάποιος να το απαγάγει (Μάλιστα είναι μία πολύ καλή ευκαιρία να αναφέρουμε ένα από τα πιο μεγάλα προβλήματα των αυτόνομων ρομποτικών συστημάτων όπου είναι η απαγωγή του ρομπότ/Kidnapped Robot Problem, όμως με τη λογική της επαναφοράς του συστήματος στο αρχικό του πλάνο και την κατανόηση του ως προς του που βρίσκεται στον κόσμο). Μπορεί λοιπόν η αντίπαλη εταιρεία να καταστρώσει ένα σχέδιο για να το αρπάξει και να το φέρει στις εγκαταστάσεις του για ανάλυση.

Όταν το σύστημα φτάσει στα χέρια εξειδικευμένων τεχνικών, στόχος τους θα είναι να κατεβάσουν το λογισμικό όπου τρέχει. Όμως, στην περίπτωση του συγκεκριμένου ενσωματωμένου συστήματος που φέρει κάποια ΚΜΕ, το πρόγραμμα είναι φορτωμένο στη μνήμη ήδη, πιθανόν μία Μνήμη Τυχαίας Προσπέλασης, που όμως είναι μη Πτητική (π.χ. NVRAM), ώστε να αποφύγουν την τοποθέτηση σκληρού δίσκου που πιθανόν να κρατάει τα δεδομένα ανασφάλιστα (π.χ. μη κρυπτογραφημένα), ή που να χρειάζεται παραπάνω μηχανισμούς ασφαλείας για να προστατέψουν το λογισμικό μέσα σε αυτόν. Επίσης, συνήθως δεν βγάζει και πολύ νόημα η χρήση ROM (Read Only Memory/Μνήμης Μόνο για Ανάγνωση) για λογισμικό που πιάνει λίγο χώρο (εκτός από αξιόπιστο λογισμικό, θα πρέπει να

είναι και ελαφρύ για καλή απόδοση). Οπότε το σύστημα μπορεί να θεωρηθεί και κλειστό, που σημαίνει ότι τα δεδομένα κάνουν έναν κύκλο που δεν μπορεί/χρειάζεται να «ανοίξει».

Οπότε οι τεχνικοί αποφασίζουν να κατεβάσουν όλο το λογισμικό με χρήση της τεχνικής του cold boot, και ύστερα από την αντιμετώπιση και υπόλοιπων, κλασσικών μηχανισμών που υπάρχουν χρόνια στα υλικά συστήματα, καταφέρνουν να κατεβάσουν αυτούσια όλη τη μνήμη και έχουν πρόσβαση σε όλο το λογισμικό αλλά και στο λογισμικό οπού έτρεχε με όλες τις παραμέτρους ρυθμισμένες και πιθανόν διάφορων ειδών κλειδιών ασφαλείας, τελικά να έχουν διαρρεύσει.

Έτσι, ο ανταγωνισμός πια μπορεί να χρησιμοποιήσει την τεχνολογία που επένδυσε η μεγάλη αυτή εταιρεία προς όφελος της. Πώς λοιπόν μπορούμε να εξασφαλίσουμε πως τα δεδομένα, ακόμη και σε περίπτωση επιτυχών επιθέσεων, όπως αναφέρθηκαν πιο πάνω, παραμένουν προστατευμένα; Είναι η κρυπτογράφηση μία πιθανή λύση; Αν ναι, μέχρι πότε; Τί θα συμβεί με την ανάπτυξη των κβαντικών υπολογιστών όπου υποστηρίζεται ότι οι σύγχρονοι αλγόριθμοι κρυπτογράφησης είναι ευάλωτοι;

Ανάλυση Προβλήματος

Η ρίζα του προβλήματος του Cold Boot Attack, βρίσκεται στο γεγονός ότι τα δεδομένα κάθονται ωμά (raw, στη μορφή όπου χρησιμοποιούνται) στη ΜΤΠ για να είναι έτοιμα για επεξεργασία από την ΚΜΕ άμεσα. Μάλιστα υπάρχει και μία κατηγοριοποίηση, ταξινόμηση καλύτερα, καθώς και κανόνες που ορίζουν πως τα προγράμματα φορτώνονται σε αυτή, που όμως με λίγη προσπάθεια μπορούμε να ενώσουμε τα «τυχαία» κομμάτια. Η αιτία του προβλήματος δηλαδή δεν αφορά κάποιο bug (προβληματικό κομμάτι κώδικα/υλικού) αλλά την ίδια την αρχιτεκτονική και την οργάνωση των Η/Υ. Αν και υπάρχουν κάποιες προστασίες (εστιασμένες σε άλλα προβλήματα) όπως η εικονική μνήμη (Virtual Memory), το πρόβλημα αυτό παραμένει καθώς επιτρέπει στον επιτιθέμενο να κάνει dump (να κατεβάσει αυτούσια τα δεδομένα) τη μνήμη και ύστερα να τα επεξεργαστεί με όποιον τρόπο χρειάζεται για να ενώσει τα κομμάτια του παζλ.

Πιο συγκεκριμένα ο τρόπος με τον οποίο συνεργάζεται η μνήμη με την ΚΜΕ είναι με τον δίαυλο επικοινωνίας (communication bus) μεταξύ της μνήμης και τις κρυφές μνήμες (cache), πιο συγκεκριμένα την πιο υψηλού επιπέδου cache. Όταν ο επεξεργαστής ζητήσει κάτι όπου δεν βρίσκεται στη cache, τότε γίνεται αναζήτηση στην κεντρική μνήμη του υπολογιστή (μέσω του ελεγκτή της μνήμης), όπου είναι η ΜΤΠ αλλά και σε περίπτωση αστοχίας της, αναζήτηση και στη σταθερή μη πτητική μνήμης π.χ. σκληρός δίσκος). Ο ελεγκτής λαμβάνει την φυσική διεύθυνση (physical address) όπου βρίσκονται τα δεδομένα και επιστρέφει ό,τι υπάρχει εκεί μέσα.

Οπότε εάν μία ουσιώδης λύση είναι η κρυπτογράφηση, πώς ακριβώς θα γίνεται η αναζήτηση; Πού θα γίνεται η κρυπτογράφηση και η αποκρυπτογράφηση; Τί είδους και σε ποιό στάδιο θα γίνεται η δημιουργία των κλειδιών; Πού θα αποθηκεύονται τα κλειδιά για τον αλγόριθμο κρυπτογράφησης; Πότε θα γίνεται η κρυπτογράφηση και τι επίπτωση θα έχει στο συνολικό σύστημα (πιθανή επιβάρυνση στην συνολική απόδοση); Με ποιόν αλγόριθμο;

Αυτά είναι τα πιο σημαντικά ζητήματα που δημιουργούνται και που όμως έχει γίνει έρευνα πάνω σε αυτά και έχουν δοθεί κάποιες λύσεις.

AMD SME και SEV/SEV-SNP

Μία από τις πιο ενδιαφέρουσες προσεγγίσεις στο πρόβλημα της προστασίας της μνήμης είναι αυτή όπου εισήγαγε η AMD με τις τεχνολογίες SME (Secure Memory Encryption) και SEV (Secure Encrypted Virtualization). Εδώ το σκεπτικό είναι απλό, όπως εξηγήθηκε και πάνω, αφού τα δεδομένα αποθηκεύονται ωμά στη ΜΤΠ (Main Memory), γιατί να μην τα κρυπτογραφούμε απευθείας εκεί;

Η τεχνολογία SME έρχεται ακριβώς για να αντιμετωπίσει αυτό. Πρόκειται για έναν μηχανισμό όπου όλα τα δεδομένα που μεταφέρονται προς ή από τη μνήμη περνάνε από μία διαδικασία κρυπτογράφησης ή αποκρυπτογράφησης. Δηλαδή το ίδιο το σύστημα συνεχίζει να λειτουργεί σαν να μην άλλαξε τίποτα, όμως στο παρασκήνιο κάθε φορά που γράφεται ή διαβάζεται κάτι από τη ΜΤΠ, περνάει από ένα επιπλέον βήμα ασφάλειας. Το πιο σημαντικό όμως είναι ότι το κλειδί κρυπτογράφησης βρίσκεται αποθηκευμένο μέσα στον ίδιο τον επεξεργαστή και δεν είναι ποτέ προσβάσιμο από κανένα επίπεδο λογισμικού, ούτε καν από το λειτουργικό σύστημα. Άρα, ακόμα και αν ο επιτιθέμενος δοκιμάσει να κάνει φυσικό dump της μνήμης (να κατεβάσει αυτούσια όλα τα δεδομένα όπου βρίσκονται στην ΜΤΠ), αυτό που θα πάρει είναι κρυπτογραφημένα, ακατανόητα δεδομένα.

Ωστόσο ενδιαφέρον είναι να εξετάσουμε και την επόμενη περίπτωση, μία πιο υψηλού επιπέδου προσέγγιση όπου αν θέλουμε να μιλάμε για προστασία σε ένα πιο πολυεπίπεδο περιβάλλον, όπως π.χ. στο cloud, τότε εκεί μπαίνει στο παιχνίδι η SEV. Η ιδέα εδώ είναι ότι πολλές εικονικές μηχανές (Virtual Machines/VMs) τρέχουν πάνω στον ίδιο φυσικό εξυπηρετητή (server, ο Η/Υ), και μπορεί θεωρητικά ο hypervisor (υπερόπτης, που τις διαχειρίζεται) να έχει πρόσβαση στη μνήμη τους. Με την SEV όμως, κάθε VM έχει το δικό της, ξεχωριστό κλειδί και η μνήμη της είναι κρυπτογραφημένη με αυτό. Αυτό σημαίνει ότι ούτε ο hypervisor ούτε άλλη εικονική μηχανή μπορεί να «δεί» τα δεδομένα της, κάθε εικονική μηχανή λειτουργεί ως μία απομονωμένη μονάδα ως προς την ΜΤΠ, με ανεξάρτητο key domain. Ας κρατήσουμε τη λογική της τμηματοποίησης.

Ακόμα πιο πρόσφατα, η AMD προχώρησε ένα βήμα παραπέρα με το SEV-SNP (Secure Nested Paging Protection/Προστασία Ασφαλούς Φωλιασμένης Σελιδοποίησης), το οποίο επεκτείνει την προστασία του Η/Υ προσθέτοντας ελέγχους ακεραιότητας και μηχανισμούς όπου αυξάνουν την εμπιστευτικότητα. Αυτό έρχεται να απαντήσει σε πιο σύνθετα ερωτήματα, όπως: τι γίνεται αν ο hypervisor είναι κακόβουλος και προσπαθήσει να αλλάξει τον τρόπο που «βλέπει» τη μνήμη η VM μέσω της σελιδοποίησης; Ή αν προσπαθήσει να παραποιήσει δεδομένα κατά την ανταλλαγή μεταξύ επεξεργαστή και μνήμης. Το SEV-SNP προσθέτει μηχανισμούς επαλήθευσης ακεραιότητας και επιτρέπει στη VM να επαληθεύσει την ίδια της την κατάσταση, δηλαδή όχι μόνο να μην της διαβάσουν τα δεδομένα, αλλά ούτε να μπορέσουν να της τα αλλοιώσουν.

Σε τεχνικό επίπεδο, η υλοποίηση της SME βασίζεται σε έναν αποκλειστικό AES κρυπτογραφητή/αποκρυπτογραφητή όπου είναι ενσωματωμένος απευθείας στον ελεγκτή μνήμης της ΚΜΕ. Κάθε φορά που ο επεξεργαστής στέλνει δεδομένα προς τη μνήμη, αυτά περνούν μέσα από αυτόν τον μηχανισμό και αποθηκεύονται κρυπτογραφημένα. Το αντίστροφο συμβαίνει όταν διαβάζονται: η αποκρυπτογράφηση γίνεται αυτόματα πριν φτάσουν στον πυρήνα του επεξεργαστή, χωρίς παρέμβαση λογισμικού. Το κλειδί κρυπτογράφησης παράγεται κατά την εκκίνηση του Η/Υ και αποθηκεύεται σε ειδικούς καταχωρητές μέσα στην ΚΜΕ, στους οποίους δεν έχει πρόσβαση κανένα επίπεδο λογισμικού, ούτε το BIOS, ούτε το OS, ούτε καν SMM (System Management Mode/Λειτουργία Διαχείρισης Συστήματος). Στην περίπτωση της SEV, επεκτείνεται το ίδιο μοντέλο για κάθε

εικονική μηχανή, με την προσθήκη διακριτών κλειδιών κρυπτογράφησης ανά εικονική μηχανή, τα οποία διαχειρίζεται η μονάδα AMD Secure Processor (PSP), ένας ARM-βασισμένος συνεπεξεργαστής ασφαλείας, ενσωματωμένος στην KME. Ο PSP είναι υπεύθυνος για τη δημιουργία, την αποθήκευση και τη διανομή των κλειδιών με ασφαλή τρόπο, ώστε ούτε ο hypervisor να μπορεί να δει τη μνήμη των εικονικών μηχανών. Στο SEV-SNP, ο PSP συντονίζεται με επιπλέον μηχανισμούς ελέγχου ακεραιότητας της μνήμης, όπως Reverse Map Tables (RMP/Πίνακες Αντίστροφης Χαρτογράφησης, πίνακες που καταγράφουν ποιο φυσικό page ανήκει σε ποιον (π.χ. VM ή hypervisor)) και έλεγχο καταχωρήσεων του πίνακα φωλιασμένων σελίδων (nested page table entries), ώστε να αποτραπούν spoofing (πλαστογράφηση) επιθέσεις και παραποιημένης χαρτογράφηση της μνήμης (memory remapping) από κακόβουλο hypervisor. Όλο αυτό το σύστημα υλοποιείται πλήρως στο υλικό και επιτρέπει στον επεξεργαστή να διαχειρίζεται την ασφάλεια της μνήμης χωρίς εξάρτηση από καμία εξωτερική πηγή εμπιστοσύνης και η επιβάρυνση της απόδοσης είναι σχετικά μικρή (περίπου 1-3% για την SME).

Όλες αυτές οι τεχνολογίες της AMD προσπαθούν να απαντήσουν ακριβώς στα ερωτήματα που γεννιούνται όταν προτείνουμε την κρυπτογράφηση της μνήμης ως λύση: πού θα γίνεται η κρυπτογράφηση; ποιος δημιουργεί και κρατά τα κλειδιά; τι επιβάρυνση έχει στο σύστημα; ποιο επίπεδο του συστήματος είναι τελικά το πιο έμπιστο; Η απάντηση της AMD είναι ξεκάθαρη: η ίδια η KME είναι το μοναδικό σημείο εμπιστοσύνης, και όλα τα υπόλοιπα, μνήμη, λογισμικό, ακόμα και ο hypervisor, δεν χρειάζεται να είναι έμπιστα.

Intel Total Memory Encryption (TME)/Multi-Key TME (MKTME)

Από την πλευρά της Intel, η προσέγγιση στο ίδιο πρόβλημα οδήγησε στην ανάπτυξη της τεχνολογίας Total Memory Encryption (Συνολική Κρυπτογράφηση Μνήμης/TME). Η λογική είναι παρόμοια με αυτή της AMD SME: αν δεν μπορούμε να εμπιστευτούμε το τι γίνεται στη φυσική μνήμη, ας την κρυπτογραφούμε ολόκληρη.

Το TME εφαρμόζει κρυπτογράφηση σε ολόκληρη τη φυσική μνήμη του συστήματος, αυτόματα και σε επίπεδο υλικού, χωρίς να απαιτείται καμία αλλαγή σε επίπεδο εφαρμογών ή λειτουργικού. Δηλαδή, ο επεξεργαστής είναι υπεύθυνος να κρυπτογραφεί τα δεδομένα λίγο πριν αυτά εγγραφούν στη μνήμη και να τα αποκρυπτογραφεί όταν ανακτώνται, χωρίς να το αντιλαμβάνεται κανένα άλλο κομμάτι του συστήματος. Τα κλειδιά δημιουργούνται δυναμικά κατά την εκκίνηση του συστήματος, αποθηκεύονται μέσα στον ίδιο τον επεξεργαστή (στον λεγόμενο Key Locker/Κλειδοκράτορας) και δεν βγαίνουν ποτέ προς τα έξω. Επομένως, ακόμα και σε περίπτωση φυσικής πρόσβασης στον υπολογιστή, το περιεχόμενο της μνήμης είναι κρυπτογραφημένο και άχρηστο για τον επιτιθέμενο, αν και αυτό εξαρτάται και από τον αλγόριθμο κρυπτογράφησης όπου χρησιμοποιείται, ας το κρατήσουμε υπόψη.

Ωστόσο, η βασική υλοποίηση του TME έχει ένα σημαντικό περιορισμό: χρησιμοποιείται ένα και μοναδικό κλειδί για όλη τη μνήμη. Αυτό σημαίνει ότι αν κάποια στιγμή αυτό το κλειδί «διαρρεύσει» ή παρακαμφθεί (π.χ. μέσω κάποιου bug στο firmware ή στην αλυσίδα εκκίνησης, η εσκεμμένης επίθεσης όπως του glitching/εσκεμμένη δημιουργία σφάλματος), τότε όλη η προστασία καταρρέει μονομιάς. Επιπλέον, δεν υπάρχει απομόνωση μεταξύ διαφορετικών εφαρμογών ή χρηστών, άρα δεν εξυπηρετεί καλά σενάρια όπου απαιτείται μεμονωμένη προστασία (π.χ. cloud, multi-tenant περιβάλλοντα).

Από τη στιγμή όμως που ένα single key δεν αρκεί για πολυμισθωτικά (πολλαπλές εικονικές μηχανές) περιβάλλοντα, η Intel εισήγαγε το MKTME. Πρόκειται για μία επέκταση της αρχικής ιδέας του TME, όπου πλέον μπορούν να υπάρχουν πολλαπλά κλειδιά κρυπτογράφησης, και κάθε περιοχή της μνήμης μπορεί να συσχετιστεί με ένα διαφορετικό κλειδί.

Έτσι, μπορούμε να φανταστούμε ότι το σύστημα χωρίζεται σε «ζώνες/θύλακες εμπιστοσύνης» (enclaves/Intel SGX), καθεμία με τη δική της πολιτική και το δικό της κλειδί. Με αυτόν τον τρόπο μπορούμε να προστατέψουμε διαφορετικούς χρήστες ή διεργασίες μεταξύ τους, ή ακόμα και να απομονώσουμε ευαίσθητες υπορουτίνες (συναρτήσεις/functions) ενός προγράμματος από το υπόλοιπο σύστημα.

Σε επίπεδο τεχνικής υλοποίησης, το Intel TME βασίζεται στην ύπαρξη μίας μονάδας κρυπτογράφησης στο υλικό (encryption engine), ενσωματωμένης στον ελεγκτή μνήμης της ΚΜΕ, ο οποίος εφαρμόζει inline (επί τόπου) κρυπτογράφηση και αποκρυπτογράφηση σε όλα τα δεδομένα που περνούν μεταξύ των κρυφών μνημών και της ΜΤΠ. Η κρυπτογράφηση γίνεται συνήθως με τον αλγόριθμο AES-XTS με κλειδί 128 bit ανά μονάδα δεδομένων, και το μυστικό κλειδί (TME key, 256 bit) αποθηκεύεται αποκλειστικά εντός της ΚΜΕ, χωρίς δυνατότητα ανάγνωσης ή εξαγωγής του από κανένα λογισμικό, ούτε καν από SMM ή BIOS. Η δημιουργία του γίνεται κατά την εκκίνηση μέσω μίας διαδικασίας εν' ονόματι «secure key derivation process» (διαδικασία ασφαλούς παραγωγής κλειδιού) όπου χρησιμοποιεί Random Number Generators (RNG), fuses, Physical Unclonable Function (PUF/μονάδα όπου δημιουργεί ξεχωριστά «αποτυπώματα»), και αποθηκεύεται σε ειδικούς καταχωρητές που «κλειδώνουν» μετά την ενεργοποίηση του TME. Ο μηχανισμός TME λειτουργεί διαφανώς, δεν απαιτεί υποστήριξη από το λειτουργικό ή τις εφαρμογές που τρέχει, και δεν αλλάζει τον τρόπο προσπέλασης της μνήμης ή τις διευθύνσεις. Από άποψη απόδοσης, επειδή εντολές που χρησιμοποιούνται για κρυπτογράφηση με τη χρήση του AES, hardware acceleration

(επιτάχυνση υλικού) μειώνει σημαντικά την επιβάρυνση της συνολικής απόδοσης, αν και η ενεργοποίησή του δεν είναι πάντα προεπιλεγμένη και αφήνεται στη διακριτική ευχέρεια των OEMs (Original Equipment Manufacturer/Κατασκευαστή Πρότυπου Εξοπλισμού) και του BIOS.

Το MKTME επεκτείνει την ίδια ιδέα, προσθέτοντας πολλαπλά κλειδιά κρυπτογράφησης και υποστηρίζοντας δυναμική αντιστοίχιση σελίδων μνήμης σε συγκεκριμένα κλειδιά μέσω ειδικών επεκτάσεων στους πίνακες σελιδοποίησης. Για την υποστήριξη αυτής της δυνατότητας, το λειτουργικό (ή ο hypervisor) συνεργάζεται με την KME ώστε να συσχετίζει διαφορετικές περιοχές στην μνήμη με συγκεκριμένα KeyIDs (Πεδίο που ταυτοποιεί το κλειδί και τη χρήση του). Ο encryption engine του MKTME περιλαμβάνει ένα KeyID πεδίο (π.χ. 5 bits για 32 κλειδιά για τους Xeon) το οποίο χρησιμοποιείται κατά την κρυπτογράφηση και αποκρυπτογράφηση κάθε γραμμής δεδομένων στη μνήμη, ενώ τα πραγματικά κλειδιά βρίσκονται σε έναν Key Locker (κλειδοκράτορα) εντός του επεξεργαστή. Η επιλογή του κλειδιού γίνεται κατά την διάρκεια λειτουργίας της συσκευής ανάλογα με τον τρέχοντα KeyID και ο μηχανισμός υποστηρίζει έλεγχο πρόσβασης με βάση πολιτικές λογισμικού ή hypervisor. Ωστόσο, επειδή το MKTME δεν προσφέρει προστασία ακεραιότητας από μόνο του, για ένα πλήρες μοντέλο ασφαλείας απαιτείται συνδυασμός με άλλες τεχνολογίες όπως Intel TDX ή SGX, οι οποίες παρέχουν απομόνωση και ταυτοποίηση (επικύρωση). Τελικά, η τεχνολογία αυτή προσφέρει ένα αρθρωτό (προστίθεται εύκολα σε υπάρχουσες αρχιτεκτονικές) εργαλείο για την εφαρμογή πολιτικών ασφάλειας μνήμης, αλλά απαιτεί συμμετοχή και εμπιστοσύνη σε επίπεδο OS/hypervisor για σωστή διαχείριση.

Φυσικά, αυτό ανοίγει μία σειρά από νέα ερωτήματα: ποιος θα διαχειρίζεται αυτά τα κλειδιά; πώς θα εξασφαλίζεται ότι το mapping (συσχέτιση διευθύνσεων) με τα κλειδιά, δεν αλλοιώνεται από κακόβουλο κώδικα; Και, ίσως πιο πρακτικά, τι αντίκτυπο έχει αυτή η πολυπλοκότητα στην απόδοση του συστήματος (σύμφωνα με την έρευνα, <3 % σε SPECint, ~5 % σε I/O benchmarks); Η Intel προτείνει συνήθως συνδυασμό της MKTME με άλλες τεχνολογίες, όπως Intel SGX ή Trust Domains, για να ενισχύσει το μοντέλο εμπιστοσύνης, όμως όλα αυτά φέρνουν ένα σημαντικό βάρος διαχείρισης και σχεδιασμού σε επίπεδο λογισμικού αλλά και hypervisor.

Intel SGX (Software Guard Extensions)

Σε αυτό το σημείο, αξίζει να αναφέρουμε έναν μηχανισμό, που μάλιστα βασίστηκαν κάποιες τεχνικές, που όμως αφορά υλικό για την προστασία του λογισμικού. Μια από τις πρώτες υλοποιήσεις των enclaves ήταν το Intel SGX. Πρόκειται για επέκταση των εντολών της x86 ISA, όπου ο επεξεργαστής μπορεί να δημιουργήσει μια απομονωμένη περιοχή μνήμης, γνωστή ως enclave, στην οποία ούτε το λειτουργικό, ούτε ο hypervisor, ούτε και το BIOS ή η SMM μπορούν να έχουν πρόσβαση.

Το enclave αυτό έχει δική του περιοχή μνήμης μέσα στην ΜΤΠ, και ό,τι δεδομένο αποθηκεύεται σε αυτή είναι αυτόματα κρυπτογραφημένο, τόσο κατά την αποθήκευση στη ΜΤΠ όσο και κατά τη μετάδοσή τους από και προς τον επεξεργαστή. Η αποκρυπτογράφηση γίνεται μέσα στην ΚΜΕ, και άρα το περιεχόμενο είναι προσβάσιμο μόνο όταν ο κώδικας τρέχει μέσα στο enclave. Επιπλέον, υποστηρίζεται απομακρυσμένη επικύρωση (remote attestation), ώστε μια εφαρμογή να μπορεί να επιβεβαιώσει σε τρίτους ότι τρέχει σε γνήσιο SGX υλικό και ότι το enclave της δεν έχει τροποποιηθεί.

Ωστόσο, το SGX έχει σοβαρούς περιορισμούς. Το μέγεθος του διαθέσιμου χώρου κρυπτογράφησης (EPC, Enclave Page Cache) είναι μικρό, με αποτέλεσμα να γίνονται συνεχώς αλλαγές σελίδας (page swapping) που μειώνουν την απόδοση. Επίσης, το SGX είναι κατάλληλο κυρίως για μικρές υπορουτίνες ή μικρά πλαίσια εφαρμογών και όχι για ολόκληρη απομόνωση εικονικής μηχανής. Γι’ αυτό και σε βάθος χρόνου αντικαταστάθηκε από το Intel TDX, το οποίο βασίζεται στις ίδιες ιδέες αλλά εφαρμόζεται σε ολόκληρες εικονικές μηχανές, με υψηλότερη απόδοση και περισσότερη συμβατότητα.

Από τεχνικής πλευράς, το Intel SGX υλοποιεί μια εξειδικευμένη περιοχή μνήμης με την ονομασία EPC (Enclave Page Cache/Κρυφή μνήμη Σελίδας Θύλακα), εντός της οποίας αποθηκεύονται οι σελίδες μνήμης του enclave σε κρυπτογραφημένη μορφή. Η κρυπτογράφηση γίνεται από τον ελεγκτή μνήμης της ΚΜΕ κατά την έξοδο προς τη ΜΤΠ, ενώ η αποκρυπτογράφηση συμβαίνει μόνο εντός του επεξεργαστή, αποκλειστικά όταν εκτελείται ο κώδικας του enclave. Η διαδικασία αυτή είναι απολύτως αδιαφανής για το λειτουργικό σύστημα ή άλλες εφαρμογές, που δεν μπορούν να παρατηρήσουν ή να παρέμβουν στα δεδομένα του enclave. Το EPC έχει αυστηρά περιορισμένο μέγεθος (συνήθως έως 128MB), και αν ξεπεραστεί, ενεργοποιείται μηχανισμός swapping (φέρει δεδομένα από την μη πτητική μνήμη του συστήματος π.χ. σκληρός δίσκος) σελίδων που προκαλεί σημαντική πτώση απόδοσης.

Κάθε σελίδα που διαχειρίζεται το SGX συνοδεύεται από metadata ακεραιότητας και nonce (ειδικός αριθμός χρήσιμος για την διαδικασία της αποκρυπτογράφησης) για αποτροπή επιθέσεων τύπου replay (επανάληψης/αναπαραγωγής) ή tampering (εσκεμμένη αλλαγή δεδομένων). Ο αλγόριθμος που χρησιμοποιείται για την προστασία είναι AES-GCM, ώστε να συνδυάζει κρυπτογράφηση και αυθεντικοποίηση, ο ίδιος αλγόριθμος που χρησιμοποιείται και στη λύση που παρουσιάζεται σε αυτή την εργασία. Επιπλέον, το SGX υποστηρίζει μηχανισμό απομακρυσμένης επικύρωσης (Remote Attestation), ο οποίος επιτρέπει σε εξωτερικό τρίτο μέρος να επαληθεύσει την ταυτότητα του enclave και την ακεραιότητα του κώδικά του, μέσω ψηφιακής υπογραφής που δημιουργεί η ίδια η ΚΜΕ χρησιμοποιώντας πιστοποιημένα κλειδιά.

Η όλη προστασία βασίζεται στην αρχή ότι ο επεξεργαστής είναι το μοναδικό έμπιστο σημείο (Root of Trust/Ρίζα Εμπιστοσύνης), με πλήρη απομόνωση από το λειτουργικό, το υλικολογισμικό (firmware), ακόμα και από πιο χαμηλού επιπέδου προνόμια όπως το SMM ή το BIOS. Παρά την ισχυρή απομόνωση, όμως, η πρακτική εφαρμογή του SGX περιορίστηκε

λόγω του μικρού EPC (128 MB ή και λιγότερο, με αποτέλεσμα η μεταφορά σελίδων στην ΜΤΠ να προκαλεί έως και 20 φορές επιβάρυνση σε διακοπές Ε/Ε, και ειδικά με επιβαρυμένους φόρτους εργασίας), πολυπλοκότητας σχεδίασης και παραγωγής και ευπάθειας σε επιθέσεις πλευρικών καναλιών (side-channel attacks) τύπου Spectre/Meltdown*, τα οποία εκμεταλλεύονται έμμεσα στοιχεία όπως χρονισμούς, cache hits/misses (ευστοχίες/αστοχίες) ή speculative execution για την εξαγωγή δεδομένων από το enclave, που επανειλημμένα έδειξαν τρόπους έμμεσης διαρροής δεδομένων σε κακόβουλους χρήστες. Για τους λόγους αυτούς, το SGX αποσύρθηκε από νεότερες σειρές επεξεργαστών, υπερκαλυπτόμενο σε μεγάλο βαθμό από την τεχνολογία TDX.

Intel TDX (Trust Domain Extensions)

Αν και το TME της Intel παρέχει ένα καλό θεμέλιο προστασίας, υστερεί στο βασικό σημείο που αφορούν οι περισσότερες απειλές μνήμης: την απομόνωση (isolation). Πιο συγκεκριμένα, το TME κρυπτογραφεί όλη την ΜΤΠ με ένα κοινόχρηστο κλειδί, το οποίο είναι προσβάσιμο (ή τουλάχιστον ορατό έμμεσα) από τον hypervisor ή το λειτουργικό σύστημα. Αυτό σημαίνει ότι, παρ' όλη την προστασία σε φυσική επίθεση (π.χ. cold boot), ο επιτιθέμενος σε επίπεδο hypervisor μπορεί ακόμα να διαβάσει τη μνήμη των εικονικών μηχανών.

Εδώ έρχεται να δώσει απάντηση το Intel TDX (Trust Domain Extensions/Επέκταση Τομέα Εμπιστοσύνης). Ουσιαστικά πρόκειται για μία αρχιτεκτονική επέκταση όπου βασίζεται πάνω στο TME, αλλά προσθέτει απομόνωση επιβαλλόμενη από το ίδιο το υλικό μεταξύ των εικονικών μηχανών και του hypervisor. Η λογική του TDX είναι ότι κάθε επισκέπτης/guest εικονική μηχανή (το οποίο ονομάζεται πλέον Trust Domain ή TD) έχει τη δική του πλήρως απομονωμένη και κρυπτογραφημένη μνήμη, και ο hypervisor δεν μπορεί να τη διαβάσει ή να την επεξεργαστεί με κανέναν τρόπο αφού τα κλειδιά δεν αποθηκεύονται στην ΜΤΠ.

Η απομόνωση αυτή δεν γίνεται μόνο σε επίπεδο μνήμης, αλλά και σε επίπεδο καταχωρητών, διαχείριση διακοπών (interrupt handling) και I/O ή E/E (Input/Output ή Είσοδος/Εξοδος). Δηλαδή, ένα TD εκτελείται μέσα σε ένα πλαίσιο ασφαλούς διαχείρισης από το υλικό, που φέρνει πολλά κοινά με τα enclaves (SGX), αλλά σε πολύ μεγαλύτερη κλίμακα (ολόκληρη εικονική μηχανή). Παράλληλα, ο κώδικας boot (εκκίνησης) του TD είναι signed/υπογεγραμμένος (πιστοποιημένος άρα έμπιστος), και υπάρχει υποστήριξη για απομακρυσμένη επικύρωση, ώστε να αποδεικνύεται ότι το TD τρέχει σε έμπιστο/ασφαλισμένο υλικό και δεν έχει τροποποιηθεί (Integrity/Ακεραιότητα).

Σε αντίθεση με το MKTME που προσπαθεί να κάνει πολλαπλά τομείς σε μία κοινή μνήμη, το TDX τα πηγαίνει ένα βήμα παραπέρα: όχι μόνο υπάρχει ξεχωριστό κλειδί ανά TD, αλλά η διαχείριση τους γίνεται αποκλειστικά εντός της μονάδας TDX, δηλαδή μιας microcode (μικροκώδικα) υποδομής εκτός ελέγχου του hypervisor. Στην πράξη, ο hypervisor δεν βλέπει τίποτα παραπάνω από πλαίσια φυσικών σελιδών, χωρίς να γνωρίζει ούτε τι τύπος δεδομένων υπάρχει εκεί, ούτε καν πότε γίνονται αλλαγές περιεχομένου.

Σε επίπεδο υλοποίησης, το TDX χτίζεται γύρω από μια νέα microcode οντότητα (μικροκώδικας), την μονάδα TDX, όπου βρίσκεται δίπλα στον ελεγκτή μνήμης και αναλαμβάνει όλο τον κύκλο ζωής ενός Trust Domain. Επίσης τρέχει σε SEAM (Secure Execution and Attestation Mode), μία νέα λογική εκτέλεσης της Intel καθώς και υποστηρίζονται λειτουργίες κοινής μνήμης μεταξύ E/E και Hypervisor. Κάθε TD διαθέτει το δικό του TD Root (TDR), ένα 4 KB metadata object (αντικείμενο) μέσα στην ΜΤΠ. Το TDR περιέχει hash-based (Αλγόριθμος Κατακερματισμού) μετρήσεις του firmware, την πολιτική ασφαλείας του TD και έναν TD Encryption Key (TDEK), ο οποίος παράγεται από μία υλική μονάδα παραγωγής τυχαίων αριθμών, σφραγίζεται με το SoC-unique Root Key, και φυλάσσεται σε στο Key Locker του τσιπ, απρόσιτο από κάθε λογισμικό. Όλες οι σελίδες που χαρτογραφούνται στη TD Private EPT (Extended Page Table/Εκτεταμένος Πίνακας Σελιδών) φέρουν επιπλέον KeyID πεδίο· κατά την έξοδο προς DRAM ο TDX Module εκτελεί κρυπτογράφηση AES-XTS 256 bit, ενώ παράλληλα ενημερώνει τον PAMT (Physical Address Metadata Table), σαν το RMP της AMD, ώστε να αποτρέψει κακόβουλη αλλαγή του περιεχομένου μνήμης από τον hypervisor. Για ακεραιότητα, κάθε 64-byte cache line συνοδεύεται από ένα MAC (Message Authentication Code/Μήνυμα Αυθεντικοποιήσης Κώδικα) που υπολογίζεται μέσω του αλγορίθμου AES-GCM. Η επαλήθευση γίνεται κατά την

διαδικασία που φέρουμε δεδομένα από τη μνήμη, πριν αυτά φτάσουν στις caches. Το ίδιο μονοπάτι διασφαλίζει και τα TD Control Registers (TDCS), που κρατούν το run-time state/την κατάσταση λειτουργίας (GP/Control Registers, interrupt bitmap, I/O bitmap). Η εκκίνηση ενός TD περνά από μία Measured Launch (TDH.MNG.INIT) διαδικασία, και η γνησιότητα πιστοποιείται μέσω Remote Attestation: ο TDX Module δημιουργεί ένα μικρό μήνυμα υπογεγραμμένο με ενσωματωμένο ECDSA key πιστοποιημένο από την Intel, επιτρέποντας στον παραλήπτη να ελέγξει ότι ο TDR hash ταιριάζει με το αναμενόμενο. Με αυτόν τον τρόπο, ακόμα κι αν ο hypervisor επιχειρήσει DMA, EPT spoofing ή register snooping (Κακόβουλη Ανάκτηση Περιεχομένων των Καταχωρητών), προσκρούει στη κρυπτογράφηση, επαλήθευση του MAC και RMP-βασισμένο έλεγχο πρόσβαση που επιβάλλονται αποκλειστικά από το ίδιο το υλικό.

Αυτό φυσικά δεν έρχεται χωρίς κόστος, η απομόνωση αυτή έχει σημαντική επίδραση στην απόδοση. Το TDX προσθέτει χρόνο αδράνειας κυρίως σε I/O και αλλαγές περιεχομένου. Επίσης, δεν είναι ακόμα διαθέσιμο σε όλο το υλικό: απαιτεί συγκεκριμένες σειρές επεξεργαστών (π.χ. Xeon Scalable 4ης γενιάς και άνω), υποστήριξη στο BIOS, και στο υλικολογισμικό της πλατφόρμας.

Παρόλα αυτά, αποτελεί την πιο ισχυρή προσπάθεια της Intel να απαντήσει στις λύσεις τύπου AMD SEV-SNP. Αν και με διαφορετική αρχιτεκτονική, μοιράζεται την ίδια βασική αρχή: η μνήμη της εικονικής μηχανής είναι πλήρως προστατευμένη ακόμα και από τον ίδιο τον οικοδεσπότη. Και τελικά, αυτή είναι η ουσία της ασφαλούς μνήμης στον κόσμο του νέφους (cloud), αλλά τί γίνεται με τις κλασσικές χρήσεις του Η/Υ, πέρα από διακομιστές (servers);

TME-Box

Ένα από τα βασικά προβλήματα που προκύπτουν από την υλοποίηση του Intel TME/MKTME, που όπως είδαμε αντιμετωπίστηκε από την TDX είναι ότι, παρά το γεγονός πως παρέχεται κρυπτογράφηση σε επίπεδο φυσικής μνήμης, λείπει το κομμάτι της ακεραιότητας (integrity). Δηλαδή, μπορεί μεν τα δεδομένα να είναι κρυπτογραφημένα, αλλά τίποτα δεν σταματάει τον επιτιθέμενο από το να αντικαταστήσει ή τροποποιήσει αυτά τα κρυπτογραφημένα δεδομένα, κάτι που σε πολλές περιπτώσεις οδηγεί σε μη ανιχνεύσιμες επιθέσεις, όπως data replay (Επανάληψη/Αναπαραγωγή Δεδομένων) ή targeted corruption (στοχευμένη αλλοίωση). Επιπλέον, η MKTME βασίζεται σε software-defined (καθορισμένες από λογισμικό) πολιτικές απομόνωσης (δηλαδή τα κλειδιά τα χειρίζεται το λειτουργικό ή ο hypervisor), πράγμα που σημαίνει ότι η εμπιστοσύνη δεν είναι πλήρως υλικοεπίπεδη (hardware-level).

Αυτό το κενό ήρθε να καλύψει μία πρόσφατη ερευνητική πρόταση με το όνομα TME-Box. Η βασική ιδέα είναι η εξής: εφόσον προσφέρουμε κρυπτογράφηση μνήμης, τότε ας δημιουργήσουμε και πραγματικά απομονωμένες περιοχές μέσα στη φυσική μνήμη, όπου και η εμπιστευτικότητα (confidentiality) και η ακεραιότητα είναι εγγυημένες εξ ολοκλήρου από το υλικό, χωρίς καμία εμπλοκή του λειτουργικού ή του hypervisor.

Το TME-Box εισάγει την έννοια του box (κουτιού/απομόνωση), μιας προστατευμένης περιοχής μνήμης η οποία έχει ξεχωριστό κλειδί, υπογραφή ακεραιότητας, και ένα σετ πολιτικών πρόσβασης όπου αποθηκεύονται μέσα στον ίδιο τον επεξεργαστή. Η δημιουργία του box γίνεται μέσω ενός ελεγχόμενου μηχανισμού (με trusted entry points/σημεία αξιόπιστης εισόδου) και, μόλις δημιουργηθεί, ούτε το OS ούτε καν ο hypervisor μπορούν να διαβάσουν ή να τροποποιήσουν τα περιεχόμενά του. Ο μόνος που μπορεί να αλληλοεπιδράσει με αυτό είναι ο κώδικας που τρέχει μέσα στο box, κάτι που θυμίζει σε κάποιο βαθμό τα enclaves της Intel SGX, αλλά σε πολύ χαμηλότερο επίπεδο (δηλαδή χωρίς να χρειάζεται ειδική υποστήριξη από το λογισμικό).

Ένα ακόμα ενδιαφέρον στοιχείο του TME-Box είναι ότι προσφέρει μικρό αποτύπωμα στον σχεδιασμό του υλικού. Δηλαδή, προτείνεται ως μία επεκτάσιμη βελτίωση πάνω από το υπάρχον TME/MKTME, χωρίς να απαιτεί ριζικές αλλαγές στον τρόπο όπου λειτουργούν οι επεξεργαστές ή η ίδια η μνήμη. Η ερευνητική εργασία που το προτείνει καταγράφει πως ο αντίκτυπος στην απόδοση είναι σχετικά μικρός (~5-6% για τις περισσότερες εφαρμογές), κάτι που καθιστά την προσέγγιση πρακτικά εφαρμόσιμη και όχι μόνο θεωρητικά ενδιαφέρουσα.

Σε τεχνικό επίπεδο, το TME-Box επεκτείνει την υπάρχουσα υποδομή του Intel MKTME προσθέτοντας μία νέα κατηγορία φυσικών σελίδων, τις box pages, οι οποίες διαθέτουν επιπλέον metadata πεδία όπως: BoxID, Access Policy Descriptor (APD/Περιγραφέας Πολιτικής Πρόσβασης) και ένα μήνυμα ακεραιότητας, το MAC, όπως ακριβώς εξηγήθηκε πιο πάνω. Οι σελίδες αυτές διαχειρίζονται από τον TME-Box Controller (Ελεγκτής), έναν μηχανισμό μέσα στο ελεγκτή μνήμης, ο οποίος διατηρεί την κατάσταση κάθε box και πιστοποιημένες λίστες επιτρεπόμενων σημείων εισόδου. Όταν μια διεργασία επιχειρεί πρόσβαση σε box page, ο TME-Box Controller/Ελεγκτής επαληθεύει ότι: (α) ο τρέχον execution context (συμφραζόμενο πλαίσιο κώδικα που εκτελείτε) ανήκει στο κατάλληλο box μέσω του BoxID, (β) η διεύθυνση αντιστοιχεί σε έγκυρο επιτρεπτό offset (μετατόπιση, π.χ. για αποφυγή buffer overflow επιθέσεις) και (γ) το MAC ταιριάζει με το αποθηκευμένο tag, χρησιμοποιώντας AES-GCM ή Chaskey για την επαλήθευσή του. Τα κλειδιά κρυπτογράφησης ανά box (BoxKey) παράγονται μέσω on-die (στο τσιπ) TRNG (True Random Number Generator/Γεννήτρια Πραγματικά Τυχαίων Αριθμών) κατά τη δημιουργία

του box, σφραγίζονται με device-unique fuses (μία ακόμη παράμετρος ασφαλείας) και παραμένουν μη προσβάσιμα από λογισμικό. Η δημιουργία ενός box γίνεται με τη χρήση ειδικής εντολής της KME (που χρειάζεται ειδικά προνόμια) που ενεργοποιεί trusted entry flow (π.χ. TMEBOX.CREATE) (ροή αξιόπιστης εισόδου), ενώ η προσπέλαση επιτρέπεται μόνο σε execution units/context (τμήματα εκτέλεσης, διεργασίες) όπου έχουν προηγουμένως ταυτοποιηθεί μέσω hardware-based attestation chain (trusted boot chain). Επιπλέον, κάθε box διατηρεί counter-based nonce (μία ακόμη παράμετρο πιστοποίησης ότι δεν έχει παραποιηθεί κάτι) για κάθε εγγραφή σελίδας, ώστε να αποτρέπεται data replay ακόμα κι αν ο επιτιθέμενος έχει φυσική πρόσβαση στην ΜΤΠ. Το σημαντικό είναι ότι οι box πολιτικές και τα integrity tags αποθηκεύονται εκτός ΜΤΠ, σε secured on-chip SRAM (στατική μνήμη) με ECC (Error Correcting Code/Κώδικα Ελέγχου Σφάλματος), έτσι ώστε ακόμα και cold-boot ή Row Hammer* επιθέσεις να μην επηρεάζουν την εύρυθμη λειτουργία. Η αρχιτεκτονική σχεδιάστηκε με ελάχιστη επέμβαση σε υπάρχουσες μικροαρχιτεκτονικές KME, με μόλις ~6% επίπτωση στην απόκριση και ~1% αύξηση στην έκταση του λογικής του ελεγκτή.

Βέβαια, το TME-Box παραμένει σε ερευνητικό στάδιο και δεν έχει υλοποιηθεί σε εμπορικούς επεξεργαστές. Όμως έχει ιδιαίτερη σημασία γιατί συνδέει την έννοια της προστατευμένης μνήμης με μηχανισμούς όπως απομόνωση επιβαλλόμενη από το ίδιο το υλικό, χωρίς να απαιτεί ριζικές αλλαγές στο λογισμικό. Σε ένα περιβάλλον όπου η εμπιστοσύνη στο λειτουργικό και στον hypervisor θεωρείται πλέον ανεπαρκής (λόγω εσωτερικών πιθανών επιθέσεων), τέτοιες τεχνολογίες φαίνονται να δείχνουν τον δρόμο για το μέλλον της υπολογιστικής ασφάλειας στο επίπεδο της μνήμης.

IBM POWER10 Transparent Memory Encryption

Μία λιγότερο γνωστή, αλλά εξαιρετικά σημαντική προσέγγιση έρχεται από την IBM με τον επεξεργαστή POWER10 και την ενσωματωμένη υποστήριξη για Transparent Memory Encryption (TME). Σε αντίθεση με τις άλλες τεχνολογίες που παρουσιάζονται ως add-ons ή extensions (προσθήκες ή επεκτάσης), εδώ η κρυπτογράφηση μνήμης αποτελεί βασικό και πλήρως ενσωματωμένο χαρακτηριστικό του αρχιτεκτονικού σχεδιασμού.

Η θεμελιώδης ιδέα πίσω από το TME της IBM είναι απλή: όλη η φυσική μνήμη του συστήματος είναι, εξ ορισμού, κρυπτογραφημένη, χωρίς να απαιτείται καμία αλλαγή σε λογισμικό, χωρίς προσαρμογές σε λειτουργικά συστήματα ή εφαρμογές, και, ίσως και το πιο σημαντικό, χωρίς καν να το αντιλαμβάνεται ο χρήστης. Όλα γίνονται διαφανώς (transparently), όπως εξηγεί και το όνομα της τεχνολογίας.

Σε επίπεδο υλοποίησης, το IBM POWER10 Transparent Memory Encryption (Διαφανής Κρυπτογράφηση Μνήμης) βασίζεται σε ενσωματωμένες μονάδες AES-256 XTS engines (μηχανές κρυπτογράφησης τύπου AES με XTS mode/λειτουργία), οι οποίες είναι τοποθετημένες ανά memory controller και λειτουργούν inline, δηλαδή κατά μήκος της διαδρομής πρόσβασης στη φυσική μνήμη, χωρίς την παραμικρή εμπλοκή από λογισμικό. Κάθε πρόσβαση στη μνήμη περνάει από την αντίστοιχη διοχέτευση του υλικού το οποίο εφαρμόζει σε πραγματικό χρόνο (real-time) τον αλγόριθμο AES-XTS με 256-bit κλειδί και αποθηκεύεται αποκλειστικά εντός της secure key vault (ασφαλής θήκη κλειδιών) του επεξεργαστή. Δεν είναι ποτέ ορατό ή εξαγώγιμο από το λειτουργικό ή οποιοδήποτε firmware, και παραμένει έγκυρο μόνο για τη διάρκεια ζωής της συνεδρίας/λειτουργίας του Η/Υ (session lifespan), ουσιαστικά όσο παραμένει ενεργοποιημένος ο Η/Υ. Παράλληλα, το TME υποστηρίζει ECC μετά την κρυπτογράφηση, διατηρώντας τη λειτουργικότητα διόρθωσης σφαλμάτων στο υλικό χωρίς να διαρρέονται δεδομένα. Το σύστημα λειτουργεί σε συνολικό επίπεδο συστήματος για πλήρη κρυπτογράφηση της ΜΤΠ, πράγμα που σημαίνει ότι όλες οι προσβάσεις στη μνήμη προστατεύονται χωρίς εξαιρέσεις, ανεξαρτήτως λειτουργικού, hypervisor ή τύπου διεργασίας. Αντίθετα με λύσεις τύπου MKTME ή SEV, δεν υποστηρίζονται πολλαπλά domains ή virtual machine isolation (απομόνωση VM), καθώς το TME της IBM είναι σχεδιασμένο περισσότερο για περιβάλλοντα single-tenant ή bare-metal, όπου το trust model βασίζεται στο έμπιστο υλικό (trusted hardware root). Παρά την απουσία πολύπλοκων πολιτικών απομόνωσης, η απόδοση του συστήματος παραμένει εξαιρετικά υψηλή: η AES μονάδα υλοποιείται με συνολικό επίπτωση στην απόδοση μικρότερη του 2% σε εφαρμογές HPC, βάσεις δεδομένων όπως π.χ. για Ανάλυση Μεγάλου Όγκου Δεδομένων.

Αυτό επιτυγχάνεται μέσω μίας αποκλειστικά ενσωματωμένη μονάδα, τον AES-256 engine (στο ίδιο το υλικό), η οποίος χειρίζεται την κρυπτογράφηση και την αποκρυπτογράφηση σε πραγματικό χρόνο, κατά την πρόσβαση στα δεδομένα μνήμης. Το κλειδί αποθηκεύεται εντός του επεξεργαστή και παράγεται κατά την εκκίνηση του συστήματος, ενώ προστατεύεται μέσω ειδικών μηχανισμών ασφαλείας που καθιστούν αδύνατη την εξαγωγή ή την υποκλοπή του, ακόμα και από κάποιον με φυσική πρόσβαση στο σύστημα.

Αυτό που κάνει την προσέγγιση της IBM ιδιαίτερα αξιοπρόσεκτη είναι η σχεδόν μηδενική απαίτηση για τροποποιήσεις στο υπάρχον λογισμικό, αλλά και το γεγονός ότι επεκτείνει την προστασία σε μεγάλα, πολυεπεξεργαστικά περιβάλλοντα, όπως αυτά που χρησιμοποιούνται σε κέντρα δεδομένων (data centers π.χ. παρόχους cloud υπηρεσιών) και για HPC (High Performance Computing). Δηλαδή, εδώ δεν μιλάμε απλά μια ΚΜΕ για τον απλό χρήστη, αλλά για συστήματα που οδηγούν κρίσιμες επιχειρησιακές εφαρμογές, βάσεις δεδομένων, και τεράστιας κλίμακας φόρτου εργασίας (π.χ. για διαχείριση Big Data/Μεγάλου Όγκου

Δεδομένων). Σε αυτά τα περιβάλλοντα, η προστασία της μνήμης δεν είναι απλώς μία ακόμα λειτουργία, αλλά προϋπόθεση για ομαλή και ασφαλή λειτουργία.

Παρ' όλα αυτά, η υλοποίηση της IBM διατηρεί έναν περισσότερο «μονολιθικό» χαρακτήρα, δηλαδή δεν προσφέρει (τουλάχιστον προς το παρόν) μηχανισμούς πολυκλειδικής απομόνωσης τύπου MKTME ή enclave-based isolation όπως η SEV-SNP. Εδώ έχουμε ένα ενιαίο κλειδί για όλη τη φυσική μνήμη, και άρα η ασφάλεια βασίζεται στην παραδοχή ότι ο επεξεργαστής και το υλικολογισμικό είναι έμπιστα. Το πλεονέκτημα, φυσικά, είναι η σταθερότητα και η απόδοση, σύμφωνα με την IBM, το κόστος απόδοσης του TME στον POWER10 είναι σχεδόν μηδενικό (<2%), λόγω της βαθιάς ενσωμάτωσης στο silicon (το υλικό όπου χρησιμοποιείται για την παραγωγή των ηλεκτρονικών/Πυρίτιο).

Εν κατακλείδι, η IBM POWER10 TME υλοποιεί με τον πιο "ώριμο" και ενσωματωμένο τρόπο την προστασία της φυσικής μνήμης, απαντώντας στο βασικό ερώτημα: "Και αν κάποιος καταφέρει να δει τα raw δεδομένα της ΜΤΠ, τι θα καταλάβει;". Η λύση που παρουσιάζεται στην εργασία αυτή είναι άμεσα συγκρίσιμη με αυτή την τεχνολογία.

Apple Memory Protection Engine (Secure Enclave/M-series)

Ακολουθώντας μία τελείως διαφορετική σχεδιαστική φιλοσοφία, όπου η ενοποίηση υλικού και λογισμικού γίνεται κανόνας, η Apple έχει ενσωματώσει στους M-series επεξεργαστές της μία τεχνολογία που δεν στοχεύει απλώς στην κρυπτογράφηση της φυσικής μνήμης, αλλά σε πλήρη απομόνωση κρίσιμων δεδομένων μέσω ενός εξειδικευμένου υποσυστήματος: το Memory Protection Engine (MPE/Μηχανισμός Προστασίας Μνήμης).

Πιο συγκεκριμένα, η Apple έχει από χρόνια επενδύσει στην ιδέα του Secure Enclave (Ασφαλής Θυλακας), μιας μικρής, ανεξάρτητης μονάδας μέσα στον επεξεργαστή, με δικό της λειτουργικό, μνήμη και μοντέλο ασφαλείας. Αυτό το Enclave διαχειρίζεται τα πιο ευαίσθητα δεδομένα του συστήματος: βιομετρικά, κωδικούς, κρυπτογραφικά κλειδιά, ακόμα και τον μηχανισμό εκκίνησης (boot integrity/ακεραιότητα κώδικα εκκινητή). Στις M1, M2 και M3 αρχιτεκτονικές, όμως, η προστασία δεν περιορίζεται μόνο εκεί: επεκτείνεται και στη φυσική μνήμη, μέσω του Memory Protection Engine.

Σε αρχιτεκτονικό επίπεδο, το Memory Protection Engine (MPE, Μηχανισμός Προστασίας Μνήμης) της Apple ενσωματώνεται απευθείας στην interconnect fabric (υφιστάμενη διασύνδεση με τον ελεγκτή μνήμης) των επεξεργαστών της σειράς M1/M2/M3, πετυχαίνοντας κρυπτογράφηση ανά γραμμή και ακεραιότητα για όλες τις μεταφορές δεδομένων προς την ΜΤΠ. Ο MPE υποστηρίζει AES-GCM (Galois/Counter Mode) με 128 ή 256-bit κλειδιά, εφαρμόζοντας παράλληλα authentication tags (ετικέτες ακεραιότητας, σαν ψηφιακές υπογραφές) σε κάθε γραμμή μνήμης. Ολόκληρος ο μηχανισμός βασίζεται σε ένα on-chip key ladder (ιεραρχία κλειδιών εντός του πυριτίου, χρήση τους ξεχωριστού ID της κάθε συσκευής/UID), όπου το τελικό memory key παράγεται κατά την εκκίνηση της συσκευής με χρήση hardware-based entropy sources (πηγές εντροπίας από το ίδιο το υλικό), και φυλάσσεται αυστηρά στο Secure Enclave, χωρίς δυνατότητα ανάγνωσης, αποθήκευσης ή αναπαραγωγής από άλλο οικοσύστημα. Επίσης να σημειωθεί ότι το MPE λειτουργεί σε επίπεδο γραμμής cache (π.χ. ανά 64 Byte)

Επιπλέον, το MPE αλληλεπιδρά άμεσα με το SMMU (System Memory Management Unit) ώστε να εφαρμόζονται policy enforcement rules (κανόνες επιβολής πολιτικής) ανάλογα με τη «ζώνη ασφάλειας» (security domain) του κάθε agent. Π.χ. το GPU (Graphics Processing Unit/Μονάδα Επεξεργασίας Γραφικών) έχει πρόσβαση μόνο σε συγκεκριμένα πεδία της μνήμης με δικά του κλειδιά ή tags, ενώ υποσύνολα του kernel space (πυρήνα) (π.χ. iBoot ή kernel extension modules) περιορίζονται δυναμικά ανάλογα με την κατάσταση του συστήματος. Τέλος, τα πρόσφατα M-series chips υποστηρίζουν ARMv8.5-A MTE (Memory Tagging Extension/Επέκταση Ετικετοποίησης Μνήμης), η οποία λειτουργεί συνεργατικά με το MPE για τον εντοπισμό use after free, buffer overflows ή παράνομες προσβάσεις σε real-time, κάτι που ενεργοποιείται σε επίπεδο υλικού. Η συνέργεια MPE και MTE οδηγεί σε ένα σύστημα μνήμης με confidentiality, integrity και fine-grained access control (εμπιστευτικότητα, ακεραιότητα και λεπτομερή έλεγχο πρόσβασης) ενσωματωμένα στον επεξεργαστή, και όχι ως πρόσθετα modules.

Το MPE λειτουργεί ως κρυπτογραφικός επιταχυντής (cryptographic accelerator), ο οποίος εφαρμόζει κρυπτογράφηση και αποκρυπτογράφηση σε δεδομένα που διακινούνται προς και από τη φυσική μνήμη. Κάθε πρόσβαση προς την ΜΤΠ περνάει από αυτόν τον μηχανισμό, ο οποίος λειτουργεί σε επίπεδο υλικού και είναι πλήρως διαφανές για το λογισμικό. Όμως, αυτό που ξεχωρίζει στην υλοποίηση της Apple είναι ότι δεν περιορίζεται στην εμπιστευτικότητα: παρέχεται και ακεραιότητα, και μάλιστα με τρόπο που δεν απαιτεί από τον προγραμματιστή να κάνει τίποτα απολύτως.

Τα κρυπτογραφικά κλειδιά παράγονται κατά την εκκίνηση της συσκευής και αποθηκεύονται εντός του Secure Enclave. Δεν είναι προσβάσιμα ούτε από τον πυρήνα του λειτουργικού συστήματος, ούτε από τον hypervisor, ούτε φυσικά από οποιαδήποτε εφαρμογή. Ούτε καν οι ίδιες οι υπηρεσίες του iOS ή macOS δεν έχουν απευθείας πρόσβαση. Επιπλέον, σε συνδυασμό με το memory tagging (ετικετοποίηση μνήμης) που προσφέρουν τα νεότερα μοντέλα ARM (π.χ. MTE, Memory Tagging Extension), δημιουργείται ένα περιβάλλον όπου ο επεξεργαστής γνωρίζει ποιος έχει πρόσβαση πού και πότε, και μπορεί να μπλοκάρει ή να αναφέρει παραβιάσεις σε επίπεδο runtime (χρονικό διάστημα εκτέλεσης).

Ουσιαστικά, η Apple δεν προσεγγίζει την κρυπτογράφηση μνήμης ως ένα αυτόνομο επίπεδο ασφάλειας, αλλά ως μέρος ενός ευρύτερου, κάθετου οικοσυστήματος προστασίας, από τη σιλικόνη μέχρι το UI (User Interface/Διεπαφή Χρήστη, Λογισμικό). Το αποτέλεσμα είναι ένα σύστημα όπου όχι μόνο τα δεδομένα στη μνήμη είναι προστατευμένα, αλλά και η πρόσβαση στη μνήμη είναι δυναμικά ελεγχόμενη και περιορισμένη, κάτι που αποτρέπει επιθέσεις τύπου cold boot.

Βέβαια, όλα αυτά λειτουργούν μέσα στο ιδιαίτερα κλειστό οικοσύστημα της Apple: ο προγραμματιστής δεν έχει τη δυνατότητα να διαχειριστεί τα κλειδιά, να ορίσει πολιτικές απομόνωσης, ή να παραμετροποιήσει τον τρόπο λειτουργίας του MPE. Η εμπιστοσύνη, λοιπόν, μεταφέρεται ολόκληρη στην Apple και στη σωστή λειτουργία του Enclave, κάτι που, αν και εξαιρετικά ασφαλές στην πράξη, δεν παύει να αποτελεί έναν ενδιαφέροντα συμβιβασμό μεταξύ ελέγχου (στον τελικό χρήστη) και ασφάλειας.

ARM-based Server SoCs (π.χ. AmpereOne)

Τα τελευταία χρόνια έχει παρατηρηθεί μια μετατόπιση από τις παραδοσιακές x86 αρχιτεκτονικές προς ARM (RISC) αρχιτεκτονικές, ιδιαίτερα στον χώρο των servers και cloud infrastructures (υποδομών νέφους). Το πλεονέκτημα; Απλός, αποδοτικός, ευέλικτος σχεδιασμός με δυνατότητα κάθετης ενσωμάτωσης (SoC/System on Chip, Σύστημα σε Τσιπ/μικρό ενσωματωμένο κύκλωμα τυπωμένο σε κομμάτι πυριτίου) και βελτιωμένα ενεργειακά χαρακτηριστικά, που τους καθιστούν ιδανικούς για μαζική παραγωγή.

Η υλοποίηση της κρυπτογράφησης μνήμης στα ARM-based Server SoCs, όπως το AmpereOne, βασίζεται σε έναν πλήρως ενσωματωμένο AES-XTS engine που λειτουργεί σε πραγματικό χρόνο εντός του chip. Η μονάδα αυτή κρυπτογραφεί κάθε γραμμή μνήμης (memory line) πριν αυτή αποθηκευτεί στη φυσική μνήμη και αποκρυπτογραφεί αντίστοιχα κατά την ανάγνωση. Τα κλειδιά κρυπτογράφησης παράγονται από έναν on-chip διαχειριστή κλειδιού κατά την εκκίνηση και φυλάσσονται αποκλειστικά εντός του SoC, προστατευμένα από μη εξουσιοδοτημένη πρόσβαση, ακόμα και σε περιπτώσεις φυσικής παρέμβασης (π.χ. cold boot attacks).

Επιπλέον, το SoC χρησιμοποιεί τμήματα (domains) επιβαλλόμενα από το υλικό όπου διαχωρίζουν τη μνήμη σε απομονωμένες περιοχές, κάθε μία με το δικό της κλειδί κρυπτογράφησης και πολιτικές πρόσβασης, εξασφαλίζοντας per(ανά)-VM isolation (απομόνωση ανά εικονική μηχανή). Αυτό επιτυγχάνεται μέσω συνεργασίας του ελεγκτή μνήμης, των ελεγκτών E/E και των πυρήνων της KME σε επίπεδο SoC, που επιβάλλουν πρόσβαση μόνο σε εξουσιοδοτημένα domains, χωρίς εξάρτηση από το λογισμικό ή τον hypervisor. Επίσης, όπως και στην λύση της Apple, η κρυπτογράφηση γίνεται σε επίπεδο γραμμής cache.

Ο σχεδιασμός αυτός επιφέρει πολύ χαμηλό χτύπημα στην απόδοση (~1-3%), λόγω της ενσωμάτωσης της κρυπτογράφησης σε όλο το data path (μονοπάτι δεδομένων), και βελτιστοποιεί ακόμα και την ενεργειακή κατανάλωση, παράγοντες κρίσιμους για χρήση σε data centers μεγάλης κλίμακας. Τέλος, παρά τη διαφάνεια και την αυτονομία του υλικού, η απουσία πολύπλοκων μηχανισμών ασφαλούς εκτέλεσης (π.χ. enclaves) περιορίζει τη λεπτομερή απομόνωση διεργασιών, κάτι που αποτελεί το επόμενο βήμα στην εξέλιξη της πλατφόρμας.

Όμως, όσο και αν η απόδοση και η κατανάλωση είναι σημαντικές, η ασφάλεια παραμένει θεμελιώδης, ειδικά όταν μιλάμε για πολυμισθωτικά (multi-tenant) περιβάλλοντα (μισθωμένες εικονικές μηχανές) όπως το cloud. Κι εδώ μπαίνει στο παιχνίδι η κατηγορία των ARM-based Server SoCs, όπως οι Ampere Altra/AmpereOne, που ενσωματώνουν κρυπτογράφηση μνήμης σε επίπεδο υλικού.

Αυτό που κάνει τις ARM server υλοποιήσεις να ξεχωρίζουν δεν είναι μόνο το ότι προσφέρουν transparent encryption, αλλά και ότι, λόγω του μοντέλου SoC, έχουν άμεσο έλεγχο σε όλο το data path (μονοπάτι δεδομένων): από τον πυρήνα της KME μέχρι το memory controller (ελεγκτή μνήμης) και τις I/O διεπαφές. Έτσι, η υλοποίηση της ασφάλειας δεν είναι απλώς μία προσθήκη πάνω από υπάρχοντα υλικά, αλλά ενσωματωμένη σχεδιαστικά σε όλα τα επίπεδα. Επιπλέον, επειδή η εστίαση γίνεται κυρίως στο cloud, οι ARM server SoCs έχουν αρχίσει να υποστηρίζουν και per-VM isolation (απομόνωση) μέσω διακριτών memory domains (τόμων στη μνήμη), κάτι που δεν απέχει πολύ από τις έννοιες που βλέπουμε στο SEV-SNP ή το MKTME, αν και δεν έχουν φτάσει ακόμα σε τόσο εξελιγμένο επίπεδο όσο της τεχνικής enclave.

Ωστόσο, αυτή η ασφάλεια παραμένει ομοιογενής και πιο μονοκόμματη: δεν έχουμε ακόμα (τουλάχιστον σύμφωνα με δημοσιευμένες έρευνες) απομόνωση ανά διεργασία ή enclave, ούτε κάποιο εξειδικευμένο secure execution (ασφαλής εκτέλεσης) περιβάλλον όπως το SGX ή το TrustZone που θα δούμε πιο κάτω. Εδώ ο στόχος είναι πιο πρακτικός: να μην αποκαλύπτεται ποτέ το περιεχόμενο της ΜΤΠ εκτός του chip, ανεξαρτήτως ποιος έχει πρόσβαση στο υλικό ή μέσω κώδικα.

Η προσέγγιση αυτή ευθυγραμμίζεται πλήρως με τις ανάγκες των hyperscalers (παρόχους υπηρεσιών στο νέφος όπως εικονικές μηχανές) (AWS, Google, Azure), που έχουν αρχίσει να επενδύουν σοβαρά σε ARM server instances (π.χ. AWS Graviton). Δεν τους ενδιαφέρει τόσο να τρέξουν κώδικα μέσα σε enclaves παρά όσο να εγγυηθούν για την ασφάλεια της απομονωμένης μνήμης ενός tenant (μισθωτή/επισκέπτη), χωρίς να βασίζονται στον hypervisor ή το λειτουργικό σύστημα. Άρα, η εστίαση των ARM server SoCs είναι περισσότερο στη διαφάνεια, την ενεργειακή απόδοση και την ελαχιστοποίηση του χώρου επίθεσης, παρά σε εξελιγμένους μηχανισμούς απομόνωσης.

Τελικά, παρόλο που ARM δεν έχει (προς το παρόν) κάποιο ανάλογο του SEV-SNP ή του TDX, η ενσωμάτωση υλικού επιπέδου κρυπτογράφησης μνήμης δείχνει ότι και αυτό το οικοσύστημα οδεύει προς μία μορφή εμπιστοσύνης, όπου το ίδιο το chip είναι υπεύθυνο για την προστασία των δεδομένων, ανεξαρτήτως του ποιος τρέχει το λειτουργικό και τι κώδικα τρέχει.

ARM TrustZone

Μια άλλη σημαντική, αν και διαφορετικής λογικής, τεχνολογία που αξίζει να αναφερθεί, είναι το ARM TrustZone. Σε αντίθεση με τις προηγούμενες λύσεις που εστιάζουν αποκλειστικά στην κρυπτογράφηση της μνήμης, η TrustZone αποτελεί μια πιο γενική αρχιτεκτονική απομόνωσης και λιγότερο μια λύση ειδικά σχεδιασμένη για την προστασία της ΜΤΠ. Ωστόσο, επειδή ο στόχος της είναι παρόμοιος, η προστασία ευαίσθητων δεδομένων από το υπόλοιπο σύστημα, μπορεί να παίξει ρόλο και σε αυτό το πεδίο.

Η βασική ιδέα πίσω από την TrustZone είναι ο διαχωρισμός του συστήματος σε δύο «κόσμους» (worlds): τον Secure World (Ασφαλή Κόσμο) και τον Normal World (Κανονικό Κόσμο). Το Secure World είναι ένα ξεχωριστό εκτελεστικό περιβάλλον με δικούς του πόρους, το οποίο έχει πρόσβαση στο σύνολο του hardware (περιλαμβανομένης της μνήμης), ενώ το Normal World δεν μπορεί να δει το Secure World ούτε να παρέμβει στη λειτουργία του. Ο διαχωρισμός αυτός υλοποιείται σε επίπεδο hardware, και πιο συγκεκριμένα χρησιμοποιείται το NS (Non-Secure) bit για τον έλεγχο πρόσβασης, με την υποστήριξη από KME, MMU (Memory Management Unit/Μονάδα Διαχείρισης Μνήμης), cache και διαύλους επικοινωνίας.

Αυτό πρακτικά σημαίνει ότι μπορούμε να αποθηκεύσουμε δεδομένα (ή και ολόκληρες εφαρμογές) στο Secure World, και να τα απομονώσουμε πλήρως από το λειτουργικό σύστημα, τον hypervisor και οποιοδήποτε άλλο στοιχείο τρέχει στο Normal World. Εφ’ όσον γίνει σωστά η παραμετροποίηση, ακόμα και αν το λειτουργικό σύστημα στο Normal World παραβιαστεί, τα δεδομένα στον Secure World παραμένουν άθικτα.

Πιο τεχνικά μιλώντας, η υλοποίηση της TrustZone βασίζεται σε ένα hardware-enforced (επιβαλλόμενο από το υλικό) περιβάλλον διαχωρισμού, το οποίο επεκτείνεται στον memory controller και στους διαύλους επικοινωνίας (bus). Συγκεκριμένα, η MMU και το system fabric υποστηρίζουν τον χαρακτηρισμό συγκεκριμένων περιοχών μνήμης ως «Secure» ή «Non-secure». Όταν μία πρόσβαση προέρχεται από το Secure World, επιτρέπεται η ανάγνωση ή εγγραφή στα secure regions (ασφαλή σημεία), ενώ όλες οι άλλες προσβάσεις απορρίπτονται ή αγνοούνται από το υλικό. Επίσης, η εναλλαγή μεταξύ αυτών γίνεται μέσω της εντολής SMC (Secure Monitor Call), που ενεργοποιεί έναν ειδικό μηχανισμό του firmware (υλικολογισμικού), τον Secure Monitor, υπεύθυνο για τον έλεγχο πρόσβασης και το context switching μεταξύ των δύο κόσμων.

Ωστόσο, υπάρχει μια βασική διαφορά: η TrustZone δεν κρυπτογραφεί εξ’ ορισμού τη μνήμη, ούτε προσφέρει προστασία έναντι επιθέσεων σε φυσικό επίπεδο (π.χ. cold boot ή DMA). Η απομόνωση της μνήμης βασίζεται στην πρόσβαση μέσω του bus, δηλαδή, αν ένα memory region (Πεδίο της Μνήμης) χαρακτηριστεί ως «Secure», μόνο οι διεργασίες στον Secure World μπορούν να το προσπελάσουν. Αυτός ο διαχωρισμός ελέγχεται, μεταξύ άλλων, και από τον TZASC (TrustZone Address Space Controller), που επιτρέπει την υλοποίηση ασφαλών περιοχών μνήμης σε επίπεδο ελεγκτή μνήμης. Όμως το περιεχόμενο της μνήμης σε αυτό το secure region παραμένει σε ωμή μορφή, εκτός αν η ίδια η εφαρμογή (ή κάποιο trusted firmware) την κρυπτογραφήσει χειροκίνητα.

Άρα, η TrustZone από μόνη της δεν αποτελεί λύση για κρυπτογράφηση της ΜΤΠ, αλλά μπορεί να συνδυαστεί με άλλες τεχνικές για την επίτευξη του στόχου. Χρησιμοποιείται ευρέως σε κινητά και embedded (ενσωματωμένα) συστήματα για secure boot (ασφαλής εκκίνηση), credential storage (αποθήκευση διαπιστευτηρίων), και λειτουργίες τύπου DRM (Digital Rights Management/Διαχείριση Ψηφιακών Δικαιωμάτων). Η Apple, για παράδειγμα, έχει υλοποιήσει όπως είδαμε πιο πάνω, ένα ανάλογο μηχανισμό, το Secure Enclave, ενώ

άλλοι ARM κατασκευαστές χτίζουν πάνω στην TrustZone πιο εξελιγμένες τεχνικές προστασίας της μνήμης.

Αν θέλαμε να συνοψίσουμε, θα λέγαμε ότι το TrustZone παρέχει απομόνωση αλλά όχι κρυπτογράφηση των δεδομένων, και σε αυτό διαφέρει από τις πιο πρόσφατες τεχνολογίες τύπου SEV, TDX ή ακόμα και Sealer. Όμως ο ρόλος του στη διαμόρφωση του secure execution model (μοντέλο ασφαλούς εκτέλεσης), ειδικά στον ARM κόσμο, είναι καθοριστικός.

Sealer

Μία πρωτοπόρα προσέγγιση στην κρυπτογράφηση μνήμης με έμφαση στην απόδοση αποτελεί το Sealer, μια ερευνητική πρόταση. Σε αντίθεση με τις προηγούμενες τεχνολογίες που τοποθετούν την κρυπτογραφική μονάδα ως ξεχωριστό στοιχείο στο μονοπάτι δεδομένων, η βασική καινοτομία του Sealer έγκειται στην ριζική ενσωμάτωση της AES κρυπτογράφησης απευθείας μέσα στα SRAM (στατική ΜΤΠ) subarrays, εκμεταλλευόμενο τη μαζική παραλληλία και τις υπολογιστικές δυνατότητες των bitlines.

Η κεντρική ιδέα πίσω από το Sealer είναι η αντιμετώπιση ενός θεμελιώδους προβλήματος που αντιμετωπίζουν οι παραδοσιακές λύσεις κρυπτογράφησης μνήμης: το σημαντικό χτύπημα στην απόδοση που προκαλεί η κρυπτογράφηση/αποκρυπτογράφηση όταν βρίσκεται στο κρίσιμο μονοπάτι (critical path) του συστήματος. Αντί να τοποθετείται ως ξεχωριστή μονάδα που επεξεργάζεται τα δεδομένα πριν ή μετά την αποθήκευση, η κρυπτογράφηση ενσωματώνεται απευθείας στην αρχιτεκτονική της SRAM, επιτρέποντας την ταυτόχρονη εκτέλεση των υπολογισμών AES με τις συνήθεις λειτουργίες ανάγνωσης και εγγραφής.

Σε τεχνικό επίπεδο, το Sealer εκμεταλλεύεται τη φυσική δομή των SRAM subarrays, όπου κάθε subarray αποτελείται από χιλιάδες κελιά μνήμης που μπορούν να προσπελαστούν παράλληλα μέσω των bitlines. Η προτεινόμενη αρχιτεκτονική επεκτείνει το περιφερειακό κύκλωμα (peripheral circuitry) της SRAM με ελάχιστες τροποποιήσεις, προσθέτοντας AES υπολογιστικές μονάδες που λειτουργούν απευθείας πάνω στις bitlines. Αυτό επιτρέπει τη διεξαγωγή των AES rounds (γύροι κρυπτογράφησης) παράλληλα με την πρόσβαση στα δεδομένα, μετατρέποντας ουσιαστικά κάθε SRAM subarray σε μια κρυπτογραφική μονάδα εντός του επεξεργαστή.

Το κλειδί στην επιτυχία του Sealer είναι η εκμετάλλευση της εγγενούς παραλληλίας των SRAM δομών. Ενώ οι παραδοσιακοί κρυπτογραφικοί επιταχυντές (μονάδες κρυπτογράφησης στο υλικό) επεξεργάζονται τα δεδομένα σειριακά, το Sealer μπορεί να εκτελέσει πολλαπλές AES λειτουργίες ταυτόχρονα σε διαφορετικά τμήματα των δεδομένων, χρησιμοποιώντας τη φυσική παραλληλία των bitlines. Αυτό οδηγεί σε δραματική βελτίωση της απόδοσης: η έρευνα αναφέρει έως και δύο τάξεις μεγέθους βελτίωση στο throughput-per-area σε σύγκριση με προηγούμενες λύσεις, ενώ παράλληλα προσφέρει τριπλάσια μείωση της ενεργειακής κατανάλωσης.

Η λειτουργία του Sealer βασίζεται στην κρυπτογράφηση των δεδομένων πριν αυτά αποσταλούν εκτός του chip (off-chip) και την αποκρυπτογράφησή τους κατά την επιστροφή τους από τη φυσική μνήμη. Αυτή η προσέγγιση εξασφαλίζει ότι τα δεδομένα που κυκλοφορούν στο memory bus και αποθηκεύονται στη φυσική μνήμη (DRAM) είναι πάντα κρυπτογραφημένα, παρέχοντας προστασία έναντι επιθέσεων που στοχεύουν στο memory bus ή στη φυσική μνήμη (π.χ. cold boot attacks, DMA attacks). Τα κρυπτογραφικά κλειδιά παράγονται και διαχειρίζονται εντός του chip, χωρίς να είναι ποτέ εξαγώγιμα ή ορατά από εξωτερικές διεργασίες.

Αυτό που κάνει την προσέγγιση του Sealer ιδιαίτερα αξιοπρόσεκτη είναι η ριζική αλλαγή στον τρόπο σκέψης σχετικά με την κρυπτογράφηση μνήμης. Αντί να θεωρείται ως επιπλέον επίπεδο που επιβαρύνει την απόδοση, η κρυπτογράφηση μετατρέπεται σε μία εγγενή ιδιότητα του υλικού μνήμης, εφαρμόζοντας την αρχή "security by design" σε βαθύτερο επίπεδο από οποιαδήποτε άλλη σύγχρονη τεχνολογία.

Παρόλα αυτά, το Sealer παραμένει στο στάδιο της ερευνητικής πρότασης και δεν έχει υιοθετηθεί εμπορικά. Οι προκλήσεις υλοποίησης περιλαμβάνουν την πολυπλοκότητα

ενσωμάτωσης των AES υπολογισμών στα υπάρχοντα SRAM designs, την ανάγκη για σημαντικές τροποποιήσεις στην αρχιτεκτονική των επεξεργαστών, και τα ζητήματα συμβατότητας με τα υπάρχοντα μνημονικά υποσυστήματα. Επιπλέον, η προσέγγιση αυτή εστιάζει κυρίως στην εμπιστευτικότητα των δεδομένων χωρίς να παρέχει τους εξελιγμένους μηχανισμούς απομόνωσης που προσφέρουν τεχνολογίες όπως το SEV-SNP ή το TDX, καθιστώντας την κατάλληλη περισσότερο για single-tenant περιβάλλοντα ή εφαρμογές όπου η βέλτιστη απόδοση έχει προτεραιότητα έναντι της λεπτομερούς πολιτικής ασφάλειας.

Εν κατακλείδι, το Sealer αποτελεί παράδειγμα του τρόπου με τον οποίο η ασφάλεια μπορεί να ενσωματωθεί στον σχεδιασμό του υλικού από τη βάση, αντί να προστίθεται ως επιπλέον επίπεδο. Η προσέγγιση αυτή θα μπορούσε να επηρεάσει τον μελλοντικό σχεδιασμό τόσο των μνημονικών υποσυστημάτων όσο και των κρυπτογραφικών λύσεων, προσφέροντας έναν πιο αποδοτικό τρόπο για την επίτευξη εμπιστευτικότητας δεδομένων χωρίς τις παραδοσιακές απώλειες απόδοσης.

*Μηχανισμός DMA (Direct Memory Access/Άμεση Πρόσβαση Μνήμης)

Ο μηχανισμός DMA (Direct Memory Access/Άμεση Πρόσβαση Μνήμης) αποτελεί θεμελιώδες υποσύστημα στις σύγχρονες αρχιτεκτονικές υπολογιστών, επιτρέποντας σε εξωτερικές συσκευές (όπως κάρτες δικτύου, ελεγκτές αποθήκευσης, GPU, περιφερειακές συσκευές) να διαβάζουν ή να γράφουν δεδομένα απευθείας στη φυσική μνήμη, χωρίς τη μεσολάβηση της κεντρικής μονάδας επεξεργασίας (ΚΜΕ).

Κατά την κανονική λειτουργία, κάθε μεταφορά δεδομένων μεταξύ μνήμης και συσκευών εισόδου/εξόδου (I/O) περνά μέσω της ΚΜΕ, η οποία: Διαβάζει δεδομένα από μία συσκευή (π.χ. δίσκο), τα αποθηκεύει προσωρινά σε καταχωρητές και τελικά τα γράφει στη μνήμη.

Αντίθετα, με τη χρήση του DMA, μία συσκευή ή ένας ειδικός ελεγκτής (DMA controller) μπορεί να ξεκινήσει μια μεταφορά απευθείας, γνωστοποιώντας τη διεύθυνση προορισμού και το μέγεθος μεταφοράς, ενώ η ΚΜΕ συνεχίζει να εκτελεί άλλες εργασίες (asynchronous data movement/Ασύγχρονη Μεταφορά Δεδομένων). Όταν ολοκληρωθεί η μεταφορά, η συσκευή ενεργοποιεί μία διακοπή (interrupt) για να ενημερώσει το λειτουργικό σύστημα.

Αυτή η προσέγγιση μειώνει τη χρονική καθυστέρηση, απελευθερώνει κύκλους επεξεργασίας για τη ΚΜΕ, και βελτιώνει τη συνολική απόδοση του συστήματος, ιδιαίτερα σε συστήματα υψηλής μεταφοράς δεδομένων.

Η χρήση DMA, αν και αποδοτική, εισάγει κρίσιμες προκλήσεις ασφάλειας, ειδικά όταν χρησιμοποιείται από συσκευές με φυσική πρόσβαση ή μη επαρκώς απομονωμένες. Ο ελεγκτής DMA λειτουργεί σε επίπεδο υλικού και μπορεί να προσπελάσει οποιαδήποτε φυσική διεύθυνση, παρακάμπτοντας τους μηχανισμούς απομόνωσης που εφαρμόζονται από το λειτουργικό σύστημα ή την MMU.

Αυτό καθιστά δυνατές επιθέσεις κατά τις οποίες ένας επιτιθέμενος χρησιμοποιεί κακόβουλη ή παραβιασμένη συσκευή για να διαβάσει ή να τροποποιήσει ευαίσθητα δεδομένα στη μνήμη του συστήματος, ακόμα και αν είναι κλειδωμένο ή σε κατάσταση αδράνειας.

Για την προστασία από επιθέσεις μέσω DMA, έχουν αναπτυχθεί εξειδικευμένοι μηχανισμοί όπως:
IOMMU (Input-Output Memory Management Unit/Μονάδα Διαχείρισης Ε/Ε Μνήμης): Πρόκειται για μία επέκταση της MMU που εφαρμόζει μεταφράσεις και περιορισμούς διευθύνσεων στις συσκευές DMA. Επιτρέπει στο λειτουργικό σύστημα να περιορίσει ποιες περιοχές μνήμης μπορεί να προσπελάσει κάθε συσκευή.
DMA remapping και isolation πολιτικές: Συστήματα όπως το Intel VT-d και AMD-Vi παρέχουν προστασία για την απόρριψη πρόσβασης σε προστατευμένα τμήματα της μνήμης.
Προστασία στο επίπεδο του υλικολογισμικού: Ορισμένα BIOS/UEFI περιλαμβάνουν ρυθμίσεις για απενεργοποίηση DMA access από συγκεκριμένες θύρες ή ενεργοποίηση pre-boot IOMMU.
Μέτρα Ασφαλείας του σειριακού πρωτοκόλλου Thunderbolt: Προβλέπουν πρόσβαση μόνο με χρήση διαπιστευτηρίων για περιφερειακά που επιχειρούν DMA μεταφορές.

Η προστασία του μηχανισμού DMA είναι κρίσιμη, ειδικά σε περιβάλλοντα όπου αξιολογείται το Trusted Computing Base (TCB) ή σε σενάρια απομόνωσης μεταξύ εικονικών μηχανών, καθώς μη ελεγχόμενες DMA προσβάσεις μπορούν να καταρρίψουν πλήρως τις εγγυήσεις απομόνωσης.

Συμπερασματικά

Μετά από την ανάλυση των επιμέρους τεχνολογιών, γίνεται φανερό ότι υπάρχουν δύο βασικές προσεγγίσεις στο πρόβλημα της προστασίας της μνήμης: η ολική κρυπτογράφηση της ΜΤΠ και η επιλεκτική απομόνωση μέσω enclaves ή προστατευμένων execution domains. Όλες οι τεχνολογίες κινούνται πάνω σε αυτόν τον άξονα, αλλά υλοποιούν διαφορετικές ισορροπίες μεταξύ ασφάλειας, απόδοσης και πολυπλοκότητας υλοποίησης.

Η Intel με το TME και η AMD το SME εφαρμόζουν πλήρη κρυπτογράφηση της μνήμης χωρίς ιδιαίτερη λογική απομόνωσης. Η μνήμη κρυπτογραφείται πλήρως στη φυσική της μορφή, και η αποκρυπτογράφηση γίνεται εντός του ελεγκτή της μνήμης. Πρόκειται για λύσεις που προστατεύουν από επιθέσεις φυσικής πρόσβασης (cold boot, DMA) αλλά δεν προστατεύουν έναντι επιθέσεων από τον hypervisor ή το λειτουργικό σύστημα, αφού τα δεδομένα είναι απλώς κρυπτογραφημένα στο υλικό, και όχι ακατανόητα από το ίδιο το λογισμικό που τρέχει.

Από την άλλη, λύσεις όπως το AMD SEV-SNP και το Intel TDX ανεβάζουν τον πήχη, καθώς συνδυάζουν κρυπτογράφηση μνήμης με πλήρη απομόνωση σε επίπεδο εικονικής μηχανής/ΕΜ. Η μνήμη κάθε ΕΜ (ή TD στην περίπτωση του TDX) είναι κρυπτογραφημένη με ξεχωριστό κλειδί, και ο hypervisor δεν έχει πρόσβαση. Επιπλέον, ενσωματώνουν μηχανισμούς απομόνωσης και επικύρωσης (ταυτοποίησης), με στόχο το την ασφάλεια διακομιστών: να μπορείς να τρέχεις μη έμπιστο OS ή κώδικα, σε έμπιστο υλικό, χωρίς δηλαδή να αποκαλύπτεις τα δεδομένα του. Ουσιαστικά, μιλάμε για τεχνολογίες όπου το hypervisor γίνεται απλός διαχειριστής πόρων, αλλά χωρίς δυνατότητα πρόσβασης στο περιεχόμενο της μνήμης του ΕΜ/VM.

Από εκεί και πέρα, υπάρχουν και άλλες υλοποιήσεις, όπως το Apple MPE και το IBM POWER10 TME, που στοχεύουν σε transparency (διαφάνεια): η κρυπτογράφηση γίνεται χωρίς να αλλάζει τίποτα στον τρόπο που προγραμματίζει ο προγραμματιστής. Δεν υπάρχει enclave, ούτε διεπαφές για το λογισμικό, απλά ό,τι γράφεται στη μνήμη, περνά από διαδικασία κρυπτογράφηση στο ίδιο το υλικό. Παρόμοια και οι ARM-based server SoCs (όπως το AmpereOne), ενσωματώνουν κρυπτογράφηση της ΜΤΠ για απομόνωση, ειδικά για εφαρμογές στο νέφος, εστιάζοντας στην απόδοση και χωρίς μεγάλη πολυπλοκότητα.

Εκεί που τα πράγματα γίνονται πιο ενδιαφέροντα είναι στις ερευνητικές προσεγγίσεις. Τεχνολογίες όπως το Sealer και το TME-Box προσπαθούν να δώσουν πιο ευέλικτες λύσεις. Το Sealer, για παράδειγμα, ενσωματώνει την κρυπτογράφηση AES απευθείας στην SRAM, επιτυγχάνοντας υψηλή απόδοση χωρίς εξάρτηση από το λειτουργικό σύστημα. Παρότι προστατεύει τα δεδομένα εκτός chip, δεν παρέχει απομόνωση ή ταυτοποίηση όπως άλλες τεχνολογίες τύπου enclave. Ενώ το TME-Box προτείνει την ιδέα των «secure boxes» μέσα στη ΜΤΠ (κάτι που θυμίζει αρκετά τις Enclaves τύπου Intel SGX), που κρυπτογραφούνται και αποκρυπτογραφούνται όταν χρειάζεται και ανάλογα με το συγκείμενο πλαίσιο που χρησιμοποιούνται.

Τέλος, το TrustZone της ARM λειτουργεί διαφορετικά: δεν κρυπτογραφεί τα δεδομένα στην ΜΤΠ, αλλά παρέχει έναν ξεχωριστό secure world όπου η πρόσβαση στη μνήμη ελέγχεται σε επίπεδο διαύλων επικοινωνίας. Παρότι δεν προστατεύει τα δεδομένα αν κάποιος αποκτήσει φυσική πρόσβαση, προστατεύει από λογισμικού/κώδικα επιπέδου επιθέσεις και είναι ιδανική για ενσωματωμένες εφαρμογές όπου χρειάζεται απλή απομόνωση χωρίς το κόστος ή την πολυπλοκότητα ενός enclave.

Συμπεραίνουμε λοιπόν, ότι η επιλογή για το ποια τεχνολογία θα χρησιμοποιήσουμε εξαρτάται από το μοντέλο απειλής όπου ακολουθείται/χρειάζεται ο χρήστης: Αν μας ενδιαφέρει να προστατευτούμε από φυσικές επιθέσεις απευθείας στο υλικό, τότε αρκεί μια απλή λύση όπως

το TME. Παρατηρούμε επίσης ότι οι περισσότερες τεχνολογίες χρησιμοποιούν τον αλγόριθμο κρυπτογράφησης AES (Advanced Encryption Standard). Επίσης, πολλές από αυτές δημιουργήθηκαν υπό την άποψη ότι οι Η/Υ χρησιμοποιούνται για να ελέγχουν εικονικές μηχανές (υπερεπόπτες/hypervisor) και θέλουν να προστατέψουν και να απομονώσουν τα δεδομένα κάθε μίας. Αν όμως θέλουμε να μην εμπιστευόμαστε ούτε το λειτουργικό σύστημα ή τον hypervisor, τότε η λύση πρέπει να ενσωματώνει απομόνωση, ακεραιότητα, και μηχανισμούς ελέγχου ταυτότητας (π.χ. απομακρυσμένη επικύρωση/remote attestation). Το κόστος, βέβαια, ανεβαίνει αντίστοιχα: τόσο σε χρόνο απόκρισης όσο και σε πολυπλοκότητα, και άρα, επιπτώσεις στην απόδοση, σε κόστος και στην ίδια την έρευνα της σχεδίασης/ενσωμάτωσης των μηχανισμών αυτών σε κάποιο σύστημα. Τέλος , με την ανάπτυξη των κβαντικών υπολογιστών και τον πιθανό σύντομο διαμοιρασμό αυτής της τεχνολογίας στην κοινωνία, είναι αρκετή η κρυπτογράφηση που εφαρμόζουμε; Τί γίνεται με επιθέσεις, όπως την κλασσική, Store Now, Decrypt Later (SNDL)* που απειλεί η κακόβουλη χρήση αυτών των νέων υπολογιστών;

Αναφορές στις προαναφερθείσες επιθέσεις

*(Σύνοψη) Spectre και Meltdown επιθέσεις

Οι επιθέσεις Meltdown και Spectre εκμεταλλεύονται μηχανισμούς βελτιστοποίησης των σύγχρονων επεξεργαστών, όπως η εκτέλεση εκτός σειράς (out of order execution) και εκ των προτέρων εκτέλεση (speculative execution), για να παρακάμψουν τους μηχανισμούς απομόνωσης στη μνήμη. Η Meltdown επιτρέπει σε έναν χρήστη, που δεν φέρει τα κατάλληλα δικαιώματα, να αποκτήσει πρόσβαση σε προστατευμένα τμήματα μνήμης του πυρήνα (kernel) παρακάμπτοντας τον διαχωρισμό χρήστη-λειτουργικού συστήματος. Αντίστοιχα, η Spectre χειραγωγεί τον μηχανισμό πρόβλεψης διακλάδωσης (branch prediction) ώστε ο επεξεργαστής να εκτελέσει εντολές speculatively και να αποκαλύψει ευαίσθητα δεδομένα που βρίσκονται στις κρυφές μνήμες (cache).

Spectre

Η επίθεση Spectre, ανήκει στην ευρύτερη κατηγορία των side-channel (πλευρικού καναλιού) επιθέσεων και αξιοποιεί τις μικροαρχιτεκτονικές βελτιστοποιήσεις των σύγχρονων επεξεργαστών, ειδικά τον μηχανισμό προβλεπτικής εκτέλεσης (speculative execution), για να διαρρεύσει δεδομένα τα οποία δεν θα έπρεπε να είναι προσβάσιμα από τον εκτελούμενο κώδικα. Η Spectre εκμεταλλεύεται το γεγονός ότι, παρόλο που τα αποτελέσματα της προβλεπτικής εκτέλεσης εντολών μπορεί να ακυρωθούν, οι παρενέργειες στους μηχανισμούς μνήμης (ιδίως στην cache) παραμένουν και μπορούν να μετρηθούν.

Η βασική ιδέα πίσω από την επίθεση είναι η εξής: ο επιτιθέμενος προκαλεί την ΚΜΕ να εκτελέσει speculatively μια ακολουθία εντολών που περιλαμβάνει πρόσβαση σε μνήμη εκτός επιτρεπτών ορίων (π.χ., παράκαμψη του ελέγχου των ορίων σε έναν πίνακα). Κατά την εκτέλεση αυτή, ο επεξεργαστής ενδέχεται να φορτώσει στη cache δεδομένα που υπό κανονικές συνθήκες δεν θα έπρεπε να είναι προσβάσιμα. Στη συνέχεια, ο επιτιθέμενος, μετρώντας τον χρόνο πρόσβασης σε διάφορα τοποθεσίες μνήμες, μπορεί να προσδιορίσει ποια δεδομένα είχαν φορτωθεί, και επομένως να ανακτήσει το περιεχόμενό τους, χωρίς ποτέ να τα έχει προσπελάσει.

Ένα χαρακτηριστικό παράδειγμα βασίζεται στον εξής κώδικα με τη χρήση πινάκων:

if (x<array1_size) {temp=array2[array1[x]*512];}

Η υπόθεση ότι το x είναι εντός των ορίων χρησιμοποιείται από την μονάδα branch prediction για να εκτελέσει speculatively τον κώδικα πρόσβασης στο array2. Αν το array1[x] αναφέρεται σε μη εξουσιοδοτημένη θέση μνήμης, το αποτέλεσμα της πρόσβασης μπορεί να επηρεάσει τη συμπεριφορά της cache, ακόμα και αν το αποτέλεσμα αυτής της πρόσβασης απορριφθεί στη συνέχεια. Μετρώντας τον χρόνο πρόσβασης στην array2, ο επιτιθέμενος μπορεί να ανιχνεύσει ποιο offset ήταν ενεργό και έτσι να ανακατασκευάσει την τιμή array1[x].

Η Spectre δεν είναι μία μόνο επίθεση, αλλά ένα σύνολο παραλλαγών που αξιοποιούν διαφορετικές οδούς πρόβλεψης και speculatively εκτελούμενων paths. Οι δύο κυριότερες εκδοχές που έχουν εντοπιστεί είναι:
Spectre Variant 1 (Bounds Check Bypass): Κατάχρηση της branch prediction για να παρακαμφθούν έλεγχοι ορίων σε πίνακες.
Spectre Variant 2 (Branch Target Injection): Poisoning (Δηλητηρίαση) του Branch Target Buffer (BTB) για εκτέλεση κώδικα σε απροσδόκητα σημεία.

Η αντιμετώπιση της Spectre είναι ιδιαίτερα σύνθετη, καθώς το πρόβλημα βρίσκεται στις θεμελιώδεις βελτιστοποιήσεις απόδοσης του hardware. Ενδεικτικά, έχουν προταθεί και υλοποιηθεί τα εξής αντίμετρα:
Retpolines (return trampolines): τεχνική περιορισμού των έμμεσων jumps (άλμα σε θέση μνήμης) σε branch prediction.
Memory fencing (π.χ., LFENCE): οδηγίες που εμποδίζουν την πρόωρη εκτέλεση εντολών.
Αναβαθμίσεις firmware (microcode patches): που εισάγουν speculative execution constraints.
Ανιχνευτές στον μεταγλωττιστή (Spectre-aware optimizations): που τροποποιούν τον κώδικα ώστε να αποφεύγει ευάλωτα μοτίβα κώδικα.

Ωστόσο, καθώς οι επιθέσεις τύπου Spectre βασίζονται σε θεμελιώδη χαρακτηριστικά της out-of-order και speculative εκτέλεσης, η πλήρης εξάλειψή τους χωρίς σημαντική απώλεια απόδοσης αποτελεί μια ενεργή πρόκληση. Πλέον, επιθέσεις τύπου Spectre επεκτείνονται και σε νέες παραλλαγές (π.χ. Spectre-RSB, Spectre-STL), καθιστώντας αναγκαία την επανεξέταση του μοντέλου εμπιστοσύνης στο επίπεδο αρχιτεκτονικής.

Meltdown

Η επίθεση Meltdown αποτελεί ένα χαρακτηριστικό παράδειγμα επίθεσης πλευρικού καναλιού μικροαρχιτεκτονικής που εκμεταλλεύεται την out-of-order execution και την ανεπαρκή απομόνωση μεταξύ user-space και kernel-space μνήμης. Αντίθετα με τη Spectre, η οποία βασίζεται κυρίως στη χειραγώγηση προβλεπτικής εκτέλεσης μέσω branch prediction, η Meltdown επιτρέπει την άμεση ανάγνωση προστατευμένων περιοχών μνήμης, όπως της μνήμης του πυρήνα, από μη προνομιούχο (unprivileged) κώδικα χρήστη.

Οι σύγχρονοι επεξεργαστές χρησιμοποιούν out-of-order execution για να βελτιώσουν την απόδοση. Κατά την εκτέλεση μιας εντολής που θα έπρεπε κανονικά να προκαλέσει σφάλμα (π.χ., πρόσβαση στη μνήμη του kernel από user-space κώδικα), η ΚΜΕ μπορεί να εκτελέσει προσωρινά επόμενες εντολές, ακόμα και αν η εκτέλεση αυτή ακυρωθεί στη συνέχεια. Παρόλο που τα αποτελέσματα αυτών των εντολών δεν καταλήγουν σε αρχιτεκτονική κατάσταση (π.χ., δεν αλλάζουν καταχωρητές), οι παρενέργειες στη cache παραμένουν και μπορούν να παρατηρηθούν από τον επιτιθέμενο.

Η επίθεση Meltdown εκτελείται τυπικά ως εξής:
Ένα πρόγραμμα χρήστη προσπαθεί να διαβάσει μια θέση μνήμης που ανήκει στον kernel (π.χ. mov al,byte [kernel_addr]). Η εντολή αυτή προκαλεί page fault (σφάλμα σελίδας). Πριν η CPU σταματήσει την εκτέλεση λόγω του σφάλματος, μπορεί να προχωρήσει σε επόμενες εντολές out-of-order, για παράδειγμα, χρησιμοποιώντας την τιμή που διάβασε από τον kernel για να προσπελάσει ένα συγκεκριμένο offset σε μια cache-friendly δομή (π.χ. mov rbx,[user_array+al*4096]). Μετά την αναίρεση των εντολών, η ΚΜΕ αποκαθιστά την αρχική της κατάσταση. Ωστόσο, η πρόσβαση στη χώρου του user (όπως σε έναν πίνακα) έχει φορτώσει δεδομένα στα συγκεκριμένα cache lines, ανάλογα με την τιμή που διέρρευσε. Ο επιτιθέμενος μπορεί στη συνέχεια να μετρήσει τον χρόνο πρόσβασης στην user_array για να εντοπίσει ποιο στοιχείο βρίσκεται στη cache, και επομένως να ανακτήσει το byte της προστατευμένης μνήμης.

Η επιτυχία της επίθεσης εξαρτάται από την παράλληλη εκτέλεση μεταξύ του σφάλματος πρόσβασης και των side-channel ενεργειών. Η χρήση τεχνικών όπως exception suppression/καταστολή εξαιρέσεων (π.χ., μέσω TSX ή signal handlers) επιτρέπει στον επιτιθέμενο να παρακάμψει ή να χειριστεί το page fault χωρίς να καταρρεύσει η εφαρμογή.

Η Meltdown επιτρέπει την ανάγνωση αυθαίρετης φυσικής μνήμης που είναι χαρτογραφημένη στο context της διεργασίας, συνήθως περιλαμβάνει όλο το kernel space, οδηγώντας σε σοβαρή παραβίαση της εμπιστευτικότητας. Οι επηρεαζόμενες αρχιτεκτονικές περιλαμβάνουν τις περισσότερες εκδόσεις επεξεργαστών Intel, καθώς και ορισμένους ARM (π.χ. Cortex-A75).

Για την αντιμετώπιση του προβλήματος, προτάθηκαν και υλοποιήθηκαν τα εξής μέτρα:
KPTI (Kernel Page-Table Isolation): διαχωρίζει πλήρως τους πίνακες σελίδων μεταξύ user και kernel mode, αποτρέποντας την παρουσία του kernel address space όταν εκτελείται μη προνομιούχος κώδικας.
Hardware-based patches: Νεότεροι επεξεργαστές περιλαμβάνουν τροποποιήσεις στη speculative/memory pipeline/σωλήνωση για να αποτρέπουν παρόμοιες επιθέσεις.
Αντιμετώπιση μέσω λειτουργικού συστήματος: όπως το UDEREF στο HardenedBSD ή το KAISER patch στο Linux.

Σε αντίθεση με τη Spectre, η Meltdown είναι πιο εύκολα εκμεταλλεύσιμη και επιτρέπει άμεση ανάγνωση προστατευμένων δεδομένων, καθιστώντας την ιδιαίτερα επικίνδυνη σε περιβάλλοντα πολλαπλών χρηστών ή cloud υποδομές.

*Row Hammer

Η επίθεση Rowhammer αποτελεί μια χαμηλού επιπέδου παραβίαση της μνήμης (hardware-based fault attack/επίθεση σφάλματος βασισμένη στο υλικό), που στοχεύει στη φυσική αρχιτεκτονική των μονάδων DRAM (Dynamic Random Access Memory)/ΜΤΠ. Η καινοτομία της έγκειται στο ότι δεν εκμεταλλεύεται κάποιο bug στο λογισμικό ή την αρχιτεκτονική των ΚΜΕ, αλλά βασίζεται σε φυσικά φαινόμενα που προκύπτουν από τη διαρροή ηλεκτρικού φορτίου μεταξύ γραμμών μνήμης. Το αποτέλεσμα είναι η ανατροπή bit (bit flip) σε γειτονικές θέσεις μνήμης, χωρίς άμεση πρόσβαση σε αυτές.

Η DRAM/ΜΤΠ είναι οργανωμένη σε γραμμές (rows) και στήλες (columns), όπου κάθε κελί μνήμης (bit) αποθηκεύεται ως ηλεκτρικό φορτίο σε έναν πυκνωτή. Λόγω των φυσικών περιορισμών και της πυκνότητας ενσωμάτωσης, η πρόσβαση σε μία σειρά κελιών (row) μπορεί να επηρεάσει τα φυσικά γειτονικά rows. Συγκεκριμένα, η συνεχής και γρήγορη ενεργοποίηση ενός συγκεκριμένου row (γνωστή ως "hammering") προκαλεί διαρροές φορτίου στα adjacent rows, οδηγώντας σε μη εξουσιοδοτημένες αλλαγές σε bit, ακόμα και χωρίς καμία άμεση εγγραφή σε αυτά.

Η επίθεση εκτελείται συνήθως ως εξής: Ο επιτιθέμενος προσδιορίζει τη φυσική τοποθεσία στη μνήμη όπου βρίσκεται μια ευαίσθητη πληροφορία ή δομή (π.χ., page tables, flags/σημαίες ελέγχου, pointers/δείκτες).
Στη συνέχεια εντοπίζει δύο rows που γειτονεύουν φυσικά με τον στόχο και τα ενεργοποιεί επανειλημμένα και ταχύτατα, χρησιμοποιώντας ειδικές εντολές (π.χ., clflush, mov, mfence) ώστε να παρακάμψει την cache και να αναγκάσει προσβάσεις στη φυσική DRAM.
Μετά από χιλιάδες ή εκατομμύρια επαναλήψεις, προκαλείται bit flip στο ενδιάμεσο row, αλλάζοντας τιμές χωρίς να έχει γίνει write σε αυτό.

Αξιοσημείωτο είναι ότι η επίθεση είναι non-invasive/διακριτική: δεν απαιτεί προνομιούχα πρόσβαση, kernel exploits ή DMA. Έχουν καταγραφεί επιτυχημένες επιθέσεις Rowhammer μέσω JavaScript (σε browser), σε smartphones, ακόμα και μέσω WebAssembly.

Η Rowhammer παρακάμπτει το θεμελιώδες δόγμα της λογικής απομόνωσης στη μνήμη, επιτρέποντας σε μία εφαρμογή να επηρεάσει δεδομένα άλλης, ακόμη κι αν οι περιοχές είναι πλήρως προστατευμένες μέσω της MMU. Μερικά παραδείγματα επιθέσεων που αξιοποιούν την Rowhammer είναι:
Privilege escalation/Κλιμάκωση Δικαιωμάτων: μέσω αναστροφής bits σε πίνακες σελίδων, μπορεί να αποκτηθεί πρόσβαση root/administrator/διαχειριστή.
Bypassing sandboxing/παράκαμψη ασφαλούς περιβάλλοντος: σε περιβάλλοντα όπως Chrome ή JavaScript sandbox.
Επιθέσεις σε μνήμη με ECC: ειδικές εκδοχές (TRRespass) δείχνουν πως ακόμα και ECC DRAM μπορεί να παραβιαστεί υπό προϋποθέσεις.
One-location hammering: πρόσφατες εργασίες δείχνουν ότι δεν απαιτείται hammering δύο διαφορετικών rows, αλλά μπορεί να είναι αρκετό το «targeted refresh denial».

Η αντιμετώπιση της Rowhammer απαιτεί συνδυασμό αντίμετρων hardware και software, καθώς οι κλασικοί μηχανισμοί ελέγχου προσβάσεων δεν έχουν ισχύ σε αυτήν την επίθεση. Ενδεικτικά:
Target Row Refresh (TRR): μηχανισμός που παρακολουθεί τις προσβάσεις σε rows και προληπτικά ανανεώνει τα γειτονικά rows (ενσωματωμένος σε DDR4/DDR5 memory blades).
ECC (Error-Correcting Code) memory: μειώνει τις πιθανότητες επιτυχημένων επιθέσεων, αν και δεν τις αποκλείει.
Software-based throttling: περιορισμός στη συχνότητα ενεργοποίησης rows από το OS ή

μέσω runtime (π.χ., στον Chrome browser).
Φυσική ανασχεδίαση: προτάσεις για αλλαγές στη φυσική τοπολογία των κελιών ώστε να αυξηθεί η απόσταση μεταξύ ευαίσθητων δεδομένων.

Ωστόσο, η Rowhammer τεχνική αναδεικνύει μια θεμελιώδη πρόκληση για την ασφάλεια: τα όρια μεταξύ φυσικής και λογικής απομόνωσης είναι ρευστά όταν οι επιθέσεις φτάνουν σε επίπεδο φυσικής συμπεριφοράς υλικού.

*Store/Harvest Now, Decrypt Later

Η επίθεση Store Now, Decrypt Later (SNDL) ή αλλιώς Harvest Now, Decrypt Later (HNDL) αποτελεί μια σοβαρή απειλή στον τομέα της κυβερνοασφάλειας, ιδιαίτερα σε σχέση με την έλευση της κβαντικής υπολογιστικής. Πρόκειται για μια στρατηγική κατά την οποία ένας επιτιθέμενος συλλέγει και αποθηκεύει σήμερα κρυπτογραφημένες επικοινωνίες ή δεδομένα, με την πρόθεση να τα αποκρυπτογραφήσει στο μέλλον, όταν θα διαθέτει πιο ισχυρούς υπολογιστικούς πόρους, κυρίως μέσω κβαντικών υπολογιστών, που θα μπορούν να «σπάνε» τις παραδοσιακές μορφές ασύμμετρης κρυπτογραφίας (π.χ. RSA, ECC).

Η επίθεση αυτή βασίζεται στην παραδοχή ότι τα σημερινά δεδομένα έχουν μακροχρόνια αξία ή ευαισθησία (π.χ. κρατικά μυστικά, ιατρικά αρχεία, επιχειρησιακά δεδομένα). Παρόλο που οι μέθοδοι αποκρυπτογράφησης είναι επί του παρόντος αναποτελεσματικές λόγω της ασφάλειας της υφιστάμενης κρυπτογραφίας, η πρόοδος στην κβαντική τεχνολογία (όπως ο αλγόριθμος του Shor) πιθανώς να επιτρέψει την αποκρυπτογράφηση αυτών των δεδομένων στο μέλλον.

Ως αποτέλεσμα, η μετάβαση σε post-quantum cryptography (PQC) είναι κρίσιμη, ώστε να διασφαλιστεί η ασφάλεια ακόμα και σε ένα μελλοντικό περιβάλλον όπου οι κβαντικοί υπολογιστές είναι λειτουργικοί.

Αρχιτεκτονική και Προσομοίωση

Για να ερευνήσουμε τρόπους να προστατεύσουμε ένα υπολογιστικό σύστημα από επιθέσεις τύπου Cold Boot, θα χρειαστεί να εξετάσουμε την αρχιτεκτονική και οργάνωση ενός Η/Υ, και πιο συγκεκριμένα να κατανοήσουμε τη λειτουργία μίας Κεντρικής Μονάδας Επεξεργασίας (ΚΜΕ). Σε αυτό το κεφάλαιο θα εξηγήσουμε ένα-ένα τα βασικότερα στοιχεία μιας ΚΜΕ και ενός Η/Υ, πως διασυνδέονται καθώς και πως λειτουργούν. Για την επίτευξη αυτού του στόχου χρήζει η χρήση κάποιας προσομοίωσης μίας ΚΜΕ. Συγκεκριμένα, για τον σκοπό αυτής της εργασίας, προσομοιώθηκε μία γενική αρχιτεκτονική ΚΜΕ βασισμένη στον MIPS. Προγραμματίστηκε λοιπόν, from scratch (από το μηδέν) ένας προσομοιωτής ηλεκτρονικού υπολογιστή βασισμένου στον επεξεργαστή του MIPS, με τη χρήση της αντικειμενοστραφούς γλώσσας προγραμματισμού C++ και παράλληλα με την παρουσίαση του βασικότερου κώδικα, θα εξηγηθεί σιγά-σιγά η αρχιτεκτονική των υπολογιστών γενικής χρήσης, όπου βασίζονται οι σύγχρονοι επεξεργαστές που βασίζονται οι Η/Υ που χρησιμοποιούμε αυτή τη στιγμή αλλά και επεξεργαστές πιο ειδικής χρήσης που ενσωματώνονται σε οποιαδήποτε χρήσης ηλεκτρονικά κυκλώματα. Πολύ σημαντικό να σημειωθεί ότι σε αυτή την αρχιτεκτονική ο ελεγκτής της μνήμης θεωρείται ότι βρίσκεται στην πλευρά της Κύριας Μνήμης.

MIPS

Ο MIPS ή Microprocessor without Interlocked Pipelined Stages (Μικροεπεξεργαστής χωρίς Ενδιάμεσα Στάδια Σωλήνωσης), κάτι που μάλιστα είναι αληθές μόνο για την πρώτη γενιά (MIPS I) των συγκεκριμένων εμπορικών επεξεργαστών (R2000, R3000), είναι ένας επεξεργαστής βασισμένος στην αρχιτεκτονική RISC (Reduced Instruction Set Computer/Υπολογιστής Περιορισμένης Αρχιτεκτονικής Συνόλου Εντολών) όπου άρχισε να αναπτύσσεται στο Πανεπιστήμιο του Stanford το 1981 από τον Καθηγητή John L. Hennessy, και την ερευνητική του ομάδα. Ο MIPS ήταν το αποτέλεσμα ενός ερευνητικού προγράμματος με σκοπό την μελέτη και την αξιολόγηση της καινούριας, για εκείνη την εποχή, αρχιτεκτονικής RISC. Ουσιαστικά ήταν η απάντηση της ομάδας του Hennessy στην αυξανόμενη πολυπλοκότητα των τότε επεξεργαστών τύπου CISC (Complex Instruction Set Computer/Υπολογιστής Σύνθετου Συνόλου Εντολών) όπως του Intel x86, όπου είχαν περίπλοκες εντολές καθώς και μεταβλητό μήκος εντολής και αυτό επέφερε δυσκολία σε περαιτέρω βελτιώσεις της ταχύτητας των επεξεργαστών.

Συγκεκριμένα, ο σκοπός του ερευνητικού αυτού έργου ήταν η απάντηση στα εξής ερωτήματα:
Μπορεί ένας επεξεργαστής με απλό και περιορισμένο σύνολο εντολών να ξεπεράσει σε ταχύτητα τους παραδοσιακούς CISC επεξεργαστές;
Μπορεί η χρήση τεχνικών όπως το pipelining (Σωλήνωση) να προσφέρουν σημαντική αύξηση στην απόδοση;
Η βασική ιδέα πίσω από αυτά τα ερωτήματα ήταν ότι η απλότητα οδηγεί σε μεγαλύτερη ταχύτητα, ευκολότερη βελτιστοποίηση και άρα σημαντική αύξηση στην συνολική απόδοση της επεξεργασίας. Δηλαδή, ουσιαστικά οι στόχοι του έργου ήταν:
Απλότητα: Μία αρχιτεκτονική απλή όπου επιτρέπει κάθε εντολή να εκτελείται γρήγορα και προβλέψιμα και άρα με σταθερό αριθμό κύκλων ρολογιού (Clock Cycle).
Ταχύτητα: Καλύτερη απόδοση με χρήση νέων τεχνικών.
Επεκτασιμότητα: Εύκολη υλοποίηση στο υλικό και άρα δυνατότητα απλής επέκτασής του.

Το 1983 παρουσιάζεται με μεγάλη επιτυχία ο πρώτος λειτουργικός MIPS επεξεργαστής, ένα πειραματικό πρωτότυπο που έφερε πολλές αλλά και θεμελιώδεις καινοτομίες για την επιστήμη της σχεδίασης επεξεργαστών. Συγκεκριμένα, μαζί με την καινοτόμα υλοποίηση της RISC αρχιτεκτονικής, εισήχθησαν νέες τεχνολογίες όπως το Pipelining, νέα μέθοδος load/store (φόρτωσης/αποθήκευσης) για αποτελεσματικότερη προσπέλαση μνήμης, απλοποίηση της αποκωδικοποίησης των εντολών καθώς και νέες μεθόδους συνεργασίας των μεταγλωττιστών. Ειδικότερα τα βασικά χαρακτηριστικά του νέου συστήματος ήταν:
Σταθερού μεγέθους εντολές (32-bit),
Σημαντικά μικρότερο ISA,
32 γενικής χρήσης καταχωρητές,
Pipeline 5 σταδίων και
πιο απλή λογική ελέγχου στο υλικό και άρα περισσότερη ευθύνη στο λογισμικό/μεταγλωττιστές, δηλαδή χωρίς τη χρήση Interlocking μηχανισμών αν και αυτό άλλαξε στις επόμενες γενιές (1991+) με την ανάπτυξη τεχνικών όπως out of order execution, branch prediction και καλύτερη ιεραρχία κρυφών μνημών.

Η επιτυχία και καινοτομία του MIPS οδήγησε στην ίδρυση της εταιρείας MIPS Computer Systems Inc από τον Hennessy το 1984, με στόχο την εμπορική αξιοποίηση της νέας RISC τεχνολογίας με την εμπορευματοποίηση του MIPS R2000 το 1985 και μεταγενέστερα μοντέλα, όπως το R3000 (1988, υποστήριξη εικονικής μνήμης) και το R4000 (1991, πρώτη 64-bit έκδοση και Interlocking) όπου χρησιμοποιήθηκαν για πολλές χρήσεις όπως σταθερούς και φορητού Η/Υ, ενσωματωμένα συστήματα (π.χ. παιχνιδομηχανές/R4300i) και

χρησιμοποιείται ακόμη και σήμερα για εκπαιδευτικούς σκοπούς και διδάσκεται σε πολλά πανεπιστήμια παγκοσμίως.

Reduced Instruction Set Computer

Η αρχιτεκτονική RISC (Reduced Instruction Set Computer) αποτελεί ένα υπολογιστικό μοντέλο που βασίζεται στον σχεδιασμό επεξεργαστών με απλό και περιορισμένο σύνολο εντολών, οι οποίες εκτελούνται όλες με ομοιόμορφο και σταθερό ρυθμό, συνήθως σε έναν μόνο κύκλο ρολογιού. Η βασική φιλοσοφία του RISC έγκειται στην παραδοχή ότι η απλότητα στο υλικό επιτρέπει την υψηλότερη απόδοση, καθώς ο επεξεργαστής μπορεί να εκτελεί περισσότερες εντολές ανά δευτερόλεπτο με μικρότερη πολυπλοκότητα.

Σε αντίθεση με την παραδοσιακή προσέγγιση της CISC (Complex Instruction Set Computer), η οποία χρησιμοποιεί σύνθετες και ποικίλες εντολές υψηλού επιπέδου, το μοντέλο RISC επιλέγει να παρέχει ένα μικρό αριθμό απλών, γενικής χρήσης εντολών, οι οποίες μπορούν να συνδυαστούν από τον μεταγλωττιστή για την επίτευξη πιο σύνθετων λειτουργιών. Κάθε εντολή έχει ομοιόμορφο μέγεθος και σταθερή μορφή, γεγονός που διευκολύνει την σωλήνωση (pipelining) και μειώνει την καθυστέρηση στη μονάδα αποκωδικοποίησης.

Κύρια χαρακτηριστικά των RISC αρχιτεκτονικών περιλαμβάνουν, ένα καινοτόμο μοντέλο φόρτωσης/αποθήκευσης (load/store architecture), στο οποίο μόνο ειδικές εντολές χειρίζονται τη μνήμη ενώ όλες οι υπόλοιπες λειτουργούν σε καταχωρητές, μεγάλος αριθμός καταχωρητών γενικής χρήσης, για την ελαχιστοποίηση της προσπέλασης στη μνήμη και απλοποιημένος έλεγχος υλικού, που επιτρέπει υψηλότερες ταχύτητες ρολογιού και καλύτερη ενεργειακή αποδοτικότητα.

Η ιδέα της αρχιτεκτονικής RISC διαμορφώθηκε αρχικά τη δεκαετία του 1970 στο εσωτερικό της IBM, μέσω του ερευνητικού έργου IBM 801, το οποίο ξεκίνησε το 1975 υπό την καθοδήγηση του John Cocke. Το IBM 801 θεωρείται το πρώτο πλήρως λειτουργικό σύστημα που εφάρμοσε τη φιλοσοφία του μειωμένου συνόλου εντολών, και έθεσε τις βάσεις τόσο για τη θεωρητική θεμελίωση όσο και για την πρακτική εφαρμογή της RISC αρχιτεκτονικής. Στη συνέχεια, στις αρχές της δεκαετίας του 1980, η ιδέα του RISC αναπτύχθηκε ανεξάρτητα και ακαδημαϊκά σε δύο σημαντικά πανεπιστημιακά έργα: στο πανεπιστήμιο Berkeley, με τα πρότζεκτ RISC I και RISC II, και στο πανεπιστήμιο Stanford, με την ανάπτυξη του MIPS. Τα έργα αυτά συνέβαλαν στη διάδοση, την τυποποίηση και την εμπορική αξιοποίηση της RISC φιλοσοφίας, οδηγώντας σε σημαντικές αρχιτεκτονικές όπως οι MIPS, SPARC, ARM και PowerPC. Σήμερα, η RISC αρχιτεκτονική έχει κυριαρχήσει κυρίως σε ενσωματωμένα και κινητά συστήματα, λόγω της υψηλής απόδοσης ανά watt, της απλότητας και της ευκολίας στην υλοποίηση και επέκταση.

Μηχανισμός Interlocking

Ο μηχανισμός Interlocking (Μηχανισμός Παγώματος Εντολών) αποτελεί κρίσιμο στοιχείο στους pipelined (με σωλήνωση) επεξεργαστές για τη διαχείριση συγκρούσεων που προκύπτουν κατά την ταυτόχρονη εκτέλεση εντολών. Σε αρχιτεκτονικές όπου οι εντολές εκτελούνται σε διαδοχικά στάδια, μπορεί να εμφανιστούν εξαρτήσεις δεδομένων, όταν μία εντολή εξαρτάται από το αποτέλεσμα προηγούμενης που δεν έχει ακόμα ολοκληρωθεί η επεξεργασίας της. Ο Interlocking μηχανισμός αναλαμβάνει τη δυναμική ανίχνευση τέτοιων εξαρτήσεων και την αυτόματη εισαγωγή καθυστερήσεων/παγωμάτων (pipeline stalls) για την αποφυγή σφαλμάτων κατά την εκτέλεση. Αυτό επιτυγχάνεται μέσω παγωμάτων συγκεκριμένων σταδίων του pipeline μέχρι να καταστούν διαθέσιμα τα απαιτούμενα δεδομένα. Ο επεξεργαστής, επομένως, μπορεί να διασφαλίσει την ορθότητα της ροής των εντολών χωρίς την ανάγκη παρέμβασης από τον μεταγλωττιστή ή τον προγραμματιστή. Αν και ο μηχανισμός αυτός προσφέρει αυξημένη αξιοπιστία και ευκολία στον προγραμματισμό, εντούτοις συνεπάγεται μεγαλύτερη πολυπλοκότητα στο υλικό και ενδέχεται να μειώσει την απόδοση λόγω των καθυστερήσεων που εισάγονται. Στις πρώτες εκδόσεις της αρχιτεκτονικής MIPS, όπως στον MIPS R2000 που δεν υπήρχε υποστήριξη interlocking σε επίπεδο υλικού, η ευθύνη για την αποφυγή των συγκρούσεων μεταβιβαζόταν στον compiler, ο οποίος εισήγαγε εντολές αναμονής (NOPs) ή αναδιάτασσε τον κώδικα. Ωστόσο, με την εμφάνιση πιο σύνθετων μοντέλων, όπως ο MIPS R4000 και οι μεταγενέστεροι επεξεργαστές, το hardware interlocking ενσωματώθηκε σταδιακά, ώστε να καλυφθούν οι αυξανόμενες απαιτήσεις σε πολυπλοκότητα, απόδοση και υποστήριξη δυναμικών τεχνικών εκτέλεσης.

Η αντικειμενοστραφής γλώσσα προγραμματισμού C++

Η γλώσσα προγραμματισμού C++ αποτελεί μία από τις πλέον διαδεδομένες και καθιερωμένες γλώσσες προγραμματισμού γενικού σκοπού, η οποία σχεδιάστηκε και υλοποιήθηκε από τον Bjarne Stroustrup στις αρχές της δεκαετίας του 1980 στο Bell Labs της AT&T. Η ανάπτυξή της βασίστηκε στην ανάγκη ενίσχυσης της γλώσσας C με υποστήριξη για αντικειμενοστραφή προγραμματισμό, διατηρώντας παράλληλα την υψηλή απόδοση και τον άμεσο έλεγχο του υπολογιστή. Η C++ υποστηρίζει πολλαπλά παραδείγματα προγραμματισμού, όπως διαδικαστικό, αντικειμενοστραφές και γενικευμένο προγραμματισμό, γεγονός που την καθιστά εξαιρετικά ευέλικτη και κατάλληλη για πλήθος εφαρμογών, από συστήματα χαμηλού επιπέδου (όπως λειτουργικά συστήματα, οδηγοί συσκευών και προσομοιωτές) έως σύνθετα συστήματα λογισμικού, εφαρμογές πραγματικού χρόνου και παιχνίδια υψηλής απόδοσης.

Ένα από τα σημαντικότερα πλεονεκτήματα της C++ είναι η δυνατότητα άμεσης διαχείρισης της μνήμης μέσω δεικτών και δυναμικής κατανομής, γεγονός που προσφέρει υψηλή απόδοση, αλλά και αυξημένες απαιτήσεις σε επίπεδο σωστής διαχείρισης από τον προγραμματιστή. Επιπλέον, η ύπαρξη χαρακτηριστικών όπως οι πρότυποι τύποι (templates), οι συναρτήσεις inline και η χρήση της Πρότυπης Βιβλιοθήκης (Standard Template Library/STL), προσφέρουν στον προγραμματιστή εργαλεία για αποδοτικό και επαναχρησιμοποιήσιμο κώδικα.

Ο Προσομοιωτής

Στα πλαίσια αυτής της εργασίας αναπτύχθηκε ένας προσομοιωτής ενός υπολογιστή βασισμένου σε ΚΜΕ μονού πυρήνα βασισμένη στον MIPS. Είναι ουσιαστικά μία εικονική μηχανή που κατανοεί ένα μεγάλο MIPS ISA (συνόλου εντολών) και εκτελεί/προσομοιώνει με έναν ρεαλιστικό τρόπο την διαδικασία όπου περνάνε οι εντολές/δεδομένα και τους κύκλους επεξεργασίας. Ο χρήστης εισαγάγει το πρόγραμμα που θέλει να εκτελέσει σε συμβολική γλώσσα και με τη χρήση του συμβολομεταφραστή (Assembler), που εκτελείται κατά την εκκίνηση του προσομοιωτή, μεταφράζεται σε γλώσσα μηχανής όπου ύστερα κατανοεί η ΚΜΕ. Συγκεκριμένα η βάση του αριθμητικού συστήματος που χρησιμοποιείται είναι η δεκαεξαδική (hexadecimal). Ύστερα, ο προσομοιωτής τροποποιήθηκε κατάλληλα για να εισαχθεί η λύση ασφαλείας που προτείνεται σε αυτή η εργασία καθώς και η τροποποίηση ήταν επιτυχής επειδή η λογική που αναπτύχθηκε πάνω ο προσομοιωτής ήταν της εύκολης προσαρμοστικότητας νέων μηχανισμών. Ο προσομοιωτής προγραμματίστηκε από το 0 (from scratch) με τη γλώσσα προγραμματισμού C++ και εκτελέστηκε σε Ubuntu 24.04 (Linux 6.11) λειτουργικό σύστημα ως εικονική μηχανή. Το συνολικό μεταγλωττισμένο πρόγραμμα/προσομοιωτής (μαζί με την τροποποίηση και εισαγωγή της λύσης ασφαλείας) είναι γύρω στα 3.8 MB. Οι λειτουργίες και τα χαρακτηριστικά του θα κατανοηθούν με την ανάλυση του κώδικα (Δεν παρατίθεται ο πλήρης κώδικας παρά μόνο τα πιο σημαντικά κομμάτια).

Ο προσομοιωτής αυτός αναπτύχθηκε ΑΠΟΚΛΕΙΣΤΙΚΑ για την εργασία αυτή και ΔΕΝ θα πρέπει να χρησιμοποιηθεί για οποιονδήποτε άλλο σκοπό εκτός από πειραματισμούς καθώς δεν θεωρείται ΚΑΘΟΛΟΥ ΑΣΦΑΛΕΙΣ ως προς την ομαλή λειτουργία του και φέρει σοβαρά προβλήματα ασφαλείας καθώς δεν γίνονται έλεγχοι όπως το τι πρόγραμμα εκτελείται, τι είδους δεδομένα προσπελαύνονται και πως χρησιμοποιείται από τον χρήστη. Η εκτέλεση του μπορεί να οδηγήσει σε κρίσιμα σφάλματα για το σύστημα όπου τρέχει και ενδέχεται οι λειτουργίες του να μην εκτελούνται ορθά καθώς και υπάρχουν κάποια σημεία κώδικα που τελικά δεν εκτελούνται. Δεν φέρω καμία ευθύνη για την χρήση του προσομοιωτή και ο κώδικας που αναλύεται πιο κάτω πρέπει να χρησιμοποιηθεί ΜΟΝΟ για εκπαιδευτικούς σκοπούς.

Η Λειτουργία του Επεξεργαστή

Ας εξηγήσουμε πολύ απλά πως λειτουργεί ο επεξεργαστής, τι μπορεί να περιλαμβάνει ένας κύκλος επεξεργασίας και πως εκτελείται μία εντολή.

Όπως έχουμε ήδη καταλάβει, τα δύο πιο βασικά στοιχεία ενός Η/Υ είναι η Κεντρική Μονάδα Επεξεργασίας και η Κύρια Μνήμη. Όταν ο υπολογιστής τροφοδοτείται με ενέργεια, κάνει κάποια σαφώς καθορισμένα βήματα, τα οποία μεταφέρουν τη λειτουργία σε κάποια άλλα σαφώς καθορισμένα βήματα (μία ρουτίνα, ένα (υπό/)πρόγραμμα) τα οποία συνεχίζουν με κάποια άλλα βήματα κ.ο.κ. μέχρι να σταματήσει να τροφοδοτείται με ηλεκτρικό ρεύμα ο επεξεργαστής, ας μην ξεχνάμε ότι ο υπολογιστής κάνει ακριβώς αυτό που εμείς του δείχνουμε ότι πρέπει να κάνει. Το αποτέλεσμα είναι συγκεκριμένο και συνεπώς είναι μία ντετερμινιστική μηχανή.

Το πρώτο πράγμα λοιπόν που εκτελείται πάντα και αυτόματα, αφού προφανώς ο επεξεργαστής κάνει τους δικούς του ελέγχους, που δεν είναι τίποτα από ένα απλό reset (επαναφορά) που μηδενίζει τα στοιχεία του, είναι να κάνει jump (να κάνει άλμα) στο BIOS/Basic Input Output System (Βασικό/Απλό Σύστημα Εισόδου Εξόδου) όπου βρίσκεται σε μία μικρή και ειδική μνήμη, ROM (Read Only Memory/Μνήμη Μόνο για Ανάγνωση) στην μητρική πλακέτα (motherboard) του συστήματος (το κομμάτι που διασυνδέει τα πάντα μεταξύ τους), αφού προφανώς, ακόμη, δεν έχει προλάβει να κάνει reset και η ίδια η ΜΤΠ, και στην πραγματικότητα ο επεξεργαστής δεν έχει ιδέα που βρίσκεται (σε ποια διεύθυνση/ποιόν δρόμο θα ακολουθήσει για να τη βρει). Το μόνο που γνωρίζει (είναι hardcoded/τυπωμένο στην πλακέτα) είναι μία και μοναδική διεύθυνση π.χ. στην 0xBFC00000 (που πρέπει να γνωρίζουν και οι κατασκευαστές των μητρικών πλακετών) όπου εκεί θα πρέπει να βρει το πρώτο πρόγραμμα που θα τρέξει, πιο συγκεκριμένα την πρώτη εντολή του BIOS που θα εκτελέσει. Άρα προφανώς, η αναφορά στην 0xBFC00000 (Συνήθως τις διευθύνσεις μνήμης τις αναφέρουμε στο δεκαεξαδικό/hexadecimal σύστημα αρίθμησης. Άμα κάποιος θέλει το αναφέρει ως 3217031168 στο δεκαδικό ή ακόμη και ως 1011 1111 1100 0000 0000 0000 0000 0000 στο δυαδικό σύστημα αρίθμησης, ειδικότερα άμα είναι επεξεργαστής και κατανοεί μόνο ρεύμα/1 ή όχι ρεύμα/0 και το διαβάζει σε κομμάτια/blocks των 32 ψηφίων, γιατί απλά ενώνονται παράλληλα με 32 γραμμές μεταφοράς/επικοινωνίας/αγώγιμου υλικού) δεν βρίσκεται στην κύρια μνήμη αλλά είναι ενωμένη με την αρχή της μνήμης/ROM της μητρικής. Παρεμπιπτόντως, ο επεξεργαστής νομίζει ότι αυτή η διεύθυνση ανήκει στην κύρια μνήμη, καθώς για αυτόν, η μνήμη είναι μία και απλά την αναφέρει με τις διευθύνσεις από 0 ως $1x2^{32}$ (για αναφορές που λειτουργούν με 32 ψηφία).

Η δουλειά του BIOS είναι να προετοιμάσει κάποια στοιχεία του Η/Υ σύμφωνα με κάποιες παραμέτρους, να ολοκληρώσει τη ρύθμιση της ΜΤΠ, να φορτώσει κάποιες ειδικές τιμές στους καταχωρητές της ΚΜΕ καθώς και να δείξει στον επεξεργαστή που πρέπει να προχωρήσει, σε ποια διεύθυνση να «χτυπήσει». Όταν ολοκληρωθεί αυτή η διαδικασία δηλαδή, πριν τελειώσει το τμήμα του BIOS, στη τελευταία διεύθυνση που θα φτάσει, υπάρχει μία εντολή που όταν την διαβάζει, κάνει άλμα (jump) στο σημείο που αναφέρει, π.χ. στην 0x80000000. Από εκεί και πέρα, εάν μιλάμε για ένα ενσωματωμένο σύστημα, συνήθως, θα προσπελάσει άλλη μία, λίγο μεγαλύτερη από την προηγούμενη, ειδική μνήμη που βρίσκεται τοποθετημένη στην μητρική πλακέτα, ενώ αν είναι ένας γενικής χρήσης υπολογιστής, θα είναι το τμήμα που έχει συμφωνηθεί να βρίσκεται το πρόγραμμα εκκίνησης στον μη πτητικό σκληρό δίσκο.

Τί υπάρχει λοιπόν σε αυτή τη διεύθυνση; Ο Bootloader ή το Πρόγραμμα Εκκίνησης, όπου είναι το τελευταίο βήμα που θα ακολουθήσει ο Η/Υ πριν δοθεί αποκλειστικά στον έλεγχο του

λειτουργικού συστήματος. Ο Bootloader λοιπόν είναι μία ρουτίνα υπεύθυνη για την καθοδήγηση και σωστή ρύθμιση της ΜΤΠ, περισσότερων καταχωρητών της ΚΜΕ καθώς και την ρύθμιση περιφερειακών συσκευών (προφανώς με τη χρήση του επεξεργαστή, γιατί απλά να κάθεται και να μην κάνει τίποτα; Επίσης, ο επεξεργαστής δεν μπορεί να διακόψει τη λειτουργία του) που βρίσκονται συνδεδεμένες (μέσω διαύλων επικοινωνίας στη μητρική κάρτα. Επίσης ας μην ξεχνάμε ότι για την ΚΜΕ όλα είναι Κύρια Μνήμη) όπως GPU (Graphics Processing Unit/Μονάδα Επεξεργασίας Γραφικών, για να μην ταλαιπωρείται και πολύ ο επεξεργαστής), κάρτα δικτύου, οθόνη, ποντίκι, πληκτρολόγιο κ.α. Όλα αυτά λοιπόν για την ΚΜΕ είναι «ένα» και απλά «ακούνε/μιλάνε» σε μία διεύθυνση/πόρτα (port), απλά συνήθως με μία συμφωνία/πρωτόκολλο (protocol) που ορίζεται με τους drivers/οδηγών των συσκευών. Αφού ολοκληρωθεί η ρύθμιση του Η/Υ, τότε γίνεται και η απόφαση για το ποιο λογισμικό (λειτουργικό σύστημα) θα φορτωθεί στην ΜΤΠ και θα λάβει τον πλήρη έλεγχο του συστήματος. Από αυτό το σημείο και μετά, τον αποκλειστικό έλεγχο τον έχει ο πυρήνας (kernel) του λειτουργικού συστήματος και οποιαδήποτε εργασία τρέξει ο χρήστης, μέχρι το σύστημα να σταματήσει να τροφοδοτείται με ηλεκτρική ενέργεια.

Άρα συμπεραίνουμε ότι αυτό που κάνει ένας Η/Υ είναι να επεξεργάζεται, όπως του δοθεί η οδηγία (εντολή), δεδομένα που βρίσκονται σε συγκεκριμένες διευθύνσεις μνήμης και να κάνει άλματα μεταξύ τους, κατά προτίμηση, με τη χρήση βελτιστοποιημένων αλγορίθμων. Εξ’ άλλου ας μην ξεχνάμε, ότι ένας υπολογιστής, υπολογίζει κάτι, και οι σύγχρονοι Η/Υ είναι αποτέλεσμα έρευνας απλών ηλεκτρονικών αριθμομηχανών, και πραγματικά, εκτελούν πράξεις μεταξύ κάποιων τελούμενων, απλά εμείς ερμηνεύουμε τις ακολουθίες των αποτελεσμάτων με τον δικό μας τρόπο, π.χ. μια κουκκίδα σε συγκεκριμένο σημείο της οθόνης και με συγκεκριμένο χρώμα, είναι απλά το αποτέλεσμα κάποιων πράξεων που τα αποτυπώνουμε στον Χ και Υ άξονα της οθόνης.

Τα βασικά στοιχεία του Η/Υ

Ο Υπολογιστής

Ο προσομοιωτής έχει αναπτυχθεί με τρόπο τέτοιο ώστε κάθε υλικό κομμάτι να αποτελεί πραγματικά ένα ξεχωριστό object, και μάλιστα αυτό θα κατανοηθεί ακόμη καλύτερα με την ανάλυση της κλάσης Processor. Έτσι λοιπόν, όλα τα αντικείμενα τοποθετούνται μέσα στην κλάση Computer καθώς και πρέπει να δοθεί η τοποθεσία και το όνομα αρχείου που περιέχει σε γλώσσα μηχανής (μετά τη συμβολομετάφραση/Assembled πρόγραμμα) του προγράμματος που θα εκτελέσει.

```
// Compiler Directives
#pragma once

// Header Files
#include "Memory.h"
#include "Processor.h"
#include <string>

class Computer
{
public:
    // Constructor
    Computer(const std::string& bootfile);

    Memory memory; // The memory component
    Processor core;

    // Methods
    void run(int cycles);
    void dump_state();
};
```

Από ότι βλέπουμε, για το απλό πείραμα χρειαζόμαστε την Κύρια Μνήμη και τον πυρήνα επεξεργασίας (επεξεργαστής) καθώς και μία μέθοδο για κάθε κύκλο επεξεργασίας (run()) και μία dump_state() που μας επιτρέπει να ελέγξουμε την κατάσταση σημαντικών στοιχείων κατά την κλήση της, για καλύτερη ανάλυση του πειράματος.

```
// Header Files
#include "Computer.h"
#include <iostream>

Computer::Computer(const std::string& bootfile):core(memory)
{
    memory.load_bootloader(bootfile);
}

void Computer::run(int cycles)
{
```

```
    for (int i=0; i<cycles; ++i)
    {
        std::cout<<std::dec<<"\n=== Cycle "<<i+1<<"
==="<<std::endl;
            core.routine();
    }
}

void Computer::dump_state()
{
    // Dump registers and caches for all processors
    std::cout<<std::dec<<"\n=== Processor State ===\n";
    core.dump_registers();
    core.print_cache_stats();
    core.dump_caches();
    core.dump_tlb();
    core.flush_all_caches();

    // Dump memory
    memory.dump_memory_block(0x00000000,0x40000000); // Physical
Address (Needs Calculation)
}
```

Η Κύρια Μνήμη

Όπως εξηγήσαμε και πιο πάνω ο Η/Υ προσπελάζει δεδομένα όπου βρίσκονται σε κάποια τοποθεσία, σε μία διεύθυνση+offset (μετατόπιση), που αντιστοιχεί στον φυσικό χώρο κάποιων ενεργητικών στοιχείων που η δουλειά τους είναι να κρατούν την κατάσταση ενός ηλεκτρικού φορτίου, τις μνήμες. Ουσιαστικά ο υπολογιστής νομίζει ότι επικοινωνεί με ένα πράγμα αλλά είναι διάφορες μνήμες (RAM, ROM, GPU κ.α.) ενοποιημένες.

Όταν αναφερόμαστε στην Κύρια Μνήμη, εννοούμε την ΜΤΠ αλλά με τον ίδιο δίαυλο επικοινωνίας (bus) γίνεται η πρόσβαση σε άλλα στοιχεία. Η Κύρια Μνήμη είναι ένα στοιχείο που χωράει Χ μονάδες (bit) ηλεκτρικά φορτισμένων πλαισίων/ειδικών κυκλωμάτων (π.χ. D-type Flip Flop). Βέβαια, για την ενορχήστρωση και καλύτερη επικοινωνία με τα υπόλοιπα κυκλώματα π.χ. ΚΜΕ, χρησιμοποιούνται ειδικοί ελεγκτές (controller). Ας είμαστε και πιο ακριβείς, οι εξηγήσεις δίνονται στην πιο απλή τους μορφή και επειδή οι σύγχρονοι Η/Υ έχουν προχωρήσει πάρα πολύ και έχουν ενσωματωθεί πάρα πολλές λειτουργίες, είναι κρίσιμο να υπάρχουν dedicated (αποκλειστικά) ελεγκτές για να εποπτεύουν και την ομαλή λειτουργία όλων των στοιχείων και των χαρακτηριστικών τους.

Στην προσομοίωση ο ελεγκτής και η μνήμη είναι μία κλάση, επειδή ουσιαστικά η μνήμη είναι ένας πίνακας Χ στοιχείων μόνο, οι υπόλοιπες συναρτήσεις (functions) κώδικα, αφορούν τον έλεγχό της και την επικοινωνία της. Επίσης, σε αυτή την αρχιτεκτονική ο ελεγκτής της μνήμης θεωρείται ότι βρίσκεται δίπλα από την Κύρια Μνήμη.

```
class Memory
{
private:
    // The memory vector
    std::vector<Byte> memory;

    // Block size for efficient encryption operations (Should
match CDU block size)
    static constexpr int MEMORY_BLOCK_SIZE=16; // 16 words*4
bytes=64 bytes

    // Helper method to align addresses to block boundaries
    Word alignToMemoryBlock(Word addr) const; // Make sure it is
the same with CDU's method

public:
    // Physical Memory Access
    Byte read_physical_byte(Address physical_addr) const;
    void write_physical_byte(Address physical_addr, Byte value);

    void write_physical_word(Word physical_addr, Word value);
    Word read_physical_word(Word physical_addr) const;

    // Block operations for efficient CDU communication
    // Memory ONLY encrypts blocks before storing
    void write_physical_block(Word physical_addr, const Word*
data, int word_count);
```

```
    // Memory returns encrypted blocks to CDU (CDU handles
decryption)
    bool read_physical_block(Word physical_addr, Word* data_out,
int word_count) const;

    // Key exchange handler for CDU communication
    // CDU calls this with its public key, memory returns its
shared secret
    bool handleKeyExchange(const uint8_t* cdu_public_key,
uint8_t* ciphertext_out, uint8_t* shared_secret_out,
KeyExchangeModule* cdu_key_exchange_bus);

    // Constructor
    explicit Memory(uint64_t size=0x40000000); // Explicit; avoid
conversions, Predefined size is 1GB // CHECK MMU

    // Methods
    // Load programs
    bool load_bootloader(const std::string& filename);

    bool isEncryptionEnabled() const;

    // Debugging
    void dump_memory_block(Address start,Address end) const;
};
```

Από ότι βλέπουμε η μνήμη είναι ένας vector (διάνυσμα/πίνακας) κατά την δημιουργία του αντικειμένου της μνήμης μηδενίζεται και δεσμεύει 1GB χώρο στο σύστημα όπου τρέχει την προσομοίωση. Αυτό το αναλαμβάνει ο constructor (κατασκευαστής αντικειμένου) όπως βλέπουμε πιο κάτω:

```
// Constructor
Memory::Memory(uint64_t size) // Predefined value is 1GB
{
    // Allocate memory and set to 0; memory is volatile
    memory.resize(size, 0); // Allocate
    std::cout<<"Memory initialized with "<<(size/(1024*1024))<<"
MB"<<std::endl;
    std::cout<<"Memory controller encryption:
"<<(encryption_enabled?"Enabled!":"DISABLED!")<<std::endl;
    std::cout<<"Memory block size: "<<MEMORY_BLOCK_SIZE<<" words
("<<(MEMORY_BLOCK_SIZE*4)<<" bytes)"<<std::endl;
}
```

Ας αγνοήσουμε για τώρα εντολές σχετικές με κρυπτογράφηση καθώς θα αναλυθούν σε επόμενο κεφάλαιο. Επίσης η φόρτωση/γράψιμο των δεδομένων γίνεται ανά block και οι write/read_physical_byte/word τελικά δεν χρησιμοποιούνται, όμως θα εξηγηθούν εδώ ενώ οι block μέθοδοι θα αναφερθούν αναλυτικά στο επόμενο κεφάλαιο καθώς διαχειρίζονται κρυπτογραφημένα δεδομένα.

```
Byte Memory::read_physical_byte(Word physical_addr) const
{
    // Simple bounds checking
    if (physical_addr>=memory.size())
    {
        std::cout<<"ERROR: Memory read out of bounds:
0x"<<std::hex<<physical_addr<<std::endl;
        return 1; // Return an error
    }
    return memory[physical_addr];
}
```

Πριν διαβαστεί η μνήμη γίνεται ένας έλεγχο εάν η φυσική διεύθυνση που ζητάνε δεν ξεπερνάει τη συνολική μνήμη.

```
void Memory::write_physical_byte(Word physical_addr, Byte value)
{
    // Simple bounds checking
    if (physical_addr>=memory.size())
    {
        std::cout<<"ERROR: Memory write out of bounds:
0x"<<std::hex<<physical_addr<<std::endl;
        return;
    }
        memory[physical_addr]=value;
}
```

Στη μνήμη γράφεται το Byte δεδομένων.

Ας εξετάσουμε και λίγο την περίπτωση που θα μπορούσαν να χρησιμοποιηθούν απευθείας words (32 bit λέξη).

```
// Read word from physical memory (returns encrypted data) DELETE
Word Memory::read_physical_word(Word physical_addr) const
{
    // Simple bounds checking
    if (physical_addr+3>=memory.size())
    {
        std::cout<<"ERROR: Memory word read out of bounds:
0x"<<std::hex<<physical_addr<<std::endl;
        return 0xFFFFFFFF; // Return error value
    }

    // Read word in little-endian format
    Word word=0;
    word|=(static_cast<Word>(memory[physical_addr+0])<<0);
    word|=(static_cast<Word>(memory[physical_addr+1])<<8);
    word|=(static_cast<Word>(memory[physical_addr+2])<<16);
    word|=(static_cast<Word>(memory[physical_addr+3])<<24);
```

```
    return word; // CDU will decrypt this
}

void Memory::write_physical_word(Word physical_addr, Word value)
{
    // Simple bounds checking
    if (physical_addr+3>=memory.size())
    {
        std::cout<<"ERROR: Memory word write out of bounds:
0x"<<std::hex<<physical_addr<<std::endl;
        return;
    }

    Word data_to_store=value;

    // Store word in little-endian format
memory[physical_addr+0]=static_cast<Byte>((data_to_store>>0)&0xFF
);

memory[physical_addr+1]=static_cast<Byte>((data_to_store>>8)&0xFF
);

memory[physical_addr+2]=static_cast<Byte>((data_to_store>>16)&0xF
F);

memory[physical_addr+3]=static_cast<Byte>((data_to_store>>24)&0xF
F);
}
```

Παρομοίως με τα Byte, γράφουμε/διαβάζουμε πολύ απλά ανά 32 bit.

Άλλη μία βασική μέθοδος για την ελεγκτή της μνήμης είναι η ρουτίνα φόρτωσης του προγράμματος (BIOS) στην μνήμη για την προσομοίωση της ειδικής μνήμης της, κάθε φορά που ενεργοποιείται ο υπολογιστής, που όπως είπαμε ο επεξεργαστής καταλαβαίνει ως μία και δεν χρειάζεται, για λόγους πειραματικούς, να την ξεχωρίσουμε.

```
// Program Loading
bool Memory::load_bootloader(const std::string& filename) // MAKE
IT READ VECTOR, READ THE FILE IN MAIN INSTEAD
{
    // Open file in binary mode
    std::ifstream file(filename);
    if (!file.is_open())
    {
        std::cout<<"Error: Could not read boot file:
"<<filename<<std::endl;
        return false;
    }
```

```
    std::string line;
    std::streamsize offset=0;

    while (std::getline(file,line))
    {
        if (line.empty()) continue;

        // Check if we're going to exceed ROM size
        if (offset+4>memory.size())
        {
            std::cout<<"ERROR: Program cannot fit in the
bootloader section! Stopped writing."<<std::endl;
            return false;
        }

        // Convert hex string to 32-bit instruction
        Word plain_instruction=std::stoul(line,nullptr,16);
        //Word instruction=0;

        // Memory controller always encrypts before storing
write_physical_word(TRANSLATED_ROM_BASE+offset,plain_instruction)
;

        // // Write instruction to memory in little-endian order
        //
write_physical_byte(TRANSLATED_ROM_BASE+offset+0,static_cast<Byte
>((instruction>>0)&0xFF));
        //
write_physical_byte(TRANSLATED_ROM_BASE+offset+1,static_cast<Byte
>((instruction>>8)&0xFF));
        //
write_physical_byte(TRANSLATED_ROM_BASE+offset+2,static_cast<Byte
>((instruction>>16)&0xFF));
        //
write_physical_byte(TRANSLATED_ROM_BASE+offset+3,static_cast<Byte
>((instruction>>24)&0xFF));

        offset+=4;
    }

    std::cout<<"Program loaded: Written "<<std::dec<<offset<<"
bytes at physical address
0x"<<std::hex<<TRANSLATED_ROM_BASE<<std::endl;

    return true;
}
```

Αυτή είναι η ρουτίνα φόρτωσης του προγράμματος που θα τρέξει για πρώτη φορά ο επεξεργαστής. Στην περίπτωση εισαγωγής κρυπτογράφησης, αυτό θα πρέπει να τροποποιηθεί κατάλληλα. Η TRANSLATED_ROM_BASED, είναι η διεύθυνση 0xBFC00000 μεταφρασμένη για 1 GB φυσικής μνήμης: #define TRANSLATED_ROM_BASE 0x1FC00000. Επίσης η write_physical_byte έχει αντικατασταθεί με την write_physical_word. Είναι δε κρίσιμο να ακολουθηθεί το endianness (ποιο bit θεωρείται το πιο σημαντικό, τελευταίο ή πρώτο από δεξιά;) του συστήματος.

Η τελευταία συνάρτηση/μέθοδος είναι αυτή όπου θα μας βοηθήσει να κατανοήσουμε το αποτέλεσμα του πειράματος, να κάνουμε dump την κατάσταση της κύριας μνήμης.

```
// Memory dump
void Memory::dump_memory_block(Word start, Word end) const
{
    // Ensure end is greater than or equal to start
    if (end<start)
    {
        std::swap(start,end);
    }

    // Check bounds
    if (start>=memory.size()) {std::cout<<"ERROR: Start address
out of bounds"<<std::endl; return;}
    if (end>=memory.size()) {end=memory.size()-1;}

    std::cout<<"Memory dump from physical 0x"<<std::hex<<start<<"
to 0x"<<end<<std::endl;
    std::cout<<"-----------------------------------------------
--------------"<<std::endl;

    for (auto addr=start; (addr<end); addr+=1)
    {
        if (memory[addr]!=0){

std::cout<<std::hex<<std::setw(8)<<std::setfill('0')<<addr<< ":
";

std::cout<<std::hex<<std::setw(2)<<std::setfill('0')<<static_cast
<int>(memory[addr])<<" "<<std::endl;}

    }
std::cout<<"-----------------------------------------------------
----------"<< std::endl;
}
```

Αν και ονομάζεται dump_memory_block, έχει σκόπιμα τροποποιηθεί για να αναζητήσει ΟΛΗ τη μνήμη για μη μηδενικές τιμές για καλύτερη επαλήθευση των περιεχομένων της μνήμης.

Ο Επεξεργαστής

Ο επεξεργαστής ή καλύτερα ο μικροεπεξεργαστής (πυρήνας επεξεργασίας) όπως έχει ήδη εξηγηθεί πολλές φορές είναι το πιο βασικό κομμάτι ενός Η/Υ, αποκωδικοποιεί και εκτελεί τις εντολές που του δίνονται κατά επανάληψη μέχρι να σταματήσει να τροφοδοτείται από ηλεκτρικό ρεύμα. Ο επεξεργαστής δεν μπορεί να διακόψει τη λειτουργία του και πάντα εκτελεί κάτι. Μαζί με κάποια άλλα συγκεκριμένα κυκλώματα, όπου θα αναλυθούν ένα-ένα παρακάτω, αποτελούν την Κεντρική Μονάδα Επεξεργασίας (ΚΜΕ/CPU).

Πριν προχωρήσουμε στην ανάλυση της κλάσης Processor, ας αναφέρουμε κάποια από τα κυκλώματα τα οποία έχουν προσομοιωθεί και είναι απαραίτητα για μία βασική ΚΜΕ:
Register File/Φάκελος Καταχωρητών
Controller/Ελεγκτής
Arithmetic Logical Unit (ALU)/Αριθμητική Λογική Μονάδα (ΑΛΜ)
ALU Controller/Ελεγκτής της ΑΛΜ
Co-processor 0/Συνεπεξεργαστής 0
Instruction Cache (Level 1)/Κρυφή Μνήμη Εντολών (Επιπέδου 1)
Data Cache (L1)/Κρυφή Μνήμη Δεδομένων (Επιπέδου 1)
Shared Cache (L2)/Κοινή (για εντολές και δεδομένα) Κρυφή Μνήμη (Ε2)
Hazard Detection Unit/Μονάδα Εντοπισμού Κινδύνων
Forwarding Unit/Μονάδα Προώθησης (δεδομένων)
Memory Management Unit (MMU)/Μονάδα Διαχείρισης Μνήμης
Translation Lookaside Buffer (TLB)/Ενδιάμεση Μνήμη Μετάφρασης Διευθύνσεων

Ο επεξεργαστής (πυρήνας) που σχεδιάστηκε έχει ενσωματωμένο pipelining μηχανισμό (σωλήνωση), εξ' ου και η ανάγκη hazard detection unit και forwarding unit, και για αυτό τον λόγο είναι απαραίτητη η χρήση κάποιων βοηθητικών καταχωρητών, αποκλειστικά για την συνεργασία, των 5 σταδίων του pipeline. Ειδικότερα, μια υλοποίηση βασισμένη στον 32-bit MIPS όπου ακολουθεί την RISC αρχιτεκτονική, όπως έχει ήδη εξηγηθεί.

```
class Processor
{
private:
    // Pipeline register structures
    struct IF_ID
    {
        Word instruction;     // The fetched instruction
        Word pc_plus_4;       // PC + 4 value (for branch/jump
calculations)
    };

    struct ID_EX
    {
        // Control signals
        bool reg_dst;         // Determines whether rd or rt is
the destination register
        bool alu_src;         // Selects ALU second operand (reg
or immediate)
        Word alu_op;          // ALU operation to perform
        bool branch;          // Branch instruction flag
        bool mem_read;        // Memory read flag
        bool mem_write;       // Memory write flag
```

```
        bool reg_write;       // Register write-back flag
        bool mem_to_reg;      // Selects data for register write-
back (ALU result or memory data)
        bool jump;            // Jump instruction flag
        bool is_reg_jump=false; // Register jump
        bool link=false;      // JR, JALR
        bool link_reg=false; // rd or $ra for register write (JR,
JALR)

        // Data values
        Word opcode;          // Opcode for memory operations
        Word pc_plus_4;       // PC + 4 value
        Word read_data_1;     // Value read from rs register
        Word read_data_2;     // Value read from rt register
        Word immediate;       // Sign-extended immediate value
        Word rs;              // Source register number (rs)
        Word rt;              // Source/destination register
number (rt)
        Word rd;              // Destination register number (rd)
        Word jump_target;     // Jump target address
        Byte shamt;           // Shift amount (for shift
instructions)
        Byte funct;           // Function code (for R-type
instructions)
    };

    struct EX_MEM
    {
        // Control signals
        bool branch;          // Branch instruction flag
        bool mem_read;        // Memory read flag
        bool mem_write;       // Memory write flag
        bool reg_write;       // Register write-back flag
        bool mem_to_reg;      // Selects data for register write-
back
        bool negative;        // For sign checks (Branches)
        bool is_reg_jump=false; // Register jump
        bool link=false;      // JR, JALR
        bool link_reg=false; // rd or $ra for register write (JR,
JALR)
        bool forward_mem_needed=false; // New field for memory
forwarding

        // Data values
        bool zero;            // ALU zero flag (for branch
decisions)
        Word alu_result;      // ALU operation result
        Word branch_target;   // Branch target address
```

```
        Word read_data_2;     // Value to write to memory
        Word reg_dst;         // Destination register
        Word pc_plus_4;
        Word opcode; // For branching
        Word rt; // For REGIMM
        Word rs;
        Word rd;
        Word funct;
        Word jump_target;
        Word link_value; // JALR, JR

        // Exceptions
        bool exception;
        Byte exception_code;
    };

    struct MEM_WB
    {
        // Control signals
        bool reg_write;       // Register write-back flag
        bool mem_to_reg;      // Selects data for register write-
back

        // Data values
        Word read_data;       // Data read from memory
        Word alu_result;      // ALU operation result
        Word reg_dst;         // Destination register
    };

    // Instantiate the Pipeline Registers
    IF_ID if_id{};
    ID_EX id_ex{};
    EX_MEM ex_mem{};
    MEM_WB mem_wb{};

    // Special Registers
    Word hi_reg;
    Word lo_reg;

    // Component Instances
    Coprocessor0 coprocessor0; // The coprocessor0 instance
    RegisterFile register_file; // Create the register file
    Controller controller; // Main Control Unit
    ALUControl alu_control; // ALU Control Unit
    ALU alu; // Arithmetic Logical Unit
    ICache i_cache;
    DCache d_cache;
    SharedCache l2_cache;
```

```
    HazardDetection hazard_detection;
    ForwardingUnit forwarding_unit;
    CacheDecryptionUnit cdu;

    // Members
    //Word instruction; // The current instruction being run
    Word PC; // Program Counter Register

    bool stall_pipeline=false; // Check for stalls
    bool flush_pipeline=false; // Flush flag
    bool branch_taken=false; // Check for issued branches
    Word branch_target=0; // Branch's target memory value

    // FOR DEBUGGING
    bool debug_mode=true; //ENABLED!

    // Methods
    void Reset(); // Processor's Reset Routine
    void execute_cp0_instruction(Byte opcode, Byte rs, Byte rt,
Byte rd, Byte funct, Word alu_result, Word rt_data, bool&
pipeline_flush, Word& mem_data);

    // Processor's Pipeline routine Methods
    IF_ID IF(void); // Instruction Fetch
    ID_EX ID(const IF_ID& if_id_reg);
    EX_MEM EX(const ID_EX& id_ex_reg);
    MEM_WB MEM(const EX_MEM& ex_mem_reg);
    void WB(const MEM_WB& mem_wb_reg);

public:
    // Constructor
    Processor(Memory& memory);

    // Methods
    void dump_registers(void) const; // Dump the register file
    //Word get_current_instruction(void) const; // Manually get
the instruction that is being run
    //Word set_current_instruction(Word ins); // manually set a
new instruction
    int routine(void); // Run the processor's routine

    // Cache Operations
    void invalidate_icache(); // Invalidate instruction cache
    void invalidate_dcache(); // Invalidate data cache
    void invalidate_shared_cache(); // Invalidate shared cache
    void invalidate_cdu();
    void invalidate_all_caches();
```

```
    void flush_icache(); // Flush instruction cache
    void flush_dcache(); // Flush data cache
    void flush_shared_cache(); // Flush shared cache
    void flush_cdu();
    void flush_all_caches(); // Flush both caches

    void print_cache_stats() const; // Print cache statistics
    void dump_caches() const;
    void reset_cache_stats();

    void dump_tlb() const;
};
```

Το CDU αφορά το κεφάλαιο της κρυπτογράφησης.

Στο παραπάνω κομμάτι κώδικα βλέπουμε αναλυτικά τα στοιχεία της κλάσης του επεξεργαστή. Παρατηρούμε τα περιεχόμενα των pipeline registers για κάθε μετάβαση από ένα στάδιο της σωλήνωσης στο άλλο. Επίσης, διακρίνουμε την εγκατάσταση των μονάδων που είναι απαραίτητες πάνω στο τσιπ. Τέλος, υπάρχει ο ορισμός των αναγκαίων ιδωτικών και δημοσίων μεθόδων με την πιο σημαντική μέθοδο να είναι η routine() όπου χρησιμοποιεί η κλάση Computer.

```
// Processor constructor
Processor::Processor(Memory&
memory):cdu(&memory),l2_cache(&cdu),i_cache(&coprocessor0,&l2_cac
he),d_cache(&coprocessor0,&l2_cache)
{
    debug_mode=true;
    Reset();
    std::cout<<"Processor instance created
successfully!"<<std::endl;
}
```

Ο κατασκευαστής του αντικειμένου Processor που καλεί ο Computer, απλά «ενεργοποιεί» τις υπόλοιπες μονάδες και περνάει δείκτες για τις αναγκαίες αναφορές. Επίσης καλεί την Reset() μέθοδο. Ας την κοιτάξουμε:

```
// Processor's Reset Routine
void Processor::Reset()
{
    PC=MemorySegments::RESET_VECTOR; // Initialize the PC
register to the RESET_VECTOR

    hi_reg=0;           // Reset HI register
    lo_reg=0;           // Reset LO register

    // Reset pipeline control flags
    stall_pipeline=false;
    flush_pipeline=true;
    branch_taken=false;
    branch_target=0;
```

```
    // Invalidate Cache
    invalidate_all_caches();

    // Reset cache statistics
    reset_cache_stats();

    // Επίσης σε αυτό το σημείο έχει τοποθετηθεί (και δεν έχει
αντιγραφεί εδώ) μία μέθοδος σημαντική για την κρυπτογράφηση που
όμως θα αναλυθεί στο επόμενο κεφάλαιο

    std::cout<<"Processor reset complete!"<<std::endl;
}
```

Η μέθοδος Reset() είναι υπεύθυνη για την επαναφορά του επεξεργαστή και καλείται με την ενεργοποίηση του H/Y. Καταχωρίζει στον ειδικό καταχωρητή, PC (Program Counter/Μετρητής Προγράμματος), την διεύθυνση όπου θα βρει την πρώτη εντολή που θα τρέξει ο επεξεργαστής. Επίσης θέτει τις κατάλληλες τιμές για τις σημαίες ελέγχου, όπως τη flush_pipeline σε true/αληθές για να εξασφαλισθεί ο καθαρισμός των καταχωρητών σωλήνωσης (συνήθως μηδενισμός των σημάτων ελέγχου/σαν να εισάγεται μία NOP εντολή). Επίσης, καθαρίζει τις κρυφές μνήμες και μηδενίζει κάποιες μετρικές στατιστικών στοιχείων.

Ας μελετήσουμε την ρουτίνα του επεξεργαστή που ξεκαθαρίζει την RISC αρχιτεκτονική.

```
// The processor's routine
int Processor::routine(void)
{
    // // Check for interrupts
    // if (coprocessor0.handle_interrupts())
    // {
    //     // Interrupt was handled, pipeline was flushed, PC
updated
    //     branch_taken=true;
    //     branch_target=coprocessor0.get_exception_pc();
    //     flush_pipeline=true;

    //     if (debug_mode)
    //     {
    //         std::cout<<"INTERRUPT: Handled interrupt, jumping
to exception vector"<<std::endl;
    //     }
    // }

    // Check for hazards before pipeline execution
    HazardDetection::HazardResult
hazard_result=hazard_detection.detect
    (
        id_ex.mem_read,             // Load instruction in EX
stage
```

```
        id_ex.rt,                    // RT register of load
instruction
        if_id.instruction>>21&0x1F, // RS register from current
instruction in ID stage
        if_id.instruction>>16&0x1F, // RT register from current
instruction in ID stage
        id_ex.branch,                // Branch instruction in EX
stage
        ex_mem.branch,               // Branch instruction in MEM
stage
        id_ex.jump,                  // Jump instruction in EX
stage
        mem_wb.mem_to_reg,             // Load instruction in WB
stage (using mem_to_reg as indicator)
        mem_wb.reg_dst,              // Destination register in
WB stage
        ex_mem.mem_write,            // Store instruction in MEM
stage
        id_ex.mem_write              // Store instruction in EX
stage
    );

    // Apply hazard detection results
    if (hazard_result.stall_needed)
    {
        stall_pipeline=true;
        if (debug_mode) {std::cout<<"HAZARD: Stall needed -
inserting bubble"<<std::endl;}
    }
    else
    {
        // If no new hazard was detected in this cycle reset the
stall condition
        stall_pipeline=false;
    }

    if (hazard_result.flush_needed)
    {
        flush_pipeline=true;
        if (debug_mode) {std::cout<<"HAZARD: Flush needed -
clearing pipeline"<<std::endl;}
    }

    // If pipeline is to be flushed clear the stages
    if (flush_pipeline)
    {
        // Clear pipeline registers with NOPs or intialize to
default values
```

```
        mem_wb=MEM_WB{};
        ex_mem=EX_MEM{};
        id_ex=ID_EX{};
        if_id=IF_ID{};

        flush_pipeline=false; // Reset
    }
    else // Normal execution
    {
        Processor::WB(mem_wb);
        mem_wb=Processor::MEM(ex_mem);
        ex_mem=Processor::EX(id_ex);

        // Only update ID if no stall is detected
        if (!stall_pipeline) {id_ex=Processor::ID(if_id);}
        else // Stall Detected!
        {
            // Reset signals to default/0/NOP
            id_ex=ID_EX{};

            // Insert NOP in ID/EX stage by clearing control
signals
            //id_ex.reg_dst=false;
            //id_ex.alu_src=false;
            //id_ex.mem_to_reg=false;
            //id_ex.reg_write=false;
            //id_ex.mem_read=false;
            //id_ex.mem_write=false;
            //id_ex.branch=false;
            //id_ex.jump=false;
            // Keep the other fields for debugging
        }
    }

    // Update IF if no stall
    if (!stall_pipeline) {if_id=Processor::IF();}

    // Return
    return 0; // If all OK return code 0;
}
```

Η αλήθεια είναι πως στην τελική της μορφή, με την εισαγωγή των μηχανισμών εντοπισμού κινδύνων και προώθησης, παγωμάτων κύκλων και επαναφοράς (flush), τα πράγματα περιπλέκονται, οπότε ας επικεντρωθούμε στο πιο σημαντικό σημείο και ας συμμαζέψουμε τη διαδικασία της σωλήνωσης (pipelining):
if_id=Processor::IF();
id_ex=Processor::ID(if_id);
ex_mem=Processor::EX(id_ex);

mem_wb=Processor::MEM(ex_mem);
Processor::WB(mem_wb);
που μάλιστα στον κώδικα είναι ανεστραμμένα λόγω της ροής των δεδομένων αφού η WB χρησιμοποιεί το αποτέλεσμα της MEM και η MEM φορτώνεται με τα επόμενα κ.ο.κ.

Παρατηρούμε λοιπόν κλήσεις για 5 μεθόδους, τα 5 στάδια του pipeline που αποθηκεύουν τα αποτελέσματα τους στους καταχωρητές σωλήνωσης. Ας εξηγήσουμε λοιπόν πολύ απλά πώς λειτουργεί το pipelining και άρα, την βασική αρχή RISC επεξεργαστών. Αντί να επεξεργάζεται ο υπολογιστής μία εντολή την φορά, οι επιστήμονες ανακάλυψαν ότι μπορούν να εξοικονομήσουν αρκετό χρόνο και να αυξήσουν σημαντικά την απόδοση του επεξεργαστή, με την μέθοδο του pipelining. Κατάφεραν να χωρίσουν ομοιόμορφα σε 5 διακριτά στάδια την επεξεργασία της κάθε εντολής, όπου το κάθε στάδιο ενεργεί σε έναν κύκλο μηχανής, με αποτέλεσμα ουσιαστικά να τρέχουν 5 εντολές ταυτόχρονα, αφού σε κάθε κύκλο, κάθε εντολή, βρίσκεται σε διαφορετικό στάδιο. Βέβαια, αυτό δημιουργεί κάποιους κινδύνους, εξ' ου και η ανάγκη μηχανισμών όπως Hazard Detection και Forwarding, καθώς ενδέχεται η μία εντολή να εξαρτάται από το αποτέλεσμα της προηγούμενης, και εφ' όσον δεν έχει ολοκληρωθεί η επεξεργασία της, τότε μπορούν να προκύψουν προβλήματα.

Instruction Fetch

```
Processor::IF_ID Processor::IF(void) // Instruction Fetch
{
    IF_ID result; // Create a local pipeline instance

    // Check if there is a stall condition
    if (stall_pipeline) {return if_id;} // Do not update anything

    // Check if there is a branch that was resolved in a previous
cycle and update the PC
    if (branch_taken)
    {
        PC=branch_target;
        branch_taken=false; // Reset the flag
    }

    // Fetch the new instruction
    result.instruction=i_cache.readInstruction(PC);

    // Update the counter and also save it for later stages
    PC=result.pc_plus_4=PC+sizeof(Word);

    // Debugging
    if (debug_mode)
    {
        std::cout<<"IF: Fetched Instruction
0x"<<std::setw(8)<<std::setfill('0')<<std::hex<<result.instructio
n<<" from PC=0x"<<PC-4<<std::endl;
    }
```

```
    return result;
}
```

Το Instruction Fetch ή το στάδιο Φόρτωσης Εντολής, είναι πολύ απλό. Ο επεξεργαστής φέρνει την εντολή από τη διεύθυνση που δείχνει ο PC. Συγκεκριμένα, λόγω της ιεραρχίας των κρυφών μνημών, το αίτημα το λαμβάνει η L1 Instruction Cache και η εντολή έρχεται από εκεί. Παράλληλα, ο PC ενημερώνεται με την επόμενη διεύθυνση, δηλαδή αυτή που δείχνει στην αρχή, το πρώτο bit, της επόμενης εντολής, δηλαδή + 4 byte, γιατί μιλάμε για αρχιτεκτονική 32-bit εντολών και 32/4=8 bit, δηλαδή μία μονάδα byte.

Instruction Decode

```
Processor::ID_EX Processor::ID(const IF_ID& if_id_reg) // Local
if_id struct for each instruction
{
    ID_EX result; // Create a local pipeline instance

    // Check for stall condition
    if (stall_pipeline) {return id_ex;} // Return the previous
values

    // Get the instruction
    Word instr=if_id_reg.instruction;

    // Separate and extract ALL the instruction fields
    Byte opcode=(instr>>26)&0x3F; // 31-26
    Byte rs=(instr>>21)&0x1F; // 25-21
    Byte rt=(instr>>16)&0x1F; // 20-16
    Byte rd=(instr>>11)&0x1F; // 15-11
    Byte shamt=(instr>>6)&0x1F; // 10-6
    Byte funct=instr&0x3F; // 5-0

    Word immediate=instr&0xFFFF; // 15-0

    // Sign-extend the immediate value to 32 bits
    if (immediate&0x8000) {immediate|=0xFFFF0000;} // If
negative; extend

    // Calculate the jump target
    Word jump_target=((if_id_reg.pc_plus_4-
4)&0xF0000000)|((instr&0x3FFFFFF)<<2);

    // Get the control signals from the controller unit
    Controller::ControlSignals
control_signals=controller.decode(opcode,rs,rt);
```

```
    // Check for ERET CP0 instruction it can cause a pipeline
flush
    //if ((funct)==FUNCT_ERET) {signals.eret=true;} // Signal for
ERET handling

    // Set the appropriate control signals based on the
controller's output
    result.reg_dst=control_signals.reg_dst;
    result.alu_src=control_signals.alu_src;
    result.mem_to_reg=control_signals.mem_to_reg;
    result.reg_write=control_signals.reg_write;
    result.mem_read=control_signals.mem_read;
    result.mem_write=control_signals.mem_write;
    result.branch=control_signals.branch;
    result.jump=control_signals.jump;
    result.alu_op=control_signals.alu_op;

    // Read the registers' values
    result.read_data_1=register_file.read_register(rs);
    result.read_data_2=register_file.read_register(rt);

    // Pass values to be used in other stages
    result.opcode=opcode;
    result.pc_plus_4=if_id_reg.pc_plus_4;
    result.immediate=immediate;
    result.rs=rs;
    result.rt=rt;
    result.rd=rd;
    result.jump_target=jump_target;
    result.shamt=shamt;
    result.funct=funct;

    // Handle jump instruction in this stage
    if (control_signals.jump)
    {
        branch_taken=true;
        branch_target=jump_target;
        // For jal save to $ra (31)
        if (opcode==OP_JAL)
{register_file.write_register(31,if_id_reg.pc_plus_4);}
    }

    // Identify the instruction type and read registers for JR
and JALR
    if (opcode==OP_RTYPE&&(funct==FUNCT_JR||funct==FUNCT_JALR))
    {
        // Set a special control signal for register jumps
        result.is_reg_jump=true;
```

```
        // For JALR, also set up the link register write
        if (funct==FUNCT_JALR)
        {
            result.link=true;
            result.link_reg=(rd!=0)?rd:31; // Use rd if
specified, else $ra
        }
    }

    // Handle link register for branch instructions CHECK!
    if (control_signals.branch && control_signals.link)
{register_file.write_register(31, if_id_reg.pc_plus_4);} // Save
return address to $31 (ra)


    // For debugging
    if (debug_mode)
    {
        std::cout<<"ID: Decoded instruction:
opcode=0x"<<std::hex<< static_cast<int>(opcode);
        if (opcode==0) {std::cout<<",
funct=0x"<<static_cast<int>(funct);}
        if (opcode==0) {std::cout<<",
shamt=0x"<<static_cast<int>(shamt);}
        std::cout<<", immediate=0x"<<static_cast<int>(immediate);
        std::cout<<",
rs=$"<<std::dec<<static_cast<int>(rs)<<"("<<result.read_data_1<<"
)";
        std::cout<<",
rt=$"<<static_cast<int>(rt)<<"("<<result.read_data_2<<")";
        if (opcode==0) {std::cout<<",
rd=$"<<static_cast<int>(rd);}
        std::cout<<std::endl;
    }

    return result;
}
```

Το στάδιο του Instruction Decoding ή Αποκωδικοποίηση Εντολής, έχει έναν βασικό σκοπό, να αναγνωρίσει την εντολή όπου περνάει. Ίσως είναι το κατάλληλο σημείο να ορίσουμε τί είναι μία εντολή.

Instruction (Εντολή): Μία ακολουθία από bit, συγκεκριμένα 32-bit για την συγκεκριμένη αρχιτεκτονική, όπου προσδιορίζουν, σαφώς, τα βήματα που θα ακολουθήσει ο επεξεργαστής κατά την επεξεργασία της. Υπάρχουν 3 βασικά είδη εντολών στο ISA (Instruction Set Architecture/Αρχιτεκτονική Συνόλου Εντολών) του MIPS: Register Type (R-type, αφορά χειρισμό καταχωρητών), Immediate Type (I-type, αφορά υπολογισμούς κυρίως με άμεσα τελούμενα) και Jump Type (J-type, ειδικές για άλματα). Το ερώτημα είναι πώς θα καταλάβει

ο επεξεργαστής τι τύπου εντολή είναι καθώς και ποια εντολή είναι; Την ευθύνη αυτή την έχει το στάδιο της αποκωδικοποίησης εντολών. Τα 32-bit χωρίζονται σε κάποια πεδία (bit fields) και το ποιο σημαντικό για την αναγνώριση του τύπου της, είναι το Operation Code field (πεδίο Κώδικα Λειτουργίας) ή opcode. Αφορά τα πρώτα MSB (Most Significant Bit/Πιο Σημαντικά Bit) 5 bit, δηλαδή στο πλαίσιο 31-26 (καθώς μετράμε από το 0). Τα υπόλοιπα bit, αναλόγως την εντολή αφορούν την επιλογή καταχωρητών, άμεσα τελούμενα (ωμοί αριθμοί), διευθύνσεις μνήμης, καθώς και το funct (function/λειτουργία) πεδίο που χρησιμοποιείται για την αναγνώριση R-type εντολών. Για παράδειγμα, η εντολή add $t0,$t1,$t2, όπου προσθέτει τις τιμές που κρατάνε, εκείνη τη στιγμή, οι καταχωρητές $t1 και $t2 μεταξύ τους και αποθηκεύει το αποτέλεσμα στον καταχωρητή $t0, μεταφράζεται από τον συμβολομεταφραστής σε γλώσσα μηχανής ως 000000 01001 01010 01000 00000 100000 (0x12A4020 σε δεκαεξαδικό) και αναλύεται ως εξής:
opcode: 00000=R-type
rs: 01001=$t1=Καταχωρητής 9
rt: 01010=$t2=Καταχωρητής 10
rd: 01000=$t0=Καταχωρητής 8
shamt: Αδιάφορο (σύμφωνα με το εγχειρίδιο του MIPS, πρέπει να είναι αυστηρά 0 όταν δεν χρησιμοποιείται αλλιώς η εντολή θεωρείται αλλοιωμένη)
funct: 100000=add, για mult θα ήταν 011000 αλλά ο opcode 0 γιατί είναι R-type.

Οπότε, όπως καταλαβαίνουμε και από τον ίδιο τον κώδικα, η βασική λειτουργία αυτής της μεθόδου είναι ναι τεμαχίζει την εντολή σε όλα τα πιθανά πεδία, κάνει κάποιους απαραίτητους ελέγχους (ειδικότερα γιατί υποστηρίζονται και κάποιες ειδικές εντολές) και κάποιες μικρές προετοιμασίες και τα στέλνει τον κύριο ελεγκτή για να επιλέξει με ποια εντολή ταιριάζει για να θέσει τα κατάλληλα σήματα ελέγχου και να αποθήκευση προσωρινά τιμές που θα χρειαστούν, για να περάσουν μέσω των καταχωρητών σωλήνωσης σε όλα τα στάδια και να επεξεργαστεί πλήρως και με επιτυχία η εντολή. Τα σήματα ελέγχου είναι όλη η ουσία καθώς χειρίζονται διάφορους πολυπλέκτες (multiplexers) για να «δείξουν» τον δρόμο όπου τα δεδομένα της εντολής πρέπει να περάσουν καθώς και να ενεργοποιήσουν/απενεργοποιήσουν διάφορες λειτουργίες (π.χ. αποθήκευση στην Κύρια Μνήμη, προσπέλαση καταχωρητών κ.α.).

Execution

```
Processor::EX_MEM Processor::EX(const ID_EX& id_ex_reg)
{
    EX_MEM result; // Local pipeline register instance

    // Check for stall condition
    if (stall_pipeline) {return ex_mem;} // Return the previous
values

    // Pass control signals for the next stage
    result.branch=id_ex_reg.branch;
    result.mem_read=id_ex_reg.mem_read;
    result.mem_write=id_ex_reg.mem_write;
    result.reg_write=id_ex_reg.reg_write;
    result.mem_to_reg=id_ex_reg.mem_to_reg;
    result.pc_plus_4=id_ex_reg.pc_plus_4;
    result.rt=id_ex_reg.rt; // For REGIMM
    // For CP0
    result.rs=id_ex_reg.rs;
```

```
    result.rd=id_ex_reg.rd;
    result.funct=id_ex_reg.funct;

    // Pass the opcode for branches
    result.opcode=id_ex_reg.opcode;

    // Forwarding logic
    // Detect forwarding conditions using the forwarding unit
    ForwardingUnit::ForwardingResult
forwarding_result=forwarding_unit.detect
    (
        id_ex_reg.rs,              // RS register in EX stage
        id_ex_reg.rt,              // RT register in EX stage
        ex_mem.reg_dst,            // Destination register in MEM
stage
        ex_mem.reg_write,          // Register write in MEM stage
        mem_wb.reg_dst,            // Destination register in WB
stage
        mem_wb.reg_write,          // Register write in WB stage
        ex_mem.mem_read,           // Memory read in MEM stage
        ex_mem.mem_write,          // Memory write in MEM stage
        id_ex_reg.mem_write        // Memory write in EX stage
    );

    // Apply forwarding for first operand (RS)
    Word forwarded_read_data_1=id_ex_reg.read_data_1;
    if
(forwarding_result.forward_a==ForwardingUnit::FORWARD_FROM_EX_MEM
)
    {
        forwarded_read_data_1=ex_mem.alu_result;
        if (debug_mode) {std::cout<<"FORWARDING: RS data from
EX/MEM to RS=$"<<id_ex_reg.rs<<std::endl;}
    }
    else if
(forwarding_result.forward_a==ForwardingUnit::FORWARD_FROM_MEM_WB
)
    {

forwarded_read_data_1=mem_wb.mem_to_reg?mem_wb.read_data:mem_wb.a
lu_result;
        if (debug_mode) {std::cout<<"FORWARDING: RS data from
MEM/WB to RS=$"<<id_ex_reg.rs<<std::endl;}
    }

    // Apply forwarding for second operand (RT)
    Word forwarded_read_data_2=id_ex_reg.read_data_2;
```

```
    if
(forwarding_result.forward_b==ForwardingUnit::FORWARD_FROM_EX_MEM
)
    {
        forwarded_read_data_2=ex_mem.alu_result;
        if (debug_mode) {std::cout<<"FORWARDING: RT data from
EX/MEM to RT=$"<<id_ex_reg.rt<<std::endl;}
    }
    else if
(forwarding_result.forward_b==ForwardingUnit::FORWARD_FROM_MEM_WB
)
    {

forwarded_read_data_2=mem_wb.mem_to_reg?mem_wb.read_data:mem_wb.a
lu_result;
        if (debug_mode) {std::cout<<"FORWARDING: RT data from
MEM/WB to RT=$"<<id_ex_reg.rt<<std::endl;}
    }

    // Memory-to-memory forwarding for store followed by store
    if
(forwarding_result.forward_mem==ForwardingUnit::FORWARD_FROM_MEM_
TO_MEM)
    {
        // Handle memory-to-memory forwarding for store
instructions
        // This will be used by the MEM stage
        result.forward_mem_needed=true;
        if (debug_mode) {std::cout<<"FORWARDING: Memory-to-memory
forwarding for store instructions"<<std::endl;}
    }
    // End of forwarding logic

    // Determine the second ALU operand (register or immediate
value)
    Word
alu_operand_2=id_ex_reg.alu_src?id_ex_reg.immediate:forwarded_rea
d_data_2; // Use forwarded value

    // Generate the ALU control signal from the ALU controller
    Byte
alu_control_signal=alu_control.generate(id_ex_reg.opcode,
id_ex_reg.alu_op, id_ex_reg.rs, id_ex_reg.rt, id_ex_reg.funct);

    // Execute the ALU operation using the generated
alu_control_signal
```

```
    ALU::Result
alu_result=alu.execute(forwarded_read_data_1,alu_operand_2,alu_co
ntrol_signal,id_ex_reg.shamt);

    // Store the ALU result
    result.alu_result=alu_result.value;
    result.zero=alu_result.zero; // Zero flag
    result.negative=((alu_result.value&0x80000000)!=0); //
Negative flag (Branches)


    // Pass exceptions
    result.exception=alu_result.exception;
    result.exception_code=alu_result.exception_code;

    // Update HI/LO registers
    if
(alu_control_signal==ALU_MULT||alu_control_signal==ALU_DIV)
{hi_reg=alu_result.hi; lo_reg=alu_result.lo;}
    // For MFHI
    if (id_ex_reg.opcode==OP_RTYPE&&id_ex_reg.funct==FUNCT_MFHI)
{result.alu_result=hi_reg;std::cout<<"MFHI
Detected!"<<std::endl;}
    // For MFLO
    if (id_ex_reg.opcode==OP_RTYPE&&id_ex_reg.funct==FUNCT_MFLO)
{result.alu_result=lo_reg;std::cout<<"MFLO
Detected!"<<std::endl;}
    // For MTHI
    if (id_ex_reg.opcode==OP_RTYPE&&id_ex_reg.funct==FUNCT_MTHI)
{hi_reg=forwarded_read_data_1;}
    // For MTLO
    if (id_ex_reg.opcode==OP_RTYPE&&id_ex_reg.funct==FUNCT_MTLO)
{lo_reg=forwarded_read_data_1;}

    if (id_ex_reg.is_reg_jump)
    {
        // Jump target is the value in rs (with any forwarding
applied)
        result.jump_target=forwarded_read_data_1;
        result.is_reg_jump=true; // Pass to MEM stage

        // Handle link register (similar to JAL) (JALR, JR)
        if (id_ex_reg.link)
        {
            result.link=true;
            result.link_reg=id_ex_reg.link_reg;
            result.link_value=id_ex_reg.pc_plus_4;
        }
    }
```

```
    // Calculate branch target address

result.branch_target=id_ex_reg.pc_plus_4+(id_ex_reg.immediate<<2)
; // (Sign-extended) Shift 2 bits for the offset

    // Pass the data to the MEM stage
    result.read_data_2=forwarded_read_data_2; // Use the
forwarded value

    // Determine the destination register (rd or rt)
    result.reg_dst=id_ex_reg.reg_dst?id_ex_reg.rd:id_ex_reg.rt;!

    // Debug output
    if (debug_mode)
    {
        std::cout<<"EX: ALU operation with
alu_control_signal="<<std::hex<<static_cast<int>(alu_control_sign
al)
                 <<", result=0x"<<result.alu_result
                 <<", zero="<<result.zero
                 <<", branch_target=0x"<<result.branch_target
                 <<", destination
register=$"<<std::dec<<result.reg_dst
                 <<std::endl;
    }

    return result;
}
```

Το στάδιο της εκτέλεσης (execution stage) είναι ουσιαστικά το στάδιο όπου γίνονται οι υπολογισμοί και αυτό φαίνεται και με την χρήση μεθόδων από τις κλάσεις της ALU και του ελεγκτή της. Διακρίνουμε επίσης την χρήση μηχανισμών εντοπισμού κινδύνων και μηχανισμών προώθησης δεδομένων, όπου σε συνεργασία με εντολές στην μέθοδο της ρουτίνας καταπολεμούν επιτυχώς αρκετούς κινδύνους και εισαγάγουν όπου χρειαστεί παγώματα (stalls), φούσκες (bubbles) ή και επαναφέρουν (flush) τα περιεχόμενα των pipeline register.

Memory

```
Processor::MEM_WB Processor::MEM(const EX_MEM& ex_mem_reg)
{
    MEM_WB result; // Local pipeline instance struct to pass the
signals

    // Check for stall condition
    if (stall_pipeline) return mem_wb; // Return the previous
values
```

```
    // Pass control signals to the next state
    result.reg_write=ex_mem_reg.reg_write;
    result.mem_to_reg=ex_mem_reg.mem_to_reg;

    // Pass ALU result
    result.alu_result=ex_mem_reg.alu_result;

    // Pass the destination register
    result.reg_dst=ex_mem_reg.reg_dst;

    Word read_success=-1;
    // Memory Read
    if (ex_mem_reg.mem_read)
    {
        read_success=d_cache.readWord(ex_mem_reg.alu_result);
        if (read_success==-1) result.read_data=0; // Read failed
        else result.read_data=read_success;
        // // Translate virtual to physical address with access
size=4 (word)
        // Word
physical_addr=coprocessor0.translate_address(ex_mem_reg.alu_resul
t,false,4);

        // // If translation resulted in an exception
        // if (physical_addr==0)
        // {
        //     std::cout<<"MEM: Translation failed!
Continuing..."<<std::endl;
        //     return result; // Exception handled by CP0 //
CHECK THE RETURN, SHOULD I RETURN RESULT?
        // }

        // // Read 4 bytes from physical memory and combine
        // Word value=0;
        //
value|=(static_cast<Word>(memory.read_physical_byte(physical_addr
))<<0);
        //
value|=(static_cast<Word>(memory.read_physical_byte(physical_addr
+1))<<8);
        //
value|=(static_cast<Word>(memory.read_physical_byte(physical_addr
+2))<<16);
        //
value|=(static_cast<Word>(memory.read_physical_byte(physical_addr
+3))<<24);
```

```
        // result.read_data=value;
        // // CHECK FOR ERROR?
    }

    int write_success=-1;
    // Memory Write with Forwarding
    if (ex_mem_reg.mem_write)

{//memory.write_word(ex_mem_reg.alu_result,ex_mem_reg.read_data_2
);
        // For store instruction forward the data from a previous
load
        Word store_data=ex_mem_reg.read_data_2;

        // // If memory-to-memory forwarding is needed, get the
data from the WB stage
        // if (ex_mem_reg.forward_mem_needed&&mem_wb.mem_to_reg)
// mem_to_reg indicates a load instruction completed in WB stage
        // {
        //     store_data=mem_wb.read_data;
        //     if (debug_mode) {std::cout<<"MEM: Using forwarded
data for store operation"<<std::endl;}
        // }


write_success=d_cache.writeWord(ex_mem_reg.alu_result,store_data)
; // If -1, failed (For debug output)
        // // Translate virtual to physical address with access
size = 4 (word)
        // Word
physical_addr=coprocessor0.translate_address(ex_mem_reg.alu_resul
t,true,4);

        // // If translation resulted in an exception
        // if (physical_addr==0)
        // {
        //     std::cout<<"MEM: Translation failed!
Continuing..."<<std::endl;
        //     return result; // Exception handled by CP0
        // }

        // Word value=store_data;
        // // Write to memory
        // memory.write_physical_byte(physical_addr,
(value>>0)&0xFF);
        //
memory.write_physical_byte(physical_addr+1,(value>>8)&0xFF);
```

```
        //
memory.write_physical_byte(physical_addr+2,(value>>16)&0xFF);
        //
memory.write_physical_byte(physical_addr+3,(value>>24)&0xFF);
    }

    // Handle CP0 instructions
    if (ex_mem_reg.opcode==OP_COP0)
    {
        bool flush_needed=false;
        Word cp0_data=0;

        // Execute the CP0 instruction
        execute_cp0_instruction(ex_mem_reg.opcode, ex_mem_reg.rs,
ex_mem_reg.rt, ex_mem_reg.rd, ex_mem_reg.funct,
ex_mem_reg.alu_result, ex_mem_reg.read_data_2, flush_needed,
cp0_data);

        // If this was MFC0, forward the CP0 data to WB stage
        if (ex_mem_reg.rs==RS_MFC0)
        {
            result.alu_result=cp0_data;
            result.mem_to_reg=false; // Data comes from CP0, not
memory
        }

        // Handle pipeline flush for ERET
        if (flush_needed)
        {
            // Flush IF, ID, EX stages and jump to exception
return address
            branch_taken=true;
            branch_target=coprocessor0.read_register(CP0_EPC);
            flush_pipeline=true;

            if (debug_mode)
            {
                std::cout<<"CP0: ERET caused pipeline flush,
jumping to EPC=0x"<<std::hex<<branch_target<<std::dec<<std::endl;
            }
        }
    }

    // Handle branch instructions //TODO ADD DELAY SLOT
    if (ex_mem_reg.branch)
    {
        bool take_branch=false;
```

```
        // Use opcode and flags to identify the type of branch
        switch (ex_mem_reg.opcode)
        {
            case OP_BEQ:
                take_branch=ex_mem_reg.zero; // Branch if equal
(zero flag set)
                break;

            case OP_BNE:
                take_branch=!ex_mem_reg.zero; // Branch if not
equal (zero flag clear)
                break;

            case OP_BLEZ:
                // Branch if less than or equal to zero
                // For BLEZ: result <= 0 => either zero flag set
or negative flag set
                take_branch=ex_mem_reg.zero||ex_mem_reg.negative;
                break;

            case OP_BGTZ:
                // Branch if greater than zero (neither zero nor
negative)

take_branch=!ex_mem_reg.zero&&!ex_mem_reg.negative;
                break;

            case OP_REGIMM:
                // Handle REGIMM branches (BLTZ, BGEZ, etc.)
                // These require the RT field to determine the
specific branch type
                switch (ex_mem_reg.rt)
                {
                    case RT_BLTZ:
                    case RT_BLTZAL:
                        // Save return address in $31 (ra)

register_file.write_register(31,ex_mem_reg.pc_plus_4);
                        take_branch=ex_mem_reg.negative;
                        break;

                    case RT_BGEZ:
                    case RT_BGEZAL:
                        // Save return address in $31 (ra)

register_file.write_register(31,ex_mem_reg.pc_plus_4);

take_branch=!ex_mem_reg.negative||ex_mem_reg.zero;
```

```
                        break;
                }
                break;

            // Handle branch likely instructions if needed
            case OP_BEQL:
                take_branch=ex_mem_reg.zero;
                if (!take_branch)
                    // If branch not taken, need to nullify delay
slot
                    flush_pipeline=true; // For branch likely,
flush if not taken
                break;

            case OP_BNEL:
                take_branch=!ex_mem_reg.zero;
                if (!take_branch)
                    // If branch not taken, need to nullify delay
slot
                    flush_pipeline=true; // For branch likely,
flush if not taken
                break;

        }

        if (take_branch)
        {
            branch_taken=true;
            branch_target=ex_mem_reg.branch_target;

            // Flush instructions
            flush_pipeline=true;

            if (debug_mode)
            {
                std::cout<<"MEM: Branch taken,
target=0x"<<std::hex
                        <<branch_target<<std::dec<<std::endl;
            }
        }
    }

    // JALR and JR
    if (ex_mem_reg.is_reg_jump)
    {
        // Take the jump
        branch_taken=true;
        branch_target=ex_mem_reg.jump_target;
```

```
        // Handle link register writing (like JAL)
        if (ex_mem_reg.link)
        {

register_file.write_register(ex_mem_reg.link_reg,ex_mem_reg.link_
value);
        }

        // Set flush flag
        flush_pipeline=true;
    }

    // Exception handling from ALU!
    if (ex_mem_reg.exception)
    {
        // Call the exception handler
        coprocessor0.handle_exception(ex_mem_reg.pc_plus_4-
4,ex_mem_reg.exception_code);

        // Update PC to the exception handler
        branch_taken=true;
        branch_target=coprocessor0.get_exception_pc();

        // Flush the pipeline
        flush_pipeline=true;

        if (debug_mode)
        {
            std::cout<<"Exception detected:
code="<<static_cast<int>(ex_mem_reg.exception_code)
                    <<", branching to exception handler at
0x"<<std::hex
                    <<branch_target<<std::dec<<std::endl;
        }
    }

    // Debug output
    if (debug_mode)
    {
        std::cout<<"MEM: ";
        if (ex_mem_reg.mem_read)
        {
            if (read_success==-1)
                std::cout<<"Could not read from memory/DCache at
address 0x"<<std::hex<<ex_mem_reg.alu_result<<"! ";
            else
```

```
                std::cout<<"Read memory[0x"<<std::hex<<
ex_mem_reg.alu_result
                        <<"] = 0x"<<result.read_data<<"; ";
        }
        if (ex_mem_reg.mem_write)
        {
            if (write_success==-1)
                std::cout<<"Could not write to memory/DCache at
address 0x"<<std::hex<<ex_mem_reg.alu_result<<"! ";
            else
                std::cout<<"Wrote 0x"<<std::hex<<
ex_mem_reg.read_data_2
                    <<" to memory[0x"<<ex_mem_reg.alu_result<<
"]; ";
        }
        //std::cout<<"ALU result=0x"<<result.alu_result
        //        <<", destination register=$"<<std::dec<<
result.reg_dst<<std::endl;
        std::cout<<"Passing "
              <<(ex_mem_reg.mem_to_reg?"MEM":"ALU")<<" result to
register $"
              <<std::dec<<result.reg_dst<<std::endl;
    }

    return result;
}
```

Η βασική λειτουργία του σταδίου της μνήμης είναι να γράψει/αποθηκεύσει στην μνήμη. Λόγω της ιεραρχίας μνήμης με τις κρυφές μνήμες όμως βλέπουμε ότι χρησιμοποιούνται μέθοδοι των κρυφών μνημών επιπέδου 1, ειδικότερα για δεδομένα, παρόλα αυτά, έχουν διατηρηθεί, σε μορφή σχολίων, οι προηγούμενες μέθοδοι πρόσβασης στην Κύρια Μνήμη για εκπαιδευτικούς σκοπούς. Επίσης παρατηρούμε ότι η μέθοδος αυτή είναι πιο πολύπλοκη. Αυτό είναι επειδή, αναλόγως με την αρχιτεκτονική λογική που ακολουθείται, συνήθως υποστηρίζει την προηγούμενη της εκτέλεσης. Στη συγκεκριμένη περίπτωση γίνονται έλεγχοι για εντολές που οι ελεγκτές και η ΑΛΜ δεν μπορούν να διαχειριστούν εύκολα, και ειδικότερα για ειδικές εντολές, όπως για άλματα. Επίσης παρατηρούμε ότι σε περίπτωση που οι εντολή αναγνωρίστηκε να αφορά τον συνεπεξεργαστή 0, είναι η κατάλληλη στιγμή να εκτελεστεί μέσω της ειδικής μεθόδου execute_cp0_instruction(). Έχει δε άμεση σχέση και με τον ομαλό χρονισμό του επεξεργαστή. Επίσης είναι και το σημείο όπου γίνεται μεταφορά ελέγχου στον συνεπεξεργαστή 0 σε περίπτωση εξαιρέσεων, όπως syscall.

Πριν συνεχίσουμε στο τελευταίο στάδιο του pipeline, ας δούμε την μέθοδο execute_cp0_instruction():

```
void Processor::execute_cp0_instruction(Byte opcode, Byte rs,
Byte rt, Byte rd, Byte funct, Word alu_result, Word rt_data,
bool& pipeline_flush, Word& mem_data)
{
    if (opcode==OP_COP0) // ADD MORE
```

```
    {
        switch (rs)
        {
            case RS_MFC0:
                // Move From CP0 - read CP0 register
                mem_data=coprocessor0.read_register(rd);
                if (debug_mode)
                {
                    std::cout<<"CP0: MFC0 - Reading
0x"<<std::hex<<mem_data
                             <<" from CP0 register
"<<std::dec<<static_cast<int>(rd)<<std::endl;
                }
                break;

            case RS_MTC0:
                // Move To CP0 - write to CP0 register
                coprocessor0.write_register(rd,rt_data);
                if (debug_mode)
                {
                    std::cout<<"CP0: MTC0 - Writing
0x"<<std::hex<<rt_data
                             <<" to CP0 register
"<<std::dec<<static_cast<int>(rd)<<std::endl;
                }
                break;

            case RS_CO:
                // Coprocessor operations
                switch (funct)
                {
                    case FUNCT_TLBR:
                        coprocessor0.tlb_read();
                        if (debug_mode)
                        {
                            std::cout<<"CP0: TLBR - TLB Read
executed"<<std::endl;
                        }
                        break;

                    case FUNCT_TLBWI:
                        coprocessor0.tlb_write_index();
                        if (debug_mode)
                        {
                            std::cout<<"CP0: TLBWI - TLB Write
Index executed"<<std::endl;
                        }
                        break;
```

```
                    case FUNCT_TLBWR:
                        coprocessor0.tlb_write_random();
                        if (debug_mode)
                        {
                            std::cout<<"CP0: TLBWR - TLB Write
Random executed"<<std::endl;
                        }
                        break;

                    case FUNCT_TLBP:
                        coprocessor0.tlb_probe();
                        if (debug_mode)
                        {
                            std::cout<<"CP0: TLBP - TLB Probe
executed"<<std::endl;
                        }
                        break;

                    case FUNCT_ERET:
                        coprocessor0.eret();
                        pipeline_flush=true; // ERET causes
pipeline flush
                        if (debug_mode)
                        {
                            std::cout<<"CP0: ERET - Exception
Return executed, flushing pipeline"<<std::endl;
                        }
                        break;

                    case FUNCT_WAIT:
//coprocessor0.check_pending_interrupts();
                        if (debug_mode)
                        {
                            std::cout<<"CP0: WAIT - Wait for
interrupt executed"<<std::endl;
                        }
                        break;

                    default:
                        std::cout<<"WARNING: Unknown CP0
coprocessor function: 0x"
                                 <<std::hex<<
static_cast<int>(funct)<<std::dec<<std::endl;
                        break;
                }
                break;
```

```
            default:
                std::cout<<"WARNING: Unknown CP0 rs field: 0x"
                        <<std::hex<<static_cast<int>(rs)<<
std::dec<<std::endl;
                break;
        }
    }
}
```

Ουσιαστικά, ελέγχει για βασικές εντολές που τον αφορούν και συνεργάζεται άμεσα με τις αντίστοιχες μεθόδους του συνεπεξεργαστη0 για τον χειρισμό τους.

Writeback

```
void Processor::WB(const MEM_WB& mem_wb_reg)
{
    // Write back to register
    if (mem_wb_reg.reg_write)
    {
        Word
write_data=mem_wb_reg.mem_to_reg?mem_wb_reg.read_data:mem_wb_reg.
alu_result;// Select data to write

register_file.write_register(mem_wb_reg.reg_dst,write_data);

        if (debug_mode)
        {
            std::cout<<"WB: Write HEX: 0x"<<std::hex<<
write_data<<std::dec<<", DEC: "<<write_data
                      <<" to register
$"<<mem_wb_reg.reg_dst<<std::endl;
        }
    }
}
```

Το writeback ή στάδιο εγγραφής καταχωρητή είναι το τελευταίο και πιο απλό στάδιο του pipeline. Όπως βλέπουμε, σε περίπτωση που το σήμα ελέγχου «reg_write» έχει ενεργοποιηθεί, το αποτέλεσμα από την ALU ή την μνήμη εγγράφεται στον καταχωρητή που έχει ζητηθεί.

Έτσι, ολοκληρώνεται το pipeline του επεξεργαστή μας, βασισμένος στη RISC αρχιτεκτονική.

Τα συμπληρωματικά στοιχεία της ΚΜΕ

Φάκελος Καταχωρητών

Όπως έχει πολλές φορές αναφερθεί, ο επεξεργαστής αποθηκεύει τα δεδομένα όπου επεξεργάζεται στους καταχωρητές του. Υπάρχουν πολλοί τύποι καταχωρητών, αλλά θα επικεντρωθούμε στους 32, General Purpose/Γενικής Χρήσης, καταχωρητές του MIPS. Κάθε καταχωρητής χωράει 32 bit δεδομένων και οργανώνονται στον Φάκελο Καταχωρητών/Register File της ΚΜΕ. Τί είναι όμως, ένας καταχωρητής;

Καταχωρητής

Οι καταχωρητές/registers είναι ένα ενεργητικό ηλεκτρονικό στοιχείο όπου αποθηκεύει ηλεκτρικό φορτίο. Είναι η πιο γρήγορη και μικρή, πτητική μνήμη ενός Η/Υ και συνήθως συμφέρει να χρησιμοποιείται σε μικρές ποσότητες. Βρίσκονται μέσα στους επεξεργαστές και οι αρχιτεκτονικές CISC προσπαθούν να χρησιμοποιούν όσους λιγότερους γίνονται. Ένας καταχωρητής που κρατάει 32 bit δεδομένων, έχει 32 συστοιχίες από Flip Flop κυκλώματα όπου το καθένα κρατάει πληροφορία ενός bit.

Πίσω στον φάκελο καταχωρητών και στην προσομοίωση. Δεν είναι τίποτα άλλο από έναν απλό πινακα 32 θέσεων τύπου Word (λέξης-32 bit).

```
// Header files
#include <array>
#include "data_types.h"

// Class
class RegisterFile
{
    // Friend the Processor class so it can access the private
constructor
    friend class Processor;
private:
    // Constructor
    RegisterFile();

    // Members
    std::array<Word,32> registers; // Register Array (For
performance)

    // Methods
    Word read_register(size_t i) const;
    void write_register(size_t i, Word value);
    void dump_register_file(void) const;
};
```

Έρχεται με 2 βασικές μεθόδους για εγγραφή και διάβασμα και μία για debugging, για να

προσπελάσει όλους τους καταχωρητές. Επίσης, για μία ρεαλιστική προσομοίωση, η κλάση Processor ορίζεται ως φίλια κλάση για τον χειρισμού του καθώς η ΚΜΕ έχει έλεγχο σε αυτόν.

```
// Header files
#include "RegisterFile.h"
#include <iostream>

// Constructor Definition
RegisterFile::RegisterFile()
{
    // Initialize all registers' values to 0
    registers.fill(0);
    // Print a message
    std::cout<<"Register File created successfully!"<<std::endl;
}

// Methods Definitions
// Read the value inside the register being indexed
Word RegisterFile::read_register(size_t i) const
{
    return registers[i];
}

// Write to register
void RegisterFile::write_register(size_t i, Word value)
{
    if (i==0) std::cout<<"Write request to Register $0 detected,
ignoring... Value: "<<value<<std::endl; // Cannot write to
register $0
    else registers[i]=value;
}

// Dump the whole Register File (For Debugging)
void RegisterFile::dump_register_file(void) const
{
    std::cout<<"Register File:"<<std::endl;
    for (size_t i=0; i<32; i++) std::cout<<"$"<<i<<": dec:
"<<registers.at(i)<<", hex:
0x"<<std::hex<<registers.at(i)<<std::dec<<std::endl;
}
```

Κατά τη κατασκευή του φακέλου καταχωρητών, μηδενίζονται όλες οι τιμές και εμφανίζεται στον χρήστη σχετικό μήνυμα. Επίσης σημαντικό να σημειωθεί ότι σε περίπτωση εγγραφής στον καταχωρητή 0/zero, το αίτημα αγνοείται όπως και στην πραγματικότητα, η εγγραφή αυτού του καταχωρητή είναι αδύνατη.

Ας σημειωθεί ότι δεν γίνεται έλεγχος υπερχείλισης του πίνακα και είναι μία ακόμη υπενθύμιση για να μην χρησιμοποιηθεί αυτός ο κώδικας για κάτι άλλο εκτός από πειράματα.

Ο Ελεγκτής του Επεξεργαστή

Όπως αναλύσαμε, κατά την φάση της αποκωδικοποίησης, ο κεντρικός ελεγκτής της ΚΜΕ ενεργοποιεί/απενεργοποιεί τα κατάλληλα σήματα ελέγχου με βάση τον opcode της εντολής.

```
// Compiler Directives
#pragma once

// Header Files
#include "data_types.h"
#include "isa_definitions.h"

class Controller
{
    friend class Processor;
private:
    // Control Signals Structure
    struct ControlSignals
    {
        bool reg_dst;    // Determines whether rd or rt is the
destination register
        bool jump;       // Jump instruction
        bool branch;     // Branch instruction
        bool mem_read;   // Memory read
        bool mem_to_reg; // Select between memory data and ALU
result
        Byte alu_op;     // ALU operation code (to ALU control)
        bool mem_write;  // Memory write
        bool alu_src;    // ALU source (reg or immediate)
        bool reg_write;  // Register write
        bool link;       // Link Instruction (save PC+4 to $31)
    };

    // Generate the control signals based on the opcode
    ControlSignals decode(const Byte opcode, const Byte rs, const
Byte rt);
};
```

Παραπάνω βλέπουμε αναλυτικά τα βασικότερα σήματα ελέγχου για την επεξεργασία μίας εντολής. Να σημειωθεί ότι οι αρχιτεκτονικές ποικίλουν και κάθε προσομοίωση μπορεί να χρησιμοποιήσει τα δικά της σήματα. Σκοπός της συγκεκριμένης αρχιτεκτονικής είναι να είναι όσο πιο ρεαλιστική γίνεται.

Η μέθοδος decode ελέγχει μέσω μίας switch τον opcode και θέτει τα κατάλληλα σήματα. Αν είναι 0 είναι R-type εντολή, αλλιώς συγκρίνει με το ISA όπου έχω ορίσει, όπως:

```
        case OP_ADDI:
        case OP_SLTI:
        case OP_ANDI:
```

```
        case OP_ORI:
        case OP_XORI:
        case OP_LUI:
        case OP_ADDIU:
        case OP_SLTIU:
            signals.alu_src=true;
            signals.reg_write=true;
            signals.alu_op=ALU_OP_IMM;
            break;
```

Η πιο σημαντική λειτουργία του επίσης είναι ότι θέτει τον κώδικα λειτουργίας της ALU. Αν και η ALU κάνει πρόσθεση, πολλαπλασιασμό και ολίσθηση, σε αυτήν την περίπτωση διαχειρίζεται περισσότερες λειτουργίες για απλοποίηση του υλικού (της προσμοίωσης).

Ίσως είναι η κατάλληλη στιγμή να δούμε κάποια μικρά κομμάτια του ISA που υποστηρίζεται, καθώς είναι πάνω από 200 καταχωρήσεις:

```
// MIPS R-type Function Codes
#define FUNCT_MOVN    0x0B
#define FUNCT_MOVZ    0x0A
#define FUNCT_SYSCALL 0x0C
#define FUNCT_BREAK   0x0D
#define FUNCT_SYNC    0x0F
```

```
// MIPS Opcodes
#define OP_RTYPE  0x00
#define OP_REGIMM 0x01
#define OP_J      0x02
#define OP_JAL    0x03
#define OP_BEQ    0x04
#define OP_BNE    0x05
```

```
// ALU Control Signals
#define ALU_AND     0x0
#define ALU_OR      0x1
#define ALU_ADD     0x2
#define ALU_ADDU    0x3
#define ALU_SUB     0x6
```

```
// ALU Operation Codes
#define ALU_OP_RTYPE    0x2
#define ALU_OP_BRANCH   0x1
#define ALU_OP_MEM_REF  0x0
#define ALU_OP_IMM      0x3
#define ALU_OP_REGIMM   0x4
#define ALU_OP_COP0     0x5
#define ALU_OP_SPECIAL2 0x6
```

```
#define ALU_OP_SPECIAL3 0x7
```

Ακόμη και κωδικοί εξαιρέσεων:

```
// Exception codes
#define EXC_INT  0    // Interrupt
#define EXC_MOD  1    // TLB modification
#define EXC_TLBL 2    // TLB load miss
#define EXC_TLBS 3    // TLB store miss
```

Οπότε, η μέθοδος decode καλείται από τον πυρήνα στην φάση του ID και επιστρέφονται σε struct (σύνθετος τύπους δεδομένων) ρυθμισμένα τα σήματα ελέγχου.

Ελεγκτής ΑΛΜ

Η δουλειά του ελεγκτή της ΑΛΜ είναι μία, να ρυθμίσει την ΑΛΜ κατάλληλα, για την λειτουργία που χρειάζεται να εκτελέσει κατά την φάση της εκτέλεσης της σωλήνωσης. Στη συγκεκριμένη αρχιτεκτονική αυτό γίνεται ακριβώς πριν την εκτέλεση της ΑΛΜ με τα τελούμενα της.

```
class ALUControl
{
    friend class Processor;
private:
    // Generate the ALU signals; alu_op&funct
    Byte generate(const Byte opcode, const Byte alu_op, const
Byte rs, const Byte rt, const Byte funct) const;
};
```

Χρησιμοποιώντας κυρίως το πεδίο funct της εντολής, κατανοεί τι είδους υπολογισμός πρέπει να γίνει. Για παράδειγμα:

```
// For R-type instructions
    if (alu_op==ALU_OP_RTYPE)
    {
        switch (funct)
        {
            // Arithmetic operations
            case FUNCT_ADD:    return ALU_ADD;
            case FUNCT_ADDU:   return ALU_ADDU;
            case FUNCT_SUB:    return ALU_SUB;
            case FUNCT_SUBU:   return ALU_SUBU;

            // Logical operations
            case FUNCT_AND:    return ALU_AND;
            case FUNCT_OR:     return ALU_OR;
            case FUNCT_XOR:    return ALU_XOR;
            case FUNCT_NOR:    return ALU_NOR;
```

Αριθμητική Λογική Μονάδα

Σκοπός του κυκλώματος της ΑΛΜ (ALU) είναι να υπολογίσει το αποτέλεσμα που προέρχεται από τα 2 τελούμενα εισόδου εφαρμόζοντας κάποιον αλγόριθμο (π.χ. πρόσθεσης, ολίσθησης) κατά τον πιο βέλτιστο τρόπο για ηλεκτρονικά κυκλώματα. Όπως, για τον πολλαπλασιασμό, χρησιμοποιείται η λογική ολίσθηση.

```
class ALU
{
    friend class Processor;
private:
    // Result Structure
    struct Result
    {
        Word value;    // ALU result
        bool zero;     // Zero Flag
        bool overflow; // Overflow Flag
        bool exception;
        Byte exception_code;
        //bool cp0_op;

        Word hi;
        Word lo;

        // Set them to 0
        Result():value(0), hi(0), lo(0), zero(false),
overflow(false), exception(false), exception_code(0){}//,
cp0_op(false) {}
    };

    // Methods
    // ALU Operation
    Result execute(const Word a,const Word b,const Byte
control,const Byte shamt=0);
};
```

Παρατηρήστε επίσης τη δομή του struct result (αποτελέσματος), εκτός της μεταβλητής όπου κρατά το αποτέλεσμα της ΑΛΜ, περιέχει κάποια βασικά σήματα/σημαίες για την διευκόλυνση του επεξεργαστή, όπως εάν υπήρξε υπερχείλιση κατά τον υπολογισμό, εάν το αποτέλεσμα είναι ίσο με το 0 ή εάν πρέπει να σηματοδοτηθεί κάποια εξαίρεση π.χ. στην περίπτωση της syscall.

```
// ALU execution
ALU::Result ALU::execute(const Word a,const Word b,const Byte
control,const Byte shamt)
{
    Result result={}; // Reset

    // Cast to signed
    int32_t signed_a=static_cast<int32_t>(a);
```

```
    int32_t signed_b=static_cast<int32_t>(b);

    switch (control)
    {
        case ALU_AND:
            result.value=a&b;
            break;

        case ALU_OR:
            result.value=a|b;
            break;

        case ALU_ADD:
            result.value=a+b;
            // Overflow Detection
            if
((signed_a>0&&signed_b>0&&static_cast<int32_t>(result.value)<0)||
(signed_a<0&&signed_b<0&&static_cast<int32_t>(result.value)>0))
{result.overflow=true;}
            break;

        case ALU_SUB:
            result.value=a-b;
            // Overflow Detection
            if
((signed_a>0&&signed_b<0&&static_cast<int32_t>(result.value)<0)||
(signed_a<0&&signed_b>0&&static_cast<int32_t>(result.value)>0))
{result.overflow=true;}
            break;
case ALU_SYSCALL:
        case ALU_BREAK:
            // These instructions cause exceptions and are
handled by the system coprocessor
            // For the ALU, we just pass the instruction through
            result.value=a; // Preserve input value
            result.exception=true; // Signal an exception

result.exception_code=(control==ALU_SYSCALL)?EXC_SYS:EXC_BP;
            break;

        // Memory operations - here the ALU just calculates the
effective address,
        // The memory access happens in the MEM stage
        case ALU_LB:
        case ALU_LH:
        case ALU_LBU:
        case ALU_LHU:
        case ALU_SB:
```

```
        case ALU_SH:
        case ALU_LL:
        case ALU_SC:
            result.value=a+b; // Calculate effective address
(base + offset)
            break;
```

```
    // Set the Zero flag
    result.zero=(result.value==0);

    // Return to caller
    return result;
```

Αν και όλο το αρχείο είναι τουλάχιστον 400 γραμμές κώδικα, ειδικά για πιο εξειδικευμένες εντολές, για τον σκοπό αυτής της εργασίας και της εξήγησης της αρχιτεκτονικής του MIPS όσο πιο απλά γίνεται, δεν παρατίθενται όλες οι εντολές για λόγους απλούστευσης.

Συνεπεξεργαστής 0

Αν και ο επεξεργαστής του MIPS είναι γενικού σκοπού όπως ήδη εξηγήσαμε, δηλαδή είναι αρκετά ικανός να κάνει διάφορες διαδικασίες, κατανοούμε ότι όπως και στον προγραμματισμό που χωρίζουμε κάποια κομμάτια κώδικα για να κάνουν ένα πράγμα, για να μπορούν να ενσωματωθούν εύκολα οι αλγόριθμοι τους, έτσι και με το υλικό, καλό είναι, όπου μπορούμε, να χωρίσουμε κάποιες δουλειές σε κάποιο άλλο υλικό, όπως π.χ. με τον ελεγκτή της ΑΛΜ. Έτσι λοιπόν και κάποιες λειτουργίες βασικές για την εύρυθμη λειτουργία του επεξεργαστή, καλό είναι να τις διαχειρίζεται κάτι άλλο, όπως ο συνεπεξεργαστής ο (co-processor 0/cp0).

Ο coprocessor0 ή συνεπεξεργαστής ελέγχου συστήματος έχει έναν πολύ σημαντικό ρόλο για τον έλεγχο της ΚΜΕ καθώς αποτελεί βασικό στοιχείο για την λειτουργία του λειτουργικού συστήματος όπου τρέχει, αφού:
Εντοπίζει και διαχειρίζεται τις εξαιρέσεις (exceptions) όπως αριθμητικά σφάλματα, μη εξουσιοδοτημένη ή μη έγκυρη πρόσβαση στη μνήμη, syscalls (κλήσεις συστήματος) κ.λ.π.
Διαχειρίζεται τις διακοπές υλικού (interrupts).
Διαχειρίζεται την Εικονική Μνήμη (Virtual Memory) σε συνεργασία με την Memory Management Unit (MMU) για μεταφράσεις διευθύνσεων με την χρήση του Translation Lookaside Buffer (TLB) και χρησιμοποιείται από το ΛΣ για την υλοποίηση της σελιδοποίησης (paging).
Επιβλέπει και προσαρμόζει τη λειτουργία του επεξεργαστή (κατάσταση πυρήνα ή χρήστη).
Χρησιμοποιείται για την διάγνωση των σφαλμάτων, όπως ένα σφάλμα κακής διεύθυνσης (Bad Virtual Address).

```
// Compiler Directives
#pragma once

// Header Files
#include "data_types.h"
#include "MMU.h"
#include <array>

// Pre-compiler Directives
// Exception Codes
#define EXC_INT     0  // Interrupt
#define EXC_MOD     1  // TLB modification exception
#define EXC_TLBL    2  // TLB exception (load or instruction
fetch)
#define EXC_TLBS    3  // TLB exception (store)
#define EXC_ADEL    4  // Address error exception (load or
instruction fetch)
#define EXC_ADES    5  // Address error exception (store)
#define EXC_IBE     6  // Bus error exception (instruction fetch)
#define EXC_DBE     7  // Bus error exception (data reference:
load/store)
#define EXC_SYSCALL 8  // Syscall exception
#define EXC_BREAK   9  // Breakpoint exception
#define EXC_RI      10 // Reserved instruction exception
#define EXC_CPU     11 // Coprocessor unusable exception
#define EXC_OV      12 // Arithmetic overflow exception
```

```
// Coprocessor 0 register numbers
#define CP0_INDEX    0
#define CP0_RANDOM   1
#define CP0_ENTRYLO0 2
#define CP0_ENTRYLO1 3
#define CP0_CONTEXT  4
#define CP0_PAGEMASK 5
#define CP0_WIRED    6
#define CP0_BADVADDR 8
#define CP0_COUNT    9
#define CP0_ENTRYHI  10
#define CP0_COMPARE  11
#define CP0_STATUS   12
#define CP0_CAUSE    13
#define CP0_EPC      14
#define CP0_PRID     15
#define CP0_CONFIG   16
#define CP0_LLADDR   17
#define CP0_WATCHLO  18
#define CP0_WATCHHI  19
#define CP0_ERRCTL   26
#define CP0_CACHERR  27
#define CP0_TAGLO    28
#define CP0_TAGHI    29
#define CP0_ERROREPC 30

class Coprocessor0
{
    friend class Processor;
    friend class ICache;
    friend class DCache;

private:
    // Constructor
    Coprocessor0();

    // Methods
    // Register Operations
    Word read_register(const Byte reg_num);
    void write_register(const Byte reg_num, const Word value);

    // Interface to MMU - TLB Operations
    void tlb_read();
    void tlb_write_index();
    void tlb_write_random();
    void tlb_probe();
```

```
    Word translate_address(Word virtual_address, bool is_write,
Word access_size=4);

    // Exception Handlers
    void handle_exception(Word pc, Byte cause, Word bad_vaddr=0);
    Word get_exception_pc() const;
    bool handle_interrupts();
    void eret();

    // Members
    // Status flags
    bool interrupts_enabled();

    // CP0 registers (Referenced MIPS R4000)
    std::array<Word,32> registers;

    // MMU part of the Coprocessor0
    MMU mmu;
};
```

Παρατηρούμε τους ορισμούς για την διευκόλυνση μετάφρασης του δείκτη του κάθε καταχωρητή καθώς και τους ορισμούς για τις εξαιρέσεις. Ύστερα, βλέπουμε τον ορισμό ως φίλιων κλάσεων του επεξεργαστή αλλά και των κρυφών μνημών επιπέδου 1. Αυτό είναι αναγκαίο να γίνει αφού αυτές οι κρυφές μνήμης είναι τύπου VIPT (Virtually Indexed–Physically Tagged/Εικονικά Δεικτοδοτημένη-Φυσικά Ετικετοποιημένη), δηλαδή δέχεται εικονικές διευθύνσεις μνήμης αλλά χειρίζεται τα δεδομένα με βάση τη φυσικού τους διεύθυνση, οπότε χρειάζονται πρόσβαση στις μεθόδους του cp0 για μετάφραση των διευθύνσεων.

```
// Constructor
Coprocessor0::Coprocessor0()
{
    // Initialize CP0 registers to 0
    registers.fill(0);

    // Set default values for special regs (For the OS)
    registers[CP0_STATUS]=0x00400000;  // BEV bit set, kernel
mode, interrupts disabled
    registers[CP0_PRID]=0x00000100;    // Implementation number
and revision IF PROBLEMS; SPOOF TO 0x00018000!
    /* Bits | Field           | Description
      31–24 | Company Options | Vendor-specific info
      23–16 | Company ID      | Identifies the manufacturer
(e.g., MIPS = 0x01)
       15–8 | Processor ID    | Tells what type of CPU (e.g.,
0x34 for 4Kc)
        7–0 | Revision        | Tells the revision/version of the
CPU */
```

```
    std::cout<<"Coprocessor 0 created successfully!"<<std::endl;
}
```

Κατά τη κατασκευή του αντικειμένου παρατηρούμε ότι ουσιαστικά μηδενίζονται τα περιεχόμενα των καταχωρητών καθώς και γίνεται ρύθμιση 2 βασικών καταχωρητών.

```
// Register access
Word Coprocessor0::read_register(const Byte reg_num)
{
    // Value check
    if (reg_num<registers.size())
    {
        if (reg_num==CP0_RANDOM)
        {   // RANDOM register decrements automatically
            //registers[CP0_RANDOM]=(registers[CP0_RANDOM]-
1)%TLB_SIZE;
            registers[CP0_RANDOM]--;
            if
(registers[CP0_RANDOM]<registers[CP0_WIRED]||registers[CP0_RANDOM
]>=MMU::TLB_SIZE) {registers[CP0_RANDOM]=MMU::TLB_SIZE-1;}
        }
        return registers[reg_num];
    }

    // If size is wrong
    std::cout<<"WARNING: Invalid CP0 register read:
"<<static_cast<int>(reg_num)<<std::endl;
    return 0; // TODO: CHANGE TO ERROR CODE VALUE!!!
}

void Coprocessor0::write_register(const Byte reg_num, const Word
value)
{   // FULL OVERWRITE IS ALLOWED, SAFETY ISSUE
    // Check for valid range
    if (reg_num<registers.size())
    {   // Special registers handling
        switch (reg_num)
        {
            case CP0_CAUSE:
                // Only software interrupt bits can be written

registers[CP0_CAUSE]=(registers[CP0_CAUSE]&~0x300)|(value&0x300);
                break;
            case CP0_STATUS:
                // Apply mask of writable bits
                registers[CP0_STATUS]=value;
                break;
            case CP0_COMPARE:
```

```
                // Clear the timer interrupt
                registers[CP0_CAUSE]&=~0x8000;
                registers[CP0_COMPARE]=value;
                break;
            default: // Other registers' behaviour
                registers[reg_num]=value;
                break;
        }
    }
    else {std::cout<<"WARNING: Invalid CP0 register write call:
"<<static_cast<int>(reg_num)<<std::endl;} //MAKEERRORCODE!
}
```

Παρατηρούμε ότι για συγκεκριμένους καταχωρητές οι λειτουργίες είναι πιο ειδικές και περιορισμένες. Αν και σε αυτές τις μεθόδους γίνεται έλεγχος ως προς το εάν πραγματικά υπάρχει ο καταχωρητής που ζητείται, δεν γίνεται επαρκής έλεγχος της εγγραφής. Άλλη μία υπενθύμιση για να μην θεωρηθεί ασφαλείς ο κώδικας αυτός.

```
Word Coprocessor0::translate_address(Word virtual_address, bool
is_write, Word access_size)
{
    Word asid=registers[CP0_ENTRYHI]&0xFF; // Current Address
Space ID
    Word bad_vaddr=0;
    Byte exception_cause=0;
    MMU::AccessMode access_mode=MMU::AccessMode::CACHED;

    // Call MMU's translate_address method
    Word physical_address=mmu.translate_address(virtual_address,
is_write, asid, bad_vaddr, exception_cause, access_size,
access_mode);

    // If an exception occurred (physical_address==0)
    if (physical_address==0)
    {
        // Map MMU exception codes to CP0 exception codes
        Byte cp0_cause;
        switch (exception_cause)
        {
            case MMU_EXC_MOD:  cp0_cause=EXC_MOD;  break;
            case MMU_EXC_TLBL: cp0_cause=EXC_TLBL; break;
            case MMU_EXC_TLBS: cp0_cause=EXC_TLBS; break;
            case MMU_EXC_ADEL: cp0_cause=EXC_ADEL; break;
            case MMU_EXC_ADES: cp0_cause=EXC_ADES; break;
            default:           cp0_cause=EXC_TLBL; break; //
Default if unknown
        }

        // Handle the exception
```

```
        handle_exception(registers[CP0_EPC],cp0_cause,bad_vaddr);
    }

    // Return the physical address and cache mode
    // Might have to use the access_mode somewhere in the memory
access code
    return physical_address;
}

// Exception Handling
void Coprocessor0::handle_exception(Word pc, Byte cause, Word
bad_vaddr)
{
    // Save the address where the exception occurred
    registers[CP0_EPC]=pc;

    // Set the exception cause
    registers[CP0_CAUSE]&=~0x7C; // Clear ExcCode field
    registers[CP0_CAUSE]|=(static_cast<Word>(cause)<<2); // New
ExcCode

    // Save the bad virtual address if applicable
    if
(cause==EXC_ADEL||cause==EXC_ADES||cause==EXC_TLBL||cause==EXC_TL
BS) {registers[CP0_BADVADDR]=bad_vaddr;}

    // Update the status register" shift KSU, EXL and IE bits
    registers[CP0_STATUS]|=0x2; // Set EXL bit (exception level)

    // Could skip, DEBUGGING,CHECK DEBUG MODE AND CHECK THE CAST
    std::cout<<"Exception handled:
CP0_STATUS=0x"<<std::hex<<registers[CP0_STATUS]<<",
CP0_BADVADDR="<<bad_vaddr<<", Cause:
"<<registers[CP0_CAUSE]<<std::dec<<std::endl;
}

Word Coprocessor0::get_exception_pc() const
{
    // Return the exception vector address
    if (registers[CP0_STATUS]&0x00400000) return 0xBFC00200; //
BEV bit set - use bootstrap exception vector
    else return 0x80000080; // Normal exception vector
}

bool Coprocessor0::interrupts_enabled() // Check if interrupts
are globally enabled (IE bit) and not disabled by exception level
(EXL, ERL)
```

```
{return
((registers[CP0_STATUS]&0x1)&&(!(registers[CP0_STATUS]&0x6)));}

bool Coprocessor0::handle_interrupts()
{
    // Check for pending interrupt attempts
    Word
pending_interrupts=((registers[CP0_CAUSE]&0xFF00)&(registers[CP0_
STATUS]&0xFF00));

    if (pending_interrupts&&interrupts_enabled())
    {
        // Handle the interrupt as an exception
        handle_exception(registers[CP0_EPC],EXC_INT,0);
        return true;
    }
    return false; // No interrupt
}

// Exception return: clear EXL bit and return to EPC
void Coprocessor0::eret() {registers[CP0_STATUS]&=~0x2;}
```

Για εκπαιδευτικούς σκοπούς έχουν παρατεθεί και οι μέθοδοί για τις εξαιρέσεις/διακοπές.

```
// TLB Operations - Use the MMU
void Coprocessor0::tlb_read()
{
    Word index=registers[CP0_INDEX]&0x3F; // Index is 6 bit
CHECK!

     // Call MMU's tlb_read method

mmu.tlb_read(index,registers[CP0_ENTRYHI],registers[CP0_ENTRYLO0]
,registers[CP0_ENTRYLO1],registers[CP0_PAGEMASK]);
}

void Coprocessor0::tlb_write_index()
{
    Word index=registers[CP0_INDEX]&0x3F; // Index is 6 bit
CHECK!

    // Call MMU's tlb_write_index method

mmu.tlb_write_index(index,registers[CP0_ENTRYHI],registers[CP0_EN
TRYLO0],registers[CP0_ENTRYLO1],registers[CP0_PAGEMASK]);
}

void Coprocessor0::tlb_write_random()
```

```
{
    Word random=registers[CP0_RANDOM]&0x3F; // Random is 6 bit
CHECK!

    // Call MMU's tlb_write_random method

mmu.tlb_write_random(random,registers[CP0_ENTRYHI],registers[CP0_
ENTRYLO0],registers[CP0_ENTRYLO1],registers[CP0_PAGEMASK]);
}

void Coprocessor0::tlb_probe()
{
    Word vpn=registers[CP0_ENTRYHI]&0xFFFFE000; // Virtual Page
Number
    Word asid=registers[CP0_ENTRYHI]&0xFF; // Address Space ID

    // Call MMU's tlb_probe method
    int result=mmu.tlb_probe(vpn,asid);

    // Found a match
    if (result>=0) {registers[CP0_INDEX]=result;}
    // No match found, set high bit of INDEX
    else {registers[CP0_INDEX]=0x80000000;}
}
```

Και οι μέθοδοι που αφορούν την TLB καλούν κυρίως την MMU καθώς ουσιαστικά, αυτή είναι υπεύθυνη για τη διαχείριση της.

Αν και έχει προσομοιωθεί μόνο ο συνεπεξεργαστής0 για αυτή την εργασία, ο MIPS έχει θέσεις για 4 συνεπεξεργαστές. Συγκεκριμένα:
Coprocessor1: Διαχείριση τελούμενων κινητής υποδιαστολής (FPU, Floating Point Unit).
Coprocessor2,3: Προαιρετικοί και δεν ορίζονται αυστηρά στα εγχειρίδια του MIPS και συνήθως αφορούν ειδικές εφαρμογές των κατασκευαστών, όπως ως κάποια πρόσθετη μονάδα γραφικών, κρυπτογράφησης ή κάποιων προσαρμοσμένων χαρακτηριστικών.

Μονάδα Διαχείρισης Μνήμης

Η Μονάδα Διαχείρισης Μνήμης (ΜΔΜ/MMU) είναι ένα πολύ βασικό στοιχείο για τις ΚΜΕ αφού είναι υπεύθυνη για την χαρτογράφηση (mapping) των εικονικών διευθύνσεων σε φυσικές διευθύνσεις καθώς και είναι υπεύθυνη για τον έλεγχο της πρόσβασης της μνήμης, π.χ. μία διεργασία να προσπελάσει μία περιοχή που δεν της ανήκει/έχει δικαιώματα ή για τον διαχωρισμό του χώρου μνήμης ανάμεσα στον πυρήνα του λειτουργικού συστήματος και του χώρου χρήστη εξασφαλίζοντας καλύτερη ασφάλεια και απομόνωση. Διαχειρίζεται δε σε συνεργασία με το ΛΣ το paging (σελιδοποίηση) για πιο αποδοτικότερη χρήση της μνήμης καθώς και χρησιμοποιεί τον πίνακα TLB για να βελτιστοποιήσει τον χρόνο μετάφρασης των διευθύνσεων.

```
// Compiler Directives
#pragma once

// Header Files
#include "data_types.h"
#include <array>
#include <iomanip>

// Pre-compiler Definitions
// Exception codes
#define MMU_EXC_MOD     1  // TLB modification exception
#define MMU_EXC_TLBL    2  // TLB exception (load or instruction
fetch)
#define MMU_EXC_TLBS    3  // TLB exception (store)
#define MMU_EXC_ADEL 4
#define MMU_EXC_ADES 5

// MIPS memory segment addresses (standard memory map)
namespace MemorySegments
{
    // Kernel segments (kseg)
    constexpr Word KSEG0_BASE=0x80000000; // Cached, unmapped
    constexpr Word KSEG0_END= 0x9FFFFFFF;

    constexpr Word KSEG1_BASE=0xA0000000; // Uncached, unmapped
    constexpr Word KSEG1_END= 0xBFFFFFFF;

    constexpr Word KSEG2_BASE=0xC0000000; // Mapped, kernel-only
    constexpr Word KSEG2_END= 0xDFFFFFFF;

    constexpr Word KSEG3_BASE=0xE0000000; // Mapped, kernel-only
    constexpr Word KSEG3_END= 0xFFFFFFFF;

    // User segments (useg)
    constexpr Word USEG_BASE=0x00000000; // Mapped user virtual
memory
```

```
    constexpr Word USEG_END= 0x7FFFFFFF;

    // Physical Memory
    // Physical memory limits (1GB)
    constexpr Word PHYS_MEM_SIZE=0x40000000; // 1GB
    constexpr Word PHYS_MEM_MASK=0x1FFFFFFF; // Mask for physical
address


    // Special memory regions
    constexpr Word RESET_VECTOR= 0xBFC00000; // Reset vector
address // Start of ROM
    constexpr Word EXCEPT_VECTOR=0x80000080; // Exception vector
address
    // ROM
    constexpr Word ROM_BASE=0xBFC00000;
    constexpr Word ROM_SIZE=0x00100000;  // 1MB
    constexpr Word ROM_END=ROM_BASE+ROM_SIZE-1;

    // I/O (Part of uncached kernel space)
    constexpr Word MMIO_START=0xB8000000;
    constexpr Word MMIO_END  =0xBFFFFFFF;
}

class MMU
{
    friend class Coprocessor0;
    friend class DCache;

private:
    // Constructor
    MMU();

    enum class AccessMode
    {
        CACHED,
        UNCACHED,
        UNCACHED_ACCELERATED
    };

    // TLB Struct
    struct TLBEntry
    {
        Word entryhi;
        Word entrylo0;
        Word entrylo1;
        Word pagemask;
        bool valid;
    };
```

```
    // Methods
    // TLB Operations
    void tlb_read(Word index,Word& entryhi,Word& entrylo0,Word&
entrylo1,Word& pagemask);
    void tlb_write_index(Word index,Word entryhi,Word
entrylo0,Word entrylo1,Word pagemask);
    void tlb_write_random(Word random,Word entryhi,Word
entrylo0,Word entrylo1,Word pagemask);
    int tlb_probe(Word vpn, Word asid);
    Word translate_address(Word virtual_address,bool
is_write,Word asid,Word& bad_vaddr,Byte& exception_cause,Word
access_size,AccessMode& access_mode);

    // TLB array (For MIPS R4000)
    static const int TLB_SIZE=32; // MOVE TO PRECOMPILER
DIRECTIVE?
    std::array<TLBEntry,TLB_SIZE> tlb;

    static bool is_kseg0(Word virtual_addr)
    {return
(virtual_addr>=MemorySegments::KSEG0_BASE&&virtual_addr<=MemorySe
gments::KSEG0_END);}

    static bool is_kseg1(Word virtual_addr)
    {return
(virtual_addr>=MemorySegments::KSEG1_BASE&&virtual_addr<=MemorySe
gments::KSEG1_END);}

    static bool is_useg(Word virtual_addr)
    {return
(virtual_addr>=MemorySegments::USEG_BASE&&virtual_addr<=MemorySeg
ments::USEG_END);}

    static bool is_kseg2(Word virtual_addr)
    {return
(virtual_addr>=MemorySegments::KSEG2_BASE&&virtual_addr<=MemorySe
gments::KSEG2_END);}

    static bool is_kseg3(Word virtual_addr)
    {return
(virtual_addr>=MemorySegments::KSEG3_BASE&&virtual_addr<=MemorySe
gments::KSEG3_END);}

    static bool is_unmapped(Word virtual_addr)
    {return (is_kseg0(virtual_addr)||is_kseg1(virtual_addr));}

    static bool is_mapped(Word virtual_addr)
```

```
    {return !is_unmapped(virtual_addr);}

    static Word virtual_to_physical(Word virtual_addr)
    {
        if (is_unmapped(virtual_addr))
        {
            return virtual_addr&MemorySegments::PHYS_MEM_MASK;
        }
        // For mapped segments, TLB lookup is required
        // This should not be called directly for mapped segments
        return 0;
    }
public:
    // Debugging
    void dump_tlb() const;
};
```

Το πιο σημαντικό κομμάτι της κλάσης αυτής είναι ο διαχωρισμός της μνήμης σε segments (τμήματα). Αυτό εξασφαλίζει την απομόνωση μεταξύ τους. Μάλιστα, είναι δεδομένο ότι το σύστημα διαχειρίζεται στο μέγιστο 4 GB φυσικού χώρου. Αυτό γίνεται διότι, χωρίς περαιτέρω λειτουργίες (εκεί βοηθάει ακόμη περισσότερο η λογική των εικονικών διευθύνσεων/για υποστήριξη περισσότερου χώρου), το σύστημα χειρίζεται το πολύ 32 bit διευθύνσεων (και μάλιστα με τη χρήση τουλάχιστον 2 εντολών, καθώς αλλιώς δεν υπάρχει χώρος για τον opcode/αναγνώριση της εντολής) και άρα η μεγαλύτερη διεύθυνση είναι 2^{32}=4GB. Επίσης, παρατηρούμε και κάποιες βοηθητικές μεθόδους καθώς και τον πίνακα TLB.

Επειδή η λειτουργία της MMU είναι αρκετά εξειδικευμένη και είναι πολύ εκτός σκοπού της εργασίας, δεν θα παρατεθεί ο σχετικός κώδικας των ορισμών των μεθόδων της (400 γραμμές κώδικα), αλλά η ουσία είναι ότι γίνονται διάφοροι έλεγχοι για να εξασφαλιστεί ότι η διεύθυνση είναι έγκυρη.

Μηχανισμός Ανίχνευσης Κινδύνων και Μηχανισμός Προώθησης Δεδομένων

Αν και η λογική της Σωλήνωσης (pipeline) επέφερε μεγάλη επανάσταση με την ενσωμάτωση της στο πρότυπο RISC, δεν έρχεται με τα μειονεκτήματα της. Το μεγαλύτερο και πιο κρίσιμο μειονέκτημα της είναι ότι η ΚΜΕ πρέπει να έχει μία σχετική επίγνωση για το τι γίνεται και στα 5 στάδια της ταυτόχρονα καθώς και εάν και πώς εξαρτώνται οι εντολές μεταξύ τους. Αν και το πρώτο μοντέλο του MIPS του Καθηγητή Hennessy άφηνε τους πιθανούς κινδύνους από εξαρτήσεις μεταξύ εντολών εντελώς στην ευθύνη του μεταγλωττιστή, εξ' ου και το «without Interlocking» του MIPS, αυτό κρίθηκε αναγκαίο αργότερα να γίνεται και από το υλικό για τους λόγους ότι στην πραγματικότητα το λογισμικό επιφέρει αρκετές φορές απρόβλεπτα άλματα και διακοπές συστήματος όπου ο μεταγλωττιστής δεν μπορεί να προβλέψει καθώς και μηχανισμοί όπως out of order execution και πολλαπλές σωληνώσεις, έκαναν αναγκαίους μηχανισμούς αλληλοκλειδώματος (interlocking) για να αποφευχθούν παγώματα κύκλων εντολών.

Παράδειγμα Κινδύνου Εξάρτησης Δεδομένων

Ας αναλύσουμε ένα απλό παράδειγμα κινδύνου εξάρτησης δεδομένων (data hazard) σε κώδικα assembly:
Ας υποθέσουμε ότι η ΚΜΕ τρέχει ένα πρόγραμμα που περιέχει αυτές τις 2 εντολές
add $t0, $t1, $t2
sub $t3, $t0, $t4
Η εντολή sub εξαρτάται από τον καταχωρητή $t0 για να υπολογίσει το αποτέλεσμά της και εφ' όσον εκτελούνται σειριακά, θεωρείται από τον προγραμματιστή πως η εκτέλεση της add έχει ολοκληρωθεί και ο $t0 περιέχει το αποτέλεσμα της add. Όμως, λόγω του pipeline, η εγγραφή του $t0 δεν έχει ολοκληρωθεί στον επόμενο κύκλο, τον κύκλο που η sub εξαρτάται σε αυτόν για την επιτυχή επεξεργασία της (Αυτό γίνεται στο 5ο στάδιο του pipeline/Write Back stage).

Αυτός ήταν ένας κλασσικός κίνδυνος Read After Write (RAW). Για να επιλυθεί ο κίνδυνος αυτός, ο μεταγλωττιστής πρέπει να εισαγάγει 2 NOP εντολές (εντολές που δεν κάνουν τίποτα) για να καθυστερήσει την εκτέλεση της sub. Αν αφεθεί σε interlocking μηχανισμό, η ΚΜΕ κάνει stall (παγώνει) αυτόματα για ένα κύκλο ή και εάν υπάρχει μηχανισμός προώθησης δεδομένων (forwarding), το αποτέλεσμα της add προωθείται απευθείας στην sub χωρίς καθυστέρηση και χωρίς να περιμένει το στάδιο του Write Back.

Επίσης ακόμη και στην περίπτωση όπου υπάρχουν τέτοιοι μηχανισμοί, από μόνοι τους δεν εξασφαλίζουν πιθανές διακοπές από περιφερειακές συσκευές και έτσι κινδυνεύουν οι εντολές που βρίσκονται στον Σωλήνα σε περίπτωση που δεν γίνει σωστό flush του σωλήνα. Ένας τρόπος επίλυσης είναι η προσωρινή αποθήκευση από την ΚΜΕ (με τη συνεργασία του συνεπεξεργαστή 0) και μετά το flush.

Για λόγους απλούστευσης δεν θα γίνει καμία αναφορά στον μηχανισμό ανίχνευσης κινδύνων και της προώθησης δεδομένων που προγραμματίστηκε για τον προσομοιωτή.

Κρυφές Μνήμες

Οι κρυφές μνήμες (cache) είναι πολύ μικρές αλλά γρήγορες μνήμες που βρίσκονται μέσα στην ΚΜΕ και χρησιμοποιούνται για να επιταχύνουν την πρόσβαση σε δεδομένα και εντολές. Αυτό γίνεται γιατί η ΜΤΠ είναι αργή και ο επεξεργαστής πολύ πιο γρήγορος και δεν μπορεί να περιμένει την ΜΤΠ. Επίσης, βασίζεται στην αρχή της χωρικής και χρονικής τοπικότητας των δεδομένων (locality), που σημαίνει ότι αν κάτι χρησιμοποιήθηκε πρόσφατα, μάλλον θα ξαναχρησιμοποιηθεί σύντομα και ότι πιθανόν να χρησιμοποιηθούν και οι διπλανές διευθύνσεις μνήμης. Επίσης, οι κρυφές μνήμες κρατάνε και πρόχειρα τα πιο χρήσιμα δεδομένα.

Στους σύγχρονους υπολογιστές υπάρχουν αρκετές κρυφές μνήμες και υπάρχει η έννοια της ιεραρχίας της μνήμης, δηλαδή, ψάχνεις κάτι στο επίπεδο 1, αν δεν το βρεις πας στο 2 κ.ο.κ. Στην συγκεκριμένη προσομοίωση χρησιμοποιούνται 3 κύρια επίπεδα direct mapped (Άμεσης Αντιστοίχισης για απλότητα) κρυφών μνημών και μάλιστα το τελευταίο παίζει πολύ σημαντικό ρόλο για την λύση ασφαλείας που προτείνω καθώς είναι το τελευταίο επίπεδο που έρχεται σε επαφή η μνήμη με τον επεξεργαστή.

Παρακάτω παρατίθενται μόνο τα header files (αρχεία κεφαλίδας) με τις δηλώσεις των σχετικών μεθόδων, των κρυφών μνημών όπου υλοποιήθηκαν.

```
// Compiler Directives
#pragma once

// Header Files
#include <vector>
#include "data_types.h"
#include "Coprocessor0.h"
//#include "Memory.h"
#include "SharedCache.h"

class ICache
// Direct-Mapped VIPT Instruction Cache
{
private:
    // Cache Configuration
    static constexpr int LINE_SIZE=4; // Each cache line will
hold 4 Words (Instructions); 16 Bytes
    static constexpr int NUM_LINES=256; // 256 cache lines=4KB
cache
    static constexpr int OFFSET_BITS=2; // log2(LINE_SIZE)
    static constexpr int INDEX_BITS=8; // log2(NUM_LINES)
    static constexpr int TAG_BITS=32-INDEX_BITS-OFFSET_BITS-2; //
32-bit address-index-offset-2 (Word addressing)

    // Cache line struct
    struct CacheLine
    {
        bool valid;
        Address tag; // Bits to identify address
```

```
        Word data[LINE_SIZE];

        // Struct Constructor
        CacheLine():valid(false),tag(0)
        {
            for (int i=0;i<LINE_SIZE;i++)
                data[i]=0; // Initialize all lines with 0
        }
    };

    // The Cache
    std::vector<CacheLine> cache;

    Coprocessor0* cp0;
    //Memory* memory;
    SharedCache* l2_cache;

    // Statistics
    int hits;
    int misses;

    // Address translation
    Word getTag(Address address) const;
    Word getIndex(Address address) const;
    Word getOffset(Address address) const;

    // Fill a line of cache
    void fillLine(Address address);

public: // FOR FUTURE FEATURE IMPLEMENTATION LEAVE PUBLIC FOR
NOW, FRIEND THE PROCESSOR CLASS AND MAKE PRIVATE!
    // Constructor
    ICache(Coprocessor0* cp0, SharedCache* l2_ptr);

    // Cache operations
    Word readInstruction(Address address);

    // Invalidate cache
    void invalidateCache();
    // Invalidate a specific address
    void invalidateAddress(Address address);
    // Invalidate a range of addresses
    void invalidateRange(Address start_addr,Address end_addr);
    // Flush the cache
    void flushCache();

    // Cache Statistics
    double getHitRate() const;
```

```
    void resetStats();
    void printStats() const;

    // Debugging
    void dump_all_cache_lines() const;
};
```

```
// Compiler Directives
#pragma once

// Header Files
#include <vector>
#include "data_types.h"
#include "Coprocessor0.h"
//#include "Memory.h"
#include "SharedCache.h"

class DCache
// Direct-Mapped VIPT Data Cache
{
private:
    // Cache Configuration
    static constexpr int LINE_SIZE=4; // Each cache line holds 4
Words; 16 Bytes
    static constexpr int NUM_LINES=256; // 256 cache lines = 4KB
cache
    static constexpr int OFFSET_BITS=2; // log2(LINE_SIZE)
    static constexpr int INDEX_BITS=8; // log2(NUM_LINES)
    static constexpr int TAG_BITS=32-INDEX_BITS-OFFSET_BITS-2; //
32-bit address-index-offset-2 (Word addressing)

    // Cache line struct
    struct CacheLine
    {
        bool valid;
        bool dirty; // Modified flag for write-back
        Address tag; // Bits to identify address
        Word data[LINE_SIZE];

        // Struct Constructor
        CacheLine():valid(false),dirty(false),tag(0)
        {
            for (int i=0; i<LINE_SIZE; i++)
                data[i]=0; // Initialize all lines with 0
        }
    };

    // The Cache
```

```
    std::vector<CacheLine> cache;

    // References
    Coprocessor0* cp0;
    //Memory* memory;
    SharedCache* l2_cache;

    // Variable for statistics
    int hits;
    int misses;
    int writebacks;

    // Address translation
    Word getTag(Address address) const;
    Word getIndex(Address address) const;
    Word getOffset(Address address) const;

    // Fill a line of cache
    void fillLine(Address address);

    // Write back a dirty line to memory
    //void writeBackLine(Word index);
    // Write to L2
    void writeBackLineToL2(Word index);

    // Helper functions
    Word readWordFromLine(const CacheLine& line, Word offset)
const;
    HalfWord readHalfWordFromLine(const CacheLine& line, Word
offset, bool upper) const;
    Byte readByteFromLine(const CacheLine& line, Word offset, int
byteOffset) const;

    void writeWordToLine(CacheLine& line, Word offset, Word
value);
    void writeHalfWordToLine(CacheLine& line, Word offset, bool
upper, HalfWord value);
    void writeByteToLine(CacheLine& line, Word offset, int
byteOffset, Byte value);

public:
    // Constructor
    DCache(Coprocessor0* cp0_ptr, SharedCache* l2_ptr);

    // Cache operations
    Word readWord(Address address);
    HalfWord readHalfWord(Address address);
    Byte readByte(Address address);
```

```
    int writeWord(Address address, Word value);
    int writeHalfWord(Address address, HalfWord value);
    int writeByte(Address address, Byte value);

    // Invalidate cache
    void invalidateCache();
    void invalidateAddress(Address address);
    void invalidateRange(Address start_addr, Address end_addr);

    // Flush cache
    void flushCache();

    // Cache Statistics
    double getHitRate() const;
    void resetStats();
    void printStats() const;

    // Debugging
    void dump_all_cache_lines() const;
};
```

```
// Compiler Directives
#pragma once

//Header Files
#include <vector>
#include "data_types.h"
//#include "Memory.h"
#include "CacheDecryptionUnit.h"

// A Direct-Mapped PIPT cache, one level above the Instruction
Cache and Data Cache and before the Memory (L2)
class SharedCache
{
private:
    // Cache Configuration
    static constexpr int LINE_SIZE=8;    // 8 Words per line (32
bytes)
    static constexpr int NUM_LINES=1024; // 1024 lines=32KB cache
    static constexpr int OFFSET_BITS=3;  // log2(LINE_SIZE)
    static constexpr int INDEX_BITS=10;  // log2(NUM_LINES)
    static constexpr int TAG_BITS=32-INDEX_BITS-OFFSET_BITS-2; //
Physical Address bits

    // Cache line structure
    struct CacheLine
    {
```

```
        bool valid;
        bool dirty;
        Address tag;
        Word data[LINE_SIZE];

        CacheLine():valid(false),dirty(false),tag(0)
        {
            for (int i=0;i<LINE_SIZE;i++)
                data[i]=0;
        }
    };

    // The cache
    std::vector<CacheLine> cache;

    //Memory* memory;
    CacheDecryptionUnit* cdu;

    // Variable for statistics
    int hits;
    int misses;
    int writebacks;
    int l1_requests;

    // Extract the fields of the address
    Word getTag(Address physical_addr) const;
    Word getIndex(Address physical_addr) const;
    Word getOffset(Address physical_addr) const;

    // Internal cache management methods
    void fillLine(Address physical_addr);
    void writeBackLine(Word index);

    // Helper functions for different access sizes
    Word readWordFromLine(const CacheLine& line, Word offset)
const;
    HalfWord readHalfWordFromLine(const CacheLine& line, Word
offset, bool upper) const;
    Byte readByteFromLine(const CacheLine& line, Word offset, int
byteOffset) const;

    void writeWordToLine(CacheLine& line, Word offset, Word
value);
    void writeHalfWordToLine(CacheLine& line, Word offset, bool
upper, HalfWord value);
    void writeByteToLine(CacheLine& line, Word offset, int
byteOffset, Byte value);
```

```
public:
    // Constuctor
    //explicit SharedCache(Memory* mem_ptr);
    explicit SharedCache(CacheDecryptionUnit* cdu_ptr);

    // Cache operations
    // Read - Return data if found, otherwise fetch from memory
    Word readWord(Address physical_addr);
    HalfWord readHalfWord(Address physical_addr);
    Byte readByte(Address physical_addr);

    // Write operations - write-back policy
    void writeWord(Address physical_addr, Word value);
    void writeHalfWord(Address physical_addr, HalfWord value);
    void writeByte(Address physical_addr, Byte value);

    // Handles for L1 Caches
    // Read an entire cache line (for L1 cache misses)
    bool readCacheLine(Address physical_addr, Word* data, int
line_size);
    // Write an entire cache line (for L1 writebacks)
    bool writeCacheLine(Address physical_addr, const Word* data,
int line_size);

    // Cache management
    void invalidateCache();
    void invalidateAddress(Address physical_addr);
    void invalidateRange(Address start_physical_addr, Address
end_physical_addr);
    void flushCache();

    // L1 cache call
    //void notifyL1Request() {l1_requests++;}

    // Statistics and debugging
    double getHitRate() const;
    void resetStats();
    void printStats() const;
    void dump_all_cache_lines() const;
};
```

Καθώς και την Κρυφή Μνήμη Επιπέδου 3 χωρίς τα στοιχεία για την αποκρυπτογράφηση, καθώς αναλύονται στο επόμενο κεφάλαιο.

```
// Compiler Directives
#pragma once
```

```
// Header files
#include "Memory.h"
#include <vector>

class CacheEncryptionUnit
{
private:
    Memory* physical_memory;

    // Cache Configuration
    static const int CDU_BLOCK_SIZE=16;  // 16 words (64 bytes)
encryption blocks (Adjust with encryption method)
    static const int CDU_CACHE_LINES=64; // Small cache for
encrypted blocks


    std::vector<EncryptedBlock> cdu_cache;

    // Statistics
    int hits,misses,decryptions;

    // Helper methods
    Address alignToCDUBlock(Address addr) const; // Make sure it
is the same with the Memory's method
    int getCDUIndex(Address addr) const;
    void loadEncryptedBlock(Address base_addr, EncryptedBlock&
block);
    void writeBackBlock(EncryptedBlock& block);

public:
    CacheDecryptionUnit(Memory* mem_ptr);

    // Block operations for efficient L2 cache line
fills/writebacks
    bool readBlock(Address physical_addr, Word* data, int
word_count);
    bool writeBlock(Address physical_addr, const Word* data, int
word_count);

    // Cache management
    void flushAll();
    void invalidateAddress(Address addr);
    void invalidateAll();

    // Statistics
    void printStats() const;
    void resetStats();
    void dumpAllBlocks() const;
```

```
};
```

Παρατηρούμε μεθόδους για γράψιμο/εγγραφή, cache flushing και invalidation, βοηθητικές μέθοδοι και στατιστικά στοιχεία.

Και έτσι, αφού παρουσιάστηκαν τα κύρια κυκλώματα της ΚΜΕ και της Κύριας Μνήμης, ολοκληρώνεται το κεφάλαιο της Αρχιτεκτονικής και της Προσομοίωσης της με την γλώσσα προγραμματισμού C++.

Ενσωμάτωση Κρυπτογράφησης

Σε αυτό το κεφάλαιο θα παρουσιαστεί η λύση που προτείνω για την ασφάλιση των δεδομένων που αποθηκεύονται στην Κύρια Μνήμη και διατρέχουν κίνδυνο διαρροής των προγραμμάτων που τρέχουν.

Η πιο απλή για το υλικό, αποδοτική και αποτελεσματική χωρίς περίπλοκα συστήματα λύση είναι μία μέθοδος κρυπτογράφησης των δεδομένων που εισέρχονται στην Κύρια Μνήμη. Το ερώτημα είναι που θα τοποθετηθεί το εν λόγω κύκλωμα, τι αλγόριθμος θα χρησιμοποιηθεί, σε ποια σημεία θα γίνεται η κρυπτογράφηση και η αποκρυπτογράφηση της πληροφορίας, που θα αποθηκευτεί το κλειδί αλλά και τι προβλήματα ενδέχεται να δημιουργηθούν.

Λαμβάνοντας τα παραπάνω ερωτήματα υπ' όψη καθώς και εστιάζοντας στην οπτική γωνία της cold boot επίθεσης, αποφάσισα να σχεδιάσω και να πειραματιστώ με μία οργάνωση αρχιτεκτονικής όπου η ευθύνη της κρυπτογράφησης/αποκρυπτογράφησης δεν γίνεται σε ένα σημείο, δεν γίνεται με μία μονάδα κυκλώματος, παρά οι ευθύνες χωρίζονται σε δύο σημεία και είναι λιγότερο επεμβατική για αρχιτεκτονικές όπου βασίζονται οι σύγχρονοι Η/Υ καθώς και μοιράζονται με τα ενσωματωμένα συστήματα και η ενσωμάτωση της είναι δυναμική και εύκολη, ανάλογα δε με μοντέλο προστασία που ακολουθεί ο κατασκευαστής.

Η λύση: Η αποκρυπτογράφηση να γίνεται αποκλειστικά κατά την είσοδο των δεδομένων στον επεξεργαστή μέσω της ειδικής κρυφής μνήμης Cache and Decryption Unit/Κρυφή Μνήμη και Μονάδα Αποκρυπτογράφησης στο επίπεδο 3 της ιεραρχίας μνημών και η κρυπτογράφηση των στοιχείων να γίνεται αποκλειστικά κατά την είσοδο τους στη Κύριας Μνήμη μέσω του ελεγκτή της transparently (διάφανα), όπου σε αυτή την αρχιτεκτονική βρίσκεται τοποθετημένος στην πλευρά της μνήμης, χωρίς να το αντιλαμβάνεται κανείς κατά την διάδοση της πληροφορίας.

Αυτή η αρχιτεκτονική και οργάνωση δημιουργεί όμως κάποια κρίσιμα ερωτήματα, ποιός θα δημιουργεί και πώς θα μοιράζεται το κλειδί; Τί θα γίνει εάν υποκλαπεί η διαδικασία διαμοιρασμού του; Πριν δώσουμε απάντηση στα ερωτήματα αυτά ας εξετάσουμε την αρχιτεκτονική, πάλι με την βοήθεια της C++.

Μονάδα Κρυπτογράφησης

Ας αρχίσουμε από την πλευρά της Μνήμης, ή πιο συγκεκριμένα του ελεγκτή της Μνήμης όπου βρίσκεται δίπλα από την Κύρια Μνήμη, στην μητρική πλακέτα. Στον memory controller λοιπόν έχει ενσωματωθεί η μονάδα κρυπτογράφησης. Τα δεδομένα έρχονται από τον επεξεργαστή, μέσω της κρυφής μνήμης 3ου επιπέδου και του κεντρικού διαύλου επικοινωνίας block ανά block. Άρα με τη χρήση της write_physical_block(), κρυπτογραφείται όλο το block που εισάγεται στην Μνήμη.

```
// Memory encrypts and stores blocks (encryption-only)
void Memory::write_physical_block(Word physical_addr, const Word*
data, int word_count)
{
    if (physical_addr+(word_count*4)>=memory.size())
    {
        std::cout<<"ERROR: Memory block write out of bounds -
addr: 0x"<<std::hex<<physical_addr<<", size:
"<<std::dec<<(word_count*4)<<" bytes"<<std::endl;
        return;
    }

    if (word_count<=0)
    {
        std::cout<<"ERROR: Invalid block size: "<<word_count<<"
words"<<std::endl;
        return;
    }

    if (encryption_enabled&&Cipher::isKeyUpdated())
    {
        // Align to block boundary for consistent encryption
        Word aligned_addr=alignToMemoryBlock(physical_addr);

        // Encrypt the entire block using AES-GCM
        std::vector<Word> encrypted_data(word_count);
        if (Cipher::encryptBlock(data, encrypted_data.data(),
word_count, aligned_addr))
        {
            // Store encrypted block in memory
            for (int i=0; i<word_count; i++)
            {
                Word encrypted_word=encrypted_data[i];
                Word addr=physical_addr+i*4;

memory[addr+0]=static_cast<Byte>((encrypted_word>>0)&0xFF);

memory[addr+1]=static_cast<Byte>((encrypted_word>>8)&0xFF);
```

```
memory[addr+2]=static_cast<Byte>((encrypted_word>>16)&0xFF);

memory[addr+3]=static_cast<Byte>((encrypted_word>>24)&0xFF);
            }

            std::cout<<"Memory: Encrypted and stored
"<<word_count<<" words at
0x"<<std::hex<<physical_addr<<std::endl;
        }
        else
        {
            std::cout<<"ERROR: Block encryption failed for
address 0x"<<std::hex<<physical_addr<<std::endl;
            // Fallback: store unencrypted (DELETE!)
            for (int i=0; i<word_count; i++)
                write_physical_word(physical_addr+i*4,data[i]);
        }
    }
    else
    {
        // Store unencrypted when encryption is disabled or no
key
        std::cout<<"Memory::write_physical_block: UNENCRYPTED!!!
Storing unencrypted block (encryption disabled or no
key)"<<std::endl;
        for (int i=0; i<word_count; i++)
            write_physical_word(physical_addr+i*4,data[i]);
    }
}
```

Αν και γίνονται κάποιοι έλεγχοι, όπως στην περίπτωση που η κρυπτογράφηση έχει απενεργοποιηθεί από τον χρήστη ή εάν αποτύχει ο μηχανισμός της κρυπτογράφησης, καλό θα είναι να αγνοηθούν εντελώς και να επικεντρωθούμε στο γεγονός ότι, όταν η CDU καλεί αυτήν την μέθοδο όταν κάνει writeback (writeBackBlock):

```
void CacheDecryptionUnit::writeBackBlock(EncryptedBlock& block)
{
    if (!block.valid||!block.dirty) return;

    std::cout<<"CDU: Writing back dirty block at
0x"<<std::hex<<block.base_addr<<std::endl;

    // Use Memory's efficient block write operation
    // Memory controller will handle the encryption automatically
    physical_memory->write_physical_block(block.base_addr,
block.decrypted_data, CDU_BLOCK_SIZE);

    block.dirty=false;
```

```
    // Could Update the cached encrypted data to reflect what's
now in memory
    // This is not strictly necessary but helps with debugging
    if (encryption_enabled&&Cipher::isKeyUpdated()) {
        // I could re-encrypt to update the encrypted_data cache,
but it's not critical
        // since I reload from memory on next access anyway
    }
}
```

Η CDU περνάει το block και τη διεύθυνση μνήμης που θα αποθηκευτεί (καθώς και το μέγεθος του block όπου η μνήμη πρέπει να αναμένει) και ο memory controller ελέγχει εάν η διεύθυνση είναι έγκυρη, κάνει align (ευθυγράμμιση) του block, αν και αυτό έχει ήδη συμμορφωθεί από την CDU, και ύστερα καλεί τον μηχανισμό κρυπτογράφησης (θα αναλυθεί αμέσως μετά) όπου επιστρέφει ολόκληρο το block κρυπτογραφημένο για να γραφτεί επιτυχώς στην μνήμη. Τελικά, η μνήμη θα είναι κρυπτογραφημένη καθώς κάθε φορά που γράφεται, τα δεδομένα κρυπτογραφούνται.

Δημιουργείται ένα μεγάλο πρόβλημα όμως, τί γίνεται κατά την πρώτη εκκίνηση του Η/Υ όπου τα δεδομένα προέρχονται από την μη πτητική μνήμη;

Ας θυμηθούμε ότι παρότι η KME νομίζει ότι όλες οι μνήμες είναι ένα, στην πραγματικότητα είναι διαφορετικά στοιχεία. Ο ελεγκτής της μνήμης λοιπόν (η MMU χρησιμοποιείται κυρίως για μετάφραση διευθύνσεων, οπότε δεν χρειάζεται να αναμειχθεί με την διαδικασία), καταλαβαίνει ότι τα δεδομένα έρχονται από αλλού, και όπως και έρχονται από την KME και κρυπτογραφούνται, έτσι κρυπτογραφούνται και από τον σκληρό δίσκο με την χρήση της write_physical_word():

```
// Program loading with block-based encryption
bool Memory::load_bootloader(const std::string& filename)
{
    std::ifstream file(filename);
    if (!file.is_open())
    {
        std::cout<<"Error: Could not read boot file:
"<<filename<<std::endl;
        return false;
    }

    std::cout<<"load_bootloader: Memory controller will encrypt
all instructions using AES-GCM!"<<std::endl;

    if (encryption_enabled&&Cipher::isKeyUpdated())
        std::cout<<"load_bootloader: Quantum-safe encryption
ENABLED!"<<std::endl;
    else
        std::cout<<"load_bootloader: WARNING - No encryption key
or encryption disabled!"<<std::endl;

    std::string line;
```

```
    std::vector<Word> instruction_buffer; // Store all
instructions here

    // Read all instructions into buffer first
    while (std::getline(file, line))
    {
        if (line.empty()) continue;

        Word instruction=std::stoul(line,nullptr,16);
        instruction_buffer.push_back(instruction);
    }

    file.close();

    if (instruction_buffer.empty())
    {
        std::cout<<"ERROR: No instructions loaded from file,
empty file."<<std::endl;
        return false;
    }

    // Check if program fits in memory
    if ((instruction_buffer.size()*4)>(memory.size()-
TRANSLATED_ROM_BASE))
    {
        std::cout<<"ERROR: Program cannot fit in
memory!"<<std::endl;
        return false;
    }

    // Pad the instruction buffer to complete block boundaries
    size_t total_instructions=instruction_buffer.size();
    size_t
instructions_needed_for_complete_blocks=((total_instructions+MEMO
RY_BLOCK_SIZE-1)/MEMORY_BLOCK_SIZE)*MEMORY_BLOCK_SIZE;

    // Pad with NOPs to fill complete blocks
    while
(instruction_buffer.size()<instructions_needed_for_complete_block
s)
        instruction_buffer.push_back(0);

    std::cout<<"Padded program from "<<total_instructions<<" to
"<<instruction_buffer.size()<<" instructions for complete
blocks"<<std::endl;

    // Write instructions using block operations
```

```
    for (size_t i=0; i<instruction_buffer.size();
i+=MEMORY_BLOCK_SIZE)
    {
        Word block_addr=TRANSLATED_ROM_BASE+i*4;


write_physical_block(block_addr,&instruction_buffer[i],MEMORY_BLO
CK_SIZE);

        std::cout<<"Wrote block "<<(i/MEMORY_BLOCK_SIZE)<<" at
address 0x"
                 <<std::hex<<block_addr<<"
("<<std::dec<<MEMORY_BLOCK_SIZE
                 <<" words)"<<std::endl;
    }

    std::cout<<"Program loaded: "<<total_instructions<<"
instructions ("
             <<std::dec<<(total_instructions*4)<<" bytes) at
physical address 0x"
             <<std::hex<<TRANSLATED_ROM_BASE<<std::endl;
    std::cout<<"Total blocks written:
"<<(instruction_buffer.size()/MEMORY_BLOCK_SIZE)<<std::endl;

    return true;
}
```

Μάλιστα, στην περίπτωση που το πρόγραμμα εκκίνησης δεν καλύπτει ολόκληρο block, τότε γράφοντα μηδενικά (NOPs) για να γεμίσει το block. Δηλαδή τώρα λόγω της κρυπτογράφησης χαλάμε παραπάνω χώρο; Όχι, είναι το κλασσικό μειονέκτημα της χρήσης block και δεν έχει να κάνει καθόλου με την κρυπτογράφηση. Παρ' όλα αυτά, ας μην ξεχνάμε ότι το πλεονέκτημα διαχείρισης δεδομένων block ανά block, ως προς την συνολική απόδοση, είναι πιο σημαντικό.

Μονάδα Αποκρυπτογράφησης

Ας αναλύσουμε καλύτερα την CDU, Cache and Decryption Unit όπου φέρει ενσωματωμένα σε μία Cache την Μονάδα Αποκρυπτογράφησης. Ένα θετικό με αυτή την αρχιτεκτονική, είναι ότι επιθέσεις τύπου Reverse Engineering στο συγκεκριμένο κύκλωμα αυτομάτως δυσκολεύουν υπερβολικά πολύ, καθώς βρίσκεται μέσα στην KME, και συγκεκριμένα μέσα σε μία Cache.

Ας αναλύσουμε ολόκληρη την CDU:

```
// Compiler Directives

#pragma once

// Header files
#include "Cipher.h"
#include "Memory.h"
#include <vector>
#include "KeyExchangeModule.h"

class CacheDecryptionUnit
{
private:
    Memory* physical_memory;

    // FIPS203 Module
    KeyExchangeModule key_exchange_module;

    // CDU Configuration
    static const int CDU_BLOCK_SIZE=16;  // 16 words (64 bytes)
encryption blocks (Adjust with encryption method)
    static const int CDU_CACHE_LINES=64; // Small cache for
encrypted blocks

    // Encryption Controller
    bool encryption_enabled;

    struct EncryptedBlock
    {
        bool valid;
        bool dirty;
        Address base_addr;
        Word encrypted_data[CDU_BLOCK_SIZE];
        Word decrypted_data[CDU_BLOCK_SIZE];

        EncryptedBlock():valid(false),dirty(false),base_addr(0)
{}
    };
```

```
    std::vector<EncryptedBlock> cdu_cache;

    // Statistics
    int hits,misses,decryptions;

    // Helper methods
    Address alignToCDUBlock(Address addr) const; // Make sure it
is the same with the Memory's method
    int getCDUIndex(Address addr) const;
    void loadEncryptedBlock(Address base_addr, EncryptedBlock&
block);
    void writeBackBlock(EncryptedBlock& block);

    // Encryption/Decryption methods
    void decryptBlock(const Word* ciphertext, Word* plaintext,
int size, Address base_addr);

public:
    CacheDecryptionUnit(Memory* mem_ptr);

    // Encryption Control
    void enableEncryption();
    void disableEncryption();

    bool performKeyExchange();
    bool isSecureKeyEstablished() const;

    // Interface for L2 Cache - these methods handle
encryption/decryption transparently (AVOID)
    Word readWord(Address physical_addr);
    void writeWord(Address physical_addr, Word value);

    // Block operations for efficient L2 cache line
fills/writebacks
    bool readBlock(Address physical_addr, Word* data, int
word_count);
    bool writeBlock(Address physical_addr, const Word* data, int
word_count);

    // Cache management
    void flushAll();
    void invalidateAddress(Address addr);
    void invalidateAll();

    // Statistics
    void printStats() const;
    void resetStats();
    void dumpAllBlocks() const;
```

```
};
```

Όπως εξηγήσαμε, η CDU είναι ουσιαστικά μία Direct-Mapped κρυφή μνήμη επιπέδου 3 οπότε πρέπει να ορίσουμε πόσα δεδομένα θα κρατάει καθώς πόσες λέξεις χωράνε σε ένα block (Αυτή η τιμή πρέπει να είναι συνεπής με τον ελεγκτή της μνήμης). Παρατηρούμε δε την κατασκευή ενός αντικειμένου KeyExchangeModule με το σχόλιο FIPS-203, θα επανέλθουμε σε αυτό αργότερα. Ορίζουμε επίσης την σύνθετη δομή που θα τοποθετήσουμε τα κρυπτογραφημένα δεδομένα όπως έρχονται απευθείας από την μνήμη καθώς με τα metadata τους (άρα πιάνει περισσότερο χώρο κάθε block). Στατιστικά στοιχεία και βοηθητικές μέθοδοι για ευθυγράμμιση του block ως προς την διεύθυνση του, flush, invalidation καθώς και μία μέθοδος performKeyExchange(), όπου αφορά άμεσα το KeyExchangeModule και συνεργάζεται με την handleKeyExchange() του ελεγκτή της μνήμης και θα αναλυθούν αργότερα.

Ας αναλύσουμε λίγο τις σχετικότερες μεθόδους:

```
Address CacheDecryptionUnit::alignToCDUBlock(Address addr) const
{
    return addr&(~((CDU_BLOCK_SIZE*4)-1)); // Make sure it is the
same with the Memory's method
}

int CacheDecryptionUnit::getCDUIndex(Address addr) const
{
    return (addr/(CDU_BLOCK_SIZE*4))%CDU_CACHE_LINES;
}
```

Υπενθύμιση, πρέπει τα χαρακτηριστικά τους να είναι συνεπή με τον ελεγκτή μνήμης.

Ο επεξεργαστής, πιο συγκεκριμένα η κρυφή μνήμη επιπέδου 2, αναζητεί τα δεδομένα που αντιστοιχούν σε μία φυσική διεύθυνση με τη χρήση της μεθόδου readBlock().

```
bool CacheDecryptionUnit::readBlock(Address physical_addr, Word*
data, int word_count)
{
    if (!data||word_count<=0)
    {
        std::cout<<"CDU: Invalid block read
parameters"<<std::endl;
        return false;
    }

    Address start_addr=physical_addr;
    int words_read=0;

    while (words_read<word_count)
    {
        Address current_addr=start_addr+(words_read*4);
        Address base_addr=alignToCDUBlock(current_addr);
        int cache_index=getCDUIndex(base_addr);
        int word_offset=(current_addr-base_addr)/4;
```

```
        if (word_offset>=CDU_BLOCK_SIZE)
        {
            std::cout<<"CDU: ERROR - Word offset
"<<word_offset<<" exceeds block size"<<std::endl;
            return false;
        }

        EncryptedBlock& block=cdu_cache[cache_index];

        // Load block if not present or wrong block
        if (!block.valid||block.base_addr!=base_addr)
        {
            if (block.valid&&block.dirty)
{writeBackBlock(block);}
            loadEncryptedBlock(base_addr,block);
            misses++;
        }
        else hits++;

        // Copy decrypted words from this block
        int words_from_this_block=std::min(word_count-words_read,
CDU_BLOCK_SIZE-word_offset);

std::memcpy(&data[words_read],&block.decrypted_data[word_offset],
words_from_this_block*sizeof(Word));

        words_read+=words_from_this_block;
    }

    return true;
}
```

Στην περίπτωση που τα ζητούμενα δεδομένα δεν βρεθούν, γίνεται έλεγχος εάν υπάρχει dirty (πειραγμένο) block και φορτώνεται το block που ζητείται απευθείας από την μνήμη (κρυπτογραφημένο). Ας δούμε πολύ γρήγορα την writeBackBlock() πριν εξηγήσουμε την loadEncryptedBlock().

```
void CacheDecryptionUnit::writeBackBlock(EncryptedBlock& block)
{
    if (!block.valid||!block.dirty) return;

    std::cout<<"CDU: Writing back dirty block at
0x"<<std::hex<<block.base_addr<<std::endl;

    // Use Memory's efficient block write operation
    // Memory controller will handle the encryption automatically
    physical_memory->write_physical_block(block.base_addr,
block.decrypted_data, CDU_BLOCK_SIZE);
```

```
    block.dirty=false;

    // Could Update the cached encrypted data to reflect what's
now in memory
    // This is not strictly necessary but helps with debugging
    if (encryption_enabled&&Cipher::isKeyUpdated()) {
        // I could re-encrypt to update the encrypted_data cache,
but it's not critical
        // since I reload from memory on next access anyway
    }
}
```

Και η loadEncryptedBlock():

```
void CacheDecryptionUnit::loadEncryptedBlock(Address base_addr,
EncryptedBlock& block)
{
    if (!physical_memory->read_physical_block(base_addr,
block.encrypted_data, CDU_BLOCK_SIZE))
    {
        std::cout<<"CDU: Failed to read encrypted block from
memory at 0x"<<std::hex<<base_addr<<std::endl;
        // Just fill with error pattern on read failure
        for (int i=0; i<CDU_BLOCK_SIZE; i++)
        {
            block.encrypted_data[i]=0xFFFFFFFF;
            block.decrypted_data[i]=0xF0F0F0F0;
        }
        block.valid=false;
        return;
    }

    // Decrypt the entire block in one AES-GCM operation
    if (encryption_enabled&&Cipher::isKeyUpdated())
    {
        if (!Cipher::decryptBlock(block.encrypted_data,
block.decrypted_data, CDU_BLOCK_SIZE, base_addr))
        {
            // Authentication failed - this is a security issue
            std::cout<<"CDU: AES-GCM decryption/Authentication
failed for block at 0x"<<std::hex<<base_addr<<" - possible
tampering detected!"<<std::endl;

            // Fill with an error pattern
            for (int i=0; i<CDU_BLOCK_SIZE; i++)
            {
                block.decrypted_data[i]=0xF0F0F0F0;
```

```
            }
            // Mark as valid but with error data
            block.valid=false;
            //block.dirty=false;
            block.base_addr=base_addr;
            //decryptions++; // Count failed decryptions too
            return;
        }

        std::cout<<"CDU: Successfully decrypted block at
0x"<<std::hex<<base_addr<<std::endl;
    }
    else
    {
        // Copy encrypted data as-is when encryption is disabled
        std::cout<<"CDU: WARNING - Encryption failed/disabled!
Copying encrypted data as plaintext!"<<std::endl;
        std::memcpy(block.decrypted_data, block.encrypted_data,
CDU_BLOCK_SIZE*sizeof(Word));
    }

    decryptions++;
    block.valid=true;
    block.dirty=false;
    block.base_addr=base_addr;
}
```

Διαβάζεται το κρυπτογραφημένο μπλοκ απευθείας από την μνήμη μέσω της read_physical_block(). Σε περίπτωση αποτυχίας, γράφονται ειδικές τιμές για να σηματοδοτήσουν πρόβλημα. Έπειτα, αναλαμβάνει η μονάδα αποκρυπτογράφησης και αποκρυπτογραφεί το μπλοκ.

Αντίστοιχα λειτουργεί και η wrteBlock():

```
bool CacheDecryptionUnit::writeBlock(Address physical_addr, const
Word* data, int word_count)
{
    if (!data||word_count<=0)
    {
        std::cout<<"CDU: Invalid block write
parameters"<<std::endl;
        return false;
    }

    Address start_addr=physical_addr;
    int words_written=0;

    while (words_written<word_count)
    {
```

```
        Address current_addr=start_addr+(words_written*4);
        Address base_addr=alignToCDUBlock(current_addr);
        int cache_index=getCDUIndex(base_addr);
        int word_offset=(current_addr-base_addr)/4;

        if (word_offset>=CDU_BLOCK_SIZE)
        {
            std::cout<<"CDU: ERROR - Word offset
"<<word_offset<<" exceeds block size"<<std::endl;
            return false;
        }

        EncryptedBlock& block=cdu_cache[cache_index];

        // Load block if not present or wrong block
        if (!block.valid||block.base_addr!=base_addr)
        {
            if (block.valid&&block.dirty)
{writeBackBlock(block);}
            loadEncryptedBlock(base_addr,block);
            misses++;
        }
        else hits++;

        // Write words to this block
        int words_to_this_block=std::min(word_count-
words_written, CDU_BLOCK_SIZE-word_offset);
        std::memcpy(&block.decrypted_data[word_offset],
&data[words_written], words_to_this_block*sizeof(Word));

        block.dirty=true;
        words_written+=words_to_this_block;
    }

    return true;
}
```

Αλγόριθμος Συμμετρικής Κρυπτογράφησης

Παραπάνω εξηγήσαμε πως και που θα τοποθετηθούν οι μονάδες αποκρυπτογράφησης/κρυπτογράφησης. Μάλιστα. αναλύσαμε πως λειτουργούν και αυτό χωρίς να αποκαλύψουμε τον συμμετρικό αλγόριθμο κρυπτογράφησης που χρησιμοποιείται στην αρχιτεκτονική αυτή.

Υπάρχουν 2 ειδών αλγορίθμων κρυπτογράφησης, συμμετρικοί και ασύμμετροι.
Η συμμετρική κρυπτογράφηση χρησιμοποιεί το ίδιο κλειδί για την κρυπτογράφηση και για την αποκρυπτογράφηση των δεδομένων και χρησιμοποιείται για γρήγορη μεταφορά μεγάλων όγκων δεδομένων καθώς οι συμμετρική αλγόριθμοι κρυπτογράφησης, αφού με κάποιον τρόπο έχει κοινοποιηθεί και στις δύο πλευρές το κλειδί, και είναι οι πιο αποδοτικοί.
Ενώ η ασύμμετρη κρυπτογράφηση, χρησιμοποιεί ένα ζεύγος κλειδιών, το δημόσιο (κρυπτογράφηση) και ιδιωτικό κλειδί (για αποκρυπτογράφηση) και είναι οι κατάλληλοι αλγόριθμοι για την δημιουργία μία συνεδρίας (session) συμμετρικής κρυπτογράφησης, δηλαδή για την ανταλλαγή του συμμετρικού κλειδιού.

Σε αυτό τον κεφάλαιο θα εξηγήσουμε την στατική κλάση Cipher όπου χρησιμοποιεί τον συμμετρικό αλγόριθμο AES-GCS, όπου έχουν «ενσωματώσει» οι κρυπτογράφησης και αποκρυπτογράφησης. Πριν μπούμε στην Cipher.h, ας αναλύσουμε τον AES-GCM.

Advanced Encryption Standard – Galois/Counter Mode

Ο AES-GCM (Advanced Encryption Standard – Galois/Counter Mode) είναι ένας σύγχρονος συμμετρικός αλγόριθμος κρυπτογράφησης που συνδυάζει την κρυπτογράφηση με τον έλεγχο ακεραιότητας και αυθεντικοποίησης (Authenticated Encryption with Associated Data/AEAD). Συνδυάζει:
Κρυπτογράφηση με Counter Mode (CTR): Σε κάθε μπλοκ δεδομένων εφαρμόζεται XOR της αρχικής μορφής με την κρυπτογραφημένη έκδοση ενός μετρητή (Counter). Μάλιστα, αυτή η προσέγγιση επιτρέπει την παραλληλοποίηση της κρυπτογράφησης/αποκρυπτογράφησης αυξάνοντας την απόδοση καθώς κάθε μπλοκ μπορεί να επεξεργαστεί ανεξάρτητα.
Έλεγχος ακεραιότητας και αυθεντικότητας (Galois Message Authentication Code/GMAC): Ο GMAC παρέχει αυθεντικοποίηση και έλεγχο ακεραιότητας χρησιμοποιώντας κατακερματισμό (hashing) με πολυωνυμικό hash (GHash) στο πεδίο Galois (Galois Field) 2^{128}, για να παράγει ένα authentication tag (ετικέτα αυθεντικοποίησης) που επιβεβαιώνει την ακεραιότητα και αυθεντικότητα των δεδομένων που περιέχονται στο μπλοκ.

Ο AES-GCM (NIST SP 800-38D) προέρχεται από τον κλασσικό AES που τυποποιήθηκε από τον NIST (FIPS 197) στις 26 Νοεμβρίου του 2001. Επίσης ενσωματώθηκε επίσημα στο TLS (Transport Layer Security) (RFC 5288), που χρησιμοποιείται για την κρυπτογράφηση επικοινωνιών στο διαδίκτυο, όπως μέσω του HTTPS (Hypertext Transfer Protocol - Secure) πρωτοκόλλου που χρησιμοποιούν οι περιηγητές/browsers καθώς και στο IPSec ESP πρωτόκολλο (RFC 4106) και για την προστασία του IEEE 802.11 (Wi-Fi).

Ο AES-GCM υποστηρίζει κλειδιά (K) μεγέθους 128,192 και 256 Bit και συνήθως χρησιμοποιείται IV (Initialization Vector, Nonce) μεγέθους 96 Bit. Παράγεται Authentication Tag (T) μεγέθους 128 bit μέσω των AAD (Additional Authenticated Data), που χρησιμοποιείται για το «σφράγισμα» και την επαλήθευση του μπλοκ.

Λίγο πιο τεχνικά, η διαδικασία της κρυπτογράφησης ξεκινά με τη δημιουργία του hash subkey $H=AES\_K(0^{128})$, όπου ένα μηδενικό μπλοκ, από 128 bit, κρυπτογραφείται με το μυστικό κλειδί K. Στη συνέχεια, κατασκευάζεται ο αρχικός μετρητής J_0 από το Initialization Vector: αν το IV έχει μήκος 96 bits, τότε $J_0=IV||0^{31}||1$, αλλιώς $J_0 = GHASH\_H(IV||0^{(s+64)}||[len(IV)]_{64})$. Κάθε μπλοκ κειμένου C_i κρυπτογραφείται ως $C\_i=P\_i \oplus AES\_K(CTR\_i)$, όπου $CTR\_i = INC^i(J_0)$ με την INC να είναι η συνάρτηση αύξησης που εφαρμόζεται i φορές. Παράλληλα, η συνάρτηση GHASH υπολογίζει το authentication tag, επεξεργαζόμενη όλα τα δεδομένα (AAD και κρυπτοκείμενο C) ως πολυώνυμο στο πεδίο Galois $GF(2^{128})$: $GHASH\_H(A,C)=(A_1 \cdot H^{m+n}+A_2 \cdot H^{m+n-1}+...+A\_m \cdot H^{n+1})+(C_1 \cdot H^n+C_2 \cdot H^{n-1}+...+C\_n \cdot H^1)+len(A,C) \cdot H$, όπου τα A_i είναι τα μπλοκ του Additional Authenticated Data, τα C_i τα μπλοκ του κρυπτογραφημένου κειμένου, και len(A,C) ένα 128-bit μπλοκ που περιέχει τα μήκη των A και C. Τελικά, το authentication tag T υπολογίζεται ως $T=GHASH\_H(A,C) \oplus S$, όπου $S=AES\_K(J_0)$ είναι η κρυπτογραφημένη έκδοση του αρχικού μετρητή, και στη συνέχεια περικόπτεται στο επιθυμητό μήκος (συνήθως 128 bit), εξασφαλίζοντας έτσι ότι το τελικό αποτέλεσμα περιλαμβάνει το κρυπτοκείμενο μαζί με την ετικέτα αυθεντικότητας που επιβεβαιώνει τόσο την ακεραιότητα όσο και την αυθεντικότητα των δεδομένων.

Τέλος, ας αναλύσουμε τα πλεονεκτήματα και τα μειονεκτήματα το αλγορίθμου:
Πλεονεκτήματα:
Έχει αποδειχθεί ότι είναι από τους πιο αποδοτικούς αλγορίθμους για το υλικό.
Ταυτόχρονα με την κρυπτογράφηση παρέχεται και αυθεντικοποίηση των δεδομένων.
Έχει χρησιμοποιηθεί σε πολλές κρίσιμες εφαρμογές για την καθημερινότητα μας.

Μειονεκτήματα:
Το κρισιμότερο μειονέκτημα, ειδικότερα για αυτήν την εφαρμογή, είναι η ευαισθησία της επαλήθευσης μέσω του authentication tag σε timing attacks (επιθέσεις χρονισμού), κάτι το οποίο έχει αποδειχθεί ότι ισχύει και για άλλους αλγορίθμους.
Επίσης, πρέπει να αποφεύγεται η επαναχρησιμοποίηση του ίδιου IV καθώς μπορεί να αποκαλυφθεί το κλειδί.

Συνεπώς, είναι ένας αρκετά καλός αλγόριθμος για την συγκεκριμένη εφαρμογή και μάλιστα υπάρχουν ήδη τυπωμένα κυκλώματα όπου τον χρησιμοποιούν, και στην περίπτωση που η λύση αυτή τυπωθεί και ενσωματωθεί στα κυκλώματα, δηλαδή σε ASIC (Application Specific Integrated Circuit/Ολοκληρωμένο Κύκλωμα Ειδικής Εφαρμογής), το κόστος μειώνεται.

Ας εξετάσουμε το Cipher header file που ορίζει τις στατικές μεθόδους που χρησιμοποιούνε οι μονάδες κρυπτογράφησης/αποκρυπτογράφησης. Ο λόγος που δεν γράφτηκαν ξεχωριστά για την κάθε μονάδα, είναι για την εύκολη χρήση της Cipher κατά την φάση του πειράματος και για καλύτερη ανάλυση του AES-GCM. Ειδικότερα, για την απόδειξη ότι η λύση αυτή λειτουργεί χωρίς προβλήματα με τον AES-GCM όπως ακριβώς έχει πρωτυποποιηθεί, χρησιμοποιήθηκε η ειδική βιβλιοθήκη OpenSSL.

```
#pragma once

#include "data_types.h"
#include <cstring>
#include <array>
#include <random>
#include <map>
#include <openssl/evp.h>
#include <openssl/rand.h>

class Cipher
{
private:
    inline static std::array<uint8_t,32>aes_key={0}; // 256-bit
key
    inline static bool key_updated=false;
    inline static std::mt19937 nonce_generator;

    // AES-GCM parameters
    static constexpr size_t CIPHER_BLOCK_SIZE=16;
    static constexpr size_t CIPHER_IV_SIZE=12;
    static constexpr size_t CIPHER_TAG_SIZE=16;

    // Storage for metadata (IV and tags) indexed by block base
address
    inline static
std::map<Address,std::array<uint8_t,CIPHER_IV_SIZE>> iv_storage;
    inline static
std::map<Address,std::array<uint8_t,CIPHER_TAG_SIZE>>
tag_storage;
```

```
public:
    // Block encryption
    static bool encryptBlock(const Word* plaintext, Word*
ciphertext, int word_count, Address base_addr)
    {
        if (!key_updated||word_count<=0){return false;} // If
encryption disabled/failed return error
        // Generate random IV for this block
        std::array<uint8_t,CIPHER_IV_SIZE> iv;
        if (RAND_bytes(iv.data(),CIPHER_IV_SIZE)!=1)
        {

std::memcpy(ciphertext,plaintext,word_count*sizeof(Word));
            return false;
        }

        // Authentication
        // Use base address as Additional Authenticated Data
(AAD)
        uint8_t aad[sizeof(Address)];
        std::memcpy(aad,&base_addr,sizeof(Address));

        // Convert word array to byte array
        size_t plaintext_len=word_count*sizeof(Word);
        uint8_t*
plaintext_bytes=reinterpret_cast<uint8_t*>(const_cast<Word*>(plai
ntext));
        uint8_t*
ciphertext_bytes=reinterpret_cast<uint8_t*>(ciphertext);

        // Encrypt the entire block
        EVP_CIPHER_CTX* ctx=EVP_CIPHER_CTX_new();
        if (!ctx)
        {
            std::memcpy(ciphertext, plaintext, plaintext_len);
            return false;
        }

        int len,ciphertext_len=0;
        std::array<uint8_t,CIPHER_TAG_SIZE> tag;

        do
        { // ADD OUTPUT
            // Initialize encryption
            if (EVP_EncryptInit_ex(ctx, EVP_aes_256_gcm(),
nullptr, nullptr, nullptr)!=1) break;
```

```
            // Set IV length
            if (EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_GCM_SET_IVLEN,
CIPHER_IV_SIZE, nullptr)!=1) break;

            // Initialize key and IV
            if (EVP_EncryptInit_ex(ctx, nullptr, nullptr,
aes_key.data(), iv.data())!=1) break;

            // Set AAD
            if (EVP_EncryptUpdate(ctx, nullptr, &len, aad,
sizeof(Address))!=1) break;

            // Encrypt entire block
            if (EVP_EncryptUpdate(ctx, ciphertext_bytes, &len,
plaintext_bytes, plaintext_len)!=1) break;
            ciphertext_len=len;

            // Finalize encryption
            if (EVP_EncryptFinal_ex(ctx, ciphertext_bytes + len,
&len)!=1) break;
            ciphertext_len+=len;

            // Get authentication tag
            if (EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_GCM_GET_TAG,
CIPHER_TAG_SIZE, tag.data())!=1) break;

            // Store IV and tag for later decryption
            iv_storage[base_addr]=iv;
            tag_storage[base_addr]=tag;

            EVP_CIPHER_CTX_free(ctx);
            return true;

        } while(false);

        // Cleanup on error
        EVP_CIPHER_CTX_free(ctx);
        return false;
    }

    // Block decryption
    static bool decryptBlock(const Word* ciphertext, Word*
plaintext, int word_count, Address base_addr)
    {
        if (!key_updated||word_count<=0){return false;}

        // Find stored IV and tag for this block
        auto iv_it=iv_storage.find(base_addr);
```

```
        auto tag_it=tag_storage.find(base_addr);

        if (iv_it==iv_storage.end()||tag_it==tag_storage.end())
{return false;} // No metadata found, encryption is required

        const auto& iv=iv_it->second;
        const auto& tag=tag_it->second;

        // Use base address as Additional Authenticated Data
(AAD)
        uint8_t aad[sizeof(Address)];
        std::memcpy(aad, &base_addr, sizeof(Address));

        // Convert word arrays to byte arrays
        size_t ciphertext_len=word_count*sizeof(Word);
        const uint8_t* ciphertext_bytes=reinterpret_cast<const
uint8_t*>(ciphertext);
        uint8_t*
plaintext_bytes=reinterpret_cast<uint8_t*>(plaintext);

        // Decrypt the entire block
        EVP_CIPHER_CTX* ctx=EVP_CIPHER_CTX_new();
        if (!ctx) {return false;}

        int len,plaintext_len_out=0;
        bool success=false;

        do { // ADD OUTPUT
            // Initialize decryption
            if (EVP_DecryptInit_ex(ctx, EVP_aes_256_gcm(),
nullptr, nullptr, nullptr)!=1) break;

            // Set IV length
            if (EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_GCM_SET_IVLEN,
CIPHER_IV_SIZE, nullptr)!=1) break;

            // Initialize key and IV
            if (EVP_DecryptInit_ex(ctx, nullptr, nullptr,
aes_key.data(), iv.data())!=1) break;

            // Set AAD
            if (EVP_DecryptUpdate(ctx, nullptr, &len, aad,
sizeof(Address))!=1) break;

            // Decrypt entire block
            if (EVP_DecryptUpdate(ctx, plaintext_bytes, &len,
ciphertext_bytes, ciphertext_len)!=1) break;
            plaintext_len_out=len;
```

```
            // Set expected tag
            if (EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_GCM_SET_TAG,
CIPHER_TAG_SIZE, const_cast<uint8_t*>(tag.data()))!=1) break;

            // Finalize decryption and verify tag
            if (EVP_DecryptFinal_ex(ctx, plaintext_bytes+len,
&len)>0)
            {
                plaintext_len_out+=len;
                success=true;
            }

        } while(false);

        EVP_CIPHER_CTX_free(ctx);

        if (success)
            return true;
        else
        { // ADD OUTPUT!
            // Authentication failed - this is a serious security
issue
            // Zero out the plaintext and return error pattern
            for (int i=0; i<word_count; i++)
                plaintext[i]=0xF0F0F0F0;

            return false;
        }
    }

    // Set encryption key from quantum-safe key exchange
    static void setKey(const uint8_t* shared_secret, size_t
secret_len)
    {
        if (secret_len>=32)
        {
            // Use full 256-bit key
            std::memcpy(aes_key.data(), shared_secret, 32);
            key_updated=true;

            // Initialize nonce generator with part of the shared
secret
            uint32_t seed;
            std::memcpy(&seed, shared_secret+28,
sizeof(uint32_t));
            nonce_generator.seed(seed);
```

```
            // Clear old metadata when key changes
            iv_storage.clear();
            tag_storage.clear();
        }
    }

    // Reset to default state (no key)
    static void resetKey()
    {
        aes_key.fill(0);
        key_updated=false;
        iv_storage.clear();
        tag_storage.clear();
    }

    static bool isKeyUpdated()
    {
        return key_updated;
    }

    // Clear metadata for a specific address range (For cache
invalidation)
    static void clearMetadata(Address start_addr, Address
end_addr)
    {
        auto it=iv_storage.lower_bound(start_addr);
        while (it!=iv_storage.end()&&it->first<=end_addr)
            it=iv_storage.erase(it);

        auto tag_it=tag_storage.lower_bound(start_addr);
        while (tag_it!=tag_storage.end()&&tag_it-
>first<=end_addr)
            tag_it=tag_storage.erase(tag_it);
    }

    // Get storage statistics (for debugging)
    static size_t getMetadataCount()
    {
        return iv_storage.size();
    }
};
```

Υπάρχει και ένα κομμάτι κώδικα που μαρτυράει την επόμενη συζήτηση μας. Την ανταλλαγή του κλειδιού μεταξύ των 2 απομακρυσμένων μονάδων.

Τελικά, η διαδικασία αυτή, της κρυπτογράφησης και αποκρυπτογράφησης, δεν κοστίζει πολύ στην KME καθώς γίνεται στις περιπτώσεις όπου οι κρυφές μνήμες δεν περιέχουν τα δεδομένα όπου ζητούνται, αν και επιβαρύνει τον χρόνο μεταφοράς δεδομένων, όταν γίνεται αυτό, από και προς την Κύρια Μνήμη.

Ανταλλαγή Κλειδιού και Προστασία από Κβαντικούς Υπολογιστές

Όπως εξηγήσαμε και στα προηγούμενα κεφάλαια, η αλήθεια είναι ότι αυτοί οι αλγόριθμοι κρυπτογράφησης δεν επαρκούν πια λόγω του προβλήματος του SNDL. Αν κάποιος καταφέρει και καταγράψει την μέθοδο ανταλλαγής του συμμετρικού κλειδιού σε όλες τις προαναφερθείσες εργασίας, και τις πετάξει σε έναν κβαντικό υπολογιστή αργότερα, θα μπορέσει να ανακτήσει το κλειδί και να αποκρυπτογραφήσει τα υπόλοιπα δεδομένα.

Για την αντιμετώπιση του προβλήματος αυτού, στην αρχιτεκτονική όπου προτείνω, χρησιμοποιώ έναν αλγόριθμο όπου έχει προταθεί από το NIST (National Institute of Standards and Technology/Εθνικό Ινστιτούτο Προτύπων και Τεχνολογίας) των ΗΠΑ, μετά από χρόνια σχολαστική έρευνας και διαγωνισμών από πολλούς επιστήμονες και πρωτυποποίηθηκε τον Αύγουστου του 2023, τον Module-Lattice-Based Key-Encapsulation Mechanism (ML-KEM) Standard, με τον κωδικό FIPS 203.

Module-Lattice-Based Key-Encapsulation Mechanism (ML-KEM) Standard (FIPS-203)

Το Federal Information Processing Standard (FIPS) 203 αποτελεί ένα καινοτόμο κρυπτογραφικό πρότυπο που αναπτύχθηκε από το National Institute of Standards and Technology (NIST) των Ηνωμένων Πολιτειών ως μέρος της προσπάθειας για τη δημιουργία post-quantum κρυπτογραφικών αλγορίθμων. Το πρότυπο αυτό καθορίζει τον αλγόριθμο ML-KEM (Module-Lattice-based Key Encapsulation Mechanism), την τελική τυποποίηση του CRYSTALS-Kyber, του αλγορίθμου όπου επελέγη κατά την τρίτη φάση του προγράμματος Post-Quantum Cryptography Standardization της NIST. Ο αλγόριθμος βασίζεται σε μαθηματικές δομές γνωστές ως πλέγματα (lattices) και σχεδιάστηκε ειδικά για να παρέχει ασφάλεια έναντι επιθέσεων από κβαντικούς υπολογιστές.

Η θεμελιώδης αρχή του FIPS 203, βασίζεται στο μαθηματικό πρόβλημα Module-LWE (Learning With Errors over Modules), το οποίο γενικεύει το κλασικό πρόβλημα LWE σε πολυώνυμα με συντελεστές σε πεπερασμένο πεδίο. Συγκεκριμένα, οι πράξεις εκτελούνται στον δακτύλιο πολυωνύμων $\mathbb{Z}_q[x]/(x^n + 1)$, όπου q είναι πρώτος αριθμός (π.χ. 3329) και n είναι δύναμη του 2 (συνήθως 256). Κατά την κρυπτογράφηση, ένα δημόσιο κλειδί αποτελείται από έναν πίνακα πολυωνύμων και ο αποστολέας παράγει το ciphertext μέσω γραμμικών συνδυασμών των πολυωνύμων αυτών, μαζί με την προσθήκη θορύβου που ακολουθεί διακριτή κατανομή (συνήθως centered binomial/κεντρική διωνυμική κατανομή). Η αποκρυπτογράφηση βασίζεται στη χρήση του μυστικού πολυωνύμου για τον υπολογισμό του shared key (κοινού/δημόσιου κλειδιού), όπου η σωστή αφαίρεση του θορύβου είναι δυνατή μόνο αν γνωρίζει κανείς το ιδιωτικό κλειδί. Η υπολογιστική ασφάλεια του σχήματος ανάγεται στην υπολογιστική δυσκολία ανάκτησης του μυστικού από τη δημόσια πληροφορία, δεδομένης της ύπαρξης του θορύβου, καθιστώντας τον αλγόριθμο ανθεκτικό σε κβαντικές επιθέσεις και κατάλληλο για την αντικατάσταση των παραδοσιακών συστημάτων βασισμένων σε RSA ή ECC. Το πλέγμα ορίζεται ως ένα διακριτό υποσύνολο του Euclidean χώρου που παράγεται από γραμμικούς συνδυασμούς ενός συνόλου βασικών διανυσμάτων.

Ο ML-KEM αλγόριθμος που περιγράφεται στο FIPS 203 λειτουργεί μέσω τριών βασικών φάσεων:
Παραγωγή Κλειδιών (Key Generation):
Η διαδικασία παραγωγής κλειδιών περιλαμβάνει τη δημιουργία ενός ζεύγους κλειδιών (δημόσιου και ιδιωτικού) μέσω της επιλογής τυχαίων στοιχείων από συγκεκριμένες

κατανομές και την εφαρμογή μαθηματικών πράξεων σε δομές πλεγμάτων. Το δημόσιο κλειδί αποτελείται από έναν πίνακα και έναν διάνυσμα που προκύπτουν από τη λύση του MLWE προβλήματος.

Ενθυλάκωση Κλειδιού (Key Encapsulation):
Κατά τη φάση του εμπλουτισμού, ο αποστολέας χρησιμοποιεί τον δημόσιο κλειδί για να δημιουργήσει ένα κοινό μυστικό κλειδί και το αντίστοιχο ciphertext. Η διαδικασία αυτή περιλαμβάνει τη δημιουργία θορύβου και την εφαρμογή κρυπτογραφικών μετασχηματισμών που εξασφαλίζουν ότι μόνο ο κάτοχος του ιδιωτικού κλειδιού μπορεί να αποκρυπτογραφήσει το μήνυμα.

Αποκρυπτογράφηση Κλειδιού (Key Decapsulation):
Ο παραλήπτης χρησιμοποιεί το ιδιωτικό κλειδί για να ανακτήσει το κοινό μυστικό κλειδί από το ciphertext. Η διαδικασία αυτή περιλαμβάνει την αντιστροφή των μαθηματικών πράξεων που εφαρμόστηκαν κατά τον εμπλουτισμό και τη χρήση αλγορίθμων διόρθωσης σφαλμάτων για την ακριβή ανάκτηση του κλειδιού.

Το FIPS 203 ορίζει τρεις διαφορετικές παραμετροποιήσεις που αντιστοιχούν σε διαφορετικά επίπεδα ασφάλειας:
ML-KEM-512: Παρέχει ασφάλεια ισοδύναμη με συμμετρικό κλειδί 128 bit (όπου χρησιμοποιήθηκε για την προσομοίωση)
ML-KEM-768: Παρέχει ασφάλεια ισοδύναμη με συμμετρικό κλειδί 192 bit
ML-KEM-1024: Παρέχει ασφάλεια ισοδύναμη με συμμετρικό κλειδί 256 bit
Κάθε παραμετροποίηση διαφοροποιείται στο μέγεθος των κλειδιών, την πολυπλοκότητα των υπολογισμών και το επίπεδο ασφάλειας που παρέχει.

Τα κύρια πλεονεκτήματα του FIPS 203 περιλαμβάνουν την αντοχή σε κβαντικές επιθέσεις, τη σχετικά υψηλή απόδοση σε σύγκριση με άλλους post-quantum αλγορίθμους, και τη δυνατότητα παραμετροποίησης ανάλογα με τις απαιτήσεις ασφάλειας. Ωστόσο, παρουσιάζει και ορισμένους περιορισμούς, όπως το σχετικά μεγάλο μέγεθος των κλειδιών σε σύγκριση με τους παραδοσιακούς αλγορίθμους και την ανάγκη για προσεκτική υλοποίηση για την αποφυγή side-channel επιθέσεων.

Το FIPS 203 αναμένεται να αποτελέσει θεμελιώδη στοιχείο της μελλοντικής κρυπτογραφικής υποδομής, ιδιαίτερα σε εφαρμογές που απαιτούν υψηλό επίπεδο ασφάλειας. Η ενσωμάτωσή του σε κρυπτογραφικές βιβλιοθήκες και πρωτόκολλα επικοινωνίας αναμένεται να αυξηθεί σημαντικά τα επόμενα χρόνια, καθώς οι οργανισμοί προετοιμάζονται για την εποχή των κβαντικών υπολογιστών.

Τέλος, Το FIPS 203 αντιπροσωπεύει μια σημαντική πρόοδο στον τομέα της post-quantum κρυπτογραφίας, παρέχοντας ένα στιβαρό και αποδοτικό μηχανισμό εμπλουτισμού κλειδιών και μάλιστα η Google πειραματίζεται μαζί του για να τον ενσωματώσει στον περιηγητή Chrome. Πιο συγκεκριμένα, όπως ανακοίνωσε η ομάδα του Chrome, η κυρίαρχη υλοποίηση ML-KEM (codepoint 0x11EC) ενεργοποιήθηκε στην έκδοση 131 του Chrome, χρησιμοποιώντας τη βιβλιοθήκη BoringSSL, αντικαθιστώντας το παλαιότερο Kyber σε μίξη (hybrid TLS). Η τυποποίησή του από το NIST υπογραμμίζει τη σημασία του για τη μελλοντική ασφάλεια των ψηφιακών επικοινωνιών και τη μετάβαση προς κβαντικά-ανθεκτικές κρυπτογραφικές λύσεις.

Για να ενσωματώσω τον αλγόριθμο στην προσομοίωση μου, και συγκεκριμένα με τη χρήση του Key Exchange Module (Μονάδα Ανταλλαγής Κλειδιού), όπως ακριβώς προτυποποιήθηκε, χρησιμοποίησα την open-quantum-safe (OQS) βιβλιοθήκη, liboqd για τον διαμοιρασμό του κλειδιού:

```
// Compiler Directives
#pragma once

// Header Files
#include "data_types.h"
#include <oqs/oqs.h>
#include <array>

// FIPS 203 ML-KEM-512 parameters
constexpr size_t MLKEM_512_PUBLIC_KEY_BYTES=800;
constexpr size_t MLKEM_512_SECRET_KEY_BYTES=1632;
constexpr size_t MLKEM_512_CIPHERTEXT_BYTES=768;
constexpr size_t MLKEM_512_SHARED_SECRET_BYTES=32;

// Forward declaration to avoid circular dependency
class Memory;

class KeyExchangeModule
{
friend class CacheDecryptionUnit;
private:
    // Key Holders
    std::array<uint8_t, MLKEM_512_PUBLIC_KEY_BYTES> public_key;
    std::array<uint8_t, MLKEM_512_SECRET_KEY_BYTES> secret_key;
    std::array<uint8_t, MLKEM_512_SHARED_SECRET_BYTES>
shared_secret;

    // Helper Booleans
    bool keys_generated;
    bool shared_secret_established;

    // CDU side key generation and decapsulation
    bool generateKeysAndDecapsulate(Memory* memory_handle); //
Insert the memory pointer

public:
    // Constructor
    KeyExchangeModule();

    // Called by memory to perform encapsulation side
    bool performMemoryEncapsulation(const uint8_t*
cdu_public_key, uint8_t* ciphertext_out, uint8_t*
shared_secret_out);

    // Get the established shared secret (for CDU to use)
    const uint8_t* getSharedSecret() const;
    bool isSharedSecretReady() const;
```

```
    // Reset the key exchange
    void reset();
};
```

```
#include "Memory.h"
#include <iostream>
#include <cstring>

KeyExchangeModule::KeyExchangeModule():keys_generated(false),
shared_secret_established(false)
{
    // Initialize arrays to zero
    public_key.fill(0);
    secret_key.fill(0);
    shared_secret.fill(0);

    std::cout<<"KeyExchangeModule: ML-KEM-512 module
initialized"<<std::endl;
}

// CDU side: Generate keypair and handle key exchange with memory
bool KeyExchangeModule::generateKeysAndDecapsulate(Memory*
memory_handle)
{
    // Initialize ML-KEM-512
    OQS_KEM* kem=OQS_KEM_new(OQS_KEM_alg_ml_kem_512);
    if (!kem)
    {
        std::cout<<"KeyExchangeModule: Failed to initialize ML-
KEM-512"<<std::endl;
        return false;
    }

    bool success=false;

    // Generate keypair
    if (OQS_KEM_keypair(kem,
public_key.data(),secret_key.data())==OQS_SUCCESS)
    {
        keys_generated=true;
        std::cout<<"KeyExchangeModule: CDU keypair generated
successfully"<<std::endl;

        // Buffer for ciphertext from memory
        std::array<uint8_t,MLKEM_512_CIPHERTEXT_BYTES>
ciphertext;
        std::array<uint8_t,MLKEM_512_SHARED_SECRET_BYTES>
memory_shared_secret;
```

```
        // Call memory to perform its side of key exchange
(encapsulation)
        if (memory_handle-
>handleKeyExchange(public_key.data(),ciphertext.data(),memory_sha
red_secret.data(),this))
        {
            std::cout<<"KeyExchangeModule: Memory-side key
exchange completed"<<std::endl;

            // Perform decapsulation using the ciphertext from
memory
            if (OQS_KEM_decaps(kem,
shared_secret.data(),ciphertext.data(),secret_key.data())==OQS_SU
CCESS)
            {
                shared_secret_established=true;
                success=true;
                std::cout<<"KeyExchangeModule: CDU-side
decapsulation successful - shared secret established"<<std::endl;

                // DEBUG Verify that both sides have the same
shared secret
                if
(memcmp(shared_secret.data(),memory_shared_secret.data(),MLKEM_51
2_SHARED_SECRET_BYTES)==0)
                {
                    std::cout<<"KeyExchangeModule: Shared secrets
match - key exchange successful!"<<std::endl;
                }
                else
                {
                    std::cout<<"KeyExchangeModule: ERROR - Shared
secrets don't match!"<<std::endl;
                    success=false;
                    shared_secret_established=false;
                }
            }
            else
            {
                std::cout<<"KeyExchangeModule: CDU-side
decapsulation failed"<<std::endl;
            }
        }
        else
        {
            std::cout<<"KeyExchangeModule: Memory-side key
exchange failed"<<std::endl;
```

```
        }
    }
    else
    {
        std::cout<<"KeyExchangeModule: Failed to generate CDU
keypair"<<std::endl;
    }

    // Delete kem object
    OQS_KEM_free(kem);

    return success;
}

// Called by memory to perform encapsulation
bool KeyExchangeModule::performMemoryEncapsulation(const uint8_t*
cdu_public_key,uint8_t* ciphertext_out,uint8_t*
shared_secret_out)
{
    // Initialize ML-KEM-512
    OQS_KEM* kem=OQS_KEM_new(OQS_KEM_alg_ml_kem_512);
    if (!kem)
    {
        std::cout<<"KeyExchangeModule: Failed to initialize ML-
KEM-512 for encapsulation"<<std::endl;
        return false;
    }

    bool success=false;

    // Encapsulate: generate ciphertext and shared secret using
CDU's public key
    if (OQS_KEM_encaps(kem,
ciphertext_out,shared_secret_out,cdu_public_key)==OQS_SUCCESS)
    {
        std::cout<<"KeyExchangeModule: Memory-side encapsulation
successful"<<std::endl;
        success=true;
    }
    else
    {
        std::cout<<"KeyExchangeModule: Failed to encapsulate
key"<<std::endl;
    }

    // Delete kem object
    OQS_KEM_free(kem);
```

```
    return success;
}

// Get the established shared secret
const uint8_t* KeyExchangeModule::getSharedSecret() const
{
    return
shared_secret_established?shared_secret.data():nullptr; // Else
return
}

bool KeyExchangeModule::isSharedSecretReady() const
{
    return shared_secret_established;
}

// Reset the key exchange
void KeyExchangeModule::reset()
{
    public_key.fill(0);
    secret_key.fill(0);
    shared_secret.fill(0);
    keys_generated=false;
    shared_secret_established=false;
    std::cout<<"KeyExchangeModule: Module reset"<<std::endl;
}
```

Όπως ήδη ειπώθηκε, το key exchange module (kem), αποφάσισα να το ενσωματώσω στην CDU.

Παρακάτω παρατίθεται ο κώδικας όπου ξεκινάει η διαδικασία του διαμοιρασμού του κλειδιού (από την CDU).

```
bool CacheDecryptionUnit::performKeyExchange()
{
    std::cout<<"CDU: Initiating quantum-safe key exchange with
memory..."<<std::endl;

    // Key exchange
    if
(!key_exchange_module.generateKeysAndDecapsulate(physical_memory)
) // Pass the memory pointer
    {
        std::cout<<"CDU: Key exchange failed"<<std::endl;
        return false;
    }

    // Update CDU's cipher with the shared secret
```

```
    const uint8_t*
shared_secret=key_exchange_module.getSharedSecret();
    if (shared_secret)
    {

Cipher::setKey(shared_secret,MLKEM_512_SHARED_SECRET_BYTES);
        std::cout<<"CDU: Quantum-safe encryption key established
successfully"<<std::endl;

        // Invalidate all CDU cached data since encryption key
changed
        invalidateAll();

        return true;
    }

    std::cout<<"CDU: Failed to retrieve shared
secret"<<std::endl;
    return false;
}
```

Καλεί την generateKeysAndDecapsulate() του kem, όπου με την σειρά του στέλνει το σχετικό αίτημα στον Ελεγκτή της Μνήμης με το memory_handle-> handleKeyExchange(public_key.data(),ciphertext.data(),memory_shared_secret.data(),this) όπου στέλνει και τον δείκτη του module για διευκόλυνση της, όπου με την σειρά της:

```
// Key exchange handler - called by MEU
bool Memory::handleKeyExchange(const uint8_t* cdu_public_key,
uint8_t* ciphertext_out, uint8_t* shared_secret_out,
KeyExchangeModule* cdu_key_exchange_bus)
{
    std::cout<<"Memory: Handling key exchange request from
CDU..."<<std::endl;
    std::cout<<"Memory: Using CDU's KeyExchangeModule via data
bus..."<<std::endl;

    // Perform memory-side encapsulation
    if (cdu_key_exchange_bus-
>performMemoryEncapsulation(cdu_public_key,ciphertext_out,shared_
secret_out))
    {
        // Update memory's cipher with the shared secret

Cipher::setKey(shared_secret_out,MLKEM_512_SHARED_SECRET_BYTES);

        std::cout<<"Memory: Key exchange completed
successfully!"<<std::endl;
        std::cout<<"Memory: Encryption key updated, all future
writes will use quantum-safe encryption"<<std::endl;
```

```
        return true;
    }
    else
    {
        std::cout<<"Memory: Key exchange FAILED!"<<std::endl;
        return false;
    }
}
```

Όπου με την σειρά της ξανα-επικοινωνεί με τον kem με τη χρήση του δείκτη που έλαβε για να κάνει το encapsulation process:

```
// Called by memory to perform encapsulation
bool KeyExchangeModule::performMemoryEncapsulation(const uint8_t*
cdu_public_key,uint8_t* ciphertext_out,uint8_t*
shared_secret_out)
{
    // Initialize ML-KEM-512
    OQS_KEM* kem=OQS_KEM_new(OQS_KEM_alg_ml_kem_512);
    if (!kem)
    {
        std::cout<<"KeyExchangeModule: Failed to initialize ML-
KEM-512 for encapsulation"<<std::endl;
        return false;
    }

    bool success=false;

    // Encapsulate: generate ciphertext and shared secret using
CDU's public key
    if (OQS_KEM_encaps(kem,
ciphertext_out,shared_secret_out,cdu_public_key)==OQS_SUCCESS)
    {
        std::cout<<"KeyExchangeModule: Memory-side encapsulation
successful"<<std::endl;
        success=true;
    }
    else
    {
        std::cout<<"KeyExchangeModule: Failed to encapsulate
key"<<std::endl;
    }

    // Delete kem object
    OQS_KEM_free(kem);

    return success;
}
```

Όλα αυτά λοιπόν, θα γίνουν κατά το reset() του επεξεργαστή, δηλαδή ας παραθέσουμε ολόκληρη τη reset() τώρα:

```
// Processor's Reset Routine
void Processor::Reset()
{
    PC=MemorySegments::RESET_VECTOR; // Initialize the PC
register to the RESET_VECTOR

    hi_reg=0;           // Reset HI register
    lo_reg=0;           // Reset LO register

    // Reset pipeline control flags
    stall_pipeline=false;
    flush_pipeline=true;
    branch_taken=false;
    branch_target=0;

    // Invalidate Cache
    invalidate_all_caches();

    // Reset cache statistics
    reset_cache_stats();

    // Perform quantum-safe key exchange
    if (cdu.performKeyExchange())
    {
        std::cout<<"Quantum-safe encryption
established!"<<std::endl;

        // Verify secure key is established
        if (meu.isSecureKeyEstablished())
        {
            std::cout<<"Secure communication channel
active"<<std::endl;
        }
    }
    else
    {
        std::cout<<"Warning: Using default encryption
key"<<std::endl;
    }

    std::cout<<"Processor reset complete!"<<std::endl;
}
```

Και αυτό συνοψίζει τη διαδικασία διαμοιρασμού του συμμετρικού κλειδιού.

Μερικές βοηθητικές μέθοδοι:

```
bool CacheDecryptionUnit::isSecureKeyEstablished() const
{
    return
key_exchange_module.isSharedSecretReady()&&Cipher::isKeyUpdated()
;
}
```

```
// Get the established shared secret
const uint8_t* KeyExchangeModule::getSharedSecret() const
{
    return
shared_secret_established?shared_secret.data():nullptr; // Else
return
}

bool KeyExchangeModule::isSharedSecretReady() const
{
    return shared_secret_established;
}

// Reset the key exchange
void KeyExchangeModule::reset()
{
    public_key.fill(0);
    secret_key.fill(0);
    shared_secret.fill(0);
    keys_generated=false;
    shared_secret_established=false;
    std::cout<<"KeyExchangeModule: Module reset"<<std::endl;
}
```

Στο υλικό, αυτό θα γινόταν είτε μέσω του κεντρικού διαύλου επικοινωνίας, είτε μέσω ειδικών διαύλων. Επίσης, αυτή η διαδικασία δεν κοστίζει καθόλου στη λειτουργία του Η/Υ, καθώς γίνεται μόνο κατά την επανεκκίνηση του επεξεργαστή, δηλαδή κυρίως κατά την εκκίνηση του Η/Υ.

Πιθανά Προβλήματα Ασφαλείας

Το πιο κρίσιμο ζήτημα αυτής της αρχιτεκτονικής και οργάνωσης είναι το γεγονός ότι τα δεδομένα μεταφέρονται raw (μη κρυπτογραφημένα) μέσω του κεντρικού διαύλου επικοινωνίας από την ΚΜΕ στην Κύρια Μνήμη και είναι ευάλωτος σε reverse engineering επιθέσεις τύπου snooping (κρυφάκουσμα). Αν και θα είναι αρκετά δύσκολο ο επιτιθέμενος να συλλέξει τα κομμάτια του παζλ, από τα μισά δεδομένα της επικοινωνίας, τίποτα δεν σταματάει εξειδικευμένα άτομα με το σωστό κίνητρο. Θα μπορούσε όμως, σχετικά να μετριαστεί σε συνδυασμό με μηχανισμούς ασφαλείας που χρησιμοποιούνται ήδη.

Επίσης, ξανά έχει σημασία για το μοντέλο προστασίας/επίθεσης όπου αναλύεται, υπάρχει περίπτωση, η μονάδα κρυπτογράφησης στον ελεγκτή μνήμης, να διακινδυνέψει καθώς η πρόσβαση σε αυτή γίνεται πιο ευκολά. Σε αυτή την περίπτωση, μπορεί να αναπαράγει το ίδιο IV, που όπως ειπώθηκε, μπορεί να αποκαλύψει το κοινό κλειδί. Μπορεί επίσης να αναπαράγει τα δικά του block ή να οδηγήσει, παράλληλα, τα δεδομένα και στη δικιά του ειδική μνήμη, για να μαζέψει πιο εύκολα τα δεδομένα, να τα συγκρίνει και να κατανοήσει καλύτερα πως έχει υλοποιηθεί ο μηχανισμός κρυπτογράφησης και να διακινδυνέψει ολόκληρη τη λύση.

Δεν θεωρώ τη λύση αυτή πλήρως ασφαλή για τους παραπάνω απλούς λόγους, όπως και καμία αρχιτεκτονική/οργάνωση που προτείνεται και κάθε σύστημα γενικά, καθώς αν κάτι κρύβεται πίσω από μία κλειδαριά με σκοπό να περιοριστεί η πρόσβαση σε κάτι πολύτιμο, σε ένα ή λίγα άτομα, τότε υπάρχει τρόπος, οποιοσδήποτε, να έχει επίσης πρόσβαση, καθώς η κλειδαριά δεν είναι ποτέ ασφαλής, αφού το μόνο που καταφέρνει να ελέγχει είναι το κλειδί, και όχι ποιος το χρησιμοποιεί. Είναι όμως μία καινούρια ιδέα, για ένα καλύτερο σύστημα, όπου ενσωματώνει όλες αυτές τις τεχνικές π.χ. ανταλλαγή κλειδιών με ασύμμετρους αλγόριθμους κρυπτογράφησης ανθεκτικούς κατά την πιθανή απειλή των κβαντικών υπολογιστών.

Πείραμα Εναλλακτικής Προσέγγισης: Μία κοινή μονάδα Κρυπτογράφησης/Αποκρυπτογράφησης

Κατά την ανάπτυξη της τελικής αρχιτεκτονικής δοκιμάστηκαν κάποιες αρχιτεκτονικές όπου η κρυπτογράφηση/αποκρυπτογράφηση γινόταν μέσα στην ΚΜΕ (Υπό το πρίσμα της αρχιτεκτονικής επεξεργαστών), όπου το πλεονέκτημα είναι ότι γίνεται από μία μονάδα σε ασφαλές σημείο π.χ. μέσω ενός Memory Encryption Unit:

```
// Compiler Directives
#pragma once

// Header files
#include "Cipher.h"
#include "Memory.h"
#include <vector>

class MemoryEncryptionUnit
{
private:
    Memory* physical_memory;

    // MEU Configuration
    static const int MEU_BLOCK_SIZE=16;  // 16 words (64 bytes)
encryption blocks
    static const int MEU_CACHE_LINES=64; // Small cache for
encrypted blocks

    // Encryption Controller
    bool encryption_enabled;

    struct EncryptedBlock
    {
        bool valid;
        bool dirty;
        Address base_addr;
        Word encrypted_data[MEU_BLOCK_SIZE];
        Word decrypted_data[MEU_BLOCK_SIZE];

        EncryptedBlock():valid(false),dirty(false),base_addr(0)
{}
    };

    std::vector<EncryptedBlock> meu_cache;

    // Statistics
    int hits,misses,encryptions,decryptions;

    // Helper methods
```

```
    Address alignToMEUBlock(Address addr) const;
    int getMEUIndex(Address addr) const;
    void loadEncryptedBlock(Address base_addr, EncryptedBlock&
block);
    void writeBackEncryptedBlock(EncryptedBlock& block);

    // Encryption/Decryption methods
    void encryptBlock(const Word* plaintext, Word* ciphertext,
int size, Address base_addr);
    void decryptBlock(const Word* ciphertext, Word* plaintext,
int size, Address base_addr);

public:
    MemoryEncryptionUnit(Memory* mem_ptr);

    // Encryption Control
    void enableEncryption();
    void disableEncryption();

    // Interface for L2 Cache - these methods handle
encryption/decryption transparently
    Word readWord(Address physical_addr);
    void writeWord(Address physical_addr, Word value);

    // Block operations for efficient L2 cache line
fills/writebacks
    bool readBlock(Address physical_addr, Word* data, int
word_count);
    bool writeBlock(Address physical_addr, const Word* data, int
word_count);

    // Cache management
    void flushAll();
    void invalidateAddress(Address addr);
    void invalidateAll();

    // Statistics
    void printStats() const;
    void resetStats();
    void dumpAllBlocks() const;
};
```

```
void MemoryEncryptionUnit::loadEncryptedBlock(Address base_addr,
EncryptedBlock& block)
{
    // Read encrypted data from physical memory
    for (int i=0; i<MEU_BLOCK_SIZE; i++)
    {
```

```
        Address mem_addr=base_addr+(i*4);
        Word encrypted_value=0;

        encrypted_value|=(static_cast<Word>(physical_memory-
>read_physical_byte(mem_addr))<<0);
        encrypted_value|=(static_cast<Word>(physical_memory-
>read_physical_byte(mem_addr+1))<<8);
        encrypted_value|=(static_cast<Word>(physical_memory-
>read_physical_byte(mem_addr+2))<<16);
        encrypted_value|=(static_cast<Word>(physical_memory-
>read_physical_byte(mem_addr+3))<<24);

        block.encrypted_data[i]=encrypted_value;
    }

    // Decrypt the block

decryptBlock(block.encrypted_data,block.decrypted_data,MEU_BLOCK_
SIZE,base_addr);
    decryptions++;

    block.valid=true;
    block.dirty=false;
    block.base_addr=base_addr;
}
```

```
void
MemoryEncryptionUnit::writeBackEncryptedBlock(EncryptedBlock&
block)
{
    if (!block.valid||!block.dirty) return;

    // Encrypt the modified data

encryptBlock(block.decrypted_data,block.encrypted_data,MEU_BLOCK_
SIZE,block.base_addr);
    encryptions++;

    // Write encrypted data back to physical memory
    for (int i=0; i<MEU_BLOCK_SIZE; i++)
    {
        Address mem_addr=block.base_addr+(i*4);
        Word encrypted_value=block.encrypted_data[i];

        physical_memory-
>write_physical_byte(mem_addr,static_cast<Byte>(encrypted_value&0
xFF));
```

```
        physical_memory-
>write_physical_byte(mem_addr+1,static_cast<Byte>((encrypted_valu
e>>8)&0xFF));
        physical_memory-
>write_physical_byte(mem_addr+2,static_cast<Byte>((encrypted_valu
e>>16)&0xFF));
        physical_memory-
>write_physical_byte(mem_addr+3,static_cast<Byte>((encrypted_valu
e>>24)&0xFF));
    }

    block.dirty=false;
}
```

```
bool MemoryEncryptionUnit::readBlock(Address physical_addr, Word*
data, int word_count)
{
    // Handle the case where L2 cache line might span multiple
MEU blocks
    Address start_addr=physical_addr;
    int words_read=0;

    while (words_read<word_count)
    {
        Address current_addr=start_addr+(words_read*4);
        Address base_addr=alignToMEUBlock(current_addr);
        int cache_index=getMEUIndex(base_addr);
        int word_offset=(current_addr-base_addr)/4;

        EncryptedBlock& block=meu_cache[cache_index];

        // Load block if not present
        if (!block.valid||block.base_addr !=base_addr)
        {
            if (block.valid&&block.dirty)
{writeBackEncryptedBlock(block);}
            loadEncryptedBlock(base_addr,block);
            misses++;
        }
        else {hits++;}

        // Copy words from this block
        int words_from_this_block=std::min(word_count-
words_read,MEU_BLOCK_SIZE-word_offset);
        for (int i=0; i<words_from_this_block; i++)
        {

data[words_read+i]=block.decrypted_data[word_offset+i];
```

```
        }

        words_read+=words_from_this_block;
    }

    return true;
}
```

```
bool MemoryEncryptionUnit::writeBlock(Address physical_addr,
const Word* data, int word_count)
{
    Address start_addr=physical_addr;
    int words_written=0;

    while (words_written<word_count)
    {
        Address current_addr=start_addr+(words_written*4);
        Address base_addr=alignToMEUBlock(current_addr);
        int cache_index=getMEUIndex(base_addr);
        int word_offset=(current_addr-base_addr)/4;

        EncryptedBlock& block=meu_cache[cache_index];

        // Load block if not present
        if (!block.valid||block.base_addr !=base_addr)
        {
            if (block.valid&&block.dirty)
{writeBackEncryptedBlock(block);}
            loadEncryptedBlock(base_addr,block);
            misses++;
        }
        else {hits++;}

        // Write words to this block
        int words_to_this_block=std::min(word_count-
words_written,MEU_BLOCK_SIZE-word_offset);
        for (int i=0; i<words_to_this_block; i++)
        {

block.decrypted_data[word_offset+i]=data[words_written+i];
        }

        block.dirty=true;
        words_written+=words_to_this_block;
    }

    return true;
}
```

```
void MemoryEncryptionUnit::encryptBlock(const Word* plaintext,
Word* ciphertext, int size, Address base_addr)
{
    if (encryption_enabled)
    {
        // Encrypt the block
        for (int i=0; i<size; i++)
        {
            ciphertext[i]=Cipher::encrypt(plaintext[i]);
        }
    }
    else
    {   // Just copy plaintext
        for (int i=0; i<size; i++)
        {
            ciphertext[i]=plaintext[i];
        }
    }
}
```

```
void MemoryEncryptionUnit::decryptBlock(const Word* ciphertext,
Word* plaintext, int size, Address base_addr)
{
    if (encryption_enabled)
    {
        // Decrypt the block
        for (int i=0; i<size; i++)
        {
            plaintext[i]=Cipher::decrypt(ciphertext[i]);
        }
    }
    else
    {
        // Just copy ciphertext to plaintext
        for (int i=0; i<size; i++)
        {
            plaintext[i]=ciphertext[i];
        }
    }
}
```

Χρησιμοποιείται απλή συνάρτηση XOR για το συγκεκριμένο πείραμα.

Αλλά αποφάσισα να ακολουθήσω την τεχνική των ξεχωριστών μονάδων, καθώς αυτό το παράδειγμα έχει εξεταστεί αρκετά.

Τί γίνεται με τον DMA μηχανισμό;

Σκοπός της εργασίας αυτής ήταν μία λύση κρυπτογράφησης/αποκρυπτογράφησης μεταξύ δεδομένων που διαδίδονται από και προς τον Επεξεργαστή και την Κύρια Μνήμη. Τί γίνεται με λοιπούς μηχανισμούς, όπως ο DMA, όπου χρησιμοποιούνται στους σύγχρονους Η/Υ; Δεν θα μπορούσε να παραλειφθεί κάποια παράγραφος για το συγκεκριμένο ζήτημα. Αν και φεύγει εκτός σκοπού αυτής της εργασίας, κρίνω απαραίτητο να αναλυθεί το ζήτημα αυτό.

Η λύση όπου προτείνω, στην πραγματικότητα είναι πιο ευέλικτη από ότι ίσως φαίνεται. Αν και πιστεύω ότι η μονάδα κρυπτογράφησης πρέπει να τοποθετηθεί εσωτερικά της ΚΜΕ, η μονάδα αποκρυπτογράφησης θα μπορούσε κάλλιστα να τοποθετηθεί στην ίδια την μητρική πλακέτα. Επίσης, συμπληρωματικοί δίαυλοι επικοινωνίας μπορούν να εισαχθούν στην συγκεκριμένη αρχιτεκτονική, όπου να οδηγούν τα σήματα από τις περιφερειακές συσκευές που εκμεταλλεύονται τον DMA, κατάλληλα στις μονάδες κρυπτογράφησης και αποκρυπτογράφησης για να μην αποκλειστούν από το όφελος της κρυπτογράφησης, αν και αυτό μπορεί να αυξήσει την περιπλοκότητα της αρχιτεκτονικής, καθώς μπορεί να χρειαστεί επιπρόσθετα πρωτόκολλα συνεργασίας με τα κυκλώματα. Θα μπορούσαν δε να χρησιμοποιηθούν συγκεκριμένα, ειδικά, segments (χώροι) στην μνήμη όπου να μην κρυπτογραφούνται τα δεδομένα όπου χρησιμοποιούνται για DMA, αν και αυτό ουσιαστικά κάνει μια τρύπα στην λογική της ασφαλούς αποθήκευσης των δεδομένων στην μνήμη.

Ο μηχανισμός DMA ήταν εκτός του σκοπού της εργασίας αυτής καθώς είναι μία μεγάλη συζήτηση από μόνος του, αλλά επιφέρει μία πρόκληση για αυτή, αλλά και τις υπόλοιπες λύσεις όπου έχουν προταθεί και ήδη αναφερθεί. Παρ’ όλα αυτά, θεωρώ την λύση νέων δίαυλων επικοινωνίας ως μία πολύ πιθανή λύση.

Πειράματα και Αποτελέσματα

Η προσομοίωση

Ας ελέγξουμε την προσομοίωση περνώντας του ένα πρόγραμμα για να εκτελέσει. Να σημειωθεί ότι κατά τη μεταγλώττιση χρησιμοποιήθηκαν οι παράμετροι -fanalyzer -fsanitize=undefined για τον έλεγχο πιθανών σφαλμάτων κατά την μεταγλώττιση με τον g++ και κατά την εκτέλεση. Επίσης ενεργοποίηθηκαν και οι σχετικές βιλβιοθήκες με τις εξής παραμέτρους -loqs -lcrypto -lssl.

Το πρόγραμμα που επιλέχθηκε για το πείραμα είναι το εξής:

```
# MIPS Test Program - 32-bit

# Sets up TLB entry and tests memory access


# Setup TLB entry

lui $t0, 0x8000          # Virtual page address (high 16 bits)

mtc0 $t0, $10            # Move to EntryHi (CP0 register 10)


lui $t1, 0x0001          # Physical page frame number

ori $t1, $t1, 0x001F     # Set valid, dirty, global bits (bits 0-5)

mtc0 $t1, $2             # Move to EntryLo0 (CP0 register 2)


lui $t2, 0x0002          # Second physical page

ori $t2, $t2, 0x001F     # Set valid, dirty, global bits

mtc0 $t2, $3             # Move to EntryLo1 (CP0 register 3)


ori $t3, $zero, 0        # TLB index 0

mtc0 $t3, $0             # Move to Index register (CP0 register 0)


tlbwi                    # Write indexed TLB entry


# Test memory access through TLB

lui $t4, 0x8000          # Virtual address to test

ori $t4, $t4, 0x0004     # Add offset
```

```
ori $t5, $zero, 0x1234   # Test data
sw $t5, 0($t4)           # Store word through TLB

lw $t6, 0($t4)           # Load word back through TLB

# Verify data
bne $t5, $t6, 8          # Branch if not equal (skip 2 instructions)
ori $t7, $zero, 0xFFFF   # Success indicator
beq $zero, $zero, 4      # Branch to end (skip 1 instruction)
ori $t7, $zero, 0x0000   # Failure indicator
nop                      # End marker
```

Το οποίο με την χρήση του συμβολομεταφραστή μεταφράζεται στο εξής δεκαεξαδικό κώδικα μηχανής:

```
3c088000
40885000
3c090001
3529001f
40891000
3c0a0002
354a001f
408a1800
340b0000
408b0000
42000002
3c0c8000
358c0004
340d1234
ad8d0000
8d8e0000
15ae0008
340fffff
```

10000004

340f0000

00000000

Πολύ συνοπτικά, στήνει κατάλληλα την TLB για να υποστηρίζονται οι μεταφράσεις για την εικονική διεύθυνση 0x80000000 καθώς και με τις φυσικές σελίδες που χρειάζονται και τα αποθηκεύει στο index=0. Ύστερα γράφει στη μνήμη την τιμή 0x1234, και συγκεκριμένα στην εικονική διεύθυνση 0x80000004 όπου πια είναι έγκυρη και την ξαναφορτώνει στον καταχωρητή $t6. Τέλος, συγκρίνει τα αποτελέσματα και σε περίπτωση που είναι όμοια, γράφει 0xFFFFFFFF στον καταχωρητή $t7.

Ας το τρέξουμε για 30 κύκλους μηχανής. Να σημειωθεί ότι είναι πολύ περισσότεροι από τις εντολές και κανονικά αυτό πρέπει να αποφεύγεται καθώς θα διαβάζει σημεία της μνήμης όπου δεν θα έπρεπε να προσπελάζει και ενδέχεται να καταστρέψει την προσομοίωση. Στην περίπτωση αυτή διαβάζει 0 από την Κύρια Μνήμη, αφού οι διευθύνσεις αυτές δεν κρυπτογραφήθηκαν ποτέ και μάλιστα, η τιμή 0 (σε 32 bit), αντιστοιχεί με την NOP εντολή.

```
teo@teo-VirtualBox:~/Ptyxiakh/L3Mod/NEW/output$ ./a.out 30
Memory initialized with 1024 MB
Memory controller encryption: Enabled!
Memory block size: 16 words (64 bytes)
MMU created successfully!
Coprocessor 0 created successfully!
Register File created successfully!
Instruction Cache initialized with 4KB, 256 lines, 4 words per line
Data Cache initialized with 4KB, 256 lines, 4 words per line.
L2 Cache initialized with 32KB, 1024 lines, 8 words per line (PIPT)
KeyExchangeModule: ML-KEM-512 module initialized
CDU initialized with 64 cache lines, 16 words per decryption block
CDU: Quantum-safe key exchange module integrated
Encryption/Decryption is ENABLED (Memory handles encryption)by default (Make sure encrypti
on directive is enabled/disabled in the memory.cpp too for the bootloader initialization!)
Processor: Invalidating all cache levels...
Instruction Cache: Invalidated all cache lines!
Processor: Invalidated L1 Instruction Cache
Data Cache: Invalidated all cache lines!
Processor: Invalidated L1 Data Cache
L2 Cache: Invalidated all cache lines!
Processor: Invalidated L2 Shared Cache
CDU: Flushed all dirty blocks
CDU: Invalidated all cache blocks
Processor: Invalidated Memory Encryption Unit Cache
Processor: All caches invalidated
```

Με το που τρέξουμε την προσομοίωση θα αρχίζουν να κατασκευάζονται όλα τα κυκλώματα που προσομοιώθηκαν με τις παραμέτρους όπου έχουμε θέσει. Επίσης γίνονται καθαρισμοί στις κρυφές μνήμες.

```
CDU: Initiating quantum-safe key exchange with memory...
KeyExchangeModule: MEU keypair generated successfully
Memory: Handling key exchange request from CDU...
Memory: Using CDU's KeyExchangeModule via data bus...
KeyExchangeModule: Memory-side encapsulation successful
Memory: Key exchange completed successfully!
Memory: Encryption key updated, all future writes will use quantum-safe encryption
KeyExchangeModule: Memory-side key exchange completed
KeyExchangeModule: MEU-side decapsulation successful - shared secret established
KeyExchangeModule: Shared secrets match - key exchange successful!
CDU: Quantum-safe encryption key established successfully
CDU: Flushed all dirty blocks
CDU: Invalidated all cache blocks
Quantum-safe encryption established!
Secure communication channel active
Processor reset complete!
Processor instance created successfully!
```

Ύστερα, θα γίνει η ασύμμετρη ανταλλαγή του κλειδιού της συμμετρικής κρυπτογράφησης με την μονάδα κρυπτογράφησης και της αποκρυπτογράφησης όπου ενορχηστρώνει ο επεξεργαστής με την μονάδα ανταλλαγής κλειδιών.

```
load_bootloader: Memory controller will encrypt all instructions using AES-GCM!
load_bootloader: Quantum-safe encryption ENABLED!
Padded program from 24 to 32 instructions for complete blocks
Memory: Encrypted and stored 16 words at 0x1fc00000
Wrote block 0 at address 0x1fc00000 (16 words)
Memory: Encrypted and stored 16 words at 0x1fc00040
Wrote block 1 at address 0x1fc00040 (16 words)
Program loaded: 24 instructions (96 bytes) at physical address 0x1fc00000
Total blocks written: 2
Starting processor emulation...
```

Τέλος, θα γίνει η συμβολομετάφραση του προγράμματος (bootloader) που θα τρέξει και θα τα φορτώσει στη Μνήμη. Εφ' όσον, κάθε μπλοκ χωράει 16 εντολές, γίνεται εισαγωγή NOP (μηδενικά) για την ολοκλήρωση του μπλοκ. Επίσης τελικά, οι εντολές είναι 3 παραπάνω με την εισαγωγή κάποιων NOP από τον συμβολομεταφραστή (Εφ' όσον είναι πείραμα, θέλησα να σιγουρέψω την ορθή εκτέλεση). Ύστερα αρχίζει η εκτέλεση εντολής προς εντολή.

Τρέχω το πείραμα σε debug mode και για αυτό και έχουν εισαχθεί πληροφορίες όπως για κινδύνους και forwarding, MMU, TLB και ενέργειες συνεπεξεργαστή0.

```
=== Cycle 1 ===
MMU: Translating VA=0xbfc00000, Write=false, Size=4
MMU: KSEG1 translation: VA=0xbfc00000 -> PA=0x1fc00000
Memory: Read 10 encrypted words fetched from 0x1fc00000
CDU: Successfully decrypted block at 0x1fc00000
IF: Fetched Instruction 0x3c088000 from PC=0xbfc00000

=== Cycle 2 ===
MEM: Passing ALU result to register $0
EX: ALU operation with alu_control_signal=2, result=0x0, zero=1, branch_target=0x0, destination register=$0
ID: Decoded instruction: opcode=0xf, immediate=0xffff8000, rs=$0(0), rt=$8(0)
IF: Fetched Instruction 0x40885000 from PC=0xbfc00004

=== Cycle 3 ===
MEM: Passing ALU result to register $0
EX: ALU operation with alu_control_signal=13, result=0x80000000, zero=0, branch_target=0xbfbe0004, destination register=$8
ID: Decoded instruction: opcode=0x10, immediate=0x5000, rs=$4(0), rt=$8(0)
IF: Fetched Instruction 0x3c090001 from PC=0xbfc00008

=== Cycle 4 ===
MEM: Passing ALU result to register $8
FORWARDING: RT data from EX/MEM to RT=$8
EX: ALU operation with alu_control_signal=4d, result=0x0, zero=1, branch_target=0xbfc14008, destination register=$8
WB: Write HEX: 0x80000000, DEC: 2147483648 to register $8
ID: Decoded instruction: opcode=0xf, immediate=0x1, rs=$0(0), rt=$9(0)
IF: Fetched Instruction 0x3529001f from PC=0xbfc0000c

=== Cycle 5 ===
CP0: MTC0 - Writing 0x80000000 to CP0 register 10
MEM: Passing ALU result to register $8
EX: ALU operation with alu_control_signal=13, result=0x10000, zero=0, branch_target=0xbfc00010, destination register=$9
ID: Decoded instruction: opcode=0xd, immediate=0x1f, rs=$9(0), rt=$9(0)
MMU: Translating VA=0xbfc00010, Write=false, Size=4
MMU: KSEG1 translation: VA=0xbfc00010 -> PA=0x1fc00010
IF: Fetched Instruction 0x40891000 from PC=0xbfc00010
```

Στο κύκλο 1-5 γίνεται η ρύθμιση της TLB.

```
=== Cycle 6 ===
MEM: Passing ALU result to register $9
FORWARDING: RS data from EX/MEM to RS=$9
FORWARDING: RT data from EX/MEM to RT=$9
EX: ALU operation with alu_control_signal=1, result=0x1001f, zero=0, branch_target=0xbfc0008c, destination register=$9
WB: Write HEX: 0x10000, DEC: 65536 to register $9
ID: Decoded instruction: opcode=0x10, immediate=0x1000, rs=$4(0), rt=$9(65536)
IF: Fetched Instruction 0x3c0a0002 from PC=0xbfc00014

=== Cycle 7 ===
MEM: Passing ALU result to register $9
FORWARDING: RT data from EX/MEM to RT=$9
EX: ALU operation with alu_control_signal=4d, result=0x0, zero=1, branch_target=0xbfc04014, destination register=$9
WB: Write HEX: 0x1001f, DEC: 65567 to register $9
ID: Decoded instruction: opcode=0xf, immediate=0x2, rs=$0(0), rt=$10(0)
IF: Fetched Instruction 0x354a001f from PC=0xbfc00018

=== Cycle 8 ===
CP0: MTC0 - Writing 0x1001f to CP0 register 2
MEM: Passing ALU result to register $9
EX: ALU operation with alu_control_signal=13, result=0x20000, zero=0, branch_target=0xbfc00020, destination register=$10
ID: Decoded instruction: opcode=0xd, immediate=0x1f, rs=$10(0), rt=$10(0)
IF: Fetched Instruction 0x408a1800 from PC=0xbfc0001c

=== Cycle 9 ===
MEM: Passing ALU result to register $10
FORWARDING: RS data from EX/MEM to RS=$10
FORWARDING: RT data from EX/MEM to RT=$10
EX: ALU operation with alu_control_signal=1, result=0x2001f, zero=0, branch_target=0xbfc00098, destination register=$10
WB: Write HEX: 0x20000, DEC: 131072 to register $10
ID: Decoded instruction: opcode=0x10, immediate=0x1800, rs=$4(0), rt=$10(131072)
MMU: Translating VA=0xbfc00020, Write=false, Size=4
MMU: KSEG1 translation: VA=0xbfc00020 -> PA=0x1fc00020
IF: Fetched Instruction 0x340b0000 from PC=0xbfc00020
```

Κατά τον κύκλο 6,7 και 8 γίνεται η εγγραφή του EntryLo0 του CP0.

```
=== Cycle 10 ===
MEM: Passing ALU result to register $10
FORWARDING: RT data from EX/MEM to RT=$10
EX: ALU operation with alu_control_signal=4d, result=0x0, zero=1, branch_target=0xbfc06020, destination register=$10
WB: Write HEX: 0x2001f, DEC: 131103 to register $10
ID: Decoded instruction: opcode=0xd, immediate=0x0, rs=$0(0), rt=$11(0)
IF: Fetched Instruction 0x408b0000 from PC=0xbfc00024

=== Cycle 11 ===
CP0: MTC0 - Writing 0x2001f to CP0 register 3
MEM: Passing ALU result to register $10
EX: ALU operation with alu_control_signal=1, result=0x0, zero=1, branch_target=0xbfc00024, destination register=$11
ID: Decoded instruction: opcode=0x10, immediate=0x0, rs=$4(0), rt=$11(0)
IF: Fetched Instruction 0x42000002 from PC=0xbfc00028

=== Cycle 12 ===
MEM: Passing ALU result to register $11
FORWARDING: RT data from EX/MEM to RT=$11
EX: ALU operation with alu_control_signal=4d, result=0x0, zero=1, branch_target=0xbfc00028, destination register=$11
WB: Write HEX: 0x0, DEC: 0 to register $11
ID: Decoded instruction: opcode=0x10, immediate=0x2, rs=$16(0), rt=$0(0)
IF: Fetched Instruction 0x3c0c8000 from PC=0xbfc0002c

=== Cycle 13 ===
CP0: MTC0 - Writing 0x0 to CP0 register 0
MEM: Passing ALU result to register $11
EX: ALU operation with alu_control_signal=51, result=0x0, zero=1, branch_target=0xbfc00034, destination register=$0
ID: Decoded instruction: opcode=0xf, immediate=0xffff8000, rs=$0(0), rt=$12(0)
MMU: Translating VA=0xbfc00030, Write=false, Size=4
MMU: KSEG1 translation: VA=0xbfc00030 -> PA=0x1fc00030
IF: Fetched Instruction 0x358c0004 from PC=0xbfc00030
```

Κατά τους κύκλους 9-11 γράφεται και ο EntryLo1, και ύστερα στον κύκλο 12, 13 και 14 ολοκληρώνεται η ρύθμιση της TLB.

```
=== Cycle 14 ===
TLB[0] written: EntryHi=0x80000000, EntryLo0=0x1001f, EntryLo1=0x2001f, PageMask=0x0
CP0: TLBWI - TLB Write Index executed
MEM: Passing ALU result to register $0
EX: ALU operation with alu_control_signal=13, result=0x80000000, zero=0, branch_target=0xbfbe0030, destination register=$12
ID: Decoded instruction: opcode=0xd, immediate=0x4, rs=$12(0), rt=$12(0)
IF: Fetched Instruction 0x340d1234 from PC=0xbfc00034

=== Cycle 15 ===
MEM: Passing ALU result to register $12
FORWARDING: RS data from EX/MEM to RS=$12
FORWARDING: RT data from EX/MEM to RT=$12
EX: ALU operation with alu_control_signal=1, result=0x80000004, zero=0, branch_target=0xbfc00044, destination register=$12
WB: Write HEX: 0x80000000, DEC: 2147483648 to register $12
ID: Decoded instruction: opcode=0xd, immediate=0x1234, rs=$0(0), rt=$13(0)
IF: Fetched Instruction 0xad8d0000 from PC=0xbfc00038

=== Cycle 16 ===
MEM: Passing ALU result to register $12
EX: ALU operation with alu_control_signal=1, result=0x1234, zero=0, branch_target=0xbfc04908, destination register=$13
WB: Write HEX: 0x80000004, DEC: 2147483652 to register $12
ID: Decoded instruction: opcode=0x2b, immediate=0x0, rs=$12(2147483652), rt=$13(0)
IF: Fetched Instruction 0x00000000 from PC=0xbfc0003c

=== Cycle 17 ===
MEM: Passing ALU result to register $13
FORWARDING: RT data from EX/MEM to RT=$13
EX: ALU operation with alu_control_signal=2, result=0x80000004, zero=0, branch_target=0xbfc0003c, destination register=$13
WB: Write HEX: 0x1234, DEC: 4660 to register $13
ID: Decoded instruction: opcode=0x0, funct=0x0, shamt=0x0, immediate=0x0, rs=$0(0), rt=$0(0), rd=$0
MMU: Translating VA=0xbfc00040, Write=false, Size=4
MMU: KSEG1 translation: VA=0xbfc00040 -> PA=0x1fc00040
Memory: Read 10 encrypted words fetched from 0x1fc00040
CDU: Successfully decrypted block at 0x1fc00040
IF: Fetched Instruction 0x00000000 from PC=0xbfc00040
```

Στον κύκλο 15 αρχίζει η δοκιμή της Κύριας Μνήμης. Στους 15 έως 17 προετοιμάζονται οι διευθύνσεις καθώς και τα δεδομένα όπου θα χρησιμοποιηθούν.

```
=== Cycle 18 ===
MMU: Translating VA=0x80000004, Write=false, Size=4
MMU: KSEG0 translation: VA=0x80000004 -> PA=0x4
Memory: Read 10 encrypted words fetched from 0x0
CDU: Successfully decrypted block at 0x0
MEM: Wrote 0x1234 to memory[0x80000004]; Passing ALU result to register $13
EX: ALU operation with alu_control_signal=10, result=0x0, zero=1, branch_target=0xbfc00040, destination register=$0
ID: Decoded instruction: opcode=0x0, funct=0x0, shamt=0x0, immediate=0x0, rs=$0(0), rt=$0(0), rd=$0
IF: Fetched Instruction 0x8d8e0000 from PC=0xbfc00044

=== Cycle 19 ===
MEM: Passing ALU result to register $0
EX: ALU operation with alu_control_signal=10, result=0x0, zero=1, branch_target=0xbfc00044, destination register=$0
Write request to Register $0 detected, ignoring... Value: 0
WB: Write HEX: 0x0, DEC: 0 to register $0
ID: Decoded instruction: opcode=0x23, immediate=0x0, rs=$12(2147483652), rt=$14(0)
IF: Fetched Instruction 0x00000000 from PC=0xbfc00048

=== Cycle 20 ===
MEM: Passing ALU result to register $0
EX: ALU operation with alu_control_signal=2, result=0x80000004, zero=0, branch_target=0xbfc00048, destination register=$14
Write request to Register $0 detected, ignoring... Value: 0
WB: Write HEX: 0x0, DEC: 0 to register $0
ID: Decoded instruction: opcode=0x0, funct=0x0, shamt=0x0, immediate=0x0, rs=$0(0), rt=$0(0), rd=$0
IF: Fetched Instruction 0x15ae0008 from PC=0xbfc0004c

=== Cycle 21 ===
MEM: Read memory[0x80000004] = 0x1234; Passing MEM result to register $14
EX: ALU operation with alu_control_signal=10, result=0x0, zero=1, branch_target=0xbfc0004c, destination register=$0
WB: Write HEX: 0x1234, DEC: 4660 to register $14
ID: Decoded instruction: opcode=0x5, immediate=0x8, rs=$13(4660), rt=$14(4660)
MMU: Translating VA=0xbfc00050, Write=false, Size=4
MMU: KSEG1 translation: VA=0xbfc00050 -> PA=0x1fc00050
IF: Fetched Instruction 0x340fffff from PC=0xbfc00050
```

Στον κύκλο 18 αποθηκεύεται η τιμή 0x1234 στην 0x80000004.
Ενώ στον κύκλο 21 φορτώνεται το στοιχείο από την μνήμη.

```
=== Cycle 22 ===
MEM: Passing ALU result to register $0
EX: ALU operation with alu_control_signal=6, result=0x0, zero=1, branch_target=0xbfc00070, destination register=$14
Write request to Register $0 detected, ignoring... Value: 0
WB: Write HEX: 0x0, DEC: 0 to register $0
ID: Decoded instruction: opcode=0xd, immediate=0xffffffff, rs=$0(0), rt=$15(0)
IF: Fetched Instruction 0x10000004 from PC=0xbfc00054

=== Cycle 23 ===
MEM: Passing ALU result to register $14
EX: ALU operation with alu_control_signal=1, result=0xffffffff, zero=0, branch_target=0xbfc00050, destination register=$15
ID: Decoded instruction: opcode=0x4, immediate=0x4, rs=$0(0), rt=$0(0)
IF: Fetched Instruction 0x340f0000 from PC=0xbfc00058

=== Cycle 24 ===
MEM: Passing ALU result to register $15
EX: ALU operation with alu_control_signal=6, result=0x0, zero=1, branch_target=0xbfc00068, destination register=$0
WB: Write HEX: 0xffffffff, DEC: 4294967295 to register $15
ID: Decoded instruction: opcode=0xd, immediate=0x0, rs=$0(0), rt=$15(4294967295)
IF: Fetched Instruction 0x00000000 from PC=0xbfc0005c

=== Cycle 25 ===
MEM: Branch taken, target=0xbfc00068
MEM: Passing ALU result to register $0
EX: ALU operation with alu_control_signal=1, result=0x0, zero=1, branch_target=0xbfc0005c, destination register=$15
ID: Decoded instruction: opcode=0x0, funct=0x0, shamt=0x0, immediate=0x0, rs=$0(0), rt=$0(0), rd=$0
MMU: Translating VA=0xbfc00060, Write=false, Size=4
MMU: KSEG1 translation: VA=0xbfc00060 -> PA=0x1fc00060
IF: Fetched Instruction 0x00000000 from PC=0xbfc00068
```

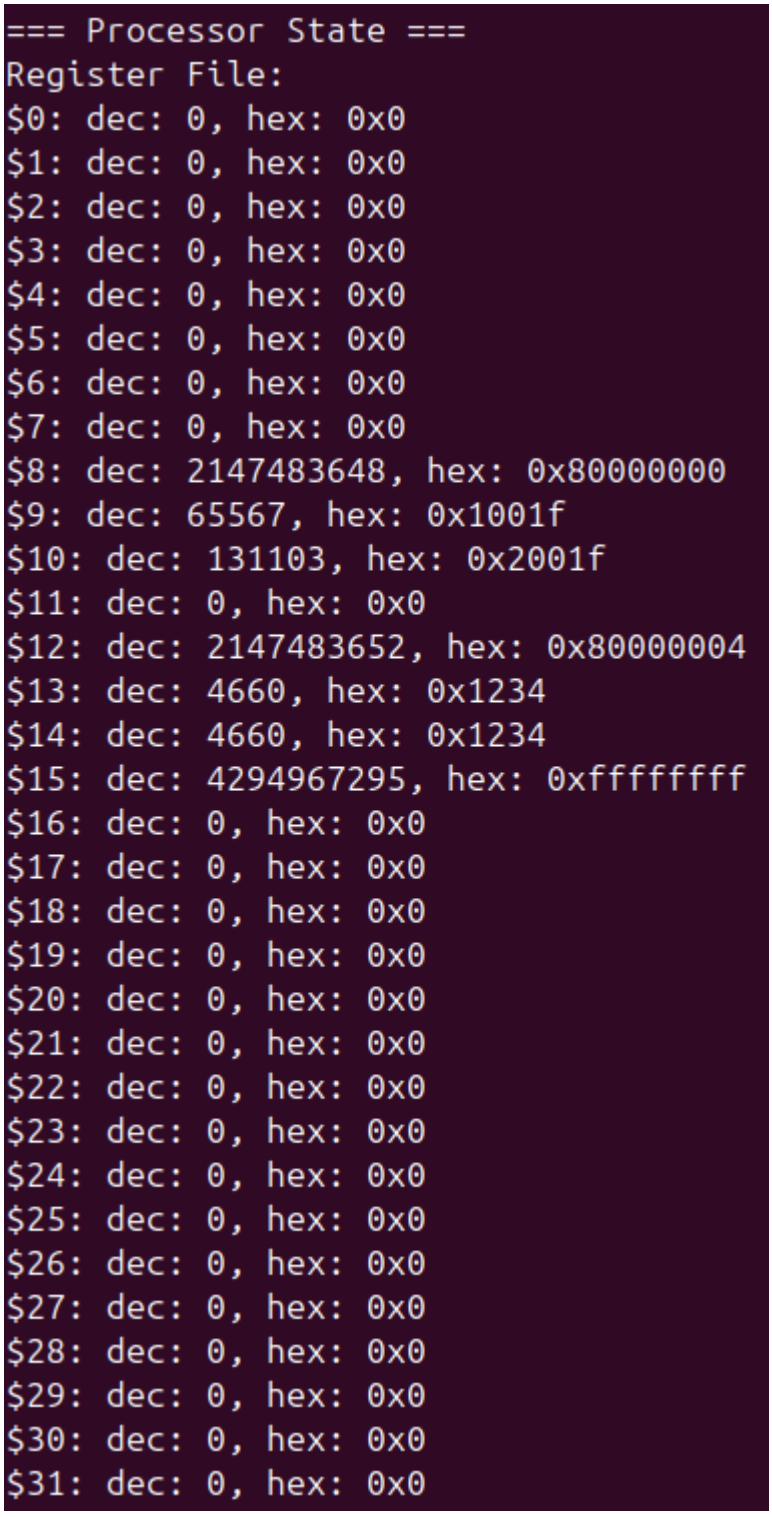
Τελικά, στον κύκλο 22 γίνεται η σύγκριση των δεδομένων και κατά των κύκλο 23-25 γράφεται η ειδική τιμή 0xFFFFFFFF.

```
=== Cycle 26 ===
IF: Fetched Instruction 0x00000000 from PC=0xbfc0006c

=== Cycle 27 ===
MEM: Passing ALU result to register $0
EX: ALU operation with alu_control_signal=2, result=0x0, zero=1, branch_target=0x0, destination register=$0
ID: Decoded instruction: opcode=0x0, funct=0x0, shamt=0x0, immediate=0x0, rs=$0(0), rt=$0(0), rd=$0
MMU: Translating VA=0xbfc00070, Write=false, Size=4
MMU: KSEG1 translation: VA=0xbfc00070 -> PA=0x1fc00070
IF: Fetched Instruction 0x00000000 from PC=0xbfc00070

=== Cycle 28 ===
MEM: Passing ALU result to register $0
EX: ALU operation with alu_control_signal=10, result=0x0, zero=1, branch_target=0xbfc00070, destination register=$0
ID: Decoded instruction: opcode=0x0, funct=0x0, shamt=0x0, immediate=0x0, rs=$0(0), rt=$0(0), rd=$0
IF: Fetched Instruction 0x00000000 from PC=0xbfc00074

=== Cycle 29 ===
MEM: Passing ALU result to register $0
EX: ALU operation with alu_control_signal=10, result=0x0, zero=1, branch_target=0xbfc00074, destination register=$0
Write request to Register $0 detected, ignoring... Value: 0
WB: Write HEX: 0x0, DEC: 0 to register $0
ID: Decoded instruction: opcode=0x0, funct=0x0, shamt=0x0, immediate=0x0, rs=$0(0), rt=$0(0), rd=$0
IF: Fetched Instruction 0x00000000 from PC=0xbfc00078

=== Cycle 30 ===
MEM: Passing ALU result to register $0
EX: ALU operation with alu_control_signal=10, result=0x0, zero=1, branch_target=0xbfc00078, destination register=$0
Write request to Register $0 detected, ignoring... Value: 0
WB: Write HEX: 0x0, DEC: 0 to register $0
ID: Decoded instruction: opcode=0x0, funct=0x0, shamt=0x0, immediate=0x0, rs=$0(0), rt=$0(0), rd=$0
IF: Fetched Instruction 0x00000000 from PC=0xbfc0007c
```

Στους τελευταίους κύκλους δεν συμβαίνει τίποτα, αν και φαίνεται πως προσπαθεί να διαβάσει από τις επόμενες διευθύνσεις που είναι κενές (Μετάφραση της διεύθυνσης στην MMU).

Φαίνεται λοιπόν πως το πρόγραμμα φορτώθηκε και εκτελέστηκε με επιτυχία. Ας εξετάσουμε τα κύρια στοιχεία του υπολογιστή κατά το τέλος της εκτέλεσης.

Ας αρχίσουμε με τον φάκελο καταχωρητών:



Βλέπουμε ότι η κατάσταση του φακέλου καταχωρητών είναι η αναμενόμενη.

Ας δούμε τα στατιστικά των κρυφών μνημών:

```
Instruction Cache Statistics:
  Hits: 22
  Misses: 8
  Hit Rate: 73.33%
Data Cache Statistics:
 Hits: 1
 Misses: 1
 Writebacks: 0
 Hit Rate: 50.00%
Shared (L2) Cache Statistics:
 Hits: 4
 Misses: 5
 Writebacks: 0
 L1 Requests: 9
 Hit Rate: 44.44%
CDU Statistics:
 Hits: 2
 Misses: 3
 Decryptions: 3
 Cipher metadata entries: 2
 Hit Rate: 40.00%
 Encryption: ACTIVE
```

Ναι, είναι ακριβώς αυτό που περιμένουμε. Ειδικότερα, η CDU επιστρέφει 3 ευστοχίες και 3 αστοχίες, που επιφέρουν και 3 ενέργειες αποκρυπτογράφησης. Λογικό αφού έχουμε 2 block για τις εντολές και 1 block, ολόκληρο μπλοκ ναι, για την καταχώρηση 0x00001234 στην 0x80000004.

Μισό λεπτό όμως, δεν έχει γίνει flush! Ας ξανατρέξουμε το πείραμα και ας κάνουμε dump τις cache.

```
Processor: Flushing all caches...
Instruction Cache: Flushed (Invalidated) all cache lines!
Processor: Flushed L1 Instruction Cache
Data Cache: Flushed all dirty lines!
Processor: Flushed L1 Data Cache
L2 Cache: Flushed all dirty lines!
Processor: Flushed Shared Cache (L2)
CDU: Writing back dirty block at 0x0
Memory: Encrypted and stored 10 words at 0x0
CDU: Flushed all dirty blocks
Processor: Flushed Memory Encryption Unit Cache
Processor: All caches flushed
```

```
Instruction Cache Statistics:
  Hits: 16
  Misses: 8
  Hit Rate: 73.33%
Data Cache Statistics:
 Hits: 1
 Misses: 1
 Writebacks: 1
 Hit Rate: 50.00%
Shared (L2) Cache Statistics:
 Hits: 5
 Misses: 5
 Writebacks: 1
 L1 Requests: 10
 Hit Rate: 50.00%
CDU Statistics:
 Hits: 3
 Misses: 3
 Decryptions: 3
 Cipher metadata entries: 3
 Hit Rate: 50.00%
 Encryption: ACTIVE
```

Ναι, να η τελική μορφή μετά το flush.

Ας κάνουμε dump την κρυφή μνήμη εντολών (L1) (αγνοείστε το 8/100, είναι το 256 στο δεκαεξαδικό σύστημα αρίθμησης) πριν το flush:

```
=== COMPLETE INSTRUCTION CACHE DUMP ===
Cache Configuration: 256 lines, 4 words per line
----------------------------------------
Line 000: Tag=0x000bfc00 (VA~0xbfc00000)
      Instructions: 0x3c088000 0x40885000 0x3c090001 0x3529001f

Line 001: Tag=0x000bfc00 (VA~0xbfc00010)
      Instructions: 0x40891000 0x3c0a0002 0x354a001f 0x408a1800

Line 002: Tag=0x000bfc00 (VA~0xbfc00020)
      Instructions: 0x340b0000 0x408b0000 0x42000002 0x3c0c8000

Line 003: Tag=0x000bfc00 (VA~0xbfc00030)
      Instructions: 0x358c0004 0x340d1234 0xad8d0000 0x00000000

Line 004: Tag=0x000bfc00 (VA~0xbfc00040)
      Instructions: 0x00000000 0x8d8e0000 0x00000000 0x15ae0008

Line 005: Tag=0x000bfc00 (VA~0xbfc00050)
      Instructions: 0x340fffff 0x10000004 0x340f0000 0x00000000

Line 006: Tag=0x000bfc00 (VA~0xbfc00060)
      Instructions: 0x00000000 0x00000000 0x00000000 0x00000000

Line 007: Tag=0x000bfc00 (VA~0xbfc00070)
      Instructions: 0x00000000 0x00000000 0x00000000 0x00000000

Summary: 8/100 lines valid
=== END ICACHE DUMP ===
```

Είναι οι εντολές που έτρεξαν μαζί με το padding (NOPs) για το μπλοκ.

Μετά το flush:

```
=== COMPLETE INSTRUCTION CACHE DUMP ===
Cache Configuration: 256 lines, 4 words per line
----------------------------------------
Summary: 0/256 lines valid
=== END ICACHE DUMP ===
```

Προφανώς, αφού δεν έχει κάτι άλλο να τρέξει.

Ας συνεχίσουμε με flush μόνο αλλιώς δεν θα βγάζει και πολύ νόημα μία non-flushed direct-mapped cache

Data Cache:

```
=== COMPLETE DATA CACHE DUMP ===
Cache Configuration: 256 lines, 4 words per line
Index bits: 8, Offset bits: 2, Tag bits: 20
----------------------------------------
Line 000: V=1 D=0 Tag=0x00080000 (VA~0x80000000)
      Data: 0x00000000 0x00001234 0x00000000 0x00000000
      ASCII: .... 4... .... ....

Summary: 1/100 lines valid, 0 dirty
=== END CACHE DUMP ===
```

Ακριβώς αυτό που περιμέναμε, η καταχώρηση 0x1234.

Ας προχωρήσουμε απευθείας στην κοινή κρυφή μνήμη (επίπεδο 2):

```
=== COMPLETE L2 CACHE DUMP ===
Cache Configuration: 400 lines, 8 words per line (PIPT)
Index bits: a, Offset bits: 3, Tag bits: 11
----------------------------------------
Line 0000: V=1 D=0 Tag=0x00000000 (PA=0x0)
      Data: 0x00000000 0x00001234 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000

Line 0001: V=1 D=0 Tag=0x00003f80 (PA=0x1fc00020)
      Data: 0x340b0000 0x408b0000 0x42000002 0x3c0c8000 0x358c0004 0x340d1234 0xad8d0000 0x00000000

Line 0002: V=1 D=0 Tag=0x00003f80 (PA=0x1fc00040)
      Data: 0x00000000 0x8d8e0000 0x00000000 0x15ae0008 0x340fffff 0x10000004 0x340f0000 0x00000000

Line 0003: V=1 D=0 Tag=0x00003f80 (PA=0x1fc00060)
      Data: 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000

Summary: 4/400 lines valid, 0 dirty
=== END L2 CACHE DUMP ===
```

Άψογα, ακριβώς αυτό που περιμένουμε μετά από τα writebacks.

Τέλος, ας δούμε και την CDU:

```
=== COMPLETE CDU CACHE DUMP ===
CDU Configuration: 40 cache lines, 10 words per decryption block
Block size: 40 bytes per block
Encryption status: ACTIVE
----------------------------------------
Block 000: V=1 D=0 Base=0x00000000
      Plaintext :
0x00000000 0x00001234 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
                0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
      Ciphertext from Memory:
0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
                0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
      Range     : 0x0 - 0x3f (64 bytes)

Block 001: V=1 D=0 Base=0x1fc00040
      Plaintext :
0x00000000 0x8d8e0000 0x00000000 0x15ae0008 0x340fffff 0x10000004 0x340f0000 0x00000000
                0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
      Ciphertext from Memory:
0xcf0cd00b 0x15f714e3 0x0537ff8d 0x94d95c71 0xe2d1c469 0x3bcf1bd8 0x14bf966a 0xeb08d924
                0x39958927 0x5ed200ee 0xb2cabc2b 0x62ee3ce8 0x8712a94d 0x93265428 0xb9b9d256 0xc4ffa1af
      Range     : 0x1fc00040 - 0x1fc0007f (64 bytes)

Summary: 2/64 blocks valid, 0 dirty
Total cached data: 128 bytes
Cipher metadata entries: 3
=== END CDU CACHE DUMP ===
```

Τέλεια, ας μην ξεχνάμε ότι είναι direct-mapped και δεν βλέπουμε όλες τις εντολές. Επίσης, εφ' όσον το 0x1234 δεν ήρθε από την μνήμη (το flush έγινε στο τέλος της προσομοιώσης,

οπότε έμεινε στην data cache στο επίπεδο 1 και δεν έφτασε ποτέ πραγματικά στην μνήμη, μέχρι φυσικά τότε που έγινε το flush) και ορθώς δεν υπάρχει ciphertext για αυτό.

Ας ελέγξουμε και την κατάσταση της TLB:

```
=== TLB DUMP ===
Index | Valid | EntryHi    | EntryLo0   | EntryLo1   | PageMask
------|-------|------------|------------|------------|----------
00000 | 00Yes | 0x80000000 | 0x0001001f | 0x0002001f | 0x00000000
      VPN=0x80000000, ASID=0, G=1, V0/V1=1/1, D0/D1=1/1
=== END TLB DUMP ===
```

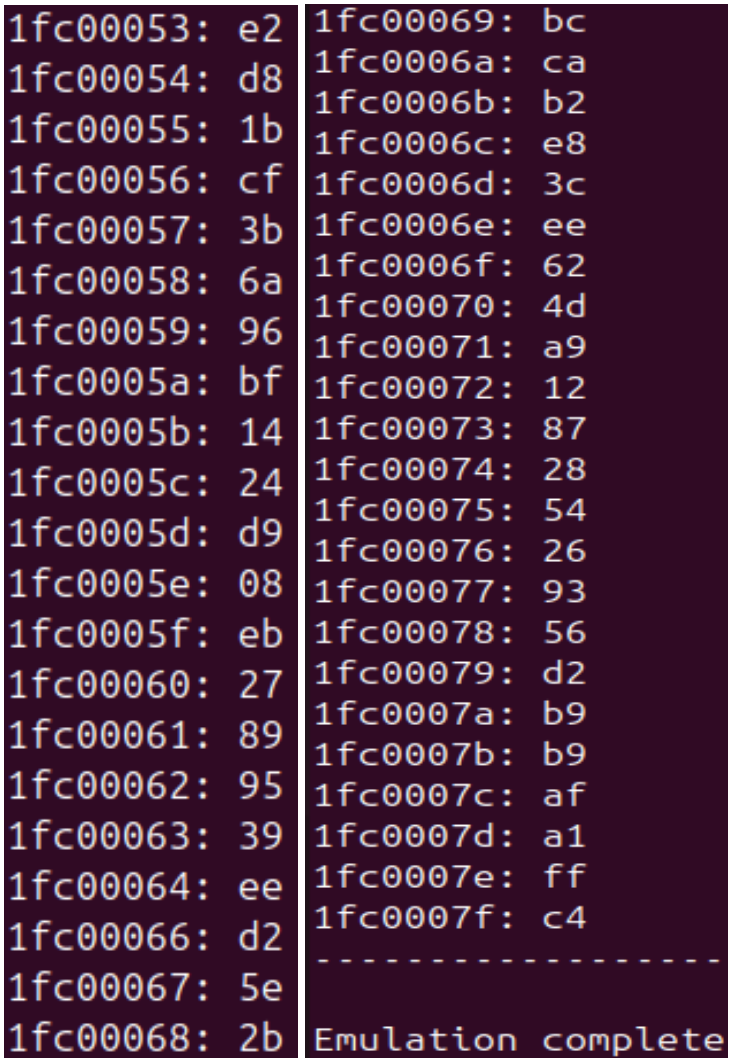
Όπως ορίστηκε από το πρόγραμμα που έτρεξε.

Το πείραμα μέχρι τώρα εκτελέστηκε με επιτυχία. Ήρθε όμως η στιγμή που θα αναλύσουμε τα στοιχεία της Κύριας Μνήμης. Εάν τα δεδομένα είναι αδιάβαστα, τότε πραγματικά, η εργασία ολοκληρώθηκε με επιτυχία.

```
Memory dump from physical 0x0 to 0x3fffffff
------------------------------------------------------------
```

Είναι υπερβολικά μεγάλη η τελική διεύθυνση για το πείραμα.

```
00000000: fe
00000001: a4
00000002: ad
00000003: 23
00000004: 57
00000005: 85
00000006: e4
00000007: d3
00000008: cf
00000009: c7
0000000a: 87
0000000b: 7e
0000000c: 69
0000000d: 2d
0000000e: 8e
0000000f: a2
00000010: 1e
00000011: 52
00000012: 9d
00000013: e5
00000014: 2c
00000015: bb
00000016: 3f
00000017: a9
00000018: 8e
00000019: f3
0000001a: d4
0000001b: c9
0000001c: 78
0000001d: a9
0000001e: c8
0000001f: b4
00000020: 0d
00000021: 95
00000022: ae
00000023: f3
00000024: 39
00000025: e2
00000026: 65
00000027: 96
00000028: 31
00000029: 46
0000002a: 8c
0000002b: 13
0000002c: 6d
0000002d: 0d
0000002e: da
0000002f: 79
00000030: 28
00000031: de
00000032: ab
00000033: 6a
00000034: f4
00000035: 39
00000036: 83
00000037: b5
00000038: 1c
00000039: 4b
0000003a: 1d
0000003b: 7a
0000003c: 10
0000003d: 03
0000003e: fc
0000003f: 2a
1fc00000: 1d
1fc00001: 4a
1fc00002: 98
1fc00003: b0
1fc00004: 3a
1fc00005: cf
1fc00006: 69
1fc00007: 39
1fc00008: 61
1fc00009: 35
1fc0000a: ca
1fc0000b: c0
1fc0000c: 78
1fc0000d: 4e
1fc0000e: da
1fc0000f: 18
1fc00010: de
1fc00011: 5f
1fc00012: 9c
1fc00013: f2
1fc00014: ea
1fc00015: 94
1fc00016: fe
1fc00017: 40
1fc00018: 78
1fc00019: 71
1fc0001a: 5e
1fc0001b: b0
1fc0001c: 24
1fc0001d: 14
1fc0001e: 9f
1fc0001f: f9
1fc00020: 49
1fc00021: dc
1fc00022: b3
1fc00023: 53
1fc00024: fa
1fc00025: bf
1fc00026: c3
1fc00027: 8c
1fc00028: 9b
1fc00029: f4
1fc0002a: b4
1fc0002b: de
1fc0002c: 8a
1fc0002d: a8
1fc0002e: ac
1fc0002f: 95
1fc00030: 69
1fc00031: 5d
1fc00032: 4c
1fc00033: e5
1fc00034: 7d
1fc00035: 01
1fc00036: a9
1fc00037: c5
1fc00038: e5
1fc00039: fb
1fc0003a: d4
1fc0003b: 1d
1fc0003c: 6c
1fc0003d: a2
1fc0003e: a6
1fc0003f: 03
1fc00040: 0b
1fc00041: d0
1fc00042: 0c
1fc00043: cf
1fc00044: e3
1fc00045: 14
1fc00046: f7
1fc00047: 15
1fc00048: 8d
1fc00049: ff
1fc0004a: 37
1fc0004b: 05
1fc0004c: 71
1fc0004d: 5c
1fc0004e: d9
1fc0004f: 94
1fc00050: 69
1fc00051: c4
1fc00052: d1
```



Το πείραμα ολοκληρώθηκε με επιτυχία. Εάν κάποιος καταφέρει να διαβάσει την ΜΤΠ, δεν θα καταλάβει τίποτα, αφού όλα είναι κρυπτογραφημένα.

Ενδιαφέρον είναι επίσης, το γεγονός ότι παρόλο που χρησιμοποιήθηκε μία εγγραφή στην 0x4, χρειάστηκε να χρησιμοποιήσουμε ένα ολόκληρο block, με πολλά μηδενικά όπου βλέπουμε μάλιστα ότι το αποτέλεσμα είναι τυχαίο, εξηγήσαμε, ήδη πως έτσι λειτουργούν τα block. Η ΜΤΠ, είναι κρυπτογραφημένη, τα σημεία όμως που χρησιμοποιήθηκαν. Η μέθοδος που χρησιμοποίησα για το dump αγνοεί τα μηδενικά και εφ' όσον ποτέ δεν προσπελάστηκαν οι υπόλοιπες διευθύνσεις, ποτέ δεν κρυπτογραφήθηκαν. Τέλος, τα στοιχεία είναι μηδέν αφού, κατά την κατασκευή της μνήμη στην αρχή της προσομοίωσης, γράφτηκε ολόκληρη με μηδενικά.

```
// Constructor
Memory::Memory(uint64_t size) // Predefined value is 1GB
{
    // Allocate memory and set to 0; memory is volatile
    memory.resize(size, 0); // Allocate
```

Συμπέρασμα

Η αρχιτεκτονική προσομοιώθηκε με επιτυχία και τα αποτελέσματα τα της ήταν θετικά και όπως τα περιμέναμε.

Τα δεδομένα της Κύριας Μνήμης είναι πλήρως κρυπτογραφημένα καθώς και υπάρχει μηχανισμός αυθεντικοποίησης τους. Ο επεξεργαστής με την χρήση της Cache and Decryption Unit, στο επίπεδο 3 των κρυφών μνημών, είναι υπεύθυνος για την αποκρυπτογράφηση των δεδομένων που προέρχονται από τον κεντρικό δίαυλο επικοινωνίας (από την Κύρια Μνήμη), καθώς και ο ελεγκτής της Κύριας Μνήμης έχει μηχανισμό όπου κρυπτογραφεί τα δεδομένα όπου εισέρχονται σε αυτή.

Υπάρχει δε το πρόβλημα snooping στον κεντρικό δίαυλο επικοινωνίας, και συγκεκριμένα για την κατεύθυνση όπου τα δεδομένα τα οποία κινούνται από την Κεντρική Μονάδα Επεξεργασίας προς την Κύρια Μνήμη αλλά και το ζήτημα της διαχείρισης του μηχανισμού Direct Memory Access (DMA) κάτι όμως, το οποίο επηρεάζει και τις υπόλοιπες προτάσεις που έχουν δημοσιευθεί.

Η αρχιτεκτονική όπου προτείνω είναι εύκολο να ενσωματωθεί με τις σύγχρονες αρχιτεκτονικές ηλεκτρονικών υπολογιστών και αποδείχθηκε αρκετά υποσχόμενη, τουλάχιστον στο λογισμικό. Αλλά, ας σημειωθεί ότι, όταν υπάρχει μία κλειδαριά, για να έχει την πρόσβαση ένας ή λίγοι, τότε μπορούν να την προσπεράσουν και οι πολλοί

Τελικά, εάν το αυτόνομο τροχοφόρο ρομποτικό όχημα που μεταφέρει αγαθά στους δημόσιους δρόμους, απαχθεί από την αντίπαλη εταιρεία, και καταφέρουν να κατεβάσουν τα δεδομένα που βρίσκονται στη μνήμη του, δεν θα καταλάβουν τίποτα, αφού τα δεδομένα είναι αδιάβαστα και οι AES αλγόριθμοι είναι ανθεκτικοί και πολύ καλά ελεγμένοι από επιστήμονες και είναι πρακτικά αδύνατο, να σπάσουν από κλασσικούς υπολογιστές ή ακόμη και υπερυπολογιστές, τουλάχιστον ακόμα. Ακόμη και εάν καταφέρουν να αναλύσουν τον τρόπο διαμοιρασμού του συμμετρικού κλειδιού, και προσπαθήσουν να ανακτήσουν το κλειδί με κβαντικούς υπολογιστές, οι προσπάθειες τους, φαίνεται πως θα είναι άκαρπες.

Ευχαριστίες

Με την ολοκλήρωση της πτυχιακής μου εργασίας θα ήθελα να ευχαριστήσω θερμά τον Καθηγητή μου κ. Δημητρίου Γεώργιο που ανέλαβε να με επιβλέψει κατά την διάρκεια της εκπόνησης της εργασίας μου καθώς και για τις πολύτιμες συμβουλές του και την καθοδήγηση του καθ' όλη την διάρκεια της εκπόνησης της.

Θα ήθελα επίσης να ευχαριστήσω όλους τους Καθηγητές και προσωπικό του τμήματος Πληροφορικής και Τηλεπικοινωνιών του Πανεπιστημίου Θεσσαλίας, οι οποίοι ο καθένας με τον δικό του τρόπο συνέβαλε στην καθοριστική μου πορεία τα τελευταία τέσσερα χρόνια. Μέσα από την διδασκαλία, τις συμβουλές τους και τη γνώση που μου μετέδωσαν, μου δόθηκε η ευκαιρία να εξερευνήσω όλες τις πλευρές των επιστημών της Πληροφορικής και των Τηλεπικοινωνιών, κάτι που με διαμόρφωσε ουσιαστικά και με βοήθησε να φτάσω στην ολοκλήρωση αυτής της εργασίας, καθώς, δεν είναι μόνο η επιστήμη της Αρχιτεκτονικής Υπολογιστών που μου έδωσε ότι χρειάζομαι να σχεδιάσω αυτή την αρχιτεκτονική, αλλά όλες οι επιστήμες και οπτικές γωνίες, όσο διαφορετικές και να φαίνεται ότι είναι, επηρέασαν καθοριστικά το αποτέλεσμα και δεν θα μπορούσε να γίνει χωρίς όλα αυτά.

Τέλος, θέλω να ευχαριστήσω ειδικώς την οικογένειά μου, τον αδερφό μου και τους φίλους μου για την υπομονή τους και την αμέριστη στήριξη τους όλα αυτά τα χρόνια καθώς και την εμπιστοσύνη τους και την ώθηση τους να ολοκληρώσω αυτή την… περιπέτεια.

Είμαι πραγματικά ευγνώμων για κάθε έναν ξεχωριστά.

Με εκτίμηση,
Θοδωρής

Αναφορές

Nunberg, G. (1996). *Farewell to the information age*. Citeseer. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.40.7487

Furnell, S., & Shah, J. N. (2020). *Home working and cyber security – an outbreak of unpreparedness? Computer Fraud & Security, 2020*(8), 6–12. https://doi.org/10.1016/S1361-3723(20)30084-1

McGuire, M., & Dowling, S. (2013). *Cyber crime: A review of the evidence* (Technical report). Home Office. Retrieved from https://assets.publishing.service.gov.uk/government/uploads/system/uploads/attachment_data/file/246749/horr75-summary.pdf

Anderson, R., Barton, C., Böhme, R., Clayton, R., van Eeten, M., Levi, M., Moore, T., & Savage, S. (2019). *Measuring the changing cost of cybercrime*. In Workshop on the Economics of Information Security (WEIS). Boston, MA, USA. Retrieved from https://www.repository.cam.ac.uk/handle/1810/299065

Clausmeier, D. (2023). Regulation of the European Parliament and the Council on digital operational resilience for the financial sector (DORA). *International Cybersecurity Law Review, 4*, 79–90. https://doi.org/10.1365/s43439-022-00076-5

Ruohonen, J. (2024). A systematic literature review on the NIS2 Directive. *arXiv*. https://doi.org/10.48550/arXiv.2412.08084

Halderman, J. A., Schoen, S. D., Heninger, N., Clarkson, W., Paul, W., Calandrino, J. A., Feldman, A. J., Appelbaum, J., & Felten, E. W. (2009). Lest we remember: Cold boot attacks on encryption keys. *Communications of the ACM, 52*(5), 91–98. https://doi.org/10.1145/1506409.1506429

AMD. (2023). AMD Secure Encrypted Virtualization (SEV). https://www.amd.com/en/developer/sev.html

AMD. (2023). SEV-SNP: Strengthening VM isolation with integrity protection and more [White paper]. https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf

The Linux Kernel Documentation. (2024). AMD Memory Encryption. https://docs.kernel.org/arch/x86/amd-memory-encryption.html

Intel Corporation. (2022). Intel Total Memory Encryption - Multi-Key whitepaper. Intel Developer Zone. https://www.intel.com/content/www/us/en/developer/articles/news/runtime-encryption-of-memory-with-intel-tme-mk.html

Intel Corporation. (2022). Intel Total Memory Encryption - Multi-Key - 009. 12th Generation Intel Core Processors Datasheet. https://edc.intel.com/content/www/us/en/design/ipla/software-development-

platforms/client/platforms/alder-lake-desktop/12th-generation-intel-core-processors-datasheet-volume-1-of-2/002/intel-multi-key-total-memory-encryption/

WikiChip. (2021). Total Memory Encryption (TME) - x86. https://en.wikichip.org/wiki/x86/tme

Costan, V., & Devadas, S. (2016). *Intel SGX explained* (Technical Report No. 2016/086). IACR Cryptology ePrint Archive. https://eprint.iacr.org/2016/086.pdf

Intel Corporation. (2024). *Intel® Trust Domain Extensions (Intel® TDX) Base Architecture Specification*. Intel Developer Zone. https://www.intel.com/content/www/us/en/developer/tools/trust-domain-extensions/overview.html

Intel Corporation. (2024). Intel® Architecture Memory Encryption Technologies Specification. Intel Developer Documentation.

Unterguggenberger, M., Weiser, S., Kogler, A., Schuster, D., Nasahl, P., & Schwarz, M. (2024). TME-Box: Scalable in-process isolation through Intel TME-MK memory encryption. arXiv preprint arXiv:2407.10740. https://arxiv.org/abs/2407.10740

IBM Corporation. (2020, August 17). IBM reveals next-generation IBM POWER10 processor. IBM Newsroom. https://newsroom.ibm.com/2020-08-17-IBM-Reveals-Next-Generation-IBM-POWER10-Processor

IBM Corporation. (2022, October 10). 10 reasons why IBM Power10 is the trusted foundation for modernization. IBM Blog. https://www.ibm.com/blog/10-reasons-why-ibm-power10-is-the-trusted-foundation-for-modernization/

Apple Inc. (2024). Apple Platform Security Guide. Apple Support. https://support.apple.com/guide/security/welcome/web

Apple. (2021). Apple Platform Security – Secure Enclave. Apple Inc. https://support.apple.com/guide/security/secure-enclave-sec59b0b31ff/web

ARM Limited. (2020). ARM Architecture Reference Manual Supplement - Memory Tagging Extension. ARM Developer Documentation.

Apple Inc. (2024). Hardware security overview. Apple Support. https://support.apple.com/guide/security/hardware-security-overview-secf020d1074/web

Arm Limited. (2020). ARM architecture security technology: Overview (DDI 0487G). https://documentation-service.arm.com/static/5f212796500e883ab8e74531

Pinto, S., & Santos, N. (2019). Demystifying Arm TrustZone: A comprehensive survey. ACM Computing Surveys, 51(6), 1-36. https://doi.org/10.1145/3291047

Ampere Computing. (2022). AmpereOne product brief [Data sheet]. Ampere Computing LLC. https://amperecomputing.com/en/briefs/ampereone-family-product-brief

Cutress, I. (2023, May 13). Sound the Siryn: AmpereOne 192-core CPU. SemiAnalysis. https://www.semianalysis.com/p/sound-the-siryn-ampereone-192-core

Shilov, A. (2023, May 20). Ampere unveils 192-core CPU, controversial benchmarks. Tom's Hardware. https://www.tomshardware.com/news/ampere-unveils-192-core-cpu

Intel. (2020). Introduction to Intel® Virtualization Technology for Directed I/O (VT-d). Intel Corporation. https://www.intel.com/content/www/us/en/virtualization/vt-directed-io.html

Zhang, J., Naghibijouybari, H., & Sadredini, E. (2022). Sealer: In-SRAM AES for High-Performance and Low-Overhead Memory Encryption. arXiv. https://arxiv.org/abs/2207.01298

Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., & Hamburg, M. (2018). Meltdown: Reading kernel memory from user space. In Proceedings of the 27th USENIX Security Symposium (pp. 973–990). USENIX Association. https://meltdownattack.com/meltdown.pdf

Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., & Yarom, Y. (2019). Spectre attacks: Exploiting speculative execution. In 2019 IEEE Symposium on Security and Privacy (pp. 1–19). IEEE. https://doi.org/10.1109/SP.2019.00002

Kim, Y., Daly, R., Kim, J., Fallin, C., Lee, J. H., Lee, D., Wilkerson, C., Lai, K., & Mutlu, O. (2014). *Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors*. In Proceedings of the 41st Annual International Symposium on Computer Architecture (ISCA '14) (pp. 361–372). IEEE. https://doi.org/10.1145/2678373.2665726

Mosca, M. (2018, September). Cybersecurity in an era with quantum computers: Will we be ready? IEEE Security & Privacy, 16(5), 38–41. https://doi.org/10.1109/MSP.2018.3761723

Patterson, D. A., & Hennessy, J. L. (2017). Computer organization and design MIPS edition: The hardware/software interface (5th ed.). Morgan Kaufmann.

Kane, G., & Heinrich, J. (1992). MIPS RISC architecture. Prentice Hall.

MIPS Technologies, Inc. (2014, Aug 20). MIPS Architecture for Programmers, Volume I-A: Introduction to the MIPS32 Architecture (Revision 6.01).

MIPS Technologies, Inc. (2015, Nov 8). MIPS Architecture for Programmers, Volume I-B: Introduction to the microMIPS32 Architecture (Revision 6.00).

MIPS Technologies, Inc. (2016, Dec 15). MIPS Architecture for Programmers, Volume II-A: The MIPS32 Instruction Set (Revision 6.06).

MIPS Technologies, Inc. (2016, Dec 15). MIPS Architecture for Programmers, Volume II-B: microMIPS32 Instruction Set (Revision 6.05).

MIPS Technologies, Inc. (2015, Jul 10). MIPS Architecture for Programmers, Volume III: MIPS32 / microMIPS32 Privileged Resource Architecture (Revision 6.02).

MIPS Technologies, Inc. (2013, Jul 16). MIPS Architecture for Programmers, Volume IV-a: The MIPS16e Application-Specific Extension to the MIPS32 Architecture (Revision 2.63).

MIPS Technologies, Inc. (2014, Dec 15). MIPS Architecture for Programmers, Volume IV-e: MIPS DSP Module for MIPS32 Architecture (Revision 3.01).

MIPS Technologies, Inc. (2013, Jul 16). MIPS Architecture for Programmers, Volume IV-f: The MIPS MT Module for the MIPS32 Architecture (Revision 1.12).

MIPS Technologies, Inc. (2013, Sep 9). MIPS Architecture for Programmers, Volume IV-h: The MCU Application-Specific Extension to the MIPS32 Architecture (Revision 1.03).

MIPS Technologies, Inc. (2013, Dec 10). MIPS Architecture for Programmers, Volume IV-i: Virtualization Module of the MIPS32 Architecture (Revision 1.06).

MIPS Technologies, Inc. (2016, Feb 3). MIPS Architecture for Programmers, Volume IV-j: The MIPS32 SIMD Architecture Module (Revision 1.12).

Sweetman, D. (2006). See MIPS run (2nd ed.). Morgan Kaufmann.

Kernighan, B. W., & Ritchie, D. M. (1988). The C programming language (2nd ed.). Prentice Hall.

MIPS Technologies, Inc. (Ημερομηνία μη διαθέσιμη). Memory map [Τεχνικό σημείωμα]. Ανακτήθηκε από το διαδίκτυο τον Ιούνιο του 2025, από https://training.mips.com/basic_mips/PDF/Memory_Map.pdf

Stroustrup, B. (2013). The C++ programming language (4th ed.). Addison-Wesley.

Χατζηγιαννάκης, Ν. Μ. (2014). Η γλώσσα C++ σε βάθος. Κλειδάριθμος. ISBN: 9789604616206.

National Institute of Standards and Technology. (2023, May 9). Advanced Encryption Standard (AES): FIPS PUB 197-upd1. U.S. Department of Commerce. https://doi.org/10.6028/NIST.FIPS.197-upd1

McGrew, D. A., & Viega, J. (2004). The security and performance of the Galois/Counter Mode (GCM) of operation. In Progress in Cryptology-INDOCRYPT 2004 (pp. 343-355). Springer. https://doi.org/10.1007/978-3-540-30556-9_27

Menezes, A. J., van Oorschot, P. C., & Vanstone, S. A. (1996). Handbook of applied cryptography. CRC Press.

Dworkin, M. J. (2007). Recommendation for block cipher modes of operation: Galois/Counter Mode (GCM) and GMAC (NIST Special Publication 800-38D). National Institute of Standards and Technology. https://doi.org/10.6028/NIST.SP.800-38D

Salowey, J. A., Choudhury, A., & McGrew, D. (2008). AES Galois Counter Mode (GCM) cipher suites for TLS (RFC 5288). Internet Engineering Task Force. https://doi.org/10.17487/RFC5288

Frankel, S., & Herbert, B. (2005). The Use of AES-GCM in IPsec ESP (RFC 4106). Internet Engineering Task Force. https://doi.org/10.17487/RFC4106

Ferguson, N. (2005). Authentication weaknesses in GCM. Comments submitted to NIST Modes of Operation Process. https://csrc.nist.gov/csrc/media/projects/block-cipher-techniques/documents/bcm/comments/cwc-gcm/ferguson2.pdf

Bernstein, D. J. (2005, April 14). Cache-timing attacks on AES (Technical Report). University of Illinois at Chicago. https://cr.yp.to/antiforgery/cachetiming-20050414.pdf

Kocher, P. C. (1996). Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In Advances in Cryptology — CRYPTO '96 (pp. 104-113). Springer. https://doi.org/10.1007/3-540-68697-5_9

Iwata, T., Ohashi, K., & Minematsu, K. (2012). Breaking and repairing GCM security proofs. In Advances in Cryptology–CRYPTO 2012 (pp. 31-49). Springer. https://doi.org/10.1007/978-3-642-32009-5_3

Almeida, J. B., Barbosa, M., Barthe, G., Dupressoir, F., Emmi, M., & Grégoire, B. (2016). Verifying constant-time implementations. In 25th USENIX Security Symposium (USENIX Security 16) (pp. 53–70). USENIX Association. https://www.usenix.org/system/files/conference/usenixsecurity16/sec16_paper_almeida.pdf

OpenSSL Software Foundation. (2024). OpenSSL: Cryptography and SSL/TLS toolkit (Version 3.0.13) [Software]. https://www.openssl.org

OpenSSL Software Foundation. (2024). EVP Symmetric Encryption and Decryption. In OpenSSL Manual Pages. https://www.openssl.org/docs/man3.2/man3/EVP_EncryptInit_ex.html

ISO/IEC 18033-3:2010. (2010). Information technology — Security techniques — Encryption algorithms — Part 3: Block ciphers. International Organization for Standardization.

National Institute of Standards and Technology. (2024). FIPS 203: Module-Lattice-Based Key Encapsulation Mechanism (ML-KEM). U.S. Department of Commerce. https://doi.org/10.6028/NIST.FIPS.203

National Institute of Standards and Technology. (2022). Submission to the NIST Post-Quantum Cryptography Standardization Process: Kyber. https://pq-crystals.org/kyber

Bos, J. W., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J. M., Schwabe, P., Seiler, G., & Stehlé, D. (2018). CRYSTALS–Kyber: A CCA-secure module-LWE cryptosystem. In 2018 IEEE European Symposium on Security and Privacy (EuroS&P) (pp. 353–367). IEEE. https://doi.org/10.1109/EuroSP.2018.00032

Lakshmanan, R. (2024, September 17). Google Chrome switches to ML-KEM for post-quantum cryptography defense. The Hacker News. https://thehackernews.com/2024/09/google-chrome-switches-to-ml-kem-for.html

Google Online Security Blog. (2024, September). A new path for Kyber on the web. Google Security Blog. https://security.googleblog.com/2024/09/a-new-path-for-kyber-on-web.html

Douglas Stebila, Michele Mosca. Post-quantum key exchange for the Internet and the Open Quantum Safe project. In Roberto Avanzi, Howard Heys, editors, Selected Areas in Cryptography (SAC) 2016, LNCS, vol. 10532, pp. 1–24. Springer, October 2017. https://openquantumsafe.org