ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

ΕΡΓΑΣΤΗΡΙΟ ΛΟΓΙΚΗΣ ΚΑΙ ΕΠΙΣΤΗΜΗΣ ΥΠΟΛΟΓΙΣΜΩΝ

Αποδεδειγμένα ασφαλής ανίχνευση εισαγωγής κακόβουλου κώδικα ή δεδομένων στη μνήμη μιας εφαρμογής

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Κωνσταντίνου Δ. Μητρόπουλου

Επιβλέποντες: Αριστείδης Παγουρτζής,
Αναπληρωτής Καθηγητής ΕΜΠ

Βασίλης Ζήκας
Επίκουρος Καθηγητής, RPI, NY

Αθήνα, Ιούνιος 2017

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΛΟΓΙΚΗΣ ΚΑΙ ΕΠΙΣΤΗΜΗΣ ΥΠΟΛΟΓΙΣΜΩΝ

Αποδεδειγμένα ασφαλής ανίχνευση εισαγωγής κακόβουλου κώδικα ή δεδομένων στη μνήμη μιας εφαρμογής

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Κωνσταντίνου Δ. Μητρόπουλου

Επιβλέποντες: Αριστείδης Παγουρτζής, Αναπληρωτής Καθηγητής ΕΜΠ
Βασίλης Ζήκας, Επίκουρος Καθηγητής, RPI, NY

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 23η Ιουνίου 2017

Αριστείδης Παγουρτζής Αναπληρωτής Καθηγητής ΕΜΠ	Νικόλαος Παπασπύρου Αναπληρωτής Καθηγητής ΕΜΠ	Δημήτριος Φωτάκης Επίκουρος Καθηγητής ΕΜΠ
.....	.....	.....

.....
Μητρόπουλος Κωνσταντίνος
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών ΕΜΠ

Περίληψη

Στην παρούσα διπλωματική εργασία μελετάμε την υλοποίηση της θεωρητικής ασφάλισης της μνήμης μιας εφαρμογής από κάποιον κακόβουλο ο οποίος μπορεί, κάποια δεδομένη χρονική στιγμή, να γράψει σε αυτήν με κάποιο εξωτερικό τρόπο (πχ με ακτινοβολία ή κάνοντας παράνομη χρήση του DMA) αλλά δεν μπορεί να τη διαβάσει. Η ιδέα ανήκει στους Richard Lipton, Rafail Ostrovsky, και Βασίλη Ζήκα (http://drops.dagstuhl.de/opus/volltexte/2016/6311/ [1]).
Η προσέγγιση έχει ενδιαφέρον γιατί είναι provably secure, δηλαδή αποδεδειγμένα ασφαλής και δεν βασίζεται σε ευριστικές μεθόδους ή απλές τεχνικές διασφάλισης που ίσως να δουλεύουν στις περισσότερες περιπτώσεις αλλά δεν είναι αποδεδειγμένα ασφαλείς. Γι' αυτόν τον λόγο στοχεύει σε συγκεκριμένο πρόβλημα με συγκεκριμένες παραδοχές (μη δυνατότητα ανάγνωσης), και το λύνει εξ' ολοκλήρου.
Η ιδέα είναι να παρεμβάλουμε, ανάμεσα στα χρήσιμα bytes της μνήμης μιας εφαρμογής κομμάτια κλειδιών (keys) καθώς και Message Authentication Codes, για ελέγχους ακεραιότητας. Προϋποθέτουμε μια ασφαλή CPU η οποία πριν λάβει κάτι από τη μνήμη ελέγχει το αντίστοιχο MAC(<χρήσιμα bytes> || <keyshares>) να είναι ορθό, και μετά προχωρά κανονικά. Όταν λάβει από έναν εξωτερικό παίκτη μια πρόκληση (challenge), συνθέτοντας τα κλειδιά των οποίων τα κομμάτια έχουν διασπαρεί στη μνήμη, αποκρυπτογραφεί ορθά το challenge και απαντάει με το plaintext. Ένας επιτιθέμενος αν αλλάξει τη μνήμη είτε θα αλλάξει κάτι που θα ακυρώνει τα MACs (οπότε η ασφαλής CPU θα σταματήσει να εκτελεί ή θα καταστρέψει τα keyshares), είτε θα αλλάξει όλο το block των χρήσιμων bytes+keyshares+MAC, οπότε θα ακυρώσει τα κλειδιά και θα το καταλάβει ο εξωτερικός παίκτης.
Στόχος της διπλωματικής είναι να προσεγγίσουμε τον χρόνο που χρειάζεται για να υλοποιηθεί στην πράξη αυτό το σχήμα σε hardware. Υλοποιούμε λοιπόν την προσομοίωση στο userspace ως μια καλή προσέγγιση (του άνω φράγματος του χρόνου), χρησιμοποιώντας κώδικα που κάνει τις ασφαλείς λειτουργίες. Στον μεν κανονικό κώδικα βάζουμε επιπρόσθετα jmps (για την αγνόηση των keyshares και macs) αλλά και verification calls όταν πρέπει να ελεγχθεί ένα MAC, ενώ την υπόλοιπη μνήμη την προσπελαύνουμε με ειδικούς getters και setters (που ελέγχουν τα MAC για την υπόλοιπη μνήμη). Φυσικά, η μνήμη της εφαρμογής δεν είναι όλη “ασφαλής”, αλλά εμείς δεσμεύουμε ένα κομμάτι της το οποίο το καλούμε ασφαλές και εκτελούμε τις προσομοιώσεις εκεί.
Τα αριθμητικά αποτελέσματα καταδεικνύουν ότι κοστίζει αρκετά η συχνή επαλήθευση των MACs, και ο ολικός χρόνος είναι μεγάλος. Προτείνουμε όμως καλύτερες (και αρκετά πιο δύσκολες στην υλοποίηση) τεχνικές ακόμα καλύτερης προσέγγισης, αλλά και ιδέες (όπως η χρήση ασφαλούς cache) που μειώνουν πολύ τον επιπρόσθετο χρόνο.

Λέξεις κλειδιά

Ανίχνευση επίθεσης, Αποδεδειγμένη ασφάλεια, Εξωτερική πιστοποίηση εγκυρότητας

Abstract

In the present diploma thesis we study the implementation of the theoretical safeguarding of a process's memory from a malicious actor who can, at a given moment, write on it using some external procedure (eg with radiation or by misusing DMA), but cannot read it. The idea belongs to Richard Lipton, Rafail Ostrovsky, and Vassilis Zikas (http://drops.dagstuhl.de/opus/volltexte/2016/6311/ [1]).
This approach is interesting because it is provably secure, and does not depend on heuristics or other techniques that may work on some occasions but are not provably secure. That's why it targets a specific problem with specific assumptions (the adversary does not have the ablitity to read the memory before injecting), and solves it completely.
The idea is to interleave, among the useful bytes in a process's memory, keyshares as well as Message Authentication Codes, for integrity. We assume a secure CPU which, before loading something from memory, checks the corresponding MAC(<useful bytes>|| <keyshares>) to be correct, and afterwards it continues the execution. When it receives a challenge from an attestator, it combines the keyshares to produce keys and decrypts the challenge answering with the plaintext. An attacker who will alter the memory, will either alter something that invalidates the MACs (so the secure CPU will stop executing or will destroy the keyshares), or will alter the entire block of useful bytes+keyshares+MAC, so the keys will be altered and the remote attestator will detect it.
The target of this diploma thesis is to approach the execution time that it takes for a schema like that being implemented in hardware. So, we implement the simulation in the userspace as a good estimate (of the upper bound of the execution time), using code that does the secure operations. We insert jmps in the normal code (so as to ignore keyshares and MACs), as well as verification calls when a MAC needs to be checked. The rest of the memory is accessed through secure getters and setters (who check the macs as well). Of course, the entire process's memory is not “secure”, but we allocate a part of it which we call secure and run the simulations there.
The numerical results show that the continuous MAC verification costs a lot, so it takes a lot for the program to complete. We propose better (and harder to impelement) techniques of an even better approach of the execution time, as well as ideas (like the utilization of a secure cache) who reduce the total time by a big amount.

Keywords

Injection detection, Provable Security, Remote attestation

Ευχαριστίες

Η παρούσα διπλωματική εργασία δεν θα μπορούσε να έχει ολοκληρωθεί χωρίς την επιστημονική επίβλεψη και την ηθική συμπαράσταση του Επίκουρου Καθηγητή του RPI (στον τομέα της Επιστήμης των Υπολογιστών) κ. Βασίλη Ζήκα, καθώς και του Αναπληρωτή Καθηγητή ΕΜΠ (στον Τομέα Τεχνολογίας Πληροφορικής και Υπολογιστών) κ. Αριστείδη Παγουρτζή.

Κομβικό ρόλο επίσης έπαιξαν ο Καθηγητής του UCLA (στον τομέα της Επιστήμης των Υπολογιστών) κ. Rafail Ostrovsky, η διδακτορική φοιτήτρια του RPI Yun Lu, ο διδακτορικός φοιτητής του RPI Avi Weinstock, και η απόφοιτος του UCLA Ashwini Nene. Τους ευχαριστώ θερμά για την πολύτιμη βοήθεια και τη συνεργασία τους.

Ευχαριστώ τους φίλους μου, Αλέξανδρο, Βασίλη, Γεωργία, Γεωργία, Δέσποινα, Κωνσταντίνο, Μανώλη και Φιλήμονα για την εμψύχωση και τη συμπόρευση κατά τη διάρκεια των φοιτητικών μου χρόνων.

Ευχαριστώ τα αδέρφια μου για την στήριξη που μου προσέφεραν όλα αυτά τα χρόνια.

Ευχαριστώ τους γονείς μου οι οποίοι μου μετέδωσαν την αγάπη για τη διαρκή μάθηση και ήταν πάντα δίπλα μου στηρίζοντας τις επιλογές μου.

Τέλος, ευχαριστώ όλους όσους αγωνίστηκαν και αγωνίζονται για την προώθηση και ανάπτυξη του Ελεύθερου Λογισμικού/Λογισμικού Ανοιχτού Κώδικα (ΕΛ/ΛΑΚ) διότι ένα τεράστιο κομμάτι των εργαλείων που χρησιμοποίησα, όχι μόνο για την εκπόνηση της παρούσας διπλωματικής αλλά καθόλη τη διάρκεια των φοιτητικών μου χρόνων, ήταν ΕΛ/ΛΑΚ.

Πίνακας Περιεχομένων

Ευρετήριο Σχημάτων

Κεφάλαιο 1: Εισαγωγή

Εδώ και πολλά χρόνια ο άνθρωπος χρησιμοποιεί τους υπολογιστές για να λύσει τα διάφορα προβλήματα που έχει, και η χρήση των υπολογιστών είναι από ερευνητική ως λογιστική, και από ενημερωτική ως ψυχαγωγική.

1.1: Επιβλαβή προγράμματα

Δυστυχώς όμως ο κόσμος δεν είναι τέλειος και δεν άργησαν να εμφανιστούν προγράμματα που μόνο στόχο είχαν τη ζημία αυτού που θα τα έτρεχε στον υπολογιστή του. Φτιάχτηκαν προγράμματα με διάφορες κακές επιπτώσεις [2].
Μερικά κατέστρεφαν τα δεδομένα του χρήστη, όπως το Friday 13. Άλλα κατασκόπευαν και συνέλεγαν πληροφορίες για τον χρήστη, ώστε να μαθευτούν passwords ή αργότερα να πουληθούν τα προσωπικά δεδομένα του χρήστη. Κάποια τρίτα ήταν κατασκευασμένα έτσι ώστε να διασπείρονται και να αποκτούν τον έλεγχο και άλλων συσκευών, περιμένοντας τις εντολές του κακόβουλου κατασκευαστή τους. Η λίστα είναι πολύ μεγάλη, αν και αξίζει να αναφέρουμε και τα πιο πρόσφατα ransomware, που κρυπτογραφούν τα αρχεία του χρήστη και ζητούν λύτρα να τα αποκρυπτογραφήσουν.
Ο λόγος ανάπτυξης επιβλαβών προγραμμάτων μπορεί να είναι πολύπλευρος. Κάποια έχουν στόχο απλά την καταστροφή. Άλλα έχουν στόχο το οικονομικό κέρδος (όπως τα ransomware), ενώ σε κάποια άλλα ο σκοπός είναι στρατιωτικός (ο ιός Stuxnet μετέβαλλε τις εντολές στους μηχανισμούς φυγοκέντρησης απεμπλουτισμού ουρανίου στο πυρηνικό πρόγραμμα του Ιράν, με αποτέλεσμα να καταστραφούν τα μηχανήματα).

Οι άνθρωποι προσπάθησαν να προστατευτούν από τα επιβλαβή προγράμματα με αντιβιοτικά προγράμματα [3], και η κατάσταση οδηγήθηκε σε μια “κούρσα εξοπλισμών”. Οι νέοι ιοί χρησιμοποιούν εξελιγμένες τεχνικές (binary obfuscation, κρυπτογραφία κλπ), τα αντιβιοτικά επιστρατεύουν το machine learning, και η σύγκρουση όσο πάει γίνεται όλο και πιο πολύπλοκη. Σε κάθε περίπτωση όμως, ανεξάρτητα από το αντιβιοτικό πρόγραμμα, πάντα θα βρεθεί κακόβουλο λογισμικό που θα περάσει την προστασία και θα μολύνει ένα σύστημα.
Η ιδέα την οποία υλοποιεί η παρούσα διπλωματική εργασία δεν είναι ένα ακόμα βήμα στην κούρσα των εξοπλισμών, αλλά, κάτω από συγκεκριμένες παραδοχές επίθεσης, ανιχνεύει την εισαγωγή κακόβουλου προγράμματος σε κάθε περίπτωση.

1.2: Τύποι επίθεσης

Υπάρχουν διάφοροι τρόποι ένα κακόβουλο πρόγραμμα να μολύνει έναν υπολογιστή. Ο πιο συνηθισμένος τρόπος είναι να το εκτελέσει ο χρήστης, νομίζοντας ότι δεν πρόκειται να γίνει κάτι κακό. Πολλοί χρήστες οι οποίοι δεν είναι ειδικοί στους υπολογιστές τρέχουν προγράμματα χωρίς να επαληθεύσουν την προέλευσή τους, με αποτέλεσμα στην καλή περίπτωση να γεμίσουν με διαφημιστικά toolbars τον φυλλομετρητή τους, και στην κακή να

κολλήσουν κάποιον ιό.
Άλλος τρόπος είναι το πρόγραμμα να δεχτεί είσοδο την οποία δεν περιμένει. Ένα παράδειγμα είναι οι επιθέσεις buffer overflow [4], που πανωγράφουν ό,τι υπάρχει μετά από έναν buffer στη μνήμη (πχ return addresses). Άλλου είδους επιθέσεις είναι τα code injection attacks [5], όπου ένα κομμάτι input θεωρείται από μια (interpreted) γλώσσα ως μια απλή συμβολοσειρά, ενώ στην πράξη είναι κώδικας που εκτελείται (πχ SQL Injection). Τέτοιες επιθέσεις απαιτούν να “αποστειρώνεται” (sanitized) κάθε input, και να γίνεται έλεγχος ώστε αυτό να βρίσκεται μέσα σε καθορισμένα πλαίσια (πχ ένα τηλέφωνο πρέπει να είναι μόνο αριθμοί, και να μην έχει γράμματα).
Ένας ακόμα τρόπος επίθεσης είναι γενικά λάθη στην υλοποίηση. Αν ο προγραμματιστής έχει ξεχάσει να συμπεριλάβει μια περίπτωση εκτέλεσης (όπως ένα race condition [6] ή έναν συγκεκριμένο συνδυασμό παραμέτρων) όταν εκτελείται ένα κομμάτι κώδικα, μπορεί το τελικό αποτέλεσμα να μην είναι το αναμενόμενο και το πρόγραμμα να δώσει στον επιτιθέμενο δυνατότητες που δεν θα έπρεπε να μπορεί να έχει (πχ privilege escalation [7]). Το ίδιο μπορεί να επιτευχθεί με λάθη στον ορισμό δικαιωμάτων χρηστών σε ένα filesystem.
Ομοίως με πριν, οι τρόποι επίθεσης μπορεί να είναι πάρα πολλοί, και η λίστα πολύ μεγάλη. Αυτός που εξετάζεται στην παρούσα διπλωματική είναι η αλλαγή, με κάποιο εξωτερικό τρόπο, της μνήμης μιας εφαρμογής. Για παράδειγμα, ο επιτιθέμενος, χρησιμοποιώντας ακτινοβολία ή με παράνομη χρήση του DMA, αλλάζει (χωρίς να χρησιμοποιήσει εντολές της CPU) τη μνήμη (άρα και χωρίς να το καταλάβει η CPU). Ίδιο αποτέλεσμα μπορεί να επιτευχθεί με τεχνικές Row Hammer [8], όπου οι συχνές αλλαγές ενός κομματιού φυσικής μνήμης επηρεάζουν τα διπλανά του με αποτέλεσμα να αλλάξουν και αυτά λόγω ηλεκτρικών ιδιοτήτων. Έτσι, τη μια χρονική στιγμή η μνήμη είναι η ορθή, και την άλλη η μνήμη περιέχει μέσα της τον κακόβουλο κώδικα (ή δεδομένα) του επιτιθέμενου, τα οποία ενεργοποιούνται αν προσπελαστούν.

1.3: Συναρτήσεις Κατακερματισμού (hash functions)

Η κρυπτογραφία χρησιμοποιεί διάφορα primitives για να επιτελέσεις διάφορες λειτουργίες. Στην παρούσα διπλωματική θα χρειαστούμε κάποια από αυτά, τα οποία θα παρουσιαστούν με συντομία.

Ένα από αυτά τα στοιχεία είναι οι συναρτήσεις κατακερματισμού (hash functions) [9], οι οποίες είναι συναρτήσεις μονής κατεύθυνσης (δεν αντιστρέφονται), και δεδομένου ενός μηνύματος A, παράγουν το αποτύπωμά του a.
Επιπλέον έχουν τις εξής ιδιότητες:
α) Είναι ντετερμινιστικές, με την έννοια ότι το ίδιο μήνυμα πάντα θα δίνει το ίδιο αποτύπωμα.
β) Είναι αρκετά γρήγορες στον υπολογισμό.
γ) Είναι αδύνατο (δηλαδή έχει αμελητέα πιθανότητα) να παραχθεί ένα μήνυμα A δεδομένου ενός αποτυπώματος a, και το καλύτερο που μπορεί να κάνει κάποιος είναι να δοκιμάσει όλους τους πιθανούς συνδυασμούς με ωμή βία.
δ) Μια μικρή αλλαγή στο μήνυμα A θα πρέπει να δίνει μεγάλες αλλαγές στο αποτύπωμά του a.

ε) Είναι υπολογιστικά αδύνατο (δηλαδή έχει αμελητέα πιθανότητα) να βρεθούν δύο διαφορετικά μηνύματα μηνύματα με ίδιο αποτύπωμα.

Δεν θα εμβαθύνουμε πάνω στις συναρτήσεις hash, και απλά θα αρκεστούμε να πούμε πως υπάρχουνε αρκετές που δείχνουν να έχουν τις επιθυμητές ιδιότητες, όπως οι SHA256, SHA512, SHA3 κλπ.
Για παράδειγμα, το sha256 αποτύπωμα της φράσης “Ο παπάς ο παχύς έφαγε παχιά φακή” είναι “5cd22fb9ab91b9b4926c40c72903807713598bdef50dac0c6aa8112895c48258”. Αν αλλάξουμε το πρώτο γράμμα και το βάλουμε μικρό: “ο παπάς ο παχύς έφαγε παχιά φακή” το hash δίνει “0cf4b6f9f52491abc6e85d8f57262cb72281016fd85abbf81b1bd91ba0b3a0ba” (το οποίο είναι εντελώς διαφορετικό).

1.4: Message Authentication Codes

Ένα ακόμα στοιχείο που θα χρησιμοποιήσουμε είναι τα Message Authentication Codes (MACs) [10]. Αυτά είναι ένα κομμάτι πληροφορίας (συνήθως μερικά bytes) τα οποία πιστοποιούν την ακεραιότητα ενός μηνύματος, καθώς και πιστοποιούν τον αποστολέα/ κατασκευαστή αυτού πως είναι ο σωστός. Έστω λοιπόν ότι έχουμε ένα μήνυμα Μ. Αν δίπλα του βάλουμε ένα MAC (ή όπως αλλιώς αποκαλείται tag) για το οποίο έχουμε προσυμφωνήσει τη μορφή του, μπορούμε να σιγουρευτούμε πως ο αποστολέας του μηνύματος είναι αυτός που νομίζουμε και το μήνυμα Μ δεν έχει παραλλαχτεί.
Αυτό προϋποθέτει το εξής: Μαζί με τον αποστολέα έχουμε προσυμφωνήσει σε ένα κοινό κλειδί (που θα πρέπει να είναι αρκετά πολύπλοκο ώστε να μην μπορεί κάποιος να το μαντέψει) και έχουμε έναν αλγόριθμο παραγωγής MAC που μας δίνει τις εξής δυνατότητες:
α) Δεδομένου μηνύματος Μ και κλειδιού k, μπορεί να παράξει ένα tag t.
β) Το tag t δεν θα μπορεί να φτιαχτεί μόνο με το μήνυμα Μ και χωρίς το κλειδί k (ή θα μπορεί να φτιαχτεί με εντελώς αμελητέα πιθανότητα).
Αλλά και πιο ισχυρές δυνατότητες:
γ) Ακόμα κι αν ο επιτιθέμενος έχει ένα μαντείο το οποίο του παράγει ορθά tags για δικά του μηνύματα, δεν μπορεί να υπολογίσει τα tags για άλλα μηνύματα χωρίς να συμβουλευτεί το μαντείο.

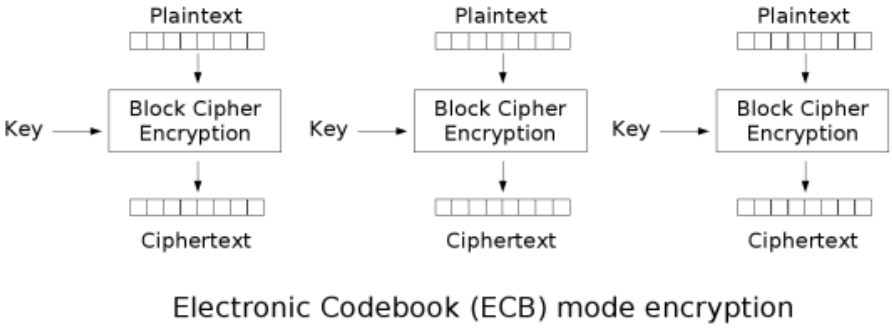
Τα MACs δεν δίνουν την μη-αποποίηση ευθύνης, μιας και οποιοσδήποτε κατέχει το κλειδί για να επαληθεύσει το MAC, μπορεί να κατασκευάσει μηνύματα και να ισχυριστεί ότι τα κατασκεύασε ένας άλλος χρήστης οποίος κατέχει κι αυτός το κλειδί. Παρ’όλα αυτά, μέσα σε μια ομάδα χρηστών που κατέχουν ένα μυστικό κλειδί k, μια επαλήθευση MAC πιστοποιεί ότι τα μηνύματα ήρθαν από κάποιο μέλος της ομάδας και δεν κατασκευάστηκαν από κάποιον τρίτο κακόβουλο.
Υπάρχουν κάποιοι τρόποι να υλοποιηθεί ένα Message Authentication Code. Οι δυο πιο γνωστοί είναι μέσω μιας συνάρτησης hash και μέσω αλγορίθμων block cipher.
Μέσω μιας συνάρτησης hash για παράδειγμα, θα μπορούσε κανείς να υπολογίσει το hash(<key>||<message>||<key>), και έτσι να παράξει ένα σχήμα MAC, γιατί μόνο αν κάποιος γνωρίζει το κλειδί μπορεί να παράξει το σωστό αποτύπωμα.

Ο δεύτερος τρόπος είναι αυτός τον οποίον χρησιμοποιούμε στην παρούσα διπλωματική (διότι έχουμε hardware που τον υλοποιεί), μέσω block ciphers, χρησιμοποιώντας έναν αλγόριθμο κρυπτογράφησης. Αυτόν θα τον παρουσιάσουμε πιο αναλυτικά στις ενότητες 1.6 και 1.7 και εδώ θα πούμε επιγραμματικά πως το tag ενός μηνύματος είναι στην πράξη το τελευταίο κομμάτι της κρυπτογράφησης του μηνύματος με το κλειδί, το οποίο όμως εξαρτάται από όλα τα προηγούμενα, άρα από όλο το μήνυμα.

1.5: Συμμετρική και Ασύμμετρη Κρυπτογραφία

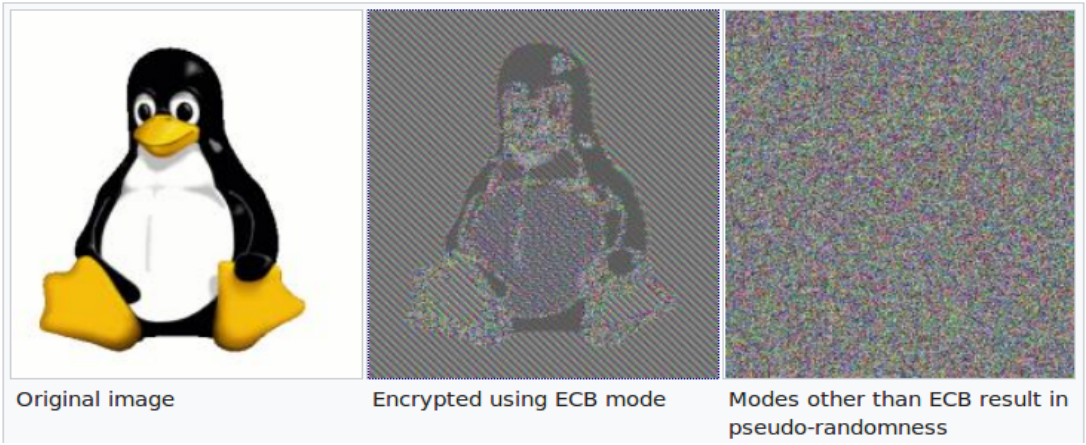
Επιγραμματικά επίσης θα μιλήσουμε για τις δύο κύριες μορφές κρυπτογραφίας στη σύγχρονη εποχή.
Η πρώτη και απλή περίπτωση είναι η συμμετρική κρυπτογραφία [11]. Σε αυτήν ο αποστολέας και ο παραλήπτης διαθέτουν το ίδιο μυστικό κλειδί, και ο ένας κρυπτογραφεί τα μηνύματα με αυτό, ενώ ο άλλος τα αποκρυπτογραφεί με το ίδιο κλειδί. Ο τρόπος αυτός της κρυπτογράφησης υπάρχει από την αρχαιότητα, και μπορεί να είναι από πολύ απλός (πχ απλά αλλαγή του γράμματος του κειμένου κατά 13 θέσεις μπροστά) ως πολύ σύνθετος (όπως ο σύγχρονος αλγόριθμος AES). Οι αλγόριθμοι συμμετρικής κρυπτογράφησης χωρίζονται σε 2 κατηγορίες:
α) Οι αλγόριθμοι ροής (stream ciphers) οι οποίοι κρυπτογραφούν ένα-ένα τα bits ενός μηνύματος (πχ Salsa20, RC5).
β) Οι αλγόριθμοι block (block ciphers) κόβουν το μήνυμα σε κομμάτια συγκεκριμένου μεγέθους (αν δεν φτάνει σε αυτό το μέγεθος εφαρμόζουν padding), και κρυπτογραφούν κάθε ένα κομμάτι (πχ AES, Blowfish).
Οι αλγόριθμοι block έχουν διάφορους τρόπους να κάνουν την κρυπτογράφηση. Οι δυο πιο γνωστοί είναι οι ECB mode και CBC mode (υπάρχουν και άλλοι όπως μπορεί να δει κάποιος στο [12]).

Ο τρόπος ECB είναι να κρυπτογραφείται κάθε block ξεχωριστά. Αυτό μπορεί να είναι γρήγορο μιας και παραλληλοποιείται, αλλά δεν είναι πάντα ασφαλές. Όταν τα plaintext είναι ίδια, τότε και τα ciphertext είναι ίδια.



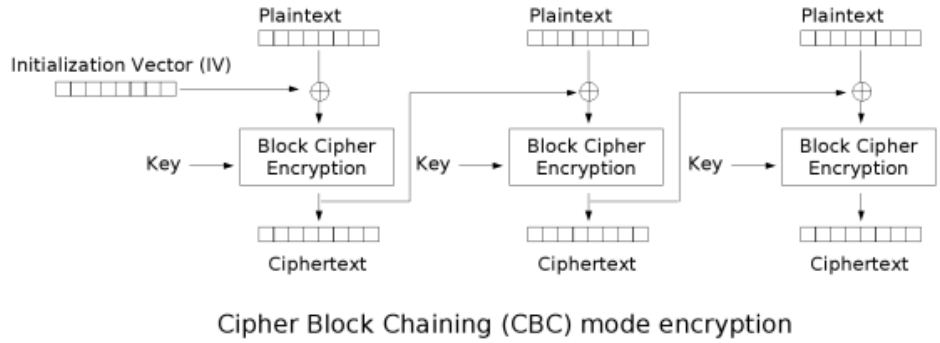
Σχήμα 1-1. Η κρυπτογράφηση (και αντίστοιχα η αποκρυπτογράφηση) του ECB.

Στην ακόλουθη εικόνα μπορούμε να δούμε το μεγάλο ελάττωμα του ECB τρόπου κρυπτογράφησης. Στην εικόνα τύπου bitmap με τον Tux, ακόμα και αν την έχουμε κρυπτογραφήσει, είναι αρκετά σαφές το plaintext.

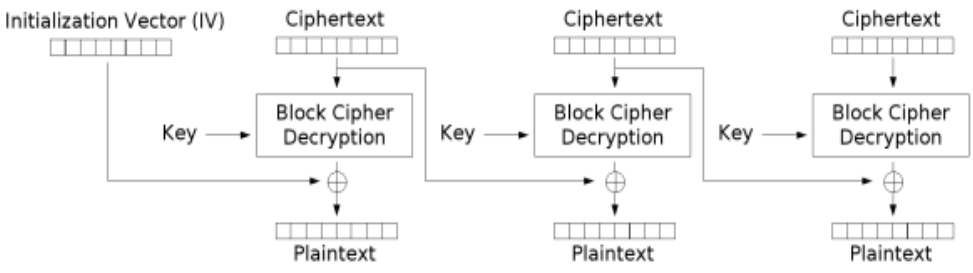


Σχήμα 1-2. Η αδυναμία του ECB τρόπου κρυπτογράφησης.

Λόγω λοιπόν της αδυναμίας του ECB τρόπου κρυπτογράφησης, προτάθηκαν και άλλοι. Ο πιο γνωστός (τον οποίο χρησιμοποιούμε στην παρούσα διπλωματική) είναι ο CBC τρόπος κρυπτογράφησης.



Σχήμα 1-3. Η κρυπτογράφηση του CBC.



Σχήμα 1-4. Η αποκρυπτογράφηση του CBC.

Όπως μπορούμε να δούμε από τα σχήματα, ξεκινάμε με το πρώτο block του μηνύματος και ένα Initialization Vector (το οποίο είναι από πριν γνωστό) και κάνοντας XOR με το IV το plaintext του πρώτου block, κρυπτογραφούμε και παράγουμε το πρώτο block του κρυπτοκειμένου. Για κάθε επόμενο block, κάνουμε XOR το ciphertext του προηγούμενου block στο plaintext, επηρεάζοντας με αυτό τον τρόπο το τι κρυπτογραφείται (άρα και το αποτέλεσμα της κρυπτογράφησης). Έτσι λοιπόν, όλα τα blocks εξαρτώνται από το plaintext των ιδίων και από τα προηγούμενά τους, λύνοντας το πρόβλημα που παρουσίασε ο τρόπος ECB. Με ανάλογο τρόπο γίνεται η αποκρυπτογράφηση.

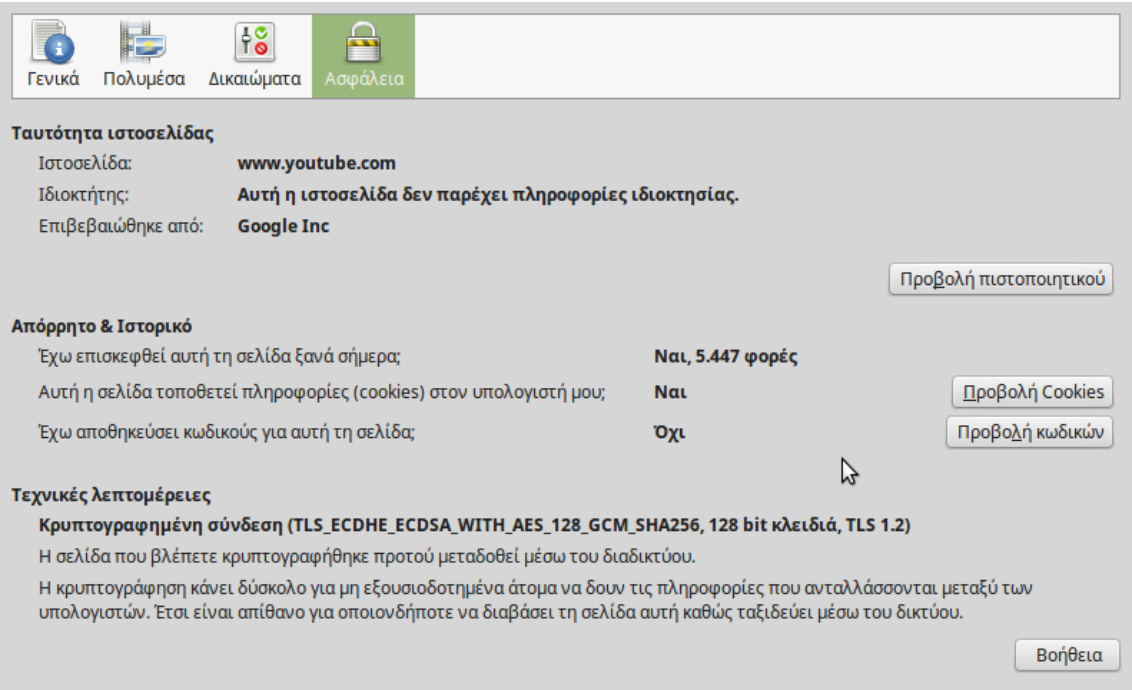
Ένα γενικό μειονέκτημα των συμμετρικών αλγορίθμων είναι η δυσκολία μετάδοσης του κλειδιού, σε περίπτωση που δεν υπάρχει φυσική επαφή. Αν οι επικοινωνούντες είναι σε διαφορετικά μέρη του κόσμου, καθίσταται πολύ δύσκολο να συμφωνήσουν σε ένα κλειδί όντας σίγουροι πως κανείς ενδιάμεσος δεν το έχει κρυφακούσει.

Ο δεύτερος τρόπος κρυπτογραφίας που χρησιμοποιείται την σημερινή εποχή είναι η ασύμμετρη κρυπτογραφία, η οποία εφευρέθηκε τη δεκαετία του 1970. Είναι επίσης γνωστή με το όνομα κρυπτογραφία δημοσίου κλειδιού [13].
Στην κρυπτογραφία δημοσίου κλειδιού κάθε χρήστης έχει δύο κλειδιά, το δημόσιο και το ιδιωτικό. Ό,τι κρυπτογραφείται με το δημόσιο, μπορεί να αποκρυπτογραφηθεί μόνο με το ιδιωτικό. Ό,τι κρυπτογραφείται με το ιδιωτικό, μπορεί να αποκρυπτογραφηθεί μόνο με το δημόσιο. Ο χρήστης κρατάει κρυφό το ιδιωτικό κλειδί, και διαμοιράζει τη δημόσιο. Αν θέλει κάποιος να του στείλει κάτι, κρυπτογραφεί με το δημόσιο κλειδί και είναι σίγουρος ότι μόνο ο κάτοχος του ιδιωτικού (άρα ο πρώτος χρήστης) μπορεί να το διαβάσει. Αν ο πρώτος χρήστης θέλει να υπογράψει κάτι, το κρυπτογραφεί με το ιδιωτικό (και άρα όλοι μπορούν να το αποκρυπτογραφήσουν), αλλά γνωρίζουν ότι μόνο ο κάτοχος του ιδιωτικού θα μπορούσε να έχει κάνει μια έγκυρη κρυπτογράφηση, επομένως είναι σαν να το υπογράφει.
Η ασύμμετρη κρυπτογραφία έρχεται να λύσει το πρόβλημα του διαμοιρασμού συμμετρικού κλειδιού για επικοινωνία. Στην πράξη η πρώτη επικοινωνία και ο διαμοιρασμός του κλειδιού γίνεται με ασύμμετρη κρυπτογραφία, και η συνέχεια της επικοινωνίας γίνεται με συμμετρική (για λόγους αποδοτικότητας, μιας και η ασύμμετρη απαιτεί πολύ μεγάλα μεγέθη κλειδιών).
Οι αλγόριθμοι ασύμμετρης κρυπτογραφίας (ο πιο γνωστός είναι ο RSA) βασίζονται πάνω σε “δύσκολα” αλγοριθμικά προβλήματα. Θεωρούν δηλαδή πως είναι πολύ πιο εύκολο να κατασκευαστεί ένα τέτοιο πρόβλημα του οποίου ξέρουμε τη λύση εμείς καθώς το κατασκευάζουμε, από το να βρει τη λύση κάποιος ο οποίος δεν ήταν παρών στη διαδικασία κατασκευής.

1.6: Ο αλγόριθμος AES

Ο πιο γνωστός αλγόριθμος συμμετρικής κρυπτογράφησης είναι ο αλγόριθμος AES (Advanced Encryption Standard), ο οποίος χρησιμοποιείται πολύ στην παρούσα διπλωματική. Είναι ένας block cipher, ο οποίος επιλέχτηκε από το NIST (Ιντστιτούτο

Προδιαγραφών και Τεχνολογίας των ΗΠΑ) το 2001 ως ο καλύτερος μεταξύ των υποψηφίων για την κρυπτογράφηση δεδομένων [14].
Από τότε έχουν βρεθεί πολύ λίγες επιθέσεις εναντίον του, πράγμα που τον έκανε να καθιερωθεί σε πάρα πολλές χρήσεις τις κρυπτογραφίας. Είναι δε τόσο κοινός, που εδώ και κάποια χρόνια πολλοί επεξεργαστές διαθέτουν ειδικό κύκλωμα που να τον υλοποιεί, ώστε η κρυπτογράφηση/αποκρυπτογράφηση να γίνεται πιο γρήγορα. Αυτός ήταν και ο λόγος που εμείς τον χρησιμοποιήσαμε, μιας και θέλαμε κάποιον ταχύ τρόπο να υλοποιήσουμε ένα σχήμα MAC.
Ο αλγόριθμος AES εργάζεται πάνω σε blocks που έχουν μέγεθος 128 bits (16 bytes), και έχει 3 πιθανά μεγέθη κλειδιών: 128, 192 και 256 bits. Όσο μεγαλύτερο το κλειδί, τόσο μεγαλύτερη και η ασφάλεια βεβαίως, αν και το κλειδί των 128 bits είναι υπεραρκετό για όλες τις κοινές εφαρμογές (εκτός από κατηγορίες TOP SECRET της Υπηρεσίας Πληροφοριών των ΗΠΑ, που φοβάται για κβαντικούς υπολογιστές). Εμείς στην παρούσα διπλωματική θεωρούμε πως τα 128 bits είναι αρκετά.

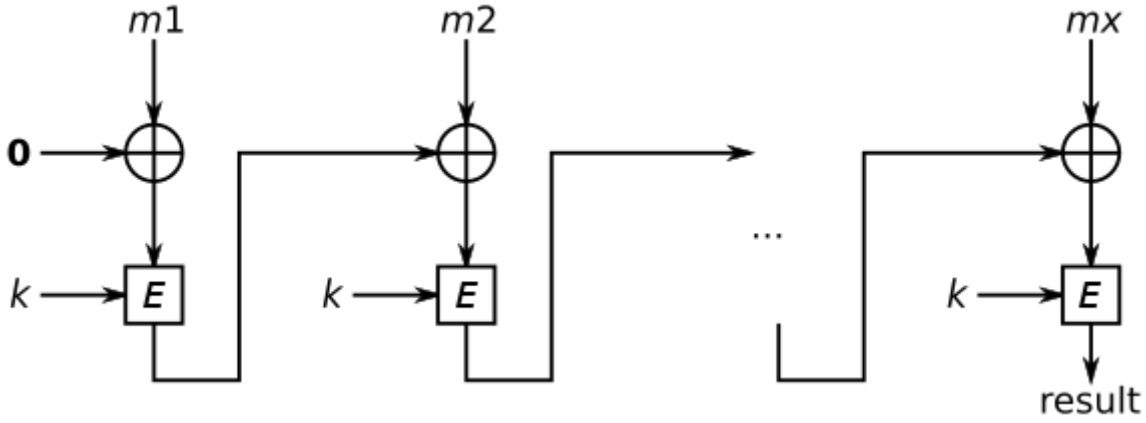


Σχήμα 1-5. Το youtube χρησιμοποιεί AES με κλειδιά 128 bit.

Δεν θα εμβαθύνουμε στην αναλυτική περιγραφή του AES, και απλά θα πούμε ότι χρησιμοποιεί επαναληπτικά μια ακολουθία αντικαταστάσεων (substitutions) – εναλλαγών (permutations). Ο AES με 128-bit κλειδί έχει 10 γύρους αντικαταστάσεων-εναλλαγών, με 192-bit κλειδί έχει 12 γύρους, και με 256-bit κλειδί έχει 14 γύρους.
Εμείς χρησιμοποιήσαμε την υλοποίηση της OpenSSL (libcrypto) για την C, η οποία καλεί το ειδικό hardware σε περίπτωση που υπάρχει στον υπολογιστή.

1.7: Το σχήμα CBC-MAC

Το σχήμα CBC-MAC [15] έχει ως στόχο να παράξει ένα MAC ενός μηνύματος, και το πετυχαίνει με τον ακόλουθο τρόπο:
Δεδομένου ενός αλγορίθμου κρυπτογράφησης τύπου block cipher (ας πούμε AES), ενός κλειδιού k, και ενός μηνύματος M, κατασκευάζει ένα σχήμα κρυπτογράφησης CBC που κρυπτογραφεί το μήνυμα M με τον αλγόριθμο AES με το κλειδί k. Έτσι, το τελευταίο block έχει επηρεαστεί από όλα τα προηγούμενα, και πληροί τις προϋποθέσεις να χρησιμοποιηθεί ως MAC (δεν μπορεί να κατασκευαστεί σωστά χωρίς το κλειδί k, και μια αλλαγή του μηνύματος θα ανιχνευτεί γιατί δεν θα καταλήγει η κρυπτογράφηση στο ίδιο MAC).



Σχήμα 1-6. Η παραγωγή ενός MAC μέσω του σχήματος CBC-MAC.

Υπάρχουν όμως κάποια λεπτά σημεία τα οποία πρέπει κάποιος να προσέξει. Κατά πρώτον, μιας και πρέπει να στέλνουμε μαζί με το μήνυμα και το MAC και το Initialization Vector, σε περίπτωση που επιτρέπουμε το IV να αλλάζει σε κάθε διαφορετικό μήνυμα, ένας επιτιθέμενος μπορεί να εισάγει το δικό του IV και να αλλάξει το πρώτο block του μηνύματος χωρίς να ακυρώσει το MAC. Η λύση σε αυτό (την οποία εφαρμόζουμε κι εμείς), είναι να “καρφώνουμε” το IV σε μια γνωστή τιμή, συνήθως όλα τα bits στο 0.
Κατά δεύτερον, αν έχουμε μηνύματα με μεταβλητό μήκος, υπάρχει η δυνατότητα για έναν επιτιθέμενο ο οποίος γνωρίζει δύο ζεύγη μηνυμάτων-tags (A,a) και (B,b), να παράξει τρίτο μήνυμα C=[A||(B1 XOR a)||B2||...||Bn] που να έχει tag b. Αφού το C είναι διάφορο του B, έχουμε δύο μηνύματα με το ίδιο tag ενώ ο επιτιθέμενος δε γνωρίζει το κλειδί. Η λύση σε αυτό είναι να μην έχουμε μηνύματα όπου το ένα μπορεί να είναι αρχικό υποσύνολο (prefix) του άλλου, και γι'αυτό σε μηνύματα με μεταβλητό μήκος, εισάγουμε στην αρχή (prepend) το μήκος του μηνύματος πριν το κάνουμε MAC.
Κατά τρίτον, αν κάποιος στέλνει μηνύματα και χρησιμοποιεί το ίδιο κλειδί και αλγόριθμο για κρυπτογράφηση του μηνύματος, αλλά και αυθεντικοποίηση/επαλήθευση ακεραιότητας, τότε κάποιος ο οποίος θα αλλάξει όλα τα blocks του μηνύματος πριν το τελευταίο, μπορεί να αλλάξει το plaintext χωρίς να ακυρώσει το MAC (που είναι το τελευταίο block). Η λύση σε αυτό είναι να χρησιμοποιούμε διαφορετικά κλειδιά για κρυπτογράφηση και MACing. Στην παρούσα διπλωματική δεν χρησιμοποιούμε το σχήμα CBC για κρυπτογράφηση αλλά μόνο για παραγωγή MACs, οπότε δεν μας επηρεάζει αυτή η ευπάθεια.

Κάποιος θα μπορούσε να αναρωτηθεί γιατί χρησιμοποιούμε AES με CBC-MAC ώστε να

παράξουμε ένα MAC. Πράγματι, θα μπορούσαμε να χρησιμοποιήσουμε ένα άλλο σχήμα (πχ HMAC, που χρησιμοποιεί κρυπτογραφικές συναρτήσεις hash). Χρειαζόμασταν όμως έναν γρήγορο τρόπο (κατά προτίμηση υλοποιημένο σε hardware), και ο μόνος διαθέσιμος κρυπτογραφικός αλγόριθμος σε hardware ήταν ο AES. Οπότε, χρησιμοποιώντας τις κλήσεις στο hardware που υλοποιούν τον AES, κατασκευάσαμε ένα σχήμα CBC-MAC που μας παράγει το MAC που θέλουμε για ένα δεδομένο μήνυμα.

Κεφάλαιο 2: Μοντέλο επιτιθέμενου και ζητούμενα

2.1: Μοντέλο επιτιθέμενου

Το μοντέλο επιτιθέμενου εναντίον του οποίου προστατεύουμε είναι το εξής: Ο επιτιθέμενος μπορεί, οποιαδήποτε χρονική στιγμή, να αλλάξει με εξωτερικό τρόπο και χωρίς να το καταλάβει η CPU (πχ με ακτινοβολία, με παράνομη χρήση του DMA κλπ) τη μνήμη. Δεν μπορεί να διαβάσει τη μνήμη πριν το κάνει αυτό, αλλά αν γράψει στη μνήμη κώδικα που τη διαβάζει και τρέξει ο κώδικας αυτός, τότε μπορεί να τη διαβάσει. Δεν ασχολούμαστε με επιτιθέμενο που διαβάζει ή γράφει στη μνήμη χρησιμοποιώντας ένα exploit λόγω κακής υλοποίησης της εφαρμογής.
Ο επιτιθέμενος δεν έχει μια υπολογιστική ισχύ που να μπορεί να σπάσει την σύγχρονη κρυπτογραφία με τα καθιερωμένα μεγέθη κλειδιών (όπως για παράδειγμα 128-bit AES key), ούτε μπορεί να λύσει αποδοτικότερα του παγκοσμίως γνωστού NP-Complete προβλήματα πάνω στα οποία βασίζεται η σύγχρονη κρυπτογραφία.
Ως άμυνα εμείς προϋποθέτουμε μια ασφαλή CPU (ο επιτιθέμενος δεν μπορεί να γράψει μέσα σε αυτή, πχ στους registers), για την οποία ζητούμε κάποια παραπάνω λειτουργικότητα. Η λειτουργικότητα αυτή αυξομειώνεται κατά τη διάρκεια του κειμένου (ώστε να εισάγονται νέα στοιχεία που την κάνουν πιο ασφαλή ή πιο αποδοτική). Μια 100% ασφαλής περίπτωση είναι να έχουμε μια CPU η οποία, καθώς διαβάζει από τη μνήμη, ελέγχει (ατομικά, πριν προλάβει να γίνει οτιδήποτε άλλο) ένα Message Authentication Code (MAC) για την εγκυρότητά του, και όταν γράφει στη μνήμη πάλι ατομικά ενημερώνει ένα MAC.
Το πρόγραμμα του οποίου τη μνήμη θα θέλαμε να προστατεύσουμε θα θέλαμε να μην έχει self-modifying κώδικα. Αυτός ο περιορισμός μπορεί να αρθεί πιο μετά, αλλά θα θέλαμε ειδική υποστήριξη από το hardware για να το πετύχουμε.

2.2: Επιπρόσθετα ζητούμενα

Το πιο σημαντικό ζητούμενο είναι να μπορούμε να ελέγξουμε την εγκυρότητα της εκτέλεσης του προγράμματος από μακριά. Για παράδειγμα, αν το πρόγραμμα υπολογίζει έναν αριθμό ο οποίος είναι σημαντικός για εμάς (και δεν θέλουμε να τον αλλάξει ο επιτιθέμενος), θα θέλαμε, κάθε Χ λεπτά, να μπορούμε να τρέξουμε έναν αλγόριθμο για μια μακρόθεν επικύρωση της (ως τώρα) εγκυρότητας την εκτέλεσης, ο οποίος θα είναι αρκετός για να ανιχνεύσουμε αν έχει γίνει εισβολή.
Ο επιτιθέμενος επιτρέπεται να είναι πάνω από τον υπολογιστή και να δοκιμάζει να επιτεθεί με ακτινοβολία στη μνήμη, και εμείς στην άλλη μεριά του κόσμου να προσπαθούμε, επικοινωνώντας με τον υπολογιστή, να το ανιχνεύσουμε. Αν η μνήμη είναι όπως είναι στην τυπική, μη ασφαλή περίπτωση, ο επιτιθέμενος θα μπορούσε να τη γράψει σχεδόν όλη με κώδικα (ή NOPS που οδηγούν σε κώδικα), στον κώδικα αυτόν να ψάξει και να διαβάσει την τιμή που υπολογίζουμε, να την αλλάξει και να μας στείλει την αλλαγμένη. Ο αλγόριθμός για την μακρόθεν επικύρωση (remote attestation) θα πρέπει να ανιχνεύει τέτοιες περιπτώσεις. Αν ανιχνεύσει την εισβολή, εμείς μπορούμε να θεωρήσουμε την τιμή που

λάβαμε ως άκυρη, και να την αγνοήσουμε.
Ένα άλλο ζητούμενο είναι ο ικανοποιητικός χρόνος εκτέλεσης. Συγκρινόμενος με τον χρόνο εκτέλεσης που παίρνει μια ανασφαλής εκτέλεση της εφαρμογής, δεν θα θέλαμε να είναι πολύ μεγαλύτερος. Το ζητούμενο αυτό, χωρίς επιπρόσθετες αλλαγές στο hardware, δεν το πετυχαίνουμε. Αν όμως απαιτήσουμε και μια μικρή μνήμη cache (η οποία θα είναι εντός της CPU, άρα ασφαλής και δεν θα μπορεί να γραφτεί από τον επιτιθέμενο), πετυχαίνουμε σαφώς πιο ικανοποιητικούς χρόνους εκτέλεσης της ασφαλούς εφαρμογής.

2.3: Παρόμοια Βιβλιογραφία

Χωρίς να θέλουμε να ασχοληθούμε σε βάθος με το τι έχει ασχοληθεί η προηγούμενη βιβλιογραφία, θα αναφέρουμε μερικές περιπτώσεις επιγραμματικά.

Όσον αφορά το attestation:
Υπάρχουν λύσεις που να παρέχουν remote attestation σε λογισμικό, στις οποίες αφενός μεν ο εξωτερικός παίκτης που θέλει να επικυρώσει πρέπει να τρέχει το ίδιο πρόγραμμα με τη συσκευή, αφετέρου δε συνήθως βασίζονται σε χρονικούς περιορισμούς (πχ αν η συσκευή αργήσει να απαντήσει έστω και λίγο, θεωρείται πως έχει δεχτεί επίθεση). Το δικό μας μοντέλο δεν απαιτεί από τον verifier να έχει ισχυρή υπολογιστική ισχύ, και είναι πολύ πιο χαλαρό σε χρονικούς περιορισμούς.
Υπάρχουν λύσεις για remote attestation σε hardware, οι οποίες αν και εύκολα υλοποιήσιμες και χρησιμοποιούνται και στην πράξη, συνήθως ζητούν πολλά από το hardware, δηλαδή μπορεί να απαιτούν ένα μεγάλο κομμάτι μνήμης (πτητικής ή μόνιμης) να είναι ασφαλές (να μην μπορεί να πανωγραφεί με ακτινοβολία) ή να μην είναι προσπελάσιμο με κανονικές εντολές assembly. Η δική μας προσέγγιση πλην της αδυναμίας ανάγνωσης της μνήμης από τον επιτιθέμενο (χωρίς να έχει βάλει δικό του κώδικα μέσα), απαιτεί μικρές αλλαγές στο hardware.

Εκτός του attestation, έχουν προταθεί αρκετοί τρόποι για προστασία της μνήμης από διάφορες μορφές επιτιθέμενων.
Μια μορφή είναι ο επιτιθέμενος να γράψει μνήμη που σε κανονική εκτέλεση δεν θα μπορούσε να γράψει, όπως buffer overflows [4]. Σε αυτή την επίθεση γίνεται κακός έλεγχος των ορίων ενός πίνακα, και ο επιτιθέμενος γράφει μετά το τέλος του πίνακα, πανωγράφοντας άλλα κομμάτια μνήμης (όπως return addresses στο stack). Αν πανωγράψει τις σωστές τιμές, μπορεί να αλλάξει την εκτέλεση του προγράμματος ώστε να εκτελέσει δικό του κώδικα. Αυτό ας πούμε μπορεί να γίνει ως εξής: Στον buffer γράφει τον κώδικα που να “σηκώνει” ένα κέλυφος (shell) για εντολές, και πανωγράφει το return address με τη διεύθυνση του buffer, ώστε να εκτελεστεί ο κώδικας.
Για την προστασία αυτών των επιθέσεων έχει προταθεί η χρήση stack canaries [16] [17] (που αν πανωγραφούν και αλλάξει η τιμή τους σηματοδοτούν πως έγινε επίθεση), σχημάτων DEP (Data execution Prevention που δεν επιτρέπουν σε ορισμένα κομμάτια μνήμης να εκτελεστούν ως κώδικας) [18], και ASLR (Address Space Layout Randomization [19], ώστε ο επιτιθέμενος να μην ξέρει πριν την εκτέλεση τις διευθύνσεις

δομών στη μνήμη, μιας και είναι ψευδοτυχαίες).
Αυτά τα σχήματα βάσει εμπειρίας λειτουργούν αρκετά καλά, παρ' όλα αυτά δεν είναι provably secure και έχουν αναπτυχθεί επιθέσεις εναντίον τους.
Η δική μας προσέγγιση πάντως δεν προστατεύει από επιθέσεις που γίνονται λόγω bug στο πρόγραμμα, και λειτουργεί συμπληρωματικά με τις προαναφερθείσες.

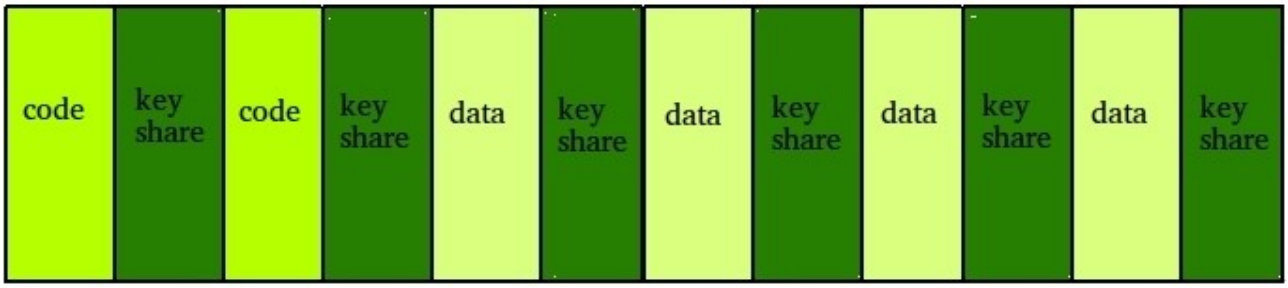
Υπάρχουν και τρόποι προστασίας για το είδος της επίθεσης εναντίον της οποίας αμυνόμαστε.
Για παράδειγμα ένας τρόπος προστασίας για Row Hammer [8] επιθέσεις , όπου αλλάζοντας γειτονικά κελιά (cells) φυσικής μνήμης επηρεάζουμε άλλα διπλανά τους, είναι να χρησιμοποιούμε Μνήμη Διόρθωσης Σφαλμάτων (Error Correcting Memory) [20] ή να προσπαθήσουμε να ανιχνεύσουμε την επίθεση Row Hammer ανανεώνοντας αρκετά συχνότερα το ηλεκτρικό φορτίο των κελιών-θυμάτων. Η Error Correcting Memory είναι μνήμη η οποία έχει ειδικό κύκλωμα για ανίχνευση ή διόρθωση μόνο μικρών αλλαγών (1-2 bits) σε αρκετά μεγαλύτερα κομμάτια μνήμης. Καθίσταται λοιπόν σαφές πως από τη μια με την Error Correcting Memory δεν μπορούμε να πιάσουμε επιτιθέμενο ο οποίος θα πανωγράψει σχετικά μεγάλο κομμάτι μνήμης, και από την άλλη οι τεχνικές συχνότερης ανανέωσης του φορτίου της μνήμης μπορεί με τον καιρό να μην αρκούν για να μην επηρεαστεί ηλεκτρικά ένα κελί από τα γειτονικά του.
Ένας τρόπος προστασίας από ακτινοβολία είναι χρησιμοποιώντας ειδικά υλικά (όπως για διαστημικές εφαρμογές) [21]. Αυτά τα υλικά όμως είναι αρκετά ακριβά, η μνήμη δεν έχει την ίδια απόδοση με τις τυπικές μνήμες της αγοράς, και ακόμα και τότε δεν δίνουν provable security αλλά προστατεύουν μέχρι ένα όριο ακτινοβολίας.

Κεφάλαιο 3: Η ιδέα που υλοποιούμε

3.1: Απλό μοντέλο, μόνο με ένα κλειδί

Σς αυτή την παράγραφο θα περιγράψουμε την απλή έκδοση της ιδέας που προτείνεται στο paper των Richard Lipton, Rafail Ostrovsky, και Βασίλη Ζήκα (http://drops.dagstuhl.de/opus/volltexte/2016/6311/ [1]). Η ιδέα λοιπόν είναι να κόψουμε τη μνήμη της εφαρμογής σε κομμάτια, και ανάμεσά τους να παρεμβάλουμε κομμάτια ενός κλειδιού. Στην απλή περίπτωση που εξετάζουμε, θεωρούμε πως η αλλαγή που θα θελήσει να κάνει ο επιτιθέμενος θα είναι αρκετά μεγάλη ώστε να πανωγράψει έστω ένα κομμάτι του κλειδιού.
Πιο συγκεκριμένα, σπάμε ένα κλειδί σε κομμάτια ως εξής: $K = \oplus_i K_i$
Δηλαδή, τα επιμέρους κλειδιά αν γίνουν XOR μεταξύ τους κατασκευάζουν το κλειδί. Με αυτόν τον τροπο, αν χαθούν (πανωγραφούν) A bits από έστω και ένα κομμάτι κλειδιού (keyshare), τότε έχουν χαθεί και A bits του κλειδιού K. Είναι επιπλέον απλό να κατασκευάσουμε ένα τέτοιο σχήμα. Δεδομένου ενός κλειδιού μεγέθους B bits, και αριθμού N για το πόσα keyshares χρειαζόμαστε, παράγουμε τυχαία N-1 keyshares μεγέθους B bits το καθένα, και το N-οστό keyshare είναι το XOR όλων των προηγουμένων και του αρχικού κλειδιού K.
Επίσης, σπάμε και τη μνήμη της εφαρμογής (και γενικότερα, του συστήματος) σε κομμάτια. Για λόγους ευκολίας ας πούμε πως αφήνουμε μαζί ομάδες των X bytes (πχ θα μπορούσε X=8), και όταν ξεπεράσουμε το X τότε φτιάχνουμε νέο κομμάτι. Ανάμεσα σε δύο κομμάτια βάζουμε ένα προηγουμένως κατασκευασθέν keyshare, και έτσι πλέον έχουμε τη μνήμη:

Σχημα 3-1. Μια αφαιρετική αναπαράσταση της μνήμης μιας εφαρμογής.



Σχήμα 3-2. Η μνήμη της εφαρμογής όταν έχουμε μοιράσει τα keyshares ανάμεσα στα χρήσιμα bytes.

Οι εικόνες αφορούν τη μνήμη μιας εφαρμογής, αλλά η ιδέα μπορεί να εφαρμοστεί γενικώς στη φυσική μνήμη του συστήματος.
Φυσικά ένα πρόγραμμα με τέτοια μνήμη όπως στις εικόνες δεν θα έτρεχε σωστά. Θα θέλαμε να χρησιμοποιεί μόνο τα κανονικά, χρήσιμα bytes της μνήμης και να αγνοεί τα keyshares. Αυτό είτε θα γίνεται με hardware, είτε ο κώδικας του προγράμματος θα πρέπει να αλλάξει ώστε να το κάνει ο ίδιος.
Όσον αφορά τον κώδικα, αυτό μπορεί να γίνει με jmps μετά από κάθε ομάδα χρήσιμων bytes. Πχ αν έχουμε στη μνήμη:

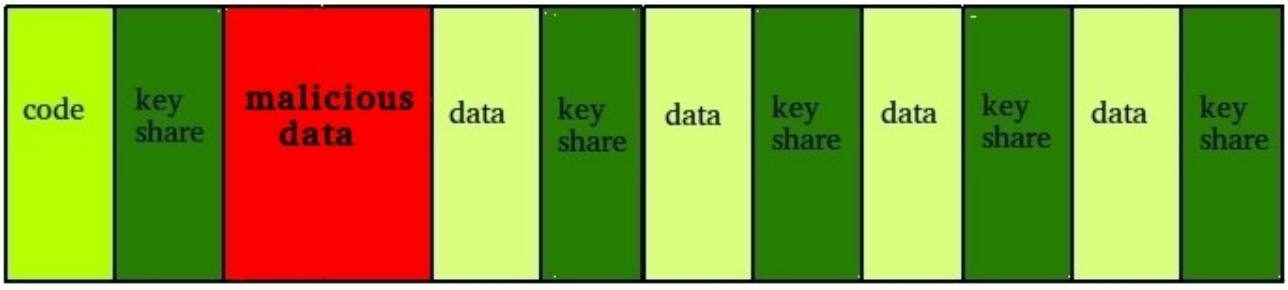
<useful_code_bytes_1>||<keyshare_1>||<useful_code_bytes_2>||<keyshare_2>||....

θα γίνει

<useful_code_bytes_1><jmp_to_useful_2>||<keyshare_1>||
<useful_code_bytes_2><jmp_to_useful_3>||<keyshare_2>||....

Όμοια θα είναι και για τη μνήμη δεδομένων. Είτε την προσεκτική προσπέλαση στο σωστό σημείο θα μας τη δίνει το hardware, είτε το software. Θα μπορούσε, δηλαδή να γίνεται η σωστή μετάφραση διευθύνσεων στη μνήμη την ώρα της εκτέλεσης (hardware) ή, για τη δεύτερη περίπτωση (software), να αλλάζουμε κάθε αναφορά στη μνήμη μέσα στο binary ώστε να γίνεται στο σωστό σημείο, την ώρα του compilation. Αυτό βέβαια δεν είναι εύκολο σε πραγματικά προγράμματα, και απαιτεί και πολύ πιο απλό instruction set (πχ πολλές εντολές x86 πειράζουν ένα συνεχόμενο κομμάτι μνήμης χωρίς ούτε καν να ονομάζονται load ή store). Επίσης δεν πρέπει να έχουμε self-modifying code, μιας και ο κώδικας πρέπει να περιέχει jumps πάνω από τα keyshares.

Προχωράμε τώρα στην επίθεση του αντιπάλου. Εδώ στο απλό μοντέλο θεωρούμε πως ο επιτιθέμενος μπορεί να γράψει μόνο ένα συμπαγές (δηλαδή όχι σπασμένο σε κομμάτια) κομμάτι μνήμης, το οποίο θα είναι και πιο μεγάλο από το μέγεθος ενός κομματιού από χρήσιμα bytes. Αυτό το αίτημα το αίρουμε μετά, και επιτρέπουμε τον αντίπαλο να μπορεί να γράψει οπουδήποτε θέλει, οσοδήποτε μικρή αλλαγή.
Μια δεδομένη χρονική στιγμή λοιπόν, ο επιτιθέμενος εισάγει με εξωτερικό τρόπο τη μνήμη που θέλει στη μνήμη της εφαρμογής. Σχηματικά λοιπόν έχουμε:



Σχήμα 3-3. Η μνήμη στο απλό μοντέλο μετά από επίθεση.

Βλέπουμε λοιπόν ότι ένα κομμάτι των κλειδιών καταστράφηκε με την επίθεση, μιας και ο κακόβουλος κώδικας/δεδομένα τα πανώγραψαν. Αυτή την καταστροφή των κλειδιών θα

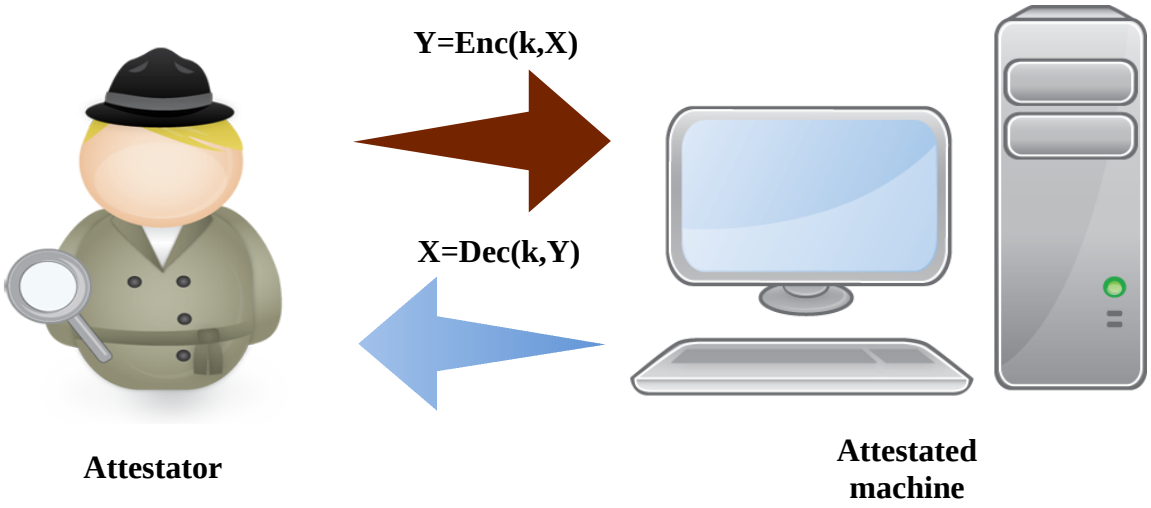
χρησιμοποιήσουμε για να ανιχνεύσουμε την επίθεση.

3.2: Η διαδικασία εξωτερικής επαλήθευσης εγκυρότητας

Η αξία της συγκεκριμένης ιδέας έρχεται με την δυνατότητα που μας παρέχει να ελέγξουμε, ως ένα τρίτος παρατηρητής, αν το σύστημα δέχτηκε επίθεση.
Ο τρίτος παρατηρητής δεν δεσμεύεται από χωρικούς περιορισμούς (μπορεί να είναι στην άλλη μεριά της υφηλίου), και δεν χρειάζεται να έχει ιδιαίτερη υπολογιστική δύναμη - αρκεί να μπορεί να υπολογίζει μερικές κρυπτογραφικές συναρτήσεις (οποιαδήποτε ηλεκτρονική συσκευή μπορεί να το κάνει σε ελάχιστο χρόνο). Ο τρίτος παρατηρητής μπορεί να έχει επίσης χαλαρές απαιτήσεις στο πότε θα λάβει την απάντηση από το σύστημα: ακόμα και μια καθυστέρηση λίγων δεκάδων δευτερολέπτων θα μπορούσε να είναι ανεκτή για κάποιον, δεδομένου ότι ένας επιτιθέμενος δεν θα μπορούσε να σπάσει την κρυπτογραφία σε τόσο χρόνο (το οποίο ισχύει). Φυσικά θα πρέπει να υπάρχει ένα όριο χρόνου στην απάντηση άνω του οποίου θα πρέπει να γίνεται δεκτό πως το σύστημα δέχτηκε επίθεση, μιας και ένας επιτιθέμενος θα μπορούσε να έχει κόψει τις απαντήσεις στο δίκτυο αφού πραγματοποιήσει την επίθεσή του.
Ο τρίτος παρατηρητής (θα τον λέμε και εξωτερικό επαληθευτή εγκυρότητας, ή αλλιώς remote attestator), ανά τακτά χρονικά διαστήματα ρωτά το σύστημα το οποίο τρέχει ώστε να δει αν έχει δεχτεί επίθεση. Έστω πως ο χρόνος μεταξύ ερώτησης και απάντησης είναι t. Αν λοιπόν ο attestator λάβει την απάντηση στον χρόνο T και την επαληθεύσει ως ορθή, ξέρει ότι τουλάχιστον ως τον χρόνο (T-t) το σύστημα έτρεχε χωρίς να δεχτεί επίθεση, και μπορεί να εμπιστευτεί τα αποτελέσματά του. Αν η απάντηση είναι λανθασμένη, ξέρει ότι το σύστημα δέχτηκε επίθεση.
Το κύριο αίτημα που έχουμε για τον attestator είναι να γνωρίζει το συνολικό κλειδί (άθροισμα των keyshares) που βρίσκεται στη μνήμη της εφαρμογής. Αυτό το αίτημα μπορούμε να το αλλάξουμε αν χρησιμοποιήσουμε κρυπτογραφία δημοσίου κλειδιού (θα μιλήσουμε γι αυτό στη συνέχεια), αλλά προς στιγμήν ας δεχτούμε πως το σχήμα αφορά τη συμμετρική κρυπτογραφία.

Ο attestator λοιπόν γνωρίζει το συνολικό κλειδί k, και παράγει μια τυχαία τιμή X. Κρυπτογραφεί το X με το κλειδί k, και στέλνει το Y=Enc(k,X) στο μηχάνημα του οποίου η εκτέλεση πρέπει να επαληθευτεί. Το Y ας του πούμε πρόκληση (challenge). Το μηχάνημα πρέπει να αποκρυπτογραφήσει το Y και να βρει το X. Όταν το βρει, το αποστέλλει πίσω στον attestator και αυτός ελέγχει αν η απάντηση είναι η ίδια την οποία παρήγαγε πιο πριν. Το X ας το πούμε απάντηση (response).
Για να αποκρυπτογραφήσει η εφαρμογή το Y και να βρει το X, πρέπει να κάνει 2 πράγματα:
α) Να συλλέξει το k από τα κομμάτια του τα οποία είναι διεσπαρμένα στη μνήμη.
β) Να ξέρει πώς να απαντήσει στην πρόκληση του attestator.
Γι'αυτό τον λόγο, στο τέλος της μνήμης της εφαρμογής εισάγουμε μια ρουτίνα επαλήθευσης (verification procedure), η οποία περιέχει τον κώδικα ο οποίος συλλέγει το γενικό κλειδί από τα κομμάτια του, αποκρυπτογραφεί την πρόκληση, και στέλνει την απάντηση στον attestator. Φυσικά, και η μνήμη την οποία καταλαμβάνει ο κώδικας της ρουτίνας αυτής, καθώς και οι τοπικές της μεταβλητές, προστατεύονται με keyshares γύρω τους.

Σχηματικά λοιπόν, έχουμε την ακόλουθη διαδικασία:



Σχήμα 3-4. Η διαδικασία του remote attestation.

Κάποιος θα μπορούσε να ισχυριστεί πως θα ήταν καλύτερα το verification procedure να το υλοποιούσαμε σε hardware. Πράγματι, αυτό θα μπορούσε να γίνει, αλλά έχει δύο σημαντικά μειονεκτήματα:
Πρώτον, απαιτεί πιο πολύπλοκο hardware. Γενικώς στην παρούσα διπλωματική προτιμούμε απλούστερο hardware, και εξετάζουμε περιπτώσεις πολυπλοκότητάς του όταν δίνει αρκετά μεγάλη αύξηση της ταχύτητας. Δεδομένου ότι το verification procedure δεν θα τρέχει συνέχεια και έχει πολύ μικρό επιπλέον κόστος σε χρόνο, δεν δίνει κάτι παραπάνω η υλοποίηση σε hardware. Μάλιστα, επειδή σπανίως δουλεύουν όλοι οι επεξεργαστικοί πυρήνες ενός συστήματος, θα μπορούσαμε την ώρα που καλείται η verification procedure να βάζουμε ένα πυρήνα να την εκτελεί, κάτι το οποίο συνήθως δεν θα επηρεάζει καθόλου την εκτέλεση των άλλων εφαρμογών του συστήματος.
Δεύτερον, “κλειδώνει” τους αλγορίθμους κρυπτογράφησης και είναι αδύνατον να αλλαχτούν, σε περίπτωση που βρεθούν ευπάθειες σε αυτούς ή στις υλοποιήσεις τους. Η μόνη περίπτωση να αλλαχτούν είναι μέσω φυσικής αντικατάστασης του επεξεργαστή, πράγμα το οποίο και μεγαλύτερο κόστος έχει και δεν κλιμακώνει στα πολλά συστήματα που μπορεί να έχει μια εταιρία. Αντιθέτως, σε περίπτωση που οι αλγόριθμοι έχουν υλοποιηθεί σε λογισμικό, γίνεται να ενημερωθούν αν εντοπιστούν τρωτότητες.

Σε όλο αυτό το σχήμα υπάρχει ένα σημαντικό σημείο το οποίο πριν το πήραμε ως δεδομένο για ευκολία: Η χρησιμοποίηση συμμετρικής κρυπτογραφίας για το attestation. Αυτό έχει ως μειονέκτημα το ότι μόνο όσοι έχουν το κλειδί μπορούν να κάνουν attestation, και αν κάποιος από αυτούς είναι κακόβουλος μπορεί να επιτεθεί στο σύστημα χωρίς οι άλλοι να τον καταλάβουν: Κάνει man-in-the-middle την κίνηση του δικτύου, και απαντάει στα challenges γνωρίζοντας το κλειδί. Επομένως καθίσταται σαφές πως θα προτιμούσαμε το attestation να γίνεται με κρυπτογραφία δημοσίου κλειδιού, όπου το ιδιωτικό κλειδί βρίσκεται στο μηχάνημο του οποίου η εγκυρότητα πρέπει να επιβεβαιωθεί, και όλοι οι υπόλοιποι επίδοξοι attestators κατέχουν το δημόσιο κλειδί.
Αυτό είναι εφικτό, και μάλιστα κλιμακώνει σε πολλούς χρήστες ενός εταιρικού υπολογιστή,

που θέλουν από διάφορα μέρη του κόσμου να επαληθεύσουν την εγκυρότητα των αποτελεσμάτων που λαμβάνουν. Υπάρχει όμως ένα πρόβλημα, και αυτό είναι το μέγεθος των κλειδιών. Τα κλειδιά της ασύμμετρης κρυπτογραφίας είναι πολύ μεγαλύτερα (για τον RSA θέλουμε κλειδιά τουλάχιστον 2048 bits, σε αντίθεση με τα 128 bits του AES). Τέτοια κλειδιά δεν μπορούν να παρεμβληθούν στη μνήμη ανάμεσα στα χρήσιμα bytes, απλά και μόνο λόγω μεγέθους. Μια λύση είναι να χρησιμοποιήσουμε ασύμμετρη κρυπτογραφία ελλειπτικών καμπυλών, που απαιτεί σαφώς μικρότερα μεγέθη κλειδιών σε σχέση με τον RSA και τους συγγενείς κρυπτογραφικούς αλγορίθμους. Στο σχήμα 3.5 (από [23]), βλέπουμε τα ενδεικνύμενα μεγέθη κλειδιών για τους διάφορους αλγορίθμους.

Symmetric Key Size (bits)	RSA and Diffie-Hellman Key Size (bits)	Elliptic Curve Key Size (bits)
80	1024	160
112	2048	224
128	3072	256
192	7680	384
256	15360	521

Table 1: NIST Recommended Key Sizes

Σχήμα 3-5. Σύγκριση μεγεθών κλειδιών αλγορίθμων κρυπτογράφησης.

Μια καλύτερη λύση όμως, για να διατηρήσουμε το ίδιο μέγεθος κλειδιών, είναι να μην βάζουμε τα κλειδιά στη μνήμη ανάμεσα στα χρήσιμα bytes, αλλά να βάζουμε το random seed που θα παράξει (ντετερμινιστικά) ένα ιδιωτικό κλειδί RSA (το ίδιο κάθε φορά, αφού το seed είναι ίδιο), το οποίο φυσικά το έχουμε σπάσει σε κομμάτια (seed-shares). Το seed αυτό, χωράει να μπει σε 128 bits χωρίς η ασφάλεια να μειωθεί, και η διαδικασία εξωτερικής επαλήθευσης αλλάζει λίγο, ώστε
α) Να μαζεύει το seed από τα κομμάτια του,
β) Να γεννά το ιδιωτικό κλειδί με βάση το seed,
γ) Να απαντά στο challenge αποκρυπτογραφώντας με το ιδιωτικό κλειδί.

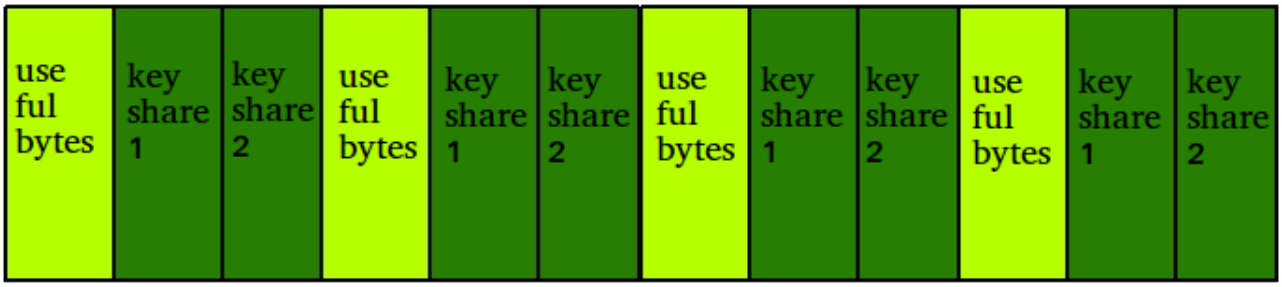
Για λόγους απλότητας στη συνέχεια της παρούσας διπλωματικής, θα θεωρήσουμε πως χρησιμοποιούμε συμμετρική κρυπτογραφία με κοινό κλειδί. Ό,τι πούμε όμως, εφαρμόζεται χωρίς αλλαγές και για την ασύμμετρη κρυπτογραφία.

3.3:Αποφυγή TOCTOU με δύο κλειδιά.

Ενίοτε στην ασφάλεια γίνεται ένα λάθος στις εφαρμογές που πιστοποιούν χρήστες, δικαιώματα κλπ. Ο χρόνος στον οποίο ελέγχουν μια συνθήκη να είναι σύμφωνη με τις επιταγές του διαχειριστή ασφαλείας, δεν είναι ο ίδιος με τον χρόνο στον οποίο

χρησιμοποιούνται τα αποτελέσματα του ελέγχου αυτού. Το λάθος αυτό αναφέρεται στη βιβλιογραφία ως Time-of-check-Time-of-use (TOCTOU) [22].Το μοντέλο της εξωτερικής επαλήθευσης εγκυρότητας υποφέρει από μια τέτοια ευπάθεια.
Έστω λοιπόν πως ένας remote attestator ζητάει να πιστοποιήσει πως η μνήμη του μηχανήματος δεν έχει δεχτεί επίθεση. Στέλνει λοιπόν την πρόκληση Υ και περιμένει να λάβει απάντηση. Το μηχάνημα λαμβάνει την πρόκληση, και ξενικά να τρέχει το attestation procedure (ας πούμε με κάποιο hardware interrupt). Το attestation procedure μαζεύει το κλειδί k, και το αποθηκεύει στην τοπική του μνήμη ώστε να ξεκινήσει την αποκρυπτογράφηση. Βρισκόμαστε λοιπόν στο σημείο που το attestation procedure έχει ένα έγκυρο κλειδί μπροστά του, και είναι έτοιμο να το χρησιμοποιήσει.
Δεν έχουμε τρόπο όμως να δεσμεύσουμε το attestation procedure να χρησιμοποιήσει το κλειδί για την αποκρυπτογράφηση αμέσως όταν το μαζέψει. Μπορεί, εκείνη ακριβώς τη στιγμή, ένας επιτιθέμενος να πανωγράψει τη μνήμη του attestation procedure και να εισάγει δικό του κώδικα, ο οποίος διαβάζει τη μνήμη ψάχνοντας για το (μαζεμένο) κλειδί k. Όταν το βρει, το τυπώνει στην οθόνη και ο επιτιθέμενος το μαθαίνει. Από τη στιγμή που το μαθαίνει, ο επιτιθέμενος ξέρει να απαντήσει όλες τις προκλήσεις του remote attestator.

Η λύση σε αυτό το πρόβλημα είναι να μην υπάρχει χρονική στιγμή, στην οποία ένας επιτιθέμενος μπορεί να αποκτήσει όλη την πληροφορία που χρειάζεται για να απαντά στις προκλήσεις.
Γι αυτό τον λόγο δεν χρησιμοποιούμε ένα κλειδί, αλλά 2 κλειδιά. Ανάμεσα στα χρήσιμα bytes λοιπόν παρεμβάλλουμε κομμάτια 2 κλειδιών, πρώτα του πρώτου και μετά του δεύτερου. Το πώς θα βρίσκονται τα bytes των 2 κλειδιών στη μνήμη δεν έχει τόση μεγάλη σημασία (θα μπορούσαν να βρίσκονται εναλλάξ, key1_byte1, key2_byte1, key1_byte2, key2_byte2 κλπ), αν και εμείς για εποπτικούς λόγους θα βάλουμε όλα τα bytes του πρώτου μαζί, και όλα τα bytes του δεύτερου μαζί, όπως στο ακόλουθο σχήμα.



Σχήμα 3-6. Η μνήμη με 2 κλειδιά ανάμεσα στα χρήσιμα bytes.

Παράλληλα, αλλάζουμε την διαδικασία επαλήθευσης εγκυρότητας ως εξής, δεδομένου ότι έχουμε 2 κλειδιά k1,k2 διασπαρμένα στη μνήμη:
α) Ο attestator στέλνει την πρόκληση Y=Enc(k2,Enc(k1,X)) , δηλαδή κρυπτογραφεί το Χ με το k1 και μετά κρυπτογραφεί δεύτερη φορά με το k2.
β) Το attestation procedure εντός του μηχανήματος τρέχει και συλλέγει το k2 από τα κομμάτια του. Αποκρυπτογραφεί το πρώτο στρώμα της πρόκλησης, και ακολούθως σβήνει το k2. Αυτό μπορεί να το πετύχει γράφοντας μηδενικά στη θέση του.
γ) Το attestation procedure συνεχίζει συλλέγοντας το k1 από τα κομμάτια του. Με το k1

αποκρυπτογραφεί και το δεύτερο στρώμα της πρόκλησης, βρίσκοντας το X το οποίο αποστέλλει στον remote attestator. Πριν το στείλει σβήνει με τον ίδιο τρόπο και το k1.

Έτσι, καμία χρονική στιγμή ο επιτιθέμενος δεν έχει όλη την πληροφορία κάπου στη μνήμη με την οποία να μπορεί να υποδυθεί ένα ανέπαφο μηχάνημα. Στη μνήμη είτε θα βρίσκεται το k1, είτε θα βρίσκεται το k2, αλλά ποτέ και τα δύο μαζί. Μιας και χρειάζεται και τα δύο για να επιτύχει την επίθεση, ο επιτιθέμενος δεν μπορεί να τα καταφέρει.

3.4: Εισαγωγή των MACs για πλήρη ασφάλεια

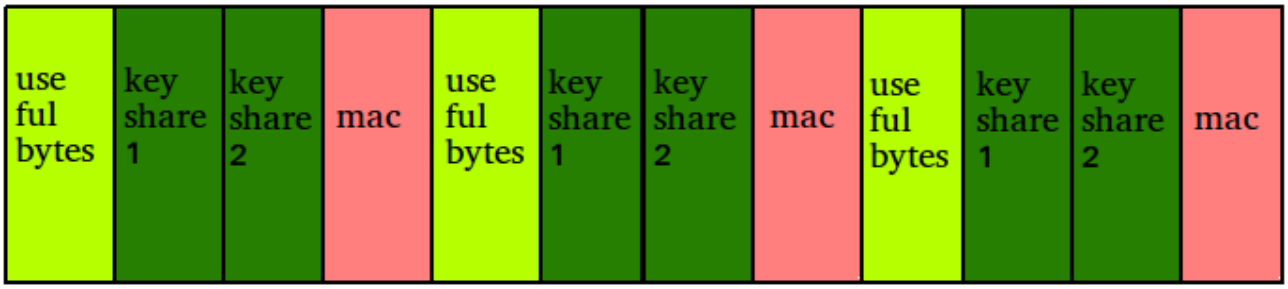
Ως τώρα είχαμε την παραδοχή πως ο επιτιθέμενος δεν μπορεί να γράψει αυθαίρετα μικρό κομμάτι μνήμης, και ακριβώς εκεί που θέλει. Ήρθε η ώρα να άρουμε αυτόν τον περιορισμό, και να εισάγουμε την τελευταία αλλαγή που δίνει στο μοντέλο αποδεδειγμένη ασφάλεια (provable security).
Η τελευταία αλλαγή λοιπόν, είναι να εισάγουμε Message Authentication Codes (MACs) στο τέλος κάθε block από <useful_bytes>||<keyshare1>||<keyshare2> . Αυτά τα MACs θα περιέχουν την τιμή των MAC των προηγούμενων bytes του block, δηλαδή κάθε νέο block θα έχει τη μορφή

<useful_bytes>||<keyshare1>||<keyshare2>||<G>

όπου *G=MAC(<useful_bytes>||<keyshare1>||<keyshare2>).*

Επιπλέον, ζητούμε αλλαγή στο hardware ώστε κάθε προσπέλαση στη μνήμη θα επαληθεύει ατομικά το MAC πριν εκτελεστεί. Δηλαδή, η CPU θα έχει ειδικό κύκλωμα που πριν κάνει READ από τη μνήμη φορτώνει το MAC και το επαληθεύει, και με κάθε WRITE το ενημερώνει. Ομοίως αν θέλει να εκτελέσει ένα κομμάτι μνήμης, είναι το ίδιο σαν να θέλει να το κάνει READ, πρέπει να το επαληθεύσει πριν το εκτελέσει.
Αυτός ο έλεγχος πρέπει να γίνεται ατομικά, δηλαδή δεν πρέπει να υπάρχει η δυνατότητα να επιτεθεί ένας κακόβουλος χρήστης αφού η CPU ελέγξει το MAC αλλά πριν φορτώσει τα bytes από τη μνήμη. Αυτό μπορεί να επιτευχθεί αν δεχτούμε πως η CPU έχει εντός της έναν buffer (ο οποίος αφού είναι μέσα στη CPU δεν απειλείται από τον επιτιθέμενο), και στον buffer αυτόν φορτώνει παράλληλα τη μνήμη η οποία πρέπει να επαληθευτεί (χρήσιμα bytes+κλειδιά+MAC). Μέσα λοιπόν στον buffer αν η πράξη είναι READ ή EXECUTE ελέγχει το MAC, ενώ αν είναι WRITE ενημερώνει το MAC και αμέσως μετά, παράλληλα πανωγράφει τη μνήμη με τις νέες τιμές (οι οποίες βρίσκονται στον buffer) στο σωστό της σημείο, εκεί που βρισκόταν το αντίστοιχο block.
Αν κατά τη διάρκεια της επαλήθευσης (σε περίπτωση READ ή EXECUTE) η επαλήθευση του MAC αποτύχει, τότε η CPU έχει 2 επιλογές:
α) Είτε να σταματήσει την εκτέλεση
β) Είτε να καταστρέψει τα keyshares των δύο κλειδιών του συγκεκριμένου block, άρα να “πιαστεί” στην επόμενη επαλήθευση εγκυρότητας από τον remote attestator.

Συνοψίζοντας λοιπόν, η εικόνα της μνήμης πλέον γίνεται:



Σχήμα 3-7. Η τελική μορφή της μνήμης με τα macs.

Αν λοιπόν έχουμε αυτές τις προδιαγραφές, πετυχαίνουμε provable security στην ανίχνευση εισαγωγής κακόβουλων bytes από έναν επιτιθέμενο.
Σημειώσεις:
α) Ο αλγόριθμος παραγωγής των MACs μπορεί να είναι ανοιχτός και όλοι να ξέρουν πώς να παράξουν ένα αν γνωρίζουν πάνω σε τι να το παράξουν (χρήσιμα bytes+keyshares). Δεν βασιζόμαστε στο να μην μπορεί κάποιος να παράξει ένα MAC λόγω άγνοιας του αλγορίθμου.
β) Όπως βλέπουμε οι απαιτήσεις σε μνήμη αυξάνονται. Ανάλογα με το μέγεθος των χρήσιμων bytes που βάζουμε σε κάθε block, χρειαζόμαστε και αντίστοιχα περισσότερη μνήμη. Μιας και τυπικά μεγέθη για τα κλειδιά είναι 128 bits έκαστο (16 bytes), και ομοίως για το MAC, έχουμε για κάθε block χρήσιμων bytes ένα overhead των 48 bytes (=3*16). Άρα αν και τα χρήσιμα bytes είναι 48 σε κάθε block, τότε χρειαζόμαστε διπλάσια μνήμη. Αν τα χρήσιμα bytes είναι 16 σε κάθε block, τότε χρειαζόμαστε τετραπλάσια μνήμη. Το πλήθος των χρήσιμων bytes ανά block είναι μια παράμετρος που μπορεί να οριστεί αναλόγως την ταχύτητα της CPU και του ποσού της μνήμης που διαθέτει ο καθένας.
Στο τέλος επίσης χρειαζόμαστε και ένα κομμάτι μνήμης όπου θα βρίσκεται ο κώδικας του attestation procedure και οι τοπικές του μεταβλητές, αλλά μιας και θα είναι μικρό σε μέγεθος σε σχέση με τη συνολική μνήμη ενός συστήματος, αυτό το overhead μπορεί να αγνοηθεί.
γ) Ο επιτιθέμενος για να νικήσει το παρόν σχήμα έχει δύο επιλογές: Είτε να αποφασίσει να καταστρέψει μέρος των keyshares είτε να αποφασίσει να τα αφήσει ως έχουν.
Αν αποφασίσει καταστρέψει μέρος των keyshares, τότε μπορεί να πανωγράψει ολόκληρα block (από τα χρήσιμα bytes ως και το MAC). Η CPU δεν θα καταλάβει κάτι (το MAC θα είναι έγκυρο) και ο επιτιθέμενος μπορεί έτσι να πάρει τον έλεγχο του συστήματος, αλλά η επίθεση θα ανιχνευτεί στο επόμενο attestation, αφού θα έχουν χαθεί τα κλειδιά.
Αν αποφασίσει να αφήσει τα keyshares ως έχουν, τότε μπορεί μόνο να πανωγράψει τα χρήσιμα bytes και το MAC. Για να είναι όμως έγκυρο το MAC, πρέπει να γνωρίζει την τιμή των keyshares (αφού το MAC επηρεάζεται από τα keyshares). Αφού δεν μπορεί να διαβάσει πριν πανωγράψει σύμφωνα με τα αρχικά αιτήματα, τότε το μόνο που μπορεί να κάνει είναι να μαντέψει τα keyshares, πράγμα αδύνατον μιας και είναι αρκετά μεγάλα σε μέγεθος (128 bits).
Αν προσπαθήσει να κάνει κάτι ενδιάμεσο, τότε σύμφωνα με το [1] :

i) Αν πανωγράψει περισσότερα από o(logk) bits κλειδιού (όπου k εδώ το μήκος του κλειδιού), τότε ανιχνεύουμε στο επόμενο attestation, μιας και έχουν καταστραφεί τα

κλειδιά.

ii) Αν πανωγράψει λιγότερα, δεν έχει αρκετή πληροφορία για να κατασκευάσει σωστά το MAC, άρα τον πιάνει επιτόπου η CPU. Πιο συγκεκριμένα, το MAC μας έχει την εξής ιδιότητα: Αν ο επιτιθέμενος πανωγράψει τα χρήσιμα bytes, λιγότερα από o(logk) bits του κλειδιού και κάποια (ή όλα τα) bytes από το MAC, στο νέο block που θα προκύψει το MAC δεν θα είναι έγυρο.

Ένα MAC που ικανοποιεί αυτή την ιδιότητα είναι το MAC= w*H(K1)+H(K2), όπου w τα χρήσιμα bytes, H() μια συνάρτηση hash, K1, K2 τα κλεδιά, και οι πράξεις του πολλαπλασιασμού και της πρόσθεσης γίνονται modulo $2^{\mu\acute{\eta}\kappa o\varsigma(w)}$. Περισσότερα γι' αυτό πιο μετά.

δ) Στο όλο αυτό σχήμα υπάρχει μια περίπτωση ο επιτιθέμενος να επιτεθεί και να μην τον καταλάβουμε, αλλά δεν επηρεάζεται τίποτα. Αν πανωγράψει κομμάτι μνήμης (αφήνοντας ανέπαφα τα keyshares, άρα κάνοντας λάθος στο MAC) το οποίο κομμάτι ποτέ δεν προσπελαύνεται, τότε πράγματι δεν θα ανιχνεύσουμε επίθεση. Επειδή η μνήμη αυτή όμως ποτέ δεν προσπελαύνεται, τότε η επίθεση ποτέ δεν εκδηλώνεται και δεν έχει επιπτώσεις.

Κεφάλαιο 4: Δυνατότητες hardware

4.1: Δυνατότητες για τον κώδικα

Ως τώρα μιλούσαμε πάνω στο θεωρητικό μοντέλο της ασφαλούς μνήμης. Τώρα ήρθε η ώρα να αρχίσουμε να συγκεκριμενοποιούμε κάποια πράγματα, και θα αρχίσουμε από τον κώδικα.
Ο κώδικας λοιπόν θα πρέπει να εκτελείται όπως θα εκτελούνταν σε ένα κανονικό πρόγραμμα το οποίο δεν θα ήταν ασφαλισμένο. Αυτό προϋποθέτει δύο πράγματα:
α) Τα κλειδιά και τα MACs δεν θα αντιμετωπίζονται ως κώδικας και δεν θα εκτελούνται.
β) Σε κάθε block κώδικα (<χρήσιμα bytes>||<keyshares>||<MACs>) θα γίνεται επαλήθευση των MACs από το hardware πριν εκτελεστούν τα χρήσιμα bytes.

Το πρώτο μπορεί να υλοποιηθεί σε λογισμικό. Μετά από κάθε block από χρήσιμα bytes μπορεί να τοποθετηθεί ένα jump στο επόμενο, ώστε να μην εκτελούνται τα keyshares και τα MACs. Αυτό βέβαια πετυχαίνει μικρότερη απόδοση από το αν ήταν υλοποιημένο όλο το σχήμα σε hardware, αλλά πάντως λειτουργεί.
Το δεύτερο δεν είναι δυνατόν να υλοποιηθεί σε λογισμικό και να είναι ασφαλές, πρέπει να υπάρχει η κατάλληλη υποστήριξη από το υλικό. Επομένως, όταν ο Intruction Pointer πέφτει πάνω σε ένα κομμάτι χρήσιμων bytes (είτε στην αρχή τους λόγω κανονικής εκτέλεσης, είτε στη μέση λόγω κάποιου jump είτε στη μέση λόγω επιστροφής από κάποιο call), το κομμάτι αυτό πρέπει να επιβεβαιώνεται ως προς την ορθότητά του ελέγχοντας το MAC, με ειδικό κύκλωμα. Αυτό βεβαίως έχει ένα σημαντικό πρόβλημα, και αυτό είναι το από πού μέχρι πού εκτείνεται ένα block χρήσιμων bytes, keyshares και MACs. Στις αρχιτεκτονικές x86 οι εντολές έχουν μεταβλητό μέγεθος, άρα και τα blocks θα έχουν μεταβλητό μέγεθος. Αυτό μπορεί να λυθεί με δύο τρόπους.
α) Κάπου στη μνήμη, να υπάρχει ένας πίνακας ο οποίος να περιγράφει πού αρχίζει το κάθε block. Δεδομένης μίας διεύθυνσης κώδικα, η CPU μπορεί να βρει σε ποιο block ανήκει η διεύθυνση αυτή και να επαληθεύσει το αντίστοιχο MAC. Αυτό είναι μη αποδοτικό για δύο λόγους: Πρώτον, η προσπέλαση στον πίνακα αυτόν θα επιβάλλει την επαλήθευση των MACs για τα στοιχεία του πίνακα (μιας και ο επιτιθέμενος μπορεί να επιτεθεί στον πίνακα, άρα πρέπει να έχουμε keyshares και MACs και εκεί), και δεύτερον η εύρεση του σωστού block είναι μια διαδικασία που μπορεί να κοστίσει κάποιες προσπελάσεις στον πίνακα αυτόν (log(μεγέθους) σε περίπτωση δυαδικής αναζήτησης, και κάπως λιγότερο σε περίπτωση hash lookup).
β) Χρήση σταθερού μεγέθους blocks. Αυτό μπορεί να γίνει είτε με χρήση άλλης αρχιτεκτονικής με τέτοια ιδιότητα (πχ ARM), είτε με τοποθέτηση bytes που να συμπληρώνουν ένα σταθερό μέγεθος block κάθε φορά. Αυτά τα bytes πρέπει να έχουν μια τιμή την οποία το hardware γνωρίζει πως δεν πρέπει να εκτελέσει, αλλά να προχωρήσει στο επόμενο block (για παράδειγμα αν συναντήσει 4 φορές συνεχόμενα την τιμή 0x42 τότε σημαίνει τέλος των χρήσιμων bytes του block). Θα μπορούσαμε να χρησιμοποιήσουμε NOPs (0x90), αλλά τα NOPs εκτελούνται (χωρίς κάποιο αποτέλεσμα βέβαια), κι έτσι έχουμε επιπλέον καθυστέρηση. Εναλλακτικά, σε περίπτωση που χρησιμοποιούμε jmps για να πάμε στο επόμενο block, θα μπορούσαμε να βάλουμε τα συμπληρωματικά bytes μετά τα jmps. Τα συμπληρωματικά bytes βέβαια θα πρέπει να συμπεριλαμβάνονται στον

υπολογισμό των MACs, μιας και θα μπορούσαν να είναι αυτά στόχος της επίθεσης ενός κακόβουλου.
Όταν χρησιμοποιούμε σταθερό μέγεθος blocks για τον κώδικα, ο επεξεργαστής πολύ εύκολα μπορεί να συμπεράνει πού αρχίζει και πού τελειώνει ένα block, προκειμένου να το επαληθεύει όταν χρειάζεται.

4.2: Δυνατότητες για τα δεδομένα

Ομοίως με τον κώδικα, απαιτούμε το hardware να επαληθεύει τα MACs όταν προσπελαύνει τα αντίστοιχα blocks. Βεβαίως εδώ δεν υπάρχει το πρόβλημα του μεταβλητού μεγέθους, μιας και μπορούμε εξαρχής να “καρφώσουμε” τα blocks των δεδομένων να είναι σταθερά. Θέλουμε όμως το hardware να μπορεί να ξεχωρίζει μεταξύ τους τα blocks, και αν μια δομή εκτείνεται σε δύο ή περισσότερα blocks, να μπορεί να φέρει ή να γράψει τα σωστά bytes.
Αυτό μπορεί να το πετύχει μεταφράζοντας τις διευθύνσεις προσπέλασης των δεδομένων την ώρα που αυτές γίνονται γνωστές, και αλλάζοντας ως ένα βαθμό τη λειτουργία του stack. Πιο συγκεκριμένα, αν το πρόγραμμα στην κανονική λειτουργία θέλει να προσπελαύσει τη διεύθυνση X, η CPU θα πρέπει να αλλάξει το X και να το κάνει Y, με τα δεδομένα στο Y να είναι τα ίδια που θα ήταν στο X στην κανονική εκτέλεση, απλώς τώρα έχουμε λάβει υπ'όψη μας πως υπάρχουν και keyshares και MACs.
Με την ίδια λογική πρέπει να αλλάζει και τις διευθύνσεις στο stack, και όταν ας πούμε ο stack pointer μειώνεται για να αυξηθεί το μέγεθος της στοίβας, να τον μειώνει κατάλληλα ώστε να μην γράφει μετά πάνω στα keyshares ή στα MACs.

Αν το hardware δεν χρησιμοποιεί jumps για να μην εκτελέσει τα keyshares και τα MACs στον κώδικα, και χρησιμοποιεί σταθερό (και ίδιο) μέγεθος blocks στον κώδικα και στα δεδομένα, τότε μπορούμε να εισάγουμε και τη δυνατότητα το πρόγραμμά μας να έχει self modifying code, ή ένα κομμάτι των δεδομένων να τίθεται ως εκτελέσιμο και να τρέχει μετά ως κώδικας. Χρειαζόμαστε όμως και τις δύο αυτές προϋποθέσεις, γιατί αλλιώς το ίδιο το πρόγραμμα θα έπρεπε να εισάγει τα jumps μόνο του στην νέο κώδικα που κατασκευάζει, ή να αλλάζει το μέγεθος του block (πράγμα που δεν γίνεται).

4.3: Δυνατότητες caching

Στο σημείο αυτό θα εισάγουμε το αίτημα για μια δυνατότητα caching που θέλουμε να έχει το hardware. Αυτή η δυνατότητα είναι προαιρετική, αλλά χωρίς αυτή η απόδοση είναι αρκετά κακή.
Ζητούμε λοιπόν, η CPU να έχει μέσα της μια περιοχή μνήμης, όπως και στους πραγματικούς επεξεργαστές, η οποία όμως θα είναι ασφαλής από επιθέσεις. Αυτό σημαίνει πως ένας επιτιθέμενος δεν θα μπορεί να γράψει πάνω της μέσω εξωτερικής επίδρασης, άρα μπορεί να θεωρηθεί ασφαλής.
Ο λόγος για τον οποίο κάνουμε ένα τέτοιο αίτημα είναι ότι στις προσομοιώσεις παρατηρήσαμε πως το πρόγραμμα καθυστερεί πολύ, και η καθυστέρηση είναι κυρίως λόγω

της επαλήθευσης των MACs την ώρα της εκτέλεσης. Ας πούμε, όταν είχαμε έναν επαναληπτικό βρόχο, ο μετρητής του βρόχου κάθε φορά αυξάνεται κατά 1, και συγκρίνεται με την τιμή που σηματοδοτεί το τέλος του βρόχου. Σε κάθε βρόχο λοιπόν έπρεπε να διαβάζουμε τον μετρητή (υπολογισμός ενός MAC), τον ξαναγράφουμε αυξημένο κατά 1 (υπολογισμός δεύτερου MAC), και τον συγκρίνουμε με την τιμή που σηματοδοτεί το τέλος του βρόχου (άλλα δύο διαβάσματα στην μη-optimized περίπτωση, άρα 4 MACs συνολικά). Καθίσταται σαφές πως η επιπλέον καθυστέρηση είναι μεγάλη, ακόμα και για την πιο απλή λειτουργία. Σκεφτήκαμε λοιπόν αν θα μπορούσαμε να μην κάνουμε αυτή την επαλήθευση για τα στοιχεία τα οποία επαληθεύουμε συχνά, και μας ήρθε στο μυαλό η ιδέα της cache.
Η cache αυτή λοιπόν θα κρατά μέσα της τα blocks τα οποία έχουν προσπελαστεί πρόσφατα. Αυτά, όταν βρίσκονται στην cache και χρειαστεί να ξαναδιαβαστούν, το MAC τους δεν επαληθεύεται (αφού είναι μέσα στην cache είναι ασφαλή), και περνούν στους registers της CPU ως έχουν. Αν χρειαστεί να γραφτούν, η τιμή τους στην cache αλλάζει χωρίς να ενημερώνεται το MAC. Όταν μόνο χρειαστεί να βγουν από την cache (είτε λόγω αντικατάστασης είτε λόγω cache flushing) το MAC υπολογίζεται και ενημερώνεται στην κύρια μνήμη (σε περίπτωση που αυτά έχουν αλλάξει από τότε που μπήκαν). Η cache δηλαδή θα είναι write-back [25] χρησιμοποιώντας dirty bits.
Ένα σημείο το οποίο δεν προσδιορίσαμε είναι το τι σημαίνει “πρόσφατη προσπέλαση” στην cache. Αυτό δεν απαντάται εύκολα, μιας και υπάρχουν πάρα πολλοί τρόποι προσέγγισης του ποιο είναι το πιο πρόσφατο στοιχείο. Αυτό το αφήνουμε σε αυτόν που θα υλοποιήσει την cache, εμείς πάντως χρησιμοποιήσαμε στην προσομοίωση 2-way assosiative για μεγάλα μεγέθη cache, και direct mapped για πιο μικρά μεγέθη. Ομοίως δεν μιλήσαμε για τα μεγέθη cache, αλλά αυτό που πρέπει να πούμε είναι πως χωρίς ιδιαίτερα μεγάλα μεγέθη (μικρότερα του μεγέθους μιας L1 cache) η καθυστέρηση λόγω των υπολογισμών των MACs σχεδόν εξαφανίζεται. Περισσότερα στοιχεία για την cache θα βρείτε στην παρουσίαση της δικιάς μας υλοποίησης, στο κεφάλαιο 5.
Υπάρχει ένα ακόμα σημαντικό σημείο που αφορά την χρήση της ασφαλούς cache από το hardware. Δεν θέλουμε να υπάρχουν σε δύο σημεία τα keyshares και τα MACs, μιας και ο επιτιθέμενος μπορεί να πανωγράψει το ένα αντίγραφο και με τον κώδικα που θα γράψει εκεί να διαβάσει το άλλο, μη χάνοντας πληροφορία με την επίθεσή του. Γι' αυτό, η CPU πριν προσπελάσει μία θέση μνήμης, ελέγχει αν αυτή υπάρχει ήδη στην cache. Αν υπάρχει, ασχολείται μόνο με το ό,τι υπάρχει στην cache και παραβλέπει το τι υπάρχει στη μνήμη. Αν δεν υπάρχει, το φέρνει στην cache επαληθεύοντας το MAC. Όταν χρειαστεί να επιστρέψει μια θέση της cache στη μνήμη, τότε πανωγράφει ό,τι υπάρχει εκεί με τα δεδομένα της cache (και ενημερώνει το MAC αν χρειάζεται).
Ο επιτιθέμενος αυτό που μπορεί να κάνει είναι να πανωγράψει τη μνήμη. Αν λοιπόν πανωγράψει ένα κομμάτι το οποίο δεν είναι στην cache, τότε ισχύει ό,τι έχουμε περιγράψει πιο πριν, και είτε καταστρέφει keyshares είτε το MAC. Αν πανωγράψει κάτι το οποίο βρίσκεται στην cache, τότε η επίθεση δεν θα εκδηλωθεί ποτέ, μιας και τα δεδομένα του δεν θα ληφθούν υπόψη (η CPU θα ασχοληθεί μόνο με αυτά της cache), και όταν έρθει η στιγμή να γυρίσουν τα δεδομένα από την cache, τότε θα πάρουν τη θέση των δεδομένων του επιτιθέμενου. Επομένως και πάλι, έχοντας αυτή την προϋπόθεση, έχουμε ασφάλεια.

Φυσικά η cache δεν είναι απαραίτητο να περιορίζεται μόνο σε ένα επίπεδο. Οι σύγχρονοι επεξεργαστές χρησιμοποιούν 3 επίπεδα cache: L1, L2 και L3. Από αυτά το ποιο ή το ποια είναι τα ασφαλή το κρίνει ο κατασκευαστής. Αυτό που χρειάζεται οπωσδήποτε είναι

τουλάχιστον το επίπεδο πιο κοντά στον επεξεργαστή (L1 cache) να είναι ασφαλές από επιθέσεις, και τα μη ασφαλή επίπεδα να αντιμετωπίζονται όπως η κύρια μνήμη, ως μη ασφαλή. Στην προσομοίωσή μας για απλότητα χρησιμοποιούμε ένα επίπεδο ασφαλούς cache, μεγέθους μικρότερο από τα τυπικά μεγέθη της L1 cache.

4.4: Δυνατότητες για το attestation procedure

Έχουμε αναφερθεί και πιο πριν σε μια ενδεχόμενη υλοποίηση του attestation procedure σε hardware. Πράγματι, θα μπορούσαμε να έχουμε ένα ειδικό κύκλωμα το οποίο μόλις λαμβάνει το σήμα για attestation να τρέχει και να πραγματοποιεί όλη τη διαδικασία του remote attestation (συλλογή των keyshares, αποκρυπτογράφηση κλπ). Δεδομένου όμως ότι γενικώς προτιμούμε όσο το δυνατόν πιο απλό hardware, και το attetation procedure δεν έχει τόσο μεγάλο κόστος σε χρόνο (εκτελείται σχετικά σπάνια και μπορεί να παραλληλοποιηθεί), δεν υπάρχει μεγάλη διαφορά στο να το έχουμε υλοποιημένο σε λογισμικό στο τέλος της υπόλοιπης μνήμης. Όπως είπαμε και πιο πριν, αυτό μας δίνει τη δυνατότητα να αλλάξουμε και τους αλγόριθμους κρυπτογράφησης σε περίπτωση που βρεθούν ευπάθειες. Άρα, δεν απαιτούμε υλοποίηση του attestation procedure σε hardware.

Κεφάλαιο 5: Η δική μας υλοποίηση

5.1: Γενική προσέγγιση υλοποίησης

Η υλοποίηση της προσομοίωσης της ασφαλούς μηχανής έγινε σε διάφορες γλώσσες προγραμματισμού. Το κύριο κομμάτι έγινε σε γλώσσα C, με κάποιες λειτουργίες χαμηλού επιπέδου σε Assembly. Χρησιμοποιήθηκαν επίσης η Java και η Python για τις δυνατότητες επεξεργασίας κειμένου που διαθέτουν. Όλα τα επιμέρους προγράμματα δέθηκαν μαζί με τη χρήση του κελύφους Bash, το οποίο είναι εξαιρετικό στο να παίρνει εξόδους από το ένα πρόγραμμα και να τις δίνει σε άλλο, αλλάζοντάς τις λίγο σε περίπτωση που χρειάζεται.
Αναλυτικά την υλοποίηση μπορείτε να την διαβάσετε, να την κατεβάσετε και να την τρέξετε από το σύνδεσμο στο github [24], https://github.com/code-injection-detection/gcc-Linux-Implementation . Εκεί θα βρείτε και βήμα-προς-βήμα οδηγίες για το τεχνικό κομμάτι της μεταγλώττισης και εκτέλεσης, κάτι το οποίο δεν αγγίζουμε εδώ.
Η γενική ιδέα λοιπόν είναι η εξής: Προσπαθούμε να προσεγγίσουμε τον χρόνο που θα πάρει μια ασφαλής μηχανή να εκτελέσει ένα υπολογιστικό πρόβλημα, και τον συγκρίνουμε με τον χρόνο τον οποίο παίρνει μια ανασφαλής, καθημερινή μηχανή. Σημαντικό είναι να πούμε ότι το δικό μας πρόγραμμα δεν είναι ασφαλές με την έννοια του provable security, αλλά προσομοιώνει και μετράει τους χρόνους ενός ασφαλούς συστήματος. Η ασφαλής μηχανή την οποία προσπαθούμε να προσεγγίσουμε έχει τα εξής χαρακτηριστικά:
α) Επιβεβαιώνει με κάθε προσπέλαση των δεδομένων το MAC (ζητούμενο και από την θεωρία), εκτός αν το block βρίσκεται στην ασφαλή cache (δοκιμάζουμε και με διάφορα μεγέθη cache και οργάνωσής τους).
β) Δεν περιέχει κώδικα (jumps) που να υπερπηδούν τα keyshares και τα MACs στο κομμάτι του κώδικα της εφαρμογής, αλλά ο επεξεργαστής το κάνει αυτόματα.
γ) Τρέχει x86 αρχιτεκτονική αλλά έχει σταθερό μέγεθος block κώδικα και δεδομένων, πράγμα που σημαίνει πως έχει padded bytes στα blocks του κώδικα προκειμένου να συμπληρώσει το σταθερό αυτό μέγεθος.
δ) Υπολογίζει το MAC πάνω στα χρήσιμα bytes, στα padded bytes αν υπάρχουν, και τα keyshares.

Αυτό είναι το βασικό σχήμα. Στη συνέχεια δοκιμάσαμε και συνδυασμούς με μεταβλητό μέγεθος block (στον κώδικα, όπου είτε χρειάζονται jump στο τέλος κάθε block είτε ένας πίνακας με τα μεγέθη των blocks). Γενικώς, επειδή το μεταβλητό μέγεθος block κάνει τα πράγματα αρκετά δύσκολα τόσο στη θεωρία όσο και στην πράξη, δεν το προτιμούμε (για παράδειγμα, σε περίπτωση προσγείωσης του Instruction Pointer στη μέση ενός block κώδικα αγνώστου μεγέθους, δεν είναι εύκολο για τη CPU να ξέρει άμεσα από πού να αρχίσει να υπολογίζει το MAC).

Η προσέγγισή μας έχει τα εξής τρία μειονεκτήματα:
α) Δεν χρησιμοποιεί αληθινό hardware, πράγμα που μειώνει πάρα πολύ την απόδοση, ακόμα κι αν βελτιστοποιήσουμε όσο μπορούμε τον κώδικα.
β) Στα blocks του κώδικα, για να προσομοιώσει την ασφαλή CPU, εισάγει παραπάνω κώδικα ο οποίος φυσικά μειώνει αισθητά την απόδοση και χειροτερεύει την απόδοση της cache (περισσότερα για αυτό όταν αναλυθεί η οργάνωση του κώδικα εις βάθος).

γ) Για να μπορέσει να μετρήσει το πλήθος των υπολογισμών των MACs στον κώδικα όταν ο IP προσγειώνεται στη μέση ενός block, αναγκάζεται να διασπάσει τα blocks με αποτέλεσμα να έχουμε αρκετά περισσότερα blocks κώδικα από όσα θα είχαμε στην πραγματικότητα, μειώνοντας ακόμα περισσότερο την απόδοση της cache (και για αυτό, περισσότερα όταν αναλυθεί η οργάνωση του κώδικα εις βάθος).
Τα δύο τελευταία μειονεκτήματα, με προσεκτική υλοποίηση καταφέρνουμε ως ένα βαθμό να τα παρακάμψουμε.

Στην προσπάθειά μας να προσεγγίσουμε όσο το δυνατόν καλύτερα τον χρόνο της ασφαλούς CPU μετράμε και προσθέτουμε δύο πράγματα:
α) Αφενός μεν τον χρόνο που θα έπαιρνε το πρόγραμμα αν δεν είχαμε να υπολογίσουμε τα MACs, και απλώς η μνήμη ήταν οργανωμένη με το να έχει keyshares και MACs. Στις προσπελάσεις των δεδομένων όμως δεν κάνουμε κάτι για τα MACs (παραβλέπουμε τον υπολογισμό τους). Φυσικά η σωστή προσέγγιση των δεδομένων (ώστε να μην διαβάζουμε keyshares ή MACs) γίνεται με λογισμικό (getters και setters), και όχι hardware. Αυτός ο χρόνος είναι ο χρόνος κάτω από τον οποίο δεν μπορούμε να πέσουμε στην προσομοίωση, μιας και δεν έχουμε hardware. Αν όλα αυτά ήταν υλοποιημένα σε κύκλωμα ο χρόνος θα ήταν πολύ μικρότερος βεβαίως.
β) Αφετέρου δε τον χρόνο που θα έπαιρνε ο υπολογισμός των MACs, τα οποία κοστίζουν και περισσότερο. Τον χρόνο αυτό τον υπολογίζουμε ως εξής: Αρχικά φτιάχνουμε ένα πίνακα με το πόσο παίρνει να υπολογιστεί ένα MAC δεδομένου ενός μεγέθους (αυτό το βρίσκουμε τρέχοντας αρκετά εκατομμύρια φορές τον αλγόριθμο του υπολογισμού). Έπειτα τρέχουμε το πρόγραμμα και μετρούμε πόσες φορές ζητήθηκε υπολογισμός του MAC, και μετά απλά πολλαπλασιάζουμε τους χρόνους. Αυτό το κάνουμε διότι αν, κατά τη διάρκεια του προγράμματος, τρέχαμε εξαρχής όλο τον κώδικα ο οποίος σώζει την κατάσταση του επεξεργαστή, υπολογίζει το MAC, και επαναφέρει την κατάσταση, ο χρόνος θα ήταν πολύ μεγαλύτερος.

Γενικές αρχές του δικού μας ασφαλούς προγράμματος

Στη συνέχεια θα πούμε τις γενικές αρχές που διέπουν ένα “ασφαλές” πρόγραμμα, όπως το κατασκευάσαμε.
Το πρόγραμμα αυτό λοιπόν έχει κώδικα ο οποίος έχει σπάσει σε blocks τα οποία χωρίζονται από keyshares και MACs. Ομοίως, η υπόλοιπή του μνήμη χωρίζεται σε κομμάτια τα οποία έχουν ανάμεσά τους keyshares και MACs.
Για τον μεν κώδικα, τον αλλάζουμε κατάλληλα πριν την εκτέλεση ώστε να πληροί τις προϋποθέσεις (περισσότερα στην ενότητα 5.2 που εστιάζουμε στον κώδικα).
Για την δε υπόλοιπη μνήμη, πράττουμε τα εξής:
α) Για το data segment (global μεταβλητές), τις δεσμεύουμε στατικά (μαζί με τα keyshares και τα MAC τους) σε ένα struct πριν την εκτέλεση. Περισσότερα στην ενότητα 5.5.
β) Για το heap και το stack, κατά τη διάρκεια της εκτέλεσης και πριν πάρουμε τις μετρήσεις, δεσμεύουμε αρκούντως μεγάλους πίνακες στο κανονικό, “ανασφαλές” heap , και τους χρησιμοποιούμε σαν ασφαλές heap και stack. Βάζουμε μέσα τους keyshares και MACs, και κατά τη διάρκεια της μέτρησης ασχολούμαστε μόνο με αυτά ως τα μόνα υπάρχοντα. Για το μεν stack υλοποιούμε το δικό μας calling convention συναρτήσεων (παρόμοιο με τα

αληθινά), για το δε heap υλοποιούμε τον δικό μας memory manager. Και τα δύο προσπελαύνονται κατάλληλα με ειδικούς getters και setters. Περισσότερες πληροφορίες στις ενότητες 5.3, 5.4 .

Σε λίγο θα μιλήσουμε για τον τρόπο με τον οποίο φτιάξαμε ένα πρόγραμμα σε μια "ασφαλή" έκδοση για να μετρήσουμε τον χρόνο.

Τι υλοποιούμε σε C

Αρχικά, υλοποιούμε τα κομμάτια που κανονικά θα ήταν υλοποιημένα σε hardware. Υλοποιούμε λοιπόν τους ασφαλείς getters και setters, τον υπολογισμό των MACs, καθώς και τις συναρτήσεις που δεσμεύουν και αποδεσμεύουν χώρο στο ασφαλές stack.
Οι ασφαλείς getters και setters είναι συναρτήσεις που προσπελαύνουν ασφαλώς το stack, το heap και το data segment (το οποίο έχει τις global μεταβλητές). Προσέχουν να μην αγγίζουν τα keyshares, και αν τους ζητηθεί ξέρουν να επαληθεύουν και να ενημερώνουν τα MACs όποτε χρειάζεται. Εμείς έχουμε getters και setters για διάφορες δομές και τύπους της γλώσσας C, αρκετούς ώστε να μπορούμε να γράψουμε ένα μεγάλο εύρος προγραμμάτων. Πιο συγκεκριμένα, έχουμε για ακεραίους (ints), μεγάλους ακεραίους (long ints), αριθμούς κινητής υποδιαστολής μονής και διπλής ακρίβειας (floats, doubles), χαρακτήρες (chars), δείκτες (pointers), καθώς και πίνακες για όλα τα ανωτέρω μεγέθη. Ακόμα έχουμε getters και setters για δομές αυθαίρετου μήκους από την ασφαλή μνήμη στη μη ασφαλή, και αντίστροφα.
Οι συναρτήσεις για τον υπολογισμό των MACs καλύπτουν τις εξής κατηγορίες:
α) Αρχικοποίηση και Καταστροφή των δομών που χρησιμοποιούν τα MACs
β) Υπολογισμός των MACs
γ) Διαχείριση της cache
Περισσότερα θα πούμε για τα MACs στην αντίστοιχη ενότητα (5.6) και το μόνο που θα αναφέρουμε εδώ είναι πως για αυτά και μόνο είχαμε την πολυτέλεια να χρησιμοποιήσουμε αληθινό hardware. Πιο συγκεκριμένα, χρησιμοποιήσαμε την επέκταση AES-NI του επεξεργαστή μας που υλοποιούσε τον αλγόριθμο AES σε hardware, και χρησιμοποιήσαμε ένα σχήμα CBC-MAC για την παραγωγή των MACs.
Οι συναρτήσεις που δεσμεύουν και αποδεσμεύουν χώρο στο ασφαλές stack δεν κάνουν κάτι άλλο από το να αυξομειώνουν μια μεταβλητή που χρησιμοποιούμε ως stack pointer. Αυτές οι λειτουργίες θα ήταν κανονικά υλοποιημένες σε hardware, αφού ο stack pointer είναι ένας register.
Αυτές τις τρεις οικογένειες συναρτήσεων (getters και setters, MACs, stack alloc/free) τις βελτιστοποιούμε όσο το δυνατόν περισσότερο (με τη χρήση του flag -O3 στον gcc), προκειμένου να προσεγγίσουμε όσο το δυνατόν περισσότερο το hardware.

Αφού λοιπόν έχουμε έτοιμες τις συναρτήσεις του hardware, καλούμαστε να γράψουμε ένα πρόγραμμα που να τις καλεί. Εδώ θα μπορούσαμε να χρησιμοποιήσουμε έναν compiler που να μετατρέπει ένα οποιοδήποτε πρόγραμμα C σε μια "ασφαλή" έκδοση που να χρησιμοποιεί τους getters και τους setters για οποιαδήποτε προσπέλαση στη μνήμη, κρίναμε πως αυτό είναι περιττό, γιατί δεν χρειαζόμασταν μια γενική λύση, αλλά μια ειδική λύση για τα benchmarks που θα τρέχαμε. Περιγράψαμε λοιπόν περιφραστικά στον κώδικα τη δομή

των συναρτήσεων που θα χρησιμοποιούσαμε, και καλούσαμε με το χέρι τους ασφαλείς getters και setters. Η περιγραφή των δομών έγινε κώδικας C από script της python μετέπειτα. Σχηματικά λοιπόν, μια συνάρτηση των benchmarks έχει την ακόλουθη μορφή.

```
/*************************calc_determinant_sec*************************************/
//PLEASE PYTHON INIT A FUNCTION HERE
NAME_OF_FUNCTION: calc_determinant_sec
RETURN_VALUE_SIZE: long
//^FOR THE ABOVE: none/int/char etc
NUM_OF_PARAMETERS: 2
    chars: 0
    ints: 1 | names: DIM
    longs: 0
    floats: 0
    doubles: 0
    pointers: 1 | names: MATRIX | size_of_pointed_elements: 2500
    arb_pointers: 0
END_OF_PARAMETERS
NUM_OF_LOCAL_VARIABLES: 11
    chars: 0
    ints: 7 | names: ISMINUS,J,P,Q,T,R,S
    longs: 2 | names: D,DET_RESULT
    floats: 0
    doubles: 0
    pointers: 0
    arb_pointers: 2 | names: SUBMATRIX,SUBDETS | size_of_objects:2500,200
    //PYTHON IGNORE: 2500= sizeof(int)*25*25, where 25 is the maximum possible size, and 200=sizeof(long)*25
END_OF_LOCAL_VARIABLES
RETURN_EXPRESSION: GET_STACK_LONG(D)
START_OF_FUNCTION : calc_determinant_sec
```

Σχήμα 5-1: Ο ορισμός του τύπου μιας ασφαλούς συνάρτησης.

Στο σχήμα βλέπουμε πως για να ορίσουμε μια συνάρτηση δίνουμε το όνομά της, τον τύπο που επιστρέφει, τα ορίσματά της και τις τοπικές μεταβλητές. Αυτή η περιγραφή θα μετατραπεί σε κώδικα C από την python, και περισσότερα θα πούμε στην ενότητα 5.4 που αφορά το stack.

Ο κώδικας της συνάρτησης θα μοιάζει ως εξής:

```
    if (GET_STACK_INT(DIM)==2)
    {
        //using get_stack_int_array_element() but if dim==2 at first it needs get_int_array_element(). The functionality is
the same though.
        SET_STACK_LONG(D,get_stack_int_array_element(GET_STACK_PTR(MATRIX),0*25+0)*
                         get_stack_int_array_element(GET_STACK_PTR(MATRIX),1*25+1) -
                         get_stack_int_array_element(GET_STACK_PTR(MATRIX),0*25+1)*
                         get_stack_int_array_element(GET_STACK_PTR(MATRIX),1*25+0));
        RETURN_POINT_OF_FUNCTION: calc_determinant_sec
    }
    else
    {

        for (SET_STACK_INT(J,0);
             GET_STACK_INT(J)<GET_STACK_INT(DIM);
             SET_STACK_INT(J,GET_STACK_INT(J)+1))
        {
            SET_STACK_INT(R,0);
            SET_STACK_INT(S,0);
            for (SET_STACK_INT(P,0);
                 GET_STACK_INT(P)<GET_STACK_INT(DIM);
                 SET_STACK_INT(P,GET_STACK_INT(P)+1))
            {
                for (SET_STACK_INT(Q,0);
                  GET_STACK_INT(Q)<GET_STACK_INT(DIM);
                  SET_STACK_INT(Q,GET_STACK_INT(Q)+1))
                {
                    if (GET_STACK_INT(P)!=0 && GET_STACK_INT(Q)!=GET_STACK_INT(J))
```

Σχήμα 5-2. Ο κώδικας μιας ασφαλούς συνάρτησης.

Ο κώδικας όπως βλέπουμε χρησιμοποιεί getters και setters για τις μεταβλητές. Είναι εμφανές το for loop που εκτελεί για τον μετρητή J από 0 ως DIM-1 χρησιμοποιώντας

αυτούς τους getters και setters, και περισσότερα θα πούμε όταν μιλήσουμε για το stack.
Με ανάλογο τρόπο (περιφραστικά) ορίζουμε και τις global μεταβλητές, οι οποίες θα μπουν σε ένα struct στην αρχή του προγράμματος. Ο ορισμός τους (ο οποίος περιμένει την python να εισάγει τον σωστό κώδικα για να τα αρχικοποιήσει) δίνεται παρακάτω.

```
//GLOBAL DECLARATION
typedef struct
{
//ATTENTION: GLOBAL VARIABLE FOLLOWING! | SIZE:double
double global_double_variable_for_testing;
//ATTENTION: GLOBAL VARIABLE FOLLOWING! | SIZE:int
int test_global;
//ATTENTION: GLOBAL VARIABLE FOLLOWING! | SIZE:int
int secured_i;
//ATTENTION: GLOBAL VARIABLE FOLLOWING! | SIZE:long
long secured_sum;
}global_vars;

global_vars globals = {
//PLEASE PYTHON INITIALISE THE GLOBAL VARS
};
```

Σχήμα 5-3. Ο ορισμός των global μεταβλητών.

Εδώ φαίνεται πως έχουμε ένα struct στο οποίοι δίνουμε τα ονόματα και τους τύπους, και περιμένουμε στο 2ο struct η python να δώσει τιμές (καθώς και να εισάγει keyshares και MACs). Περισσότερα για τις global μεταβλητές στην ενότητα 5.5.

Έχουμε λοιπόν υλοποιημένη την ασφαλή συνάρτηση (η συγκεκριμένη της οποίας ένα απόσπασμα είναι στην εικόνα, υπολογίζει τη ορίζουσα ενός τετραγωνικού πίνακα), και πρέπει να την εκτελέσουμε. Το υπόλοιπο C πρόγραμμα δεν έχει πάρα να αρχικοποιήσει τις κρυπτογραφικές δομές καθώς και την cache αν υπάρχει, να αρχικοποιήσει το ασφαλές stack και heap, να θέσει τις μεταβλητές που λαμβάνουν τις μετρήσεις στο 0 και να τρέξει τη συνάρτηση. Έχοντας λοιπόν δεδομένο τον κώδικα C που τα κάνει όλα αυτά, θα μιλήσουμε πώς κινούμαστε ώστε να παράξουμε ένα ασφαλές εκτελέσιμο.

Διαδικασία Compilation

Το πρώτο βήμα είναι να θέσουμε τις κατάλληλες σταθερές. Ένα πρόγραμμα python λοιπόν αναλαμβάνει να θέσει τιμές σε ένα C header file (.h). Οι τιμές αυτές είναι το πλήθος των κλειδιών, το πλήθος των MACs, το πλήθος των χρήσιμων bytes, καθώς και διάφορες άλλες επιλογές του χρήστη.
Ακολούθως ένα άλλο πρόγραμμα python περνά τον κώδικα και διαβάζει τον ορισμό των ασφαλών συναρτήσεων. Όπου βλέπει ορισμό συνάρτησης, κλήση συνάρτησης ή τέλος συνάρτησης αντικαθιστά με τον σωστό κώδικα C ο οποίος χρησιμοποιεί το ασφαλές stack για τις αντίστοιχες λειτουργίες.
Στη συνέχεια κάνουμε compile με όλα τα optimizations ενεργά τον κώδικα που υλοποιεί τα MACs, την cache, τους ασφαλείς getters και setters και τις συναρτήσεις που δεσμεύουν και αποδεσμεύουν χώρο στο ασφαλές stack (την υλοποίηση του hardware δηλαδή).

Μετά από αυτό σειρά έχει να τρέξει το πρόγραμμα σε python το οποίο αρχικοποιεί τις global μεταβλητές σε ένα stuct, καθώς και εισάγει keyshares και MACs ανάμεσά τους.
Πλέον λοιπόν έχουμε έναν κώδικα C ο οποίος υλοποιεί όλες τις ασφαλές λειτουργίες, και το μόνο που μένει είναι να ασφαλίσουμε τον κώδικα assembly ο οποίος παράγεται από αυτόν τον κώδικα C. Μεταγλωττίζουμε λοιπόν το πρόγραμμα (κάνοντας link και τις υπόλοιπες, optimized συναρτήσεις που υλοποιούν το hardware) από κώδικα C σε assembly. Η μεταγλώττιση αυτή γίνεται με όλα τα optimizations απενεργοποιημένα, μιας και προσπαθούμε να συγκρίνουμε τον χρόνο μεταξύ δύο προγραμμάτων, και όχι απαραίτητα optimized προγραμμάτων. Αν επιβάλαμε optimization σε όλα, τότε θα έπρεπε να έχουμε υλοποιήσει με τον βέλτιστο τρόπο όλες τις δομές μας, το calling convention των συναρτήσεων κλπ. Αυτά όμως εκτός από ανέφικτα για ένα ή ακόμα και λίγα άτομα, δεν οδηγούν απαραίτητα προς τη λύση του προβλήματος: να βρούμε το συγκριτικό χρόνο στον οποίο τρέχουν τα ασφαλή προγράμματα και τα μη ασφαλή. Απενεργοποιούμε λοιπόν όλες τις βελτιστοποιήσεις και στις δύο περιπτώσεις, και συγκρίνουμε εκεί τους χρόνους.
Δεδομένου τώρα του αρχείου assembly, πράττουμε τα εξής:
Χωρίζουμε τον κώδικα σε κομμάτια χρήσιμων bytes, και στο τέλος του καθενός βάζουμε ένα jump στην αρχή του επόμενου. Στον ενδιάμεσο βάζουμε NOPs, τόσα όσα να χωράνε τα keyshares και τα MACs, καθώς και 3-4 bytes για δική μας διευκόλυνση (αυτά βοηθούν πολύ στην υλοποίηση). Αυτά τα ενδιάμεσα bytes δεν εκτελούνται ποτέ γιατί ο κώδικας κάνει jump αποπάνω τους. Στην αρχή κάθε block εισάγουμε κώδικα που να κάνει verify το αντίστοιχο MAC, σώζοντας την κατάσταση του επεξεργαστή και επαναφέροντάς τη μόλις τελειώσει. Τέλος, για να μετρήσουμε σωστά τις κλήσεις των MACs, επιβάλλουμε αλλαγή block όταν συναντάμε label (άρα ο κώδικας μπορεί να κάνει jump εκεί) και κλήση συνάρτησης (όπου ο κώδικας επιστρέφει μετά από το call). Με την επιβολή της αλλαγής του block σιγουρεύουμε πως οποιαδήποτε μη σειριακή εκτέλεση κώδικα θα επιφέρει άμεση επαλήθευση ενός MAC, όπως θα έκανε και στο πραγματικό ασφαλές hardware. Προσέχουμε όμως να μην επαληθεύσουμε το MAC στην περίπτωση που δεν πηγαίνει ο κώδικας στο label μέσω άλματος αλλά συνεχίζει κανονικά η εκτέλεση του block. Επίσης να πούμε πως η διάσπαση των blocks (πέραν των labels και calls) γίνεται με τον τρόπο με τον οποίο θα διασπούσε το ασφαλές hardware, και σε αυτό προσθέτουμε τις ειδικές εμβόλιμες διασπάσεις λόγω labels και calls.
Το επόμενο βήμα είναι να βάλουμε μερικά ακόμα NOPs ανάμεσα στα blocks του κώδικα ώστε να πετύχουμε blocks σταθερού μεγέθους, σε περίπτωση που αυτό ζητήσει ο χρήστης.
Αμέσως μετά μεταγλωττίζουμε σε εκτελέσιμο (πάλι χωρίς optimizations), και διατρέχουμε το εκτελέσιμο ώστε να αντικαταστήσουμε τα keyshares και τα MACs (αυτή τη στιγμή είναι NOPs) με τις σωστές τιμές τους.

Έτσι λοιπόν, μετά από αυτά τα βήματα έχουμε ένα εκτελέσιμο το οποίο
α) Έχει ασφαλή κώδικα
β) Χρησιμοποιεί ασφαλείς global μεταβλητές
γ) Χρησιμοποιεί ασφαλές heap και ασφαλές stack

Άρα προσομοιώνει σωστά ένα πρόγραμμα σε μια ασφαλή CPU, πράγμα το οποίο μας δίνει τη δυνατότητα να λάβουμε μετρήσεις.

5.2: Υλοποίηση του κώδικα

Σε αυτή την ενότητα θα δούμε με αρκετή λεπτομέρεια το πώς υλοποιείται η ασφάλιση του κώδικα.
Ξεκινούμε με ένα αρχείο assembly και θέλουμε να το μετατρέψουμε στην ασφαλή του έκδοση. Το αρχείο assembly θα μοιάζει κάπως έτσι:

```
find_primes_up_to_a_number:
.LFB552:
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    subq    $48, %rsp
    movl    %edi, -36(%rbp)
    movb    $0, -21(%rbp)
    movl    -36(%rbp), %eax
    movl    %eax, %esi
    movl    $.LC126, %edi
    movl    $0, %eax
    call    printf
    movl    -36(%rbp), %eax
    cltq
    salq    $2, %rax
    movl    $123, %edx
    movl    $__func__.14113, %esi
    movq    %rax, %rdi
    call    error_checking_malloc
```

Σχήμα 5-4. Η απλή assembly χωρίς προσθήκες.

Την assembly αυτή την σπάμε σε κομμάτια και στο τέλος καθενός βάζουμε ένα jmp και NOPs. Η ακόλουθη εικόνα δίνεται για κατανόηση, ο αριθμός των εντολών και τον NOPs γενικώς είναι αρκετά μεγαλύτερος. Επίσης για τον ίδιο λόγο δεν χρησιμοποιούμε padded NOPs στην εικόνα.

```
secure_find_primes_up_to_a_number:
.LFB551:
    .cfi_startproc
.START_OF_FUNCTION_secure_find_primes_up_to_a_number:
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    pushq   %r12
 jmp .UNIQUE1991
NOP
NOP
NOP
NOP
NOP
NOP
.UNIQUE1991:
    pushq   %rbx
    subq    $32, %rsp
    .cfi_offset 12, -24
    .cfi_offset 3, -32
    movl    %edi, -36(%rbp)
 jmp .UNIQUE1992
NOP
NOP
NOP
NOP
NOP
```

Σχήμα 5-5. Τα χωρισμένα blocks του κώδικα χωρίς verification procedure και με λίγα NOPs.

Το επόμενο βήμα είναι να εισάγουμε και το verification procedure στην αρχή του κάθε block. Έχουμε λοιπόν

```
NOP
NOP
.UNIQUE3157:
pushfq
call do_verify_code_on_the_fly
popfq
    movl    %edi, -36(%rbp)
    movl    -36(%rbp), %eax
    movl    %eax, %esi
 jmp .UNIQUE3158
NOP
NOP
NOP
```

Σχήμα 5-6. Ένα block κώδικα το οποίο έχει και verification procedure.

Στο σχήμα βλέπουμε πλέον ένα block κώδικα (πάλι, εν γένει θα είναι μεγαλύτερο σε μέγεθος) που στην αρχή του έχει κώδικα ώστε να επιβεβαιώνεται το MAC του την ώρα της εκτέλεσης (οι 3 πρώτες εντολές μετά το label). Αυτό συμβαίνει για κάθε block. Το verification γίνεται στη συνάρτηση do_verify_code_on_the_fly(), και σώζουμε τα flags στην στοίβα πριν την κλήση διότι στην αρχή της εκτελεί μία αφαίρεση (εντολή sub) η οποία

τα χαλάει. Η συνάρτηση αυτή σώζει την κατάσταση του επεξεργαστή (τους registers), λαμβάνει τη διεύθυνση του block από την (ανασφαλή) στοίβα στην οποία μπήκε με την εκτέλεση της call, ελέγχει αν χρειάζεται να επικυρώσουμε το MAC σε περίπτωση που έχουμε cache, επικυρώνει το MAC σε περίπτωση που ο έλεγχος αποτύχει, και τέλος επαναφέρει την κατάσταση του επεξεργαστή στο πρότερο σημείο.
Γίνεται σαφές πως όλα αυτά τα πράγματα παίρνουν αρκετή ώρα και ένας επεξεργαστής που θα τα είχε υλοποιημένα σε hardware δεν θα είχε όλη την επιβάρυνση. Γι αυτό στην πράξη το μόνο που κάνουμε στα benchmarks είναι να μετράμε το πόσες φορές κλήθηκε ο υπολογισμός των MAC και να βρίσκουμε τον χρόνο που παίρνει μόνο ο υπολογισμός αυτός καθεαυτός (βρίσκοντας εξωτερικά τον χρόνο που παίρνει το ένα MAC να υπολογιστεί χωρίς όλη την άλλη διαδικασία).

Ας μιλήσουμε τώρα για την διάσπαση των blocks. Στην ασφαλή CPU, αν ο Instruction Pointer πέσει στη μέση ενός block κώδικα, τότε όλο το block πρέπει να επαληθευτεί ως προς το MAC. Το ίδιο ισχύει αν υπάρχει κλήση συνάρτησης στη μέση ενός block – με την επιστροφή, η εκτέλεση “προσγειώνεται” αμέσως μετά το call, άρα χρειάζεται πάλι επαλήθευση όλου του block.
Ο τρόπος με τον οποίο εμείς επαληθεύουμε ένα block είναι, μιας και δεν έχουμε hardware, να τρέχουμε στην αρχή του κάθε block ειδικό κώδικα τον οποίο τον έχουμε εισάγει εμβόλιμα στην υπόλοιπη assembly. Όταν όμως η εκτέλεση προσγειώνεται στη μέση ενός block εμείς δεν έχουμε τρόπο να τρέξουμε το verification procedure. Αυτό που κάνουμε λοιπόν όταν συναντάμε ένα label (πιθανός στόχος κάποιου jump) ή ένα call είναι να επιβάλλουμε αλλαγή στο block. Το μεν label θα πρέπει να βρεθεί στην αρχή του νέου block, το δε call στο τέλος του παλιού.
Όταν διασπάμε κώδικα λόγω ενός label, ενδεχομένως να μην πέσει ο κώδικας στο label (μετά από κάποιο jump). Οπότε, στην αρχή του νέου block ο κώδικας για το verification εκτελείται όταν ο κώδικας μέσω άλματος βρεθεί στο label, και όχι αν απλά συνεχίζεται η εκτέλεση από το προηγούμενο block (το οποίο θα ήταν το ίδιο στην ασφαλή CPU η οποία δεν διασπά σε labels. Σχηματικά, θα είναι κάπως έτσι, όπου L1 το label που μας επιβάλλει διάσπαση:

```
        <previous block>
        <jmp inserted_label>
        <keyshares>
        <MACs>
L1:
        <MAC verification call>
inserted_label:
        <next block>
```

Τα διασπασμένα blocks είναι και αυτά σταθερού μεγέθους με τη χρήση επιπρόσθετων NOPs, ακόμα κι αν έχουν λιγότερες “χρήσιμες” εντολές.
Έχουμε λοιπόν πως αν συναντήσουμε call έχουμε διάσπαση block (και στο νέο block τρέχει verification στην αρχή), και αν συναντήσουμε label έχουμε διάσπαση block (χωρίς να είναι σίγουρη η κλήση του verification στο νέο block, μιας και η εκτέλεση μπορεί να συνεχίζεται από το προηγούμενο). Εκτός αυτών, υπάρχει και η κανονική διάσπαση block όταν

συμπληρωθεί το μέγιστο μέγεθος σε bytes των εντολών που χωράνε στα χρήσιμα bytes ενός block. Αυτή η κανονική διάσπαση δεν υπολογίζει τα νέα, διασπασμένα blocks που παράγουμε λόγω labels ή calls, και λειτουργεί σαν να μην είχαμε καθόλου διάσπαση λόγω αυτών, όπως θα δούλευε η ασφαλής CPU. Όταν λοιπόν αποφασίσει πως εξαντλήθηκε ένα (μη διασπασμένο από labels ή calls όπως το έχει στο μυαλό του) block, επιβάλλει διάσπαση και MAC verification στο επόμενο block. Με αυτόν τον τρόπο μετράμε με ακρίβεια το πόσα MAC verifications θα τρέξει η ασφαλής CPU, το οποίο είναι και το ζητούμενο.

Εδώ να σημειώσουμε κάτι: Οι δικές μας “ασφαλείς” συναρτήσεις που χρησιμοποιούν το ασφαλές stack δεν κάνουν χρήση της εντολής “call”, αλλά λειτουργούν με jumps και χειροκίνητα pushes-pops στο ασφαλές stack. Η επιβολή της αλλαγής του block για τα calls γίνεται για όλες τις άλλες συναρτήσεις που μπορεί να χρησιμοποιούμε (πχ printf). Αυτές οι συναρτήσεις υλοποιούνται στη libc και τις θεωρούμε “ασφαλείς”, αλλά επειδή στην πράξη δεν είναι δεν κάνουμε μετρήσεις με αυτές, μη συμπεριλαμβάνοντάς τες . Η διάσπαση των blocks επίσης δεν συμβαίνει όταν καλούμε τις συναρτήσεις που υλοποιούν το hardware, μιας και αυτές δεν θα υπήρχαν καν στον κώδικα κανονικά.

Η επόμενη κίνηση είναι μεταγλωττίσουμε τον κώδικα assembly σε εκτελέσιμο και να αλλάξουμε τα NOPs σε keyshares και MACs. Αφού το κάνουμε αυτό διατρέχουμε όλο το εκτελέσιμο και όπου βλέπουμε jump με τον σωστό αριθμό NOPs μετά από αυτό, αντικαθιστούμε:
α) Τα bytes που προσθέτουμε και βοηθούν στην υλοποίηση (canaries για εντοπισμό των jumps πάνω από keyshares + MACs, μετρητές κλπ)
β) Τα keyshares
γ) Τα MACs
Επίσης θέτουμε, σε περίπτωση μεταβλητού μεγέθους block, το μέγεθος του block σε μια global μεταβλητή πριν κληθεί η συνάρτηση που επιβεβαιώνει το MAC για τον κώδικα (ώστε να ξέρει τι μέγεθος θα επιβεβαιώσει). Σε περίπτωση σταθερού μεγέθους block δεν συμβαίνει κάτι τέτοιο.

5.3: Υλοποίηση heap

Το heap προσομοιώνεται δεσμεύοντας έναν μεγάλο πίνακα, και γεμίζοντάς τον με keyshares και MACs πριν αρχίσουμε να τον χρησιμοποιούμε.
Αφού τον αρχικοποιήσουμε έτσι τον χρησιμοποιούμε όπως θα χρησιμοποιούσαμε το heap, με ειδικές, “ασφαλείς” malloc()/free() και ειδικούς getters/setters.
Τις secure_malloc() και secure_free() τις υλοποιούμε με ένα δικό μας memory manager, ο οποίος διατηρεί δύο λίστες:
α) Μία διπλά συνδεδεμένη λίστα που κρατάει τα κομμάτια του ελεύθερου χώρου (πρώτη λίστα).
β) Μία διπλά συνδεδεμένη λίστα που κρατάει τα κομμάτια του δεσμευμένου χώρου (δεύτερη λίστα).
Όταν αρχικοποιείται ο memory manager η πρώτη λίστα έχει ένα στοιχείο, όλο το heap που

είναι άδειο, και η δεύτερη είναι κενή. Μόλις ο χρήστης δεσμεύσει χώρο με την secure_malloc(), τότε δημιουργείται ένας κόμβος στη λίστα με τα κομμάτια του δεσμευμένου χώρου. Παράλληλα, ο ένας κόμβος που έχει η λίστα με τους ελεύθερους κόμβους ενημερώνεται ώστε να έχει μικρότερο μέγεθος και διαφορετικά αρχή, μιας και το αρχικό του κομμάτι δεσμεύτηκε. Αν ο χρήστης δεσμεύσει άλλες δύο φορές μνήμη, η δεύτερη λίστα των δεσμευμένων κομματιών έχει πια 3 κόμβους, και η πρώτη λίστα με τα ελεύθερα κομμάτια έναν, με ακόμα μικρότερο μέγεθος και διαφορετική αρχή. Αν ο χρήστης τώρα αποδεσμεύσει τη δεύτερη μνήμη που δέσμευσε με τη secure_free(), η πρώτη λίστα αποκτά δύο κόμβους, ο ένας με το δεύτερο, αποδεσμευμένο κομμάτι και ο άλλος με τον υπόλοιπο αδέσμευτο χώρο. Η δεύτερη λίστα χάνει έναν κόμβο και μένει με δύο. Τέλος, αν ο χρήστης αποδεσμεύσει και την τρίτη μνήμη που δέσμευσε, η δεύτερη λίστα επιστρέφει στην κατάσταση με τον ένα (δεσμευμένο) κόμβο και η πρώτη λίστα των ελεύθερων κόμβων έχει τα κομμάτια της να συνενώνονται (αφού είναι συνεχόμενα) σε ένα μεγάλο κομμάτι αδέσμευτης μνήμης.
Όταν δεσμεύεται ένα κομμάτι μνήμης, στον αιτούντα επιστρέφονται ακέραιος αριθμός blocks, τόσα ώστε να ικανοποιείται το αίτημα. Αν το τελευταίο block δεν ζητηθεί ολόκληρο εμάς δεν μας πειράζει, το παραχωρούμε διότι θα έπρεπε να κρατάμε πληροφορία για δεσμευμένη μνήμη χωρισμένη ακόμα και στη μέση ενός block.

Το ασφαλές heap προσπελαύνεται με ειδικούς getters και setters. Αυτοί όπως έχουμε πει λειτουργούν για χαρακτήρες (chars), ακεραίους (ints), μεγάλους ακεραίους (long ints), αριθμούς κινητής υποδιαστολής μονής και διπλής ακρίβειας (floats, doubles), δείκτες (pointers) και για πίνακες με όλους τους ανωτέρω τύπους. Ακόμα, λειτουργούν για δομές αυθαίρετου μεγέθους, όπου τις μεταφέρουν από την ανασφαλή μνήμη στην ασφαλή, και αντίστροφα. Αυτό είναι ένα υποσύνολο όλων των τύπων της C, το οποίο όμως μας επιτρέπει να γράψουμε προγράμματα για ένα μεγάλο μέρος προβλημάτων.
Όταν ένας getter ή setter γράφει έναν τύπο δεδομένων, θεωρεί δεδομένο πως ο τύπος αυτός ξεκινάει στην αρχή ενός block. Εξαίρεση αποτελούν οι πίνακες, των οποίων τα στοιχεία είναι το ένα μετά το άλλο εντός των blocks, πράγμα που σημαίνει πως ενδεχομένως κάποια από αυτά να είναι διασπασμένα σε δύο blocks. Δηλαδή αν ένα block έχει 6 bytes για χρήσιμα δεδομένα και θέλουμε να γράψουμε δύο ακέραιους (4 bytes έκαστος), θα δεσμεύσουμε δύο blocks και θα τους γράψουμε στις αρχές των block. Αν όμως θέλουμε να γράψουμε έναν πίνακα 3 ακεραίων θα δεσμεύσουμε πάλι δύο blocks, και οι 3 ακέραιοι θα χωρέσουν σε αυτά τα δύο, με τον δεύτερο να έχει τα 2 πρώτα byte του στο πρώτο και τα 2 τελευταία στο δεύτερο block.

Σχηματικά, οι δύο ακέραιοι:

<int1_byte1><int1_byte2><int1_byte3><int1_byte4><unused_byte><unused_byte>||
<keyshare1_block1><keyshare2_block1>||<mac_block1>||
<int2_byte1><int2_byte2><int2_byte3><int2_byte4><unused_byte><unused_byte>||
<keyshare1_block2><keyshare2_block2>||<mac_block2>

Ενώ ο πίνακας 2 ακεραίων:

<int1_byte1><int1_byte2><int1_byte3><int1_byte4><int2_byte1><int2_byte2>||

<keyshare1_block1><keyshare2_block1>||<mac_block1>||
<int2_byte3><int2_byte4><int3_byte1><int3_byte2><int3_byte3><int3_byte4>||
<keyshare1_block2><keyshare2_block2>||<mac_block2>

Αξίζει να δούμε το πώς περίπου υλοποιείται μια τέτοια συνάρτηση. Για λόγους εποπτείας, παρουσιάζεται ένας setter σε μια θέση μνήμης η οποία βρίσκεται στην αρχή ενός block στο heap.

```
long insert_data_into_mem(long data_size,unsigned char * data, unsigned char * mem_where_to_insert)
{
 long i,j;
 unsigned char * p;
 long chunks=0;
 long total_data_inserted=0;
 long data_remaining;

 p=&mem_where_to_insert[0]; //p points to where we start writing
 i=0;

 while(total_data_inserted<data_size)
 {
        chunks++;
        //find how much data we still have to write
        data_remaining=data_size-total_data_inserted;
        //actual insertion if we have less data to insert than the useful bytes in one block
        //bytes_for_useful_data = size of useful data per block
        if (data_remaining<=bytes_for_useful_data)
        {
                memcpy(&p[i],&data[total_data_inserted],data_remaining);
                total_data_inserted=data_size;
                //update mac
                if (!ignore_macs_even_if_there_are_mac_bytes)
                {
                        update_mac_when_setting_data(&p[i],
                                                     bytes_for_useful_data+bytes_used_for_keyshares,
                                                      bytes_for_useful_data,
                                                     &p[i]+bytes_for_useful_data+bytes_used_for_keyshares);
                }
        }
        else  //if we have more data to insert than the useful bytes in one block
        {
                //write as many as allowed, i.e. <bytes_for_useful_data>
                memcpy(&p[i],&data[total_data_inserted],bytes_for_useful_data);
                total_data_inserted+=bytes_for_useful_data;
                //update mac
                if (!ignore_macs_even_if_there_are_mac_bytes)
                {
                        update_mac_when_setting_data(&p[i],
                                                     bytes_for_useful_data+bytes_used_for_keyshares,
                                                     bytes_for_useful_data,
                                                     &p[i]+bytes_for_useful_data+bytes_used_for_keyshares);
                }
        }
        //jump over to reach the start of the next block
        i+=bytes_for_useful_data+bytes_used_for_keyshares+number_of_mac_bytes;
 }
 return chunks;
}
```

και για εγγραφή ακεραίου:

```
void set_int( void * start_of_secure_data,int source)
{
  insert_data_into_mem(sizeof(int),(unsigned char *)&source,(unsigned char *)start_of_secure_data);
}
```

Η λογική είναι πολύ απλή. Η set_int() λαμβάνει τη διεύθυνση στο ασφαλές heap στην οποία πρέπει να γραφεί ένας ακέραιος, και τον ακέραιο αυτόν. Καλεί την insert_data_into_mem() με τις κατάλληλες παραμέτρους, η οποία όσο δεν έχουν τελειώσει τα δεδομένα που πρέπει να γράψει, περνά από το ένα block στο άλλο και γράφει όσα της επιτρέπει το μέγεθος των χρήσιμων bytes, ενημερώνοντας το MAC. Στο τέλος κάθε block, υπερπηδά τα κλειδιά και το MAC.

5.4: Υλοποίηση stack και συναρτήσεων

Προχωράμε τώρα στο πώς υλοποιείται το stack και το πώς καλούνται οι συναρτήσεις.
Το ασφαλές stack δεσμεύεται και αρχικοποιείται όπως και το heap, δηλαδή είναι ένας μεγάλος πίνακας στον οποίο εισάγουμε κλειδιά και MACs. Ο “stack pointer” δεν είναι τίποτε άλλο από μια global μεταβλητή που παίζει το ρόλο του %rsp register, ο οποίος δείχνει στο τέλος του stack. Εδώ η διαχείριση μνήμης δεν χρειάζεται ειδικό manager, αλλά έχουμε μία συνάρτηση η οποία δεσμεύει και μία που αποδεσμεύει μνήμη από τη ασφαλή στοίβα.
Ομοίως επίσης με το ασφαλές heap, το ασφαλές stack μπορούμε να το προσπελάσουμε με ειδικούς getters και setters, ίδιων τύπων με το heap. Εκτός αυτών δεν υπάρχει κάτι άλλο αξιόλογο να πούμε για την προσπέλαση του ασφαλούς stack.
Το σημείο με ιδιαίτερο ενδιαφέρον είναι η υλοποίηση της κλήσης συναρτήσεων. Για να το πετύχουμε πρέπει να προσομοιώσουμε έναν κανόνα κλήσης συναρτήσεων (calling convention) που είναι παρόμοιος με τους κοινούς [26], τουλάχιστον σε απόδοση. Αυτός ο κανόνας συνοψίζεται στα εξής:
α) Η στοίβα μεγαλώνει από τα αριστερά προς τα δεξιά, αν δεχτούμε πως ο πίνακας που δεσμεύσαμε είναι οριζόντιος.
β) Κάθε function frame αποτελείται από τα εξής:

i) Τα ορίσματα της συνάρτησης, από το πρώτο ως το n-οστό (άρα οι συναρτήσεις δεν μπορούν να έχουν μεταβλητό πλήθος παραμέτρων)

ii) Χώρο για την τιμή επιστροφής (return value), αν υπάρχει

iii) Χώρο για τη διεύθυνση επιστροφής (return address)

iv) Χώρο για τον δείκτη βάσης (base pointer) της προηγούμενου function frame, ο οποίος ουσιαστικά είναι ένας δείκτης (τον προσομοιώνουμε με μια άλλη global μεταβλητή) ο οποίος λειτουργεί ως σημείο αναφοράς για την προσπέλαση των δεδομένων ενός function frame. Όλα τε δεδομένα γράφονται και προσπελαύνονται με τις διευθύνσεις τους να αποτελούν ένα offset από αυτόν τον δείκτη.

v) Τις τοπικές μεταβλητές της συνάρτησης.

Κάθε μια από αυτές τις κατηγορίες δεσμεύει χώρο ξεχωριστά στο ασφαλές stack (δηλαδή σε ξεχωριστά blocks). Οι παράμετροι και οι τοπικές μεταβλητές δεσμεύουν ξεχωριστά χώρο για κάθε ένα τύπο μεταβλητών τους (ακέραιοι, χαρακτήρες κλπ), και μόνο όταν υπάρχουν πολλές μεταβλητές ίδιου τύπου τις βάζουν σε ένα πίνακα. Δηλαδή αν μια συνάρτηση έχει τοπικές μεταβλητές ένα χαρακτήρα (char), έναν μεγάλο ακέραιο (long int) και τρεις ακεραίους (ints), τότε θα δεσμεύσει ένα block για τον χαρακτήρα, όσα blocks χρειάζονται για να χωρέσουν τον μεγάλο ακέραιο (τυπικά ένα), και όσα blocks χρειάζονται για να χωρέσουν έναν πίνακα με 3 ακεραίους.

Οι κλήσεις των συναρτήσεων, επειδή χρησιμοποιούμε δικιά μας σύμβαση, δεν χρησιμοποιούν την εντόλη call της assembly (η οποία δεν ξέρει να χρησιμοποιήσει το ασφαλές stack). Επομένως την προσομοιώνουμε, βάζοντας χειροκίνητα την διεύθυνση επιστροφής (return address) στο ασφαλές stack, και πηγαίνοντας (με goto) στη διεύθυνση της συνάρτησης. Ομοίως στην επιστροφή, με goto πηγαίνουμε στη διεύθυνση επιστροφής. Για να δουλέψει το goto, πρέπει τα άλματα να είναι εντός της ίδιας συνάρτησης C. Επομένως γράφουμε όλες τις ασφαλείς συναρτήσεις μας μέσα σε μία μεγάλη συνάρτηση (συγκεκριμένα great_function_that_wraps_the_tests()), όπως και τον κώδικα τον οποίο τις καλεί. Το ενδιαφέρον είναι πως ακόμα και μέσα στη συνάρτηση μπορούμε να γράψουμε κώδικα που να την καλεί (αναδρομικά), όπως και στις κοινές συναρτήσεις.
Όπως είπαμε και πιο πάνω, η περιγραφή και η κλήση μιας συνάρτησης γίνεται περιφραστικά, και δεν χρησιμοποιούμε τον κώδικα της C. Ένα πρόγραμμα python διαβάζει την περιγραφή μας και παράγει κώδικα ο οποίος να χρησιμοποιεί το ασφαλές stack. Όπως πριν παραθέσαμε την περιγραφή της συνάρτησης υπολογισμού ορίζουσας, τώρα θα παραθέσουμε την περιγραφή της συνάρτησης υπολογισμού πρώτων αριθμών.

```
/*************************find_primes_up_to_a_number_sec*************************************/
//PLEASE PYTHON INIT A FUNCTION HERE
NAME_OF_FUNCTION: find_primes_up_to_a_number_sec
RETURN_VALUE_SIZE: none
//^FOR THE ABOVE: none/int/char etc
NUM_OF_PARAMETERS: 1
    chars: 0
    ints: 1 | names: NUM
    longs: 0
    floats: 0
    doubles: 0
    pointers: 0
    arb_pointers: 0
END_OF_PARAMETERS
NUM_OF_LOCAL_VARIABLES: 5
    chars: 1 | names: BOOL
    ints: 3 | names: I,J,NUM_OF_PRIMES_FOUND
    longs: 0
    floats: 0
    doubles: 0
    pointers: 1 | names: PRIMES_FOUND_SO_FAR | size_of_pointed_elements: 4  //sizeof(int)
    arb_pointers: 0
END_OF_LOCAL_VARIABLES
RETURN_EXPRESSION: NULL
START_OF_FUNCTION : find_primes_up_to_a_number_sec
```

Σχήμα 5-7: Η περιγραφή της ασφαλούς συνάρτησης υπολογισμού πρώτων αριθμών.

Η συνάρτηση αυτή παίρνει έναν ακέραιο ως όρισμα (τον αριθμό μέχρι τον οποίο να βρει τους πρώτους), δεν επιστρέφει κάτι, και χρησιμοποιεί ως τοπικές μεταβλητές έναν χαρακτήρα, 3 ακεραίους και έναν δείκτη σε ακεραίους. Βλέπουμε πως οι μεταβλητές αυτές έχουν και κάποιο όνομα. Το όνομά τους έπειτα ορίζεται ως ένα C macro το οποίο περιέχει το σωστό offset από τον base pointer, ώστε όταν το περνάμε στον κατάλληλη getter ή setter συνάρτηση, αυτή να παίρνει τη σωστή διεύθυνση.

Ιδού λοιπόν ένα κομμάτι της υλοποίησης της συνάρτησης υπολογισμού πρώτων αριθμών, χρησιμοποιώντας ασφαλείς getters και setters. Στην υλοποίηση βλέπουμε στην αρχή να δεσμεύουμε ένα πίνακα για την αποθήκευση των πρώτων, ακολούθως να εισάγουμε στην αρχή το 2, και για κάθε επόμενο περιττό αριθμό να ελέγχουμε διαιρέτες ως το μισό του. Η υλοποίηση δεν είναι η πιο αποδοτική που υπάρχει, αλλά μιας και συγκρίνουμε με την ίδια υλοποίηση υλοποιημένη σε ανασφαλή συνάρτηση, η μέτρηση είναι έγκυρη.

```
//allocate space
SET_STACK_PTR(PRIMES_FOUND_SO_FAR, error_checking_managed_secure_malloc(
                                                                    sizeof(int)*GET_STACK_INT(NUM)
                                                                    ,__func__,__LINE__));
printf("Allocated space\n");

//num_of_primes=1; //2 is a prime
SET_STACK_INT(NUM_OF_PRIMES_FOUND,1);
//primes_found_so_far[num_of_primes-1]=2;
set_int_array_element(GET_STACK_PTR(PRIMES_FOUND_SO_FAR),
                      GET_STACK_INT(NUM_OF_PRIMES_FOUND)-1,
                      2);
printf("Set 2 as first prime\n");

//for (i=3;i<=num;i+=2)
for (SET_STACK_INT(I,3);
     GET_STACK_INT(I)<= GET_STACK_INT(NUM);
     SET_STACK_INT(I,GET_STACK_INT(I)+2))
     {
         //bool=0;
         SET_STACK_CHAR(BOOL,0);

         //for (j=2;j<=i/2+1;j++) //not using sqrt here
         for (SET_STACK_INT(J,2);
              GET_STACK_INT(J)<= GET_STACK_INT(I)/2+1;
              SET_STACK_INT(J,GET_STACK_INT(J)+1))
         {

            //if (i%j==0)
            if (GET_STACK_INT(I)%GET_STACK_INT(J) ==0 )
                {
                    //bool=1;
```

Σχήμα 5-8. Κομμάτι του κώδικα της συνάρτησης υπολογισμού πρώτων αριθμών.

Οι αντίστοιχες εντολές “ανασφαλούς” κώδικα φαίνονται στα σχόλια.

Ο τρόπος με τον οποίο καλούμε την ασφαλή συνάρτηση φαίνεται στο ακόλουθο σχήμα.

```
//primes
_securestart=clock();
count_mac_invocations_in_this_code_part=1;
//HEY PYTHON CALLING FUNCTION : find_primes_up_to_a_number_sec | PARAMETERS TO CALL WITH: 150000
count_mac_invocations_in_this_code_part=0;
_secureend=clock();
_securetime=((double) (_secureend - _securestart)) / CLOCKS_PER_SEC;
printf("\n");
printf("New Secure find_primes_up_to_a_number time:%lg cpu seconds\n",_securetime);
```

Σχήμα 5-9. Η κλήση της συνάρτησης υπολογισμού πρώτων αριθμών.

Το σημείο στο οποίο μπαίνει ο κώδικας για την κλήση της συνάρτησης είναι η γραμμή που ξεκινάει με “HEY PYTHON CALLING FUNCTION”. Η python διαβάζοντας αυτή τη γραμμή εισάγει κώδικα C ο οποίος κάνει ακριβώς αυτή τη δουλειά, δηλαδή δεσμεύει το function frame, αρχικοποιεί τις παραμέτρους της συνάρτησης και μεταφέρει την εκτέλεση στον κώδικα της ασφαλούς συνάρτησης, με ένα goto.
Ο κώδικας μετά την εκτέλεση του προγράμματος python, στη σημείο κλήσης της συνάρτησης φαίνεται στο ακόλουθο σχήμα.

```
    //HEY PYTHON CALLING FUNCTION : find_primes_up_to_a_number_sec | PARAMETERS TO CALL WITH: 150000
returned_addr_after_allocating=allocate_mem_into_secure_stack_in_chunks(7);
if (returned_addr_after_allocating==NULL) {printf("ERROR,no stack mem left, line %d\n",__LINE__);exit(8);}

set_stack_pointer(returned_addr_after_allocating+(1+0)*(stack_bytes_used_for_keyshares+number_of_mac_bytes+stack_bytes_for_useful_data),
&&return_label_find_primes_up_to_a_number_sec_no_1);
size_of_array_for_array_fun_parameters=1*4;
array_for_int_fun_find_primes_up_to_a_number_sec_params[0]=150000;
insert_data_into_stack_mem(size_of_array_for_array_fun_parameters,(unsigned char*)array_for_int_fun_find_primes_up_to_a_number_sec_params,(unsigned
char*)returned_addr_after_allocating+(0)*(stack_bytes_used_for_keyshares+number_of_mac_bytes+stack_bytes_for_useful_data));
set_stack_pointer(returned_addr_after_allocating+(1+0+1)*(stack_bytes_used_for_keyshares+number_of_mac_bytes+stack_bytes_for_useful_data),base_pointer_for_stack);
base_pointer_for_stack=returned_addr_after_allocating+(1+0+1)*(stack_bytes_used_for_keyshares+number_of_mac_bytes+stack_bytes_for_useful_data);
goto find_primes_up_to_a_number_sec_start_label;
return_label_find_primes_up_to_a_number_sec_no_1:
```

Σχήμα 5-10. Ο κώδικας που προστέθηκε στην κλήση της συνάρτησης υπολογισμού πρώτων αριθμών.

Όπως μπορούμε να δούμε δεσμεύει τον χώρο, θέτει το return address, εισάγει τις παραμέτρους, θέτει το base pointer και πηγαίνει στον κώδικα της συνάρτησης. Στον δε κώδικα της συνάρτησης υπολογισμού πρώτων έχουν γίνει οι εξής δύο αλλαγές. Στην αρχή, έχουν οριστεί τα “ονόματα” των μεταβλητών ως offsets από τον base pointer.

```
find_primes_up_to_a_number_sec_start_label:
#define NUM base_pointer_for_stack-(2)*(stack_bytes_used_for_keyshares+number_of_mac_bytes+stack_bytes_for_useful_data)+0*(stack_bytes_used_for_keyshares
+number_of_mac_bytes+stack_bytes_for_useful_data),0
;int array_for_int_fun_find_primes_up_to_a_number_sec_params[1];
#define BOOL base_pointer_for_stack+(1)*(stack_bytes_used_for_keyshares+number_of_mac_bytes+stack_bytes_for_useful_data)+0*(stack_bytes_used_for_keyshares
+number_of_mac_bytes+stack_bytes_for_useful_data),0
#define I base_pointer_for_stack+(1)*(stack_bytes_used_for_keyshares+number_of_mac_bytes+stack_bytes_for_useful_data)+1*(stack_bytes_used_for_keyshares
+number_of_mac_bytes+stack_bytes_for_useful_data),0
#define J base_pointer_for_stack+(1)*(stack_bytes_used_for_keyshares+number_of_mac_bytes+stack_bytes_for_useful_data)+1*(stack_bytes_used_for_keyshares
+number_of_mac_bytes+stack_bytes_for_useful_data),1
#define NUM_OF_PRIMES_FOUND base_pointer_for_stack+(1)*(stack_bytes_used_for_keyshares+number_of_mac_bytes+stack_bytes_for_useful_data)+1*
(stack_bytes_used_for_keyshares+number_of_mac_bytes+stack_bytes_for_useful_data),2
#define PRIMES_FOUND_SO_FAR base_pointer_for_stack+(1)*(stack_bytes_used_for_keyshares+number_of_mac_bytes+stack_bytes_for_useful_data)+3*
(stack_bytes_used_for_keyshares+number_of_mac_bytes+stack_bytes_for_useful_data),0
```

Σχήμα 5-11. Ο κώδικας που προστέθηκε στην αρχή της συνάρτησης υπολογισμού πρώτων αριθμών.

Στο δε τέλος, έχει προστεθεί ο κώδικας ο οποίος επαναφέρει το ασφαλές stack στην πρότερη κατάσταση (απελευθέρωση χώρου, επαναφορά του base pointer), και επιστρέφει στο return address, δηλαδή αμέσως μετά την κλήση της συνάρτησης. Επίσης, ακυρώνει τον ορισμό των macros που έγιναν στην αρχή της συνάρτησης.

```
temp_base_pointer=base_pointer_for_stack;
base_pointer_for_stack=get_stack_pointer(base_pointer_for_stack);
free_mem_from_secure_stack_in_chunks(7);
#undef NUM
#undef BOOL
#undef I
#undef J
#undef NUM_OF_PRIMES_FOUND
#undef PRIMES_FOUND_SO_FAR
goto *(get_stack_pointer(temp_base_pointer-(1)*(stack_bytes_used_for_keyshares+number_of_mac_bytes+stack_bytes_for_useful_data)));
find_primes_up_to_a_number_sec_end_label:
```

Σχήμα 5-12. Ο κώδικας που προστέθηκε στο τέλος της συνάρτησης υπολογισμού πρώτων αριθμών.

5.5: Υλοποίηση global μεταβλητών

Οι global μεταβλητές, οποίες βρίσκονται στο data segment, ασφαλίζονται με τον εξής τρόπο.
α) Όλες βρίσκονται μέσα σε ένα struct, ώστε να είναι σε συνεχόμενες θέσεις μνήμης.
β) Ανάμεσά τους είναι κλειδιά και MACs
γ) Προσπελαύνονται με getters και setters, αλλά οι τύποι τους είναι μόνο ακέραιοι (ints), χαρακτήρες (chars), μεγάλοι ακέραιοι (long ints), αριθμοί κινητής υποδιαστολής μονής και διπλής ακρίβειας (floats, doubles), και δείκτες (pointers). Δηλαδή δεν μπορούμε να δεσμεύσουμε έναν global πίνακα, αν θέλουμε έχουμε έναν global δείκτη και δεσμεύουμε τον πίνακα στο ασφαλές heap.
δ) Οι ασφαλείς global μεταβλητές δεν διασπώνται ανάμεσα σε δύο ή περισσότερα blocks, τουτέστιν τα χρήσιμα bytes στα blocks των global μεταβλητών θα είναι τουλάχιστον 8 (όσο το μέγεθος των long ints, doubles και pointers).

Υπενθυμίζουμε το πώς δηλώνονται οι global μεταβλητές, στο ακόλουθο σχήμα.

```
//GLOBAL DECLARATION
typedef struct
{
//ATTENTION: GLOBAL VARIABLE FOLLOWING! | SIZE:double
double global_double_variable_for_testing;
//ATTENTION: GLOBAL VARIABLE FOLLOWING! | SIZE:int
int test_global;
//ATTENTION: GLOBAL VARIABLE FOLLOWING! | SIZE:int
int secured_i;
//ATTENTION: GLOBAL VARIABLE FOLLOWING! | SIZE:long
long secured_sum;
}global_vars;

global_vars globals = {
//PLEASE PYTHON INITIALISE THE GLOBAL VARS
};
```

Σχήμα 5-13. Ο ορισμός των global μεταβλητών.

Όταν η python βλέπει το αίτημα για αρχικοποίηση των global μεταβλητών, μετατρέπει τον κώδικα αυτόν στον ακόλουθο, όπου για λόγους εποπτείας έχουμε μόνο 1 byte για κάθε keyshare και 1 byte για τα MACs. Αυτό φυσικά δεν συμβαίνει κανονικά, όπου έχουμε 32 bytes για τα 2 keyshares και 16 bytes για τα MACs. Επίσης, έχουμε θέσει το μέγεθος των χρήσιμων bytes κάθε block ίσο με 8.
Εκτός αυτού η python εισάγει “άχρηστα” bytes μέχρι να γεμίσει τα κομμάτια των χρήσιμων bytes, σε περίπτωση που η μεταβλητή που έχουμε στη σημείο αυτό δεν συμπληρώνει τα 8 bytes με το μέγεθός της. Αυτό διότι πρέπει κάθε block να έχει ίσο μέγεθος με τα υπόλοιπα.

```
//GLOBAL DECLARATION
typedef struct
{
double global_double_variable_for_testing;
 unsigned char key_1_1;
 unsigned char key_1_2;
 unsigned char mac_1_1;
int test_global;
 char useless_byte_1;
 char useless_byte_2;
 char useless_byte_3;
 char useless_byte_4;
 unsigned char key_2_1;
 unsigned char key_2_2;
 unsigned char mac_2_1;
int secured_i;
 char useless_byte_5;
 char useless_byte_6;
 char useless_byte_7;
 char useless_byte_8;
 unsigned char key_3_1;
 unsigned char key_3_2;
 unsigned char mac_3_1;
long secured_sum;
 unsigned char key_4_1;
 unsigned char key_4_2;
 unsigned char mac_4_1;
}global_vars;
```

Σχήμα 5-14. Οι global μεταβλητές μαζί με τα keyshares και τα MACs.

Κάθε byte το οποίο έχουμε εισάγει έχει και το δικό του ξεχωριστό όνομα, όταν χρειαστεί να το χρησιμοποιήσουμε στο attestation procedure ή στην επαλήθευση των MACs.

Στο επόμενο σχήμα μπορούμε να δούμε το struct με τις αρχικές τιμές των global μεταβλητών. Σε αυτό μπορούμε να δούμε πως οι μεν μεταβλητές αρχικοποιούνται στο 0, τα “άχρηστα” bytes αρχικοποιούνται στο 1, και τα μεν keyshares έχουν τυχαίες τιμές, ενώ τα MAC bytes παίρνουν ως τιμή το MAC των προηγουμένων bytes. Επειδή στο παράδειγμά μας χρησιμοποιούμε μόνο 1 byte για MAC, παίρνουμε μόνο το 1ο byte του αποτελέσματος του αλγορίθμου υπολογισμού των MACs.

Ένα άλλα σημείο το οποίο πρέπει να τονίσουμε είναι πως οι global μεταβλητές, τα “άχρηστα” bytes, τα keyshares και τα MACs παίρνουν τιμή με την εκκίνηση του προγράμματος, αφού αρχικοποιούνται στο struct. Δεν λαμβάνει χώρα δηλαδή η αρχικοποίησή τους αμέσως πριν την εκτέλεση των προσομοιώσεων, όπως συμβαίνει στο ασφαλές heap και ασφαλές stack.

```
global_vars globals = {
//PLEASE PYTHON INITIALISE THE GLOBAL VARS
0,
4,
58,
187,
0,
1,
1,
1,
1,
204,
123,
161,
0,
1,
1,
1,
1,
5,
162,
112,
0,
100,
250,
88
};
```

Σχήμα 5-14. Η αρχικοποίηση των τιμών των global μεταβλητών.

Οι global μεταβλητές προσπελαύνονται με getters και setters, όπως τα δεδομένα στο ασφαλές heap και stack. Εδώ βέβαια τα πράγματα είναι πιο απλά, μιας και δεν χρειάζεται να προσέξουμε από πού ως πού εκτείνεται μια global μεταβλητή, ώστε να επιβεβαιώσουμε ή να ενημερώσουμε το MAC των αντίστοιχων blocks. Η global μεταβλητή θα είναι σε ένα block, και το MAC θα είναι ένα και συγκεκριμένο. Ο τυπικός setter λοιπόν είναι ο εξής:

```
#define UPDATE_GLOBAL_VAR(global_var,new_value) { \
                        global_var=(new_value); \
                        if (!ignore_macs_even_if_there_are_mac_bytes) \
                        { \
                        update_mac_when_setting_data((unsigned char *)&(global_var),number_of_global_useful_bytes
+bytes_used_for_keyshares,number_of_global_useful_bytes,((unsigned char*) &(global_var))+(number_of_global_useful_bytes+bytes_used_for_keyshares)) ;\
                        } \
                    }
```

Σχήμα 5-16. Ο τυπικός setter global μεταβλητών.

Όπως βλέπουμε δεν κάνει κάτι αξιόλογο, εκτός του να θέτει τη νέα τιμή και να ενημερώνει το MAC. Το ενδιαφέρον είναι πως δουλεύει για κάθε τύπο μεταβλητής, αφού χρησιμοποιεί την ανάθεση τιμής (“=”) της γλώσσας C, και δεν θέτει κάθε byte ξεχωριστά όπως στο ασφαλές heap και stack.

Αντίστοιχα, ο getter ενός ακεραίου ακολουθεί, ο οποίος επιβεβαιώνει το MAC και επιστρέφει την τιμή του ακεραίου.

```
#define GET_GLOBAL_INT(global_var) get_global_int(&(global_var))

int get_global_int(int *global_var)
{
    if (!ignore_macs_even_if_there_are_mac_bytes)
    {
        verify_mac_on_the_fly(global_var,number_of_global_useful_bytes+number_of_interleaved_keys,number_of_global_useful_bytes);
    }
    return *global_var;
}
```

Σχήμα 5-17. Ο getter global ακεραίων.

5.6: Υλοποίηση των MACs

Ο υπολογισμός των MACs είναι αυτός που εν γένει παίρνει τον περισσότερο χρόνο στο ασφαλές πρόγραμμα. Αυτό είναι λογικό μιας και πρέπει να τον εκτελούμε για κάθε block κώδικα (κάθε λίγες εντολές), και επίσης σε κάθε προσπέλαση στη μνήμη δεδομένων, είτε είναι διάβασμα είτε γράψιμο.
Το paper των Lipton κ.ά. [1] θέτει ως MAC το εξής, όπου w είναι τα χρήσιμα bytes, K1 το πρώτο keyshare, K2 το δεύτερο και H() μια κρυπτογραφικά ασφαλής συνάρτηση Hash.

*MAC=w*H(K1)+H(K2)*

Οι πράξεις της πρόσθεσης και του πολλαπλασιασμού είναι πράξεις modulo του μεγέθους των λέξεων-words (εδώ θεωρούμε πως w, K1, K2 έχουν ίδιο μέγεθος και ίσο με μία λέξη), δηλαδή αν η λέξη έχει μήκος 128 bits, όλες οι πράξεις γίνονται modulo του 2^{128} .
Αυτός ο τύπος MAC έχει τις ιδιότητες που θέλουμε, δηλαδή να μην μπορεί ένας επιτιθέμενος ο οποίος μπορεί να αλλάξει το w κατά βούληση αλλά δεν γνωρίζει τα K1,K2 να μπορέσει με μη αμελητέα πιθανότητα να κατασκευάσει ένα ορθό MAC.
Το ίδιο ισχύει αν ο επιτιθέμενος μπορεί εκτός του w, να πανωγράψει και ένα κομμάτι μικρότερο του o(log(μεγέθους)) των κλειδιών, πράγμα το οποίο του επιτρέπει να μην τα καταστρέψει ανεπανόρθωτα. Δηλαδή ακόμα κι αν τα έχει κάνει αυτά, να μην μπορεί να κατασκευάσει ένα ορθό MAC (είτε γράφοντας εξαρχής το δικό του πάνω στο παλιό, είτε απλά αλλάζοντας κάποια κομμάτια του παλιού).

Δοκιμάσαμε την συγκεκριμένη προσέγγιση χρησιμοποιώντας τα BIGNUMS της libcrypto, αλλά η καθυστέρηση προέκυπτε πάρα πολύ μεγάλη. Σκεφτήκαμε όμως, μιας και που στο paper υπάρχει η απαίτηση τα MACs να υλοποιούνται σε hardware, να χρησιμοποιήσουμε κι

εμείς hardware για την εκτέλεσή τους. Το hardware που διαθέταμε ήταν η υλοποίηση του AES, και ο πιο απλός τρόπος να τον χρησιμοποιήσουμε ήταν να τον βάλουμε σε ένα σχήμα CBC-MAC. Δηλαδή κρυπτογραφούμε τα <useful bytes>||<keyshare1>||<keyshare2> με τον AES, και λαμβάνουμε το τελευταίο block ως το MAC.
Αιτούμαστε λοιπόν, τόσο στο paper όσο και στην παρούσα διπλωματική, το σχήμα CBC-MAC που χρησιμοποιήσαμε να ικανοποιεί την ιδιότητα του προηγούμενου MAC ώστε να προστατεύει από τον επιτιθέμενο. Αυτό το αίτημα δεν μπορούμε να το αποδείξουμε, αλλά εν γένει, όντας ένα καλό MAC που χρησιμοποιείται και στην πράξη, αλλά και ο AES έχοντας ψευδοτυχαία έξοδο με ψευδοτυχαίο ένα αρκετά μεγάλο κομμάτι της εισόδου, το MAC που παράγουμε δείχνει να έχει αυτή την ιδιότητα.
Ένα ακόμα πράγμα που πρέπει να αναφέρουμε, είναι πως για την παραγωγή αυτού του MAC χρησιμοποιείται ένα κλειδί και ένα Initialization Vector για κρυπτογράφηση. Αυτά τα χρησιμοποιούμε για τεχνικούς λόγους, γιατί είναι απαραίτητα στην παραγωγή αυτή. Είναι σημαντικό ότι δεν είναι απαραίτητο να κρατηθούν κρυφά, και επιτρέπουμε στον επιτιθέμενο να τα γνωρίζει. Βεβαίως πρέπει να ενημερώσουμε και τα αιτήματά μας να μπορεί το σχήμα CBC-MAC με AES να παραμένει ασφαλές ακόμα κι αν το κλειδί και το IV είναι γνωστά. Για να γίνει κατανοητό πως το κλειδί δεν είναι απαραίτητα κρυφό, σκεφτείτε ότι θα μπορούσαμε να χρησιμοποιήσαμε HMAC(<useful bytes>|| <keyshares>), δηλαδή MAC με τη χρήση hash (χωρίς κλειδί), το οποίο όμως θα πρέπει να έχει τις επιθυμητές ιδιότητες. Ο λόγος που δεν το κάναμε είναι διότι δεν το είχαμε υλοποιημένο σε hardware. Παρ'ολα αυτά καθίσταται σαφές πως επιτρέπουμε στον επιτιθέμενο να γνωρίζει τον αλγόριθμο παραγωγής των MACs, άρα και το κλειδί και τον IV στο σχήμα AES-CBC.

Όσον αφορά την υλοποίηση αυτή καθεαυτή, αναφέραμε τα επικίνδυνα σημεία μιας υλοποίησης CBC-MAC στον κεφάλαιο 1. Έτσι λοιπόν προσέξαμε στην υλοποίηση και
α) Έχουμε πάντα το ίδιο IV , έναν πίνακα 16 bytes γεμάτο με μηδενικά, και
β) Όποτε υπάρχει η πιθανότητα να υπολογίσουμε MACs σε στοιχεία διαφορετικού μεγέθους (αυτό εξαρτάται από τις επιλογές του χρήστη), εισάγουμε στην αρχή του MAC 1 byte με το μέγεθος του μηνύματος (length prepending). Αν το μήνυμα έχει μέγεθος μεγαλύτερο από 255 bytes (αυτό στην πράξη δεν συμβαίνει), εισάγουμε 4 bytes στην αρχή τα οποία περιέχουν το μήκος (δηλαδή έναν ακέραιο της C).

5.7: Υλοποίηση της cache

Η ασφαλής cache όπως έχουμε πει δεν είναι απαραίτητη για την θεωρητική ασφάλεια του μοντέλου, αλλά είναι απαραίτητη για την πρακτική του αποδοτικότητα.
Όσον αφορά την υλοποίησή της, υπάρχουν κάποια σημεία τα οποία καλό είναι να αναφέρουμε.
Το πρώτο είναι πως δεν δεσμεύουμε κάποιον ιδιαίτερο χώρο στον οποίο αντιγράφουμε τα δεδομένα τα οποία πιο πρόσφατα προσπελαύνουμε, απλά κρατάμε δομές οι οποίες μας λένε ποια είναι αυτά (βάσει των διευθύνσεών τους). Το να αντιγράφουμε τα δεδομένα είναι εκ των πραγμάτων περιττό, διότι αφενός μεν δεν τα περνάμε σε cache αλλά στη μνήμη, αφετέρου δε όταν χρησιμοποιούμε την cache για τα MACs δεν μετράμε χρόνο, αλλά πλήθος κλήσεων στη συνάρτηση που υπολογίζει τα MACs.

Το δεύτερο αφορά τον τρόπο οργάνωσης της cache. Στον τομέα αυτόν όπως είπαμε και πιο πριν υπάρχουν πάρα πολλοί τρόποι προσέγγισης του τι σημαίνει “πρόσφατη προσπέλαση” στα δεδομένα (δηλαδή το πώς οργανώνεται μια cache), και φυσικά δεν μπορούμε να τους υλοποιήσουμε όλους. Αυτό όμως που μπορούμε να κάνουμε είναι να υλοποιήσουμε κάποιους αντιπροσωπευτικούς, και να ισχυριστούμε χωρίς να κάνουμε μεγάλο λάθος ότι τα αποτελέσματα και για άλλους τρόπους θα είναι παρόμοια.
Υλοποιούμε λοιπόν δύο cache, η μία για τα δεδομένα και η άλλη για τον κώδικα, οι οποίες:
α) Όταν κάνουμε READ μια θέση μνήμης, η ασφαλής CPU ελέγχει αν η διεύθυνσή της υπάρχει στην cache, και αν ναι τη διαβάζει από εκεί χωρίς να επαληθεύσει το MAC. Αν όχι, φέρνει όλο το block στην cache (επαληθεύοντας το MAC), και διαβάζει από την cache.
β) Όταν κάνουμε WRITE μια θέση μνήμης, η ασφαλής CPU ελέγχει αν η διεύθυνσή της υπάρχει στην cache, και αν ναι γράφει εκεί χωρίς να ενημερώσει το MAC στην cache ή στην μνήμη. Το μόνο που κάνει είναι να ενημερώνει μια δομή που κρατάει διευθύνσεις και dirty bits, δηλαδή ποιες έχουν ενημερωθεί στην cache και όχι στη μνήμη (η cache μας δηλαδή είναι write-back). Η κύρια μνήμη ενημερώνεται όταν αντικαθιστούμε το block στην cache με κάποιο άλλο, ή κάνουμε flush την cache. Αν τώρα το block εξ’αρχής δεν υπάρχει στην cache, ενημερώνουμε το MAC και το φέρνουμε στην cache.
γ) Το cache line είναι ίσο με ένα block δεδομένων, δηλαδή <useful bytes> || <keyshares> || <mac>
δ) Στα μικρά μεγέθη (μέχρι και 10 θέσεις για blocks) την βάζουμε στα benchmarks να είναι direct mapped, δηλαδή κάθε διεύθυνση πηγαίνει σε ένα slot στην cache που υπολογίζεται από τον αριθμό της modulo το μέγεθος της cache. Στα μεγαλύτερα μεγέθη είναι 2-way assosiative, δηλαδή κάθε διεύθυνση έχει 2 δυνατές θέσεις να μπει, και ο αριθμός του ζεύγους υπολογίζεται από τον αριθμό της διεύθυνσης modulo το μισό του μεγέθους της cache.

Όπως θα δούμε στη συνέχεια, η cache βοηθάει πάρα πολύ στην επιτάχυνση των προσομοιώσεων.

Κεφάλαιο 6: Αποτελέσματα προσομοιώσεων

6.1: Τρόπος συλλογής αποτελεσμάτων

Τις προσομοιώσεις τις τρέξαμε σε ένα laptop Acer Aspire V13 V3-372, με επεξεργαστή τετραπύρηνο Intel Core i5-6267U στα 2.9 GHz με υποστήριξη AES-NI και 8 GB μνήμης RAM. Το λειτουργικό ήταν Linux Mint Serena 18.1 64 bit, το οποίο είναι βασισμένο στα Ubuntu 16.04. Ο κώδικας C μεταγλωττίστηκε με τον gcc (έκδοση 5.4.0), ο κώδικας Java με το OpenJDK 8 (1.8.0_131) και ο κώδικας Python έτρεξε με την Python 3.5.2. Οι βοηθητικές εφαρμογές που χρησιμοποιήσαμε (as, objdump) ήταν μέρος των GNU binutils για τα Ubuntu 2.26. Όλος ο κώδικας ήταν single-threaded και γραμμένος για 64-bit επεξεργαστές.

Υπενθυμίζουμε πως για τη συλλογή των αποτελεσμάτων ακολουθήθηκε η εξής διαδικασία:
α) Τρέξαμε τα "ανασφάλιστα" προγράμματα για κάποια συγκεκριμένη είσοδο και καταγράψαμε τους χρόνους τους. Τα προγράμματα αυτά μεταγλωττίστηκαν χωρίς optimizations.
β) Τρέξαμε τα "ασφαλισμένα" προγράμματα (πάλι χωρίς optimizations) αγνοώντας τα MACs. Δηλαδή δεν εισάγαμε verification κώδικα στην αρχή των blocks στον κώδικα, δεν εκτελούσαμε το verification όταν προσπελαύναμε μνήμη δεδομένων, και δεν διασπούσαμε τα blocks του κώδικα σε περίπτωση που συναντούσαμε label ή call. Η μόνη διάσπαση που γίνεται είναι όταν γεμίσει ένα block από εντολές. Αυτό το κάναμε γιατί έτσι ακριβώς θα λειτουργούσε και η ασφαλής CPU. Παρ'όλα αυτά, αναγκαστικά εισάγαμε jumps στο τέλος του κώδικα για να μην εκτελέσουμε τα keyshares και τα MACs. Ομοίως με πριν, καταγράψαμε τον χρόνο που παίρνουν αυτά τα προγράμματα, ο οποίος ήταν η βάση (baseline) κάτω από την οποία δεν μπορούμε να πέσουμε. Αυτό θα μπορούσε να γίνει μόνο σε περίπτωση που υλοποιούσαμε σε hardware το μοντέλο, και μιας και το έχουμε μόνο σε software, είναι φυσικό να έχουμε μια καθυστέρηση.
γ) Τρέξαμε τα "ασφαλισμένα" προγράμματα με τον υπολογισμό των MACs ενεργοποιημένο (verification calls σε κάθε block κώδικα και επαλήθευση των MACs στους getters και setters), καθώς και διάσπαση blocks. Κατά την εκτέλεση, μετράμε το πλήθος των κλήσεων των MAC και το μέγεθος για το οποίο καλείται (αν έχουμε σταθερού μεγέθους block θα είναι συγκεκριμένο και δεν θα αλλάζει). Έχουμε λοιπόν, στο τέλος του προγράμματος το ιστόγραμμα των συχνοτήτων των κλήσεων MAC. Υπολογίζουμε ακολούθως το πόσο παίρνει να εκτελεστεί ένα MAC κάθε τέτοιου μεγέθους, και πολλαπλασιάζουμε τους χρόνους.
Υπενθυμίζουμε πως στην τρίτη εκτέλεση η διάσπαση των blocks γίνεται όταν βρούμε μπροστά μας label, call ή όταν η ασφαλής cpu θα διασπούσε λόγω γεμίσματος block (αυτή όμως υπολογίζει πάνω στα αδιάσπαστα από label ή calls blocks).

Προσθέτοντας τα β) και γ), έχουμε μια καλή εκτίμηση για το πόσο χρόνο θα έπαιρνε η εκτέλεση σε μια ασφαλή CPU. Ο λόγος για τον οποίο δεν υπολογίσαμε με μιας το πόσο χρόνο παίρνει ένα πρόγραμμα το οποίο υπολογίζει τα MACs, είναι διότι σε κάθε υπολογισμό των MACs, αναγκαζόμαστε να σώσουμε την κατάσταση της CPU (registers, flags κλπ), να υπολογίσουμε το MAC αφού πρώτα ελέγξουμε τις δομές της cache, και να επαναφέρουμε την κατάσταση της CPU στο πρότερο σημείο. Αυτές οι διεργασίες είτε δεν

θα παίρνουν καθόλου είτε θα παίρνουν ελάχιστο χρόνο από την ασφαλή CPU, μιας και τα έχει όλα αυτά υλοποιημένα σε κύκλωμα. Η διάσωση της κατάστασης και επαναφορά της δεν χρειάζεται να γίνεται καθόλου, αφού τον υπολογισμό του MAC τον αναλαμβάνει ειδικό hardware που έχει δικό του state. Η αναζήτηση στην cache στους μοντέρνους επεξεργαστές δεν παίρνει παρά ελάχιστους κύκλους CPU. Η μέτρησή μας από την άλλη, ακόμα κι όταν είναι “καθαρή” και εκτελεί μόνο τον υπολογισμό των MACs, και μόνο ότι καλεί τη συνάρτηση υπολογισμού (η οποία με τη σειρά της καλεί άλλες 3), παίρνει περισσότερο από ένα cache lookup, άρα υπερεκτιμούμε τον χρόνο που θα πάρει ένας υπολογισμός του MAC (ελάχιστα βέβαια). Το δε ενδεχόμενο cache miss που θα κοστίσει το να φέρουμε στην cache τα δεδομένα και να τα υπολογίσουμε εκεί, πληρώνεται έτσι κι αλλιώς στον getter ή τον setter όταν συμβαίνει. Άρα δεν έχει νόημα να υπολογίζουμε στο MAC τον χρόνο που παίρνει ένα ενδεχόμενο cache miss, μιας και θα διπλομετράμε.

Πιο μετά θα παρουσιάσουμε τα τέσσερα benchmarks που τρέξαμε, τα οποία θεωρούμε αντιπροσωπευτικά για ένα μεγάλο εύρος προβλημάτων.

6.2: Επεξήγηση προεπιλεγμένων τιμών

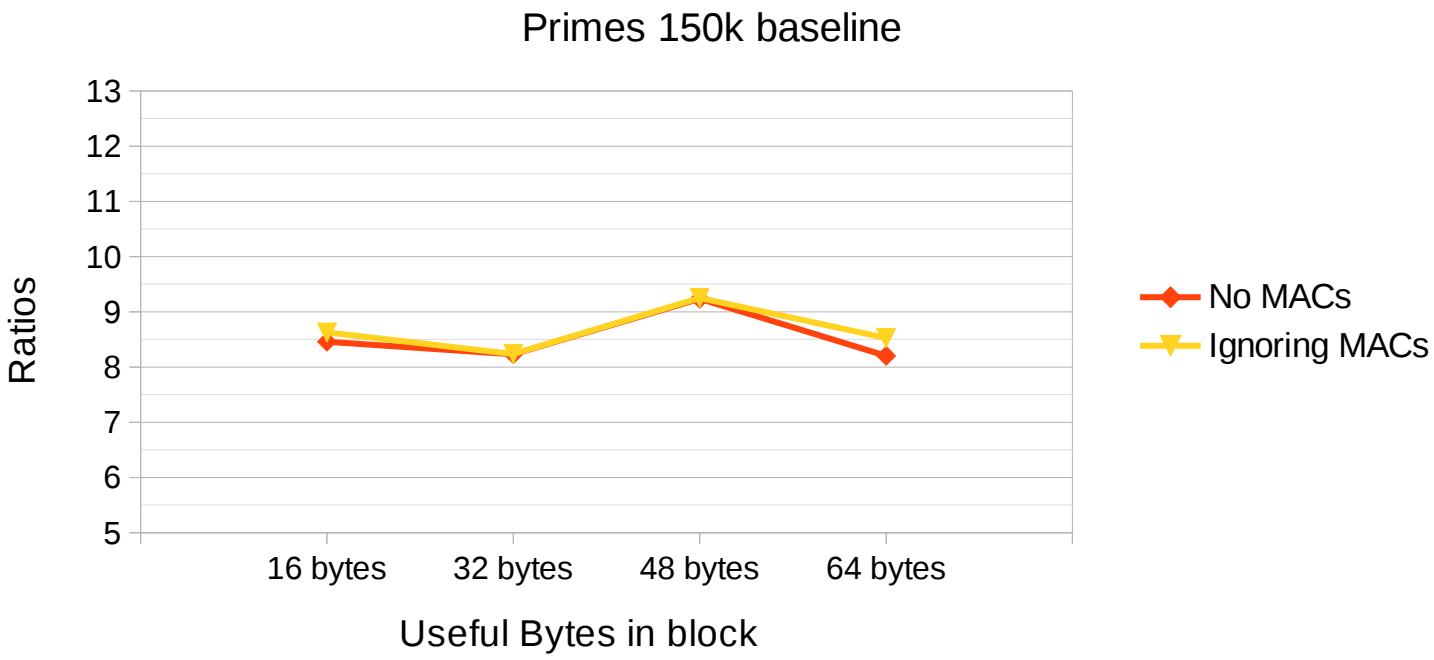
Το πρόβλημα το οποίο καλούμαστε να λύσουμε (προσομοίωση ασφαλούς hardware) έχει πάρα πολλές μεταβλητές, και φυσικά δεν είναι δυνατόν να εξερευνήσουμε όλο το εύρος των τιμών τους, για κάθε μία από αυτές. Θέτουμε λοιπόν τιμές στις περισσότερες από αυτές, και αλλάζουμε μόνο λίγες, αυτές που θεωρούμε ότι έχει πραγματικά νόημα.
Έτσι λοιπόν έχουμε:
α) Ανάμεσα στα χρήσιμα bytes και τα MACs, θέτουμε 32 bytes για τα 2 keyshares. Το κάθε keyshare έχει δηλαδή 16 bytes (=128 bits), αρκετά μεγάλο ώστε να μην μπορεί να το μαντέψει ο επιτιθέμενος. Τα 128 bits θεωρούνται, ως παράμετρος ασφαλείας (security parameter), αρκετά καλά για τις σύγχρονες κρυπτογραφικές ανάγκες.
β) Το μέγεθος των MACs είναι 16 bytes για κάθε block. Η λογική είναι ακριβώς η ίδια, θέλουμε ένα αρκετά μεγάλο MAC το οποίο ο επιτιθέμενος να μην μπορεί να το κατασκευάσει.
γ) Μετά το jmp πάνω από τα keyshares και τα MACs θέτουμε 3 bytes για canaries (με την τιμή 0x42), ώστε να μπορούμε να βρούμε το jmp αυτό όταν χρειάζεται. Στατιστικά είναι πολύ απίθανο να βρούμε άλλο jmp στην θέση του.
δ) Το μέγεθος του stack και του heap το αυξομειώνουμε ανάλογα με το τι benchmark τρέχουμε, αλλά πάντα για τον ίδιο τύπο benchmark τα κρατάμε σταθερά.
ε) Το μέγεθος των blocks, κώδικα και δεδομένων, είναι σταθερό και ίσο και στον κώδικα, και στο heap, και στο stack και στις global μεταβλητές. Σε αυτό το σημείο να τονίσουμε ότι πειραματιστήκαμε και με μεταβλητό μέγεθος block κώδικα, αλλά αυτό θα έκανε την υλοποίηση σε hardware πολύ δύσκολη, και εμείς προσπαθούμε να προσομοιώσουμε αληθοφανές hardware. Αυτό γίνεται μόνο αν η μνήμη έχει διασπαστεί σε ίσα κομμάτια, και κάποια από αυτά έχουν κώδικα και κάποια δεδομένα.
στ) Τα μεγέθη των χρήσιμων bytes των blocks τα αυξομειώνουμε σε βήματα του 16 από το 16 ως το 64 (δηλαδή 16-32-48-64). Θα είναι πολλαπλάσιο του 16 γιατί ο AES κρυπτογραφεί σε blocks των 16 bytes, και θα ήταν κακή χρήση του χώρου να είχαμε κάτι

διαφορετικό. Επίσης επιλέξαμε να μην το αυξήσουμε περισσότερο από 64, διότι (μαζί με τα keyshares και τα MACs) το συνολικό μέγεθος του block θα γίνει τόσο μεγάλο που το atomic fetch θα είναι ένα πρόβλημα. Επίσης μετά από ένα σημείο η επαλήθευση του MAC ενός block θα παίρνει πολύ μεγάλο χρόνο (λόγω του μεγέθους του), ενώ εμείς θα χρειαζόμασταν απλώς να διαβάσουμε ένα μικρό κομμάτι του block. Δηλαδή θα πληρώνουμε πολύ στον υπολογισμό του MAC ενώ η πραγματική χρήση της μνήμης μπορεί να είναι αρκετά μικρότερη.
ε) Το μέγεθος της ασφαλούς cache το μεταβάλλουμε στα εξής μεγέθη block (για κώδικα και για δεδομένα): 1,2,4,6,8,10 (με οργάνωση direct mapped), και 20,40,60,80,100,150,200 (με οργάνωση 2-way set-assosiative). Όταν έχουμε cache μεγέθους X blocks, σημαίνει πως έχουμε μία τέτοια cache για δεδομένα και μία τέτοια cache για κώδικα, να χρησιμοποιούνται παράλληλα.

Επίσης ας αναφέρουμε δύο σημεία τα οποία μειώνουν ως ένα βαθμό τον χρόνο εκτέλεσης.
α) Τον κώδικα του code verification και το jmp πάνω από τα keyshares+MACs δεν τα υπολογίζουμε στο MAC, αφού η ασφαλής CPU δεν θα το έκανε (δεν θα υπήρχαν καν εντός του κώδικα αλλά θα ήταν κομμάτι του hardware). Το ίδιο ισχύει για τα βοηθητικά bytes που εισάγουμε (όπως τα canaries). Αναγκαζόμαστε όμως να βάλουμε στο MAC τα padded NOPs που εισάγουμε για να συμπληρώσουμε σταθερό μέγεθος block (όπως θα υπήρχαν και στην μνήμη σε περίπτωση που υλοποιούσαμε το σχήμα σε hardware).
β) Τα keyshares, για να γλιτώσουμε έναν γύρο κρυπτογράφησης AES, τους μειώνουμε το μέγεθος στο μισό κατά τη διάρκεια του MAC (δηλαδή από 32 bytes τα μειώνουμε στα 16). Αυτό δεν μειώνει την ασφάλεια (τα 16 bytes είναι ακόμα αρκετά πολλά), και συμβαίνει ως εξής: κάνουμε XOR το πρώτο byte του πρώτου με το πρώτο byte του δεύτερου, το δεύτερο του πρώτου με το δεύτερο του δεύτερου κλπ. Το θεωρητικό μοντέλο δεν απειλείται, αλλά ο χρόνος υπολογισμού μειώνεται ελαφρώς.

6.3: Υπολογισμός πρώτων αριθμών

Το πρώτο πρόβλημα που εξετάζουμε είναι ένα απλό μαθηματικό πρόβλημα, που υπολογίζει τους πρώτους αριθμούς μέχρι ένα νούμερο (στην περίπτωσή μας το 150000).
Ο αλγόριθμος είναι πολύ απλός (και σε καμία περίπτωση ο βέλτιστος): Προσθέτουμε το 2 στη λίστα των πρώτων αριθμών, και έπειτα για κάθε περιττό αριθμό ως το 150000 κοιτάζουμε όλους τους αριθμούς ως το μισό του (δεν χρησιμοποιούμε τη ρίζα γιατί δεν έχουμε γράψει εμείς την υλοποίησή της ώστε να είναι ασφαλής), για να βρει διαιρέτη. Αν δεν βρει, εισάγει τον αριθμό σε έναν πίνακα με πρώτους.
Το πρόβλημα αυτό αντιπροσωπεύει ένα κοινό υπολογιστικό πρόβλημα το οποίο κυρίως καθυστερεί στους υπολογισμούς και δεν οφείλει την καθυστέρησή του τον επικοινωνία με τη μνήμη. Όταν αγγίζει τη μνήμη συνήθως το κάνει στις ίδιες θέσεις, άρα έχει καλή χρησιμοποίηση της cache.
Κατ’αρχάς θα παρουσιάσουμε το baseline που έχουμε για τους πρώτους αριθμούς, δηλαδή τον χρόνο που παίρνει το πρόγραμμα να εκτελέσει τις προσομοιώσεις αγνοώντας τα MACs.

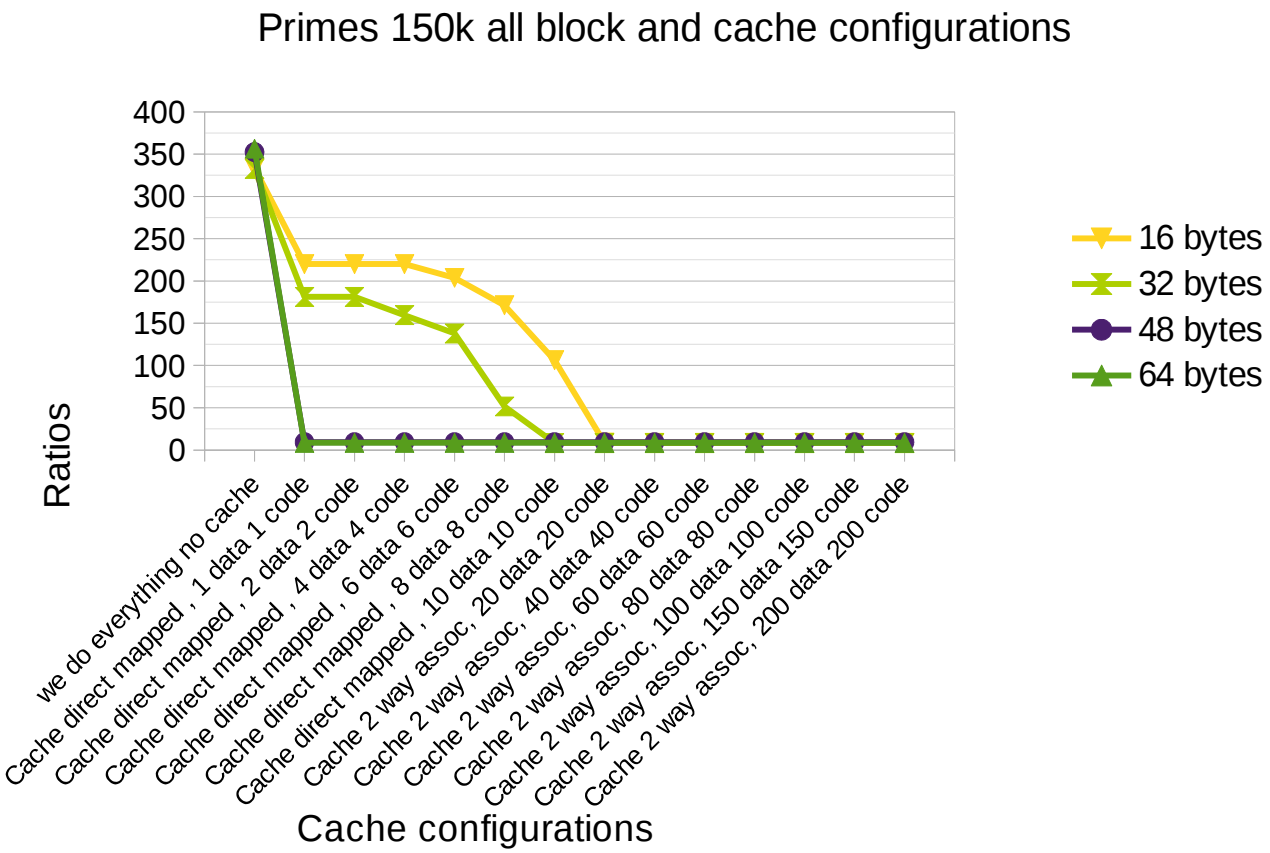


Σχήμα 6-1. Το baseline των πρώτων αριθμών.

Στο σχήμα αυτό βλέπουμε τα εξής:
Οι δύο καμπύλες είναι οι δύο “κόσμοι”, αφενός μεν αυτός ο οποίος δεν έχει καθόλου MACs στη μνήμη (μόνο keyshares), και αφετέρου αυτός ο οποίος έχει τα MACs, αλλά δεν κάνει τίποτε γι’αυτά. Δεν εκτελεί verification στον κώδικα, δεν εκτελεί verification στους getters/setters, και δεν διασπά blocks στον κώδικα λόγω labels και calls.
Ο άξονας των X αφορά το μέγεθος των χρήσιμων bytes στο block (κώδικα ή δεδομένων) που χρησιμοποιούμε, και ο άξονας των Y αφορά την καθυστέρηση σε σχέση με το ανασφαλές πρόγραμμα.

Από το γράφημα προκύπτουν τα εξής δύο συμπεράσματα:
α) Ο κόσμος χωρίς MACs και με αυτά στη μνήμη (χωρίς όμως να κάνουμε κάτι γι’ αυτά), δεν διαφέρουν σχεδόν καθόλου.
β) Το πρόγραμμα, χωρίς τον υπολογισμό των MACs και απλώς και μόνο επειδή χρησιμοποιεί την υλοποίηση του υλικού που κάναμε σε λογισμικό (getters/setters, jmp πάνω από keyshares+MACs κλπ), έχει μια καθυστέρηση ~8 φορές μεγαλύτερη του ανασφαλούς προγράμματος. Αυτή η καθυστέρηση οφείλεται ακριβώς στην υλοποίηση του υλικού σε λογισμικό, αν είχαμε κυκλωματική πραγμάτωση των λειτουργιών αυτών, η απόδοση θα ήταν πολύ καλύτερη. Αυτό που πρέπει να κρατήσουμε είναι πως κάτω από αυτόν τον αριθμό είναι αδύνατον να πέσουμε στην παρούσα διπλωματική.

Ακολούθως θα παρουσιάσουμε την εκτέλεση συμπεριλαμβάνοντας και τον υπολογισμό των MACs.



Σχήμα 6-2. Το γράφημα χρόνου εκτέλεσης για τους πρώτους αριθμούς.

Το ανωτέρω γράφημα ερμηνεύεται ως εξής:
Πρόκειται για την εκτέλεση του προγράμματος υπολογισμού πρώτων αριθμών για διάφορα block sizes και μεγέθη cache. Οι 4 καμπύλες που υπάρχουν αντιπροσωπεύουν τις εκτελέσεις για τα τέσσερα μεγέθη χρήσιμων byte στα block (16, 32, 48, 64). Ο άξονας των X έχει στήλες για κάθε cache configuration (όχι cache, 1-10 direct mapped, 20-200 2-way set assosiative). Ο άξονας των Y δείχνει για τις συγκεκριμένες παραμέτρους το πόσο πιο αργό είναι το ασφαλές πρόγραμμα από το μη ασφαλές (τον λόγο των χρόνων τους).

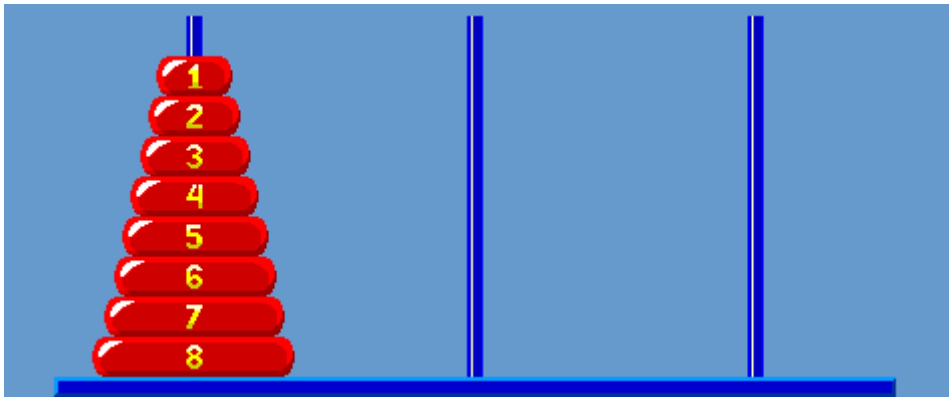
Σε αυτό το γράφημα παρατηρούμε τα εξής:
α) Χωρίς cache, η εκτέλεση του ασφαλούς προγράμματος παίρνει γύρω στις 330-350 φορές τον χρόνο του μη ασφαλούς προγράμματος, κάτι απαγορευτικό.
β) Με την εισαγωγή της cache, ο χρόνος πέφτει δραματικά. Για τα μεν μικρότερα μεγέθη στα χρήσιμα bytes στα blocks (16, 32 bytes) η καθυστέρηση πέφτει βαθμηδόν, ενώ για μεγαλύτερα μεγέθη (48, 64) η καθυστέρηση πέφτει άμεσα.
Τα benchmarks σε όλα τα block sizes, στην περίπτωση της τελευταίας cache (200 blocks για κώδικα και 200 για δεδομένα) αλλά και αρκετά πιο πριν, έχουν πάρει την ελάχιστη τιμή τους. Αυτή η τιμή είναι 8 φορές περίπου ο χρόνος του μη ασφαλούς προγράμματος, και είναι ακριβώς η τιμή του baseline που παρουσιάσαμε στο προηγούμενο διάγραμμα. Ο χρόνος που παίρνει ο υπολογισμός των MACs στις στήλες αυτές είναι 0 δευτερόλεπτα, δηλαδή τα MACs δεν επιβαρύνουν καθόλου σε αυτές τις περιπτώσεις.
Το ενδιαφέρον είναι πως αυτό συμβαίνει και χωρίς ιδιαίτερα μεγάλη cache, όπως θα

αναλύσουμε στη συνέχεια.

6.4: Λύση Πύργων του Ανόι

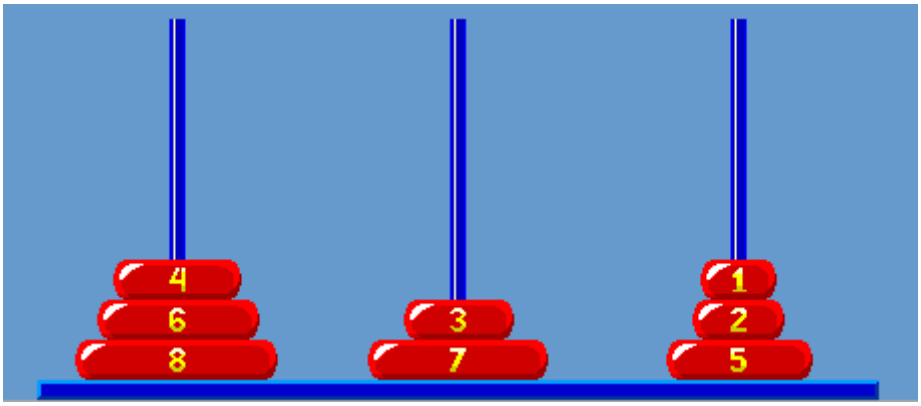
Το δεύτερο πρόβλημα που εξετάζουμε είναι το πρόβλημα των πύργων του Ανόι [27] [28]. Το πρόβλημα αυτό περιγράφεται ως εξής: Έχουμε 3 στύλους, και N δίσκους διαφορετικού μεγέθους, ο ένας πάνω στον άλλο περασμένοι στον πρώτο στύλο. Οι δίσκοι είναι διατεταγμένοι από τον μεγαλύτερο (κάτω-κάτω) στον μικρότερο. Στόχος είναι να μεταφέρουμε τους δίσκους από τον πρώτο στύλο στον τελευταίο στην ίδια διάταξη, με τους εξής δύο κανόνες:
α) Σε μια κίνηση δεν μπορούμε να μεταφέρουμε πάνω από έναν δίσκο
β) Ποτέ δεν πρέπει δίσκος μικρότερου μεγέθους να βρίσκεται κάτω από δίσκο μεγαλύτερου μεγέθους.

Το πρόβλημα στην αρχική του διάταξη φαίνεται στην ακόλουθη εικόνα:



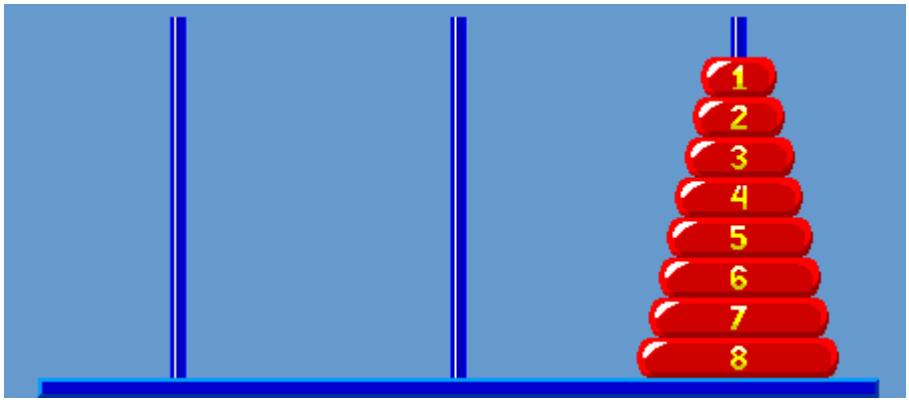
Σχήμα 6-3. Το αρχικό στιγμιότυπο του προβλήματος των πύργων του Ανόι.

Σε μια ενδιάμεση κατάσταση είναι κάπως έτσι:



Σχήμα 6-4. Ένα ενδιάμεσο στιγμιότυπο του προβλήματος των πύργων του Ανόι.

Ενώ θεωρείται λυμένο όταν οι δίσκοι είναι στην εξής διάταξη:



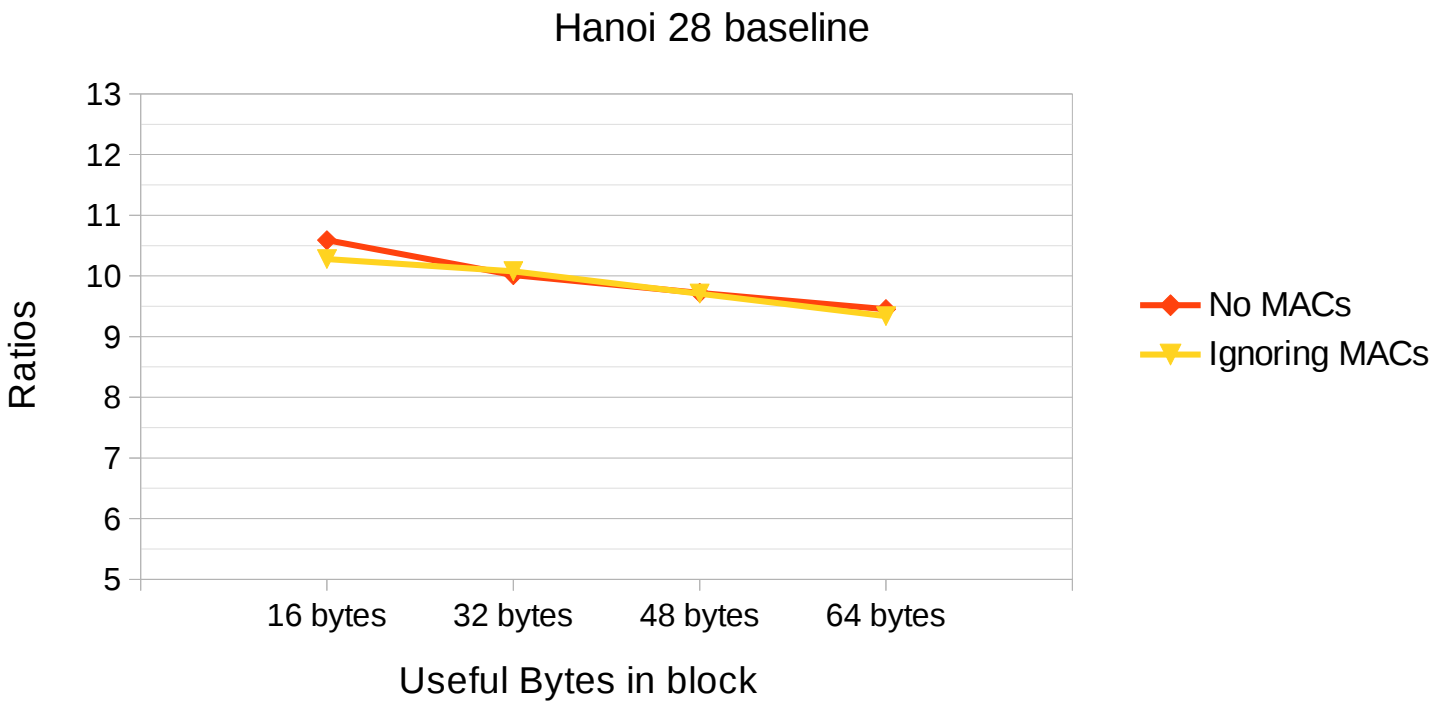
Σχήμα 6-5. Το τελικό στιγμιότυπο του προβλήματος των πύργων του Ανόι.

Το πρόβλημα έχει μια πολύ απλή αναδρομική λύση:
α) Αν έχεις μόνο ένα δίσκο να μεταφέρεις από τον στύλο X στον Z (χρησιμοποιώντας ως βοηθητικό τον Y), πάρε τον δίσκο και πήγαινέ τον από τον X στον Z.
β) Αν έχεις παραπάνω από έναν δίσκους (έστω N) να μεταφέρεις από τον στύλο X στον Z (χρησιμοποιώντας ως βοηθητικό τον Y), μετάφερε αναδρομικά τους N-1 από τον X στον Y (χρησιμοποιώντας ως βοηθητικό τον Z), μετάφερε από τον X στον Z τον ένα δίσκο που απομένει, και επίσης αναδρομικά μετάφερε από τον Y στον Z τους N-1 δίσκους (χρησιμοποιώντας ως βοηθητικό τον X).

Αυτή η λύση είναι βέλτιστη και κάνει $2^N - 1$ βήματα να τελειώσει.
Το πρόβλημα το υλοποιούμε για να δούμε πόσο καλά η ασφαλής CPU εκτελεί αναδρομικά προγράμματα, και θέτουμε N=28.

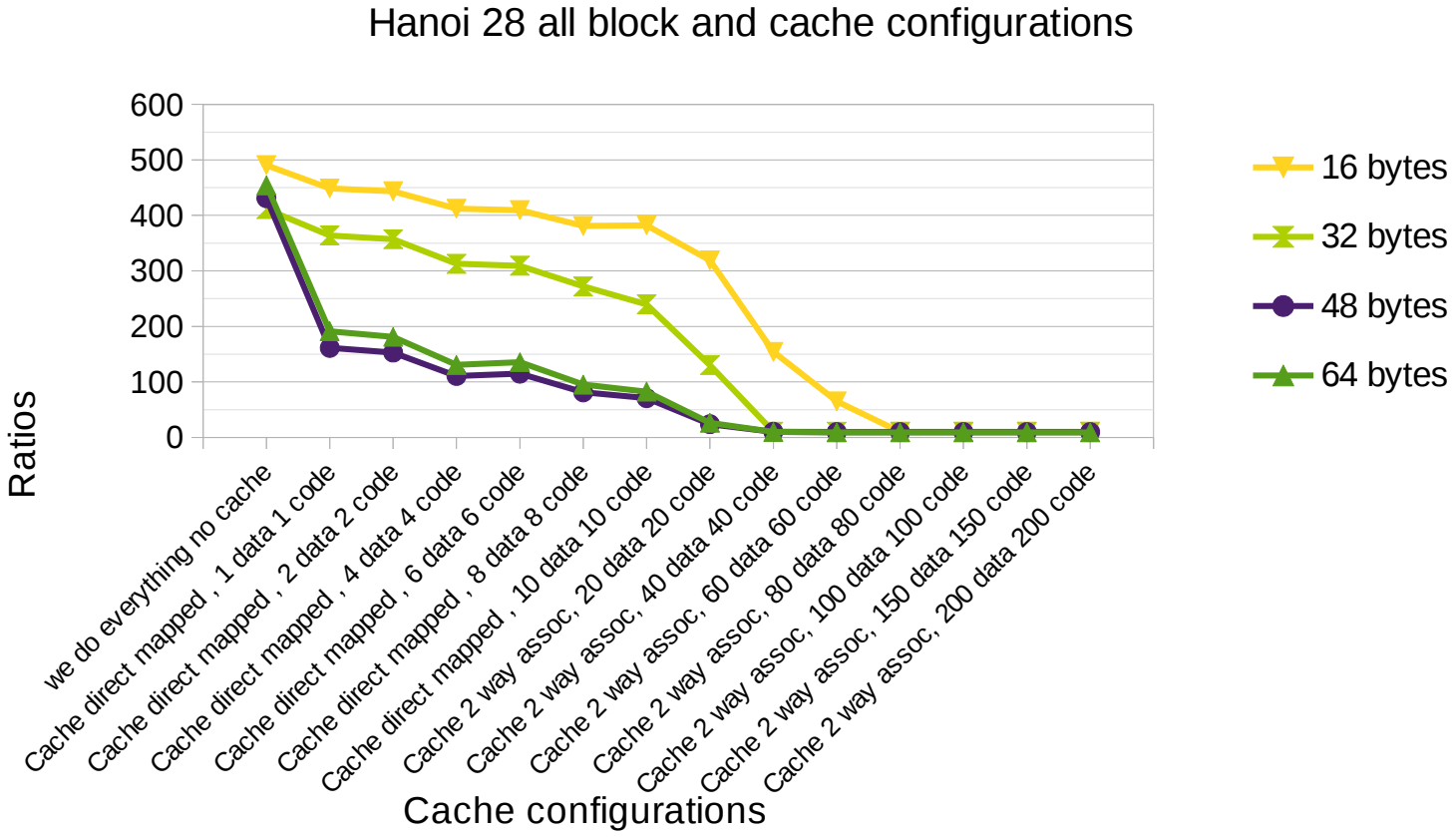
Ακολουθεί το γράφημα που δίνει το baseline για τους πύργους του Ανόι. Μπορούμε να δούμε πως πάλι το καλύτερο που μπορούμε να κάνουμε είναι ~9-10 φορές τον χρόνο που παίρνει το ανασφαλές πρόγραμμα λόγω της υλοποίησής μας σε λογισμικό.

Θα μπορούσε κάποιος να πει πως η αύξηση του block size δίνει και καλύτερους χρόνους για το πρόβλημα αυτό. Ο συγκεκριμένος ισχυρισμός εν γένει δεν είναι λανθασμένος, αλλά η διαφορά στις τιμές είναι αρκετά μικρή που θα λέγαμε ότι βρίσκεται στα όρια του στατιστικού λάθους. Δεν είναι άγνωστο ότι όταν τρέχουμε ένα πρόγραμμα σε ένα πραγματικό σύστημα, αν το τρέξουμε αρκετές φορές μπορεί να μην πάρουμε εντελώς ίδιους χρόνους εκτέλεσης.



Σχήμα 6-6. Το baseline των πύργων του Ανόι.

Στη συνέχεια παραθέτουμε και τους χρόνους που έκανε το πρόγραμμα με τον υπολογισμό των MACs ενεργοποιημένο.



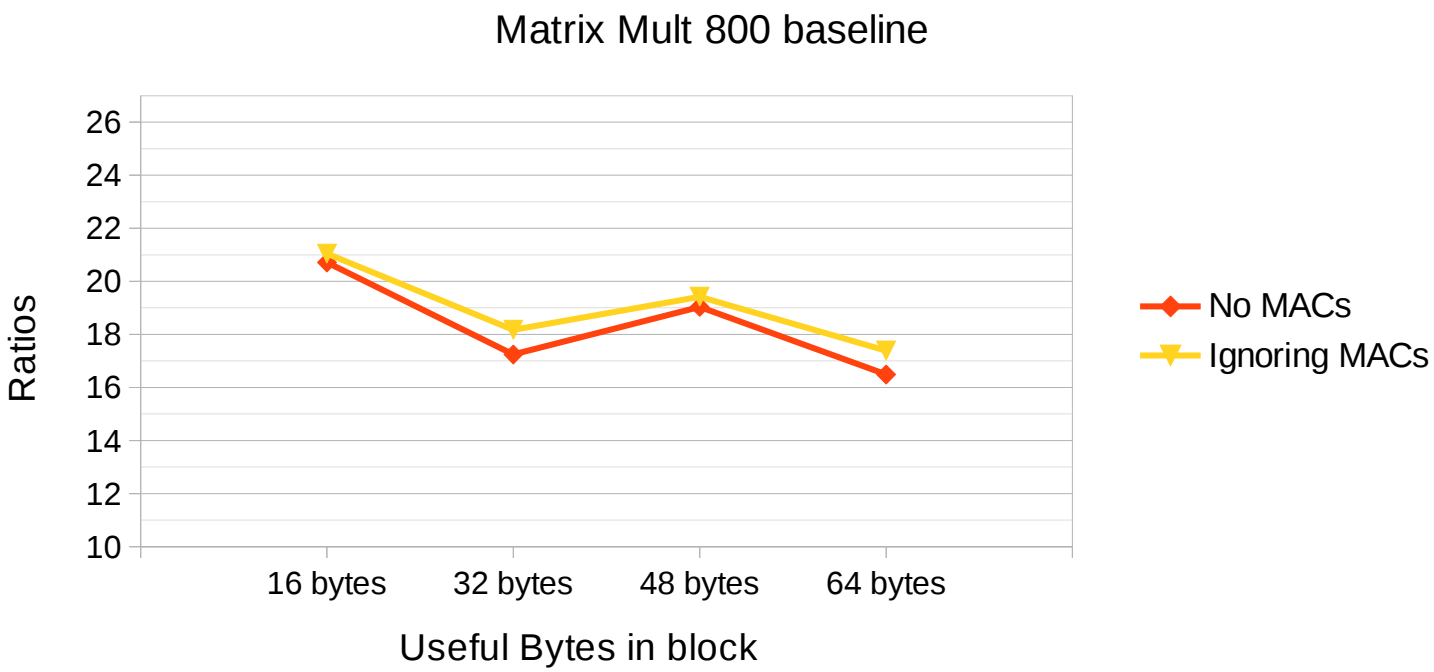
Σχήμα 6-7. Το γράφημα χρόνου εκτέλεσης για τους πύργους του Ανόι.

Όπως και με τους πρώτους αριθμούς, βλέπουμε πως ενώ ξεκινάμε με πολύ άσχημη απόδοση (400-500 φορές πιο αργά σε σχέση με το ανασφαλές πρόγραμμα), όσο αυξάνουμε την cache το αποτέλεσμα πλησιάζει το baseline, και για μεγάλες cache το φτάνει (η επιβάρυνση λόγω MACs πάει στο 0).
Βλέπουμε πως εν γένει όσο αυξάνουμε το μέγεθος του block τόσο πιο γρήγορα φτάνουμε στο baseline, με μια μικρή μόνο εξαίρεση, από τα 48 στα 64 bytes. Αν και η διαφορά είναι στα όρια του στατιστικού λάθους, μπορεί να δοθεί μία εξήγηση γιατί να συμβαίνει κάτι τέτοιο:
Με την αύξηση του block, έχουμε περισσότερο κώδικα στο ίδιο block που μπορούμε να τον εκτελέσουμε χωρίς jump στο επόμενο και χωρίς verification, άρα εκεί έχουμε επιτάχυνση. Μπορεί όμως να έχουμε κακή χρήση των δεδομένων. Σε ένα μεγάλο block παίρνει παραπάνω ώρα να επικυρώσουμε το MAC, και αν εμείς χρειαζόμαστε μόνο ένα μικρό κομμάτι του, τότε πληρώνουμε πολύ για λίγο όφελος. Επομένως, αναλόγως το πόσο κάθε benchmark προσπελαύνει μακρινά μεταξύ τους δεδομένα που βρίσκονται σε διαφορετικά blocks (δηλαδή έχει κακή απόδοση στην cache), τα μεγάλα μεγέθη block μπορεί να επιβραδύνουν την εκτέλεση.

6.5: Πολλαπλασιασμός πινάκων

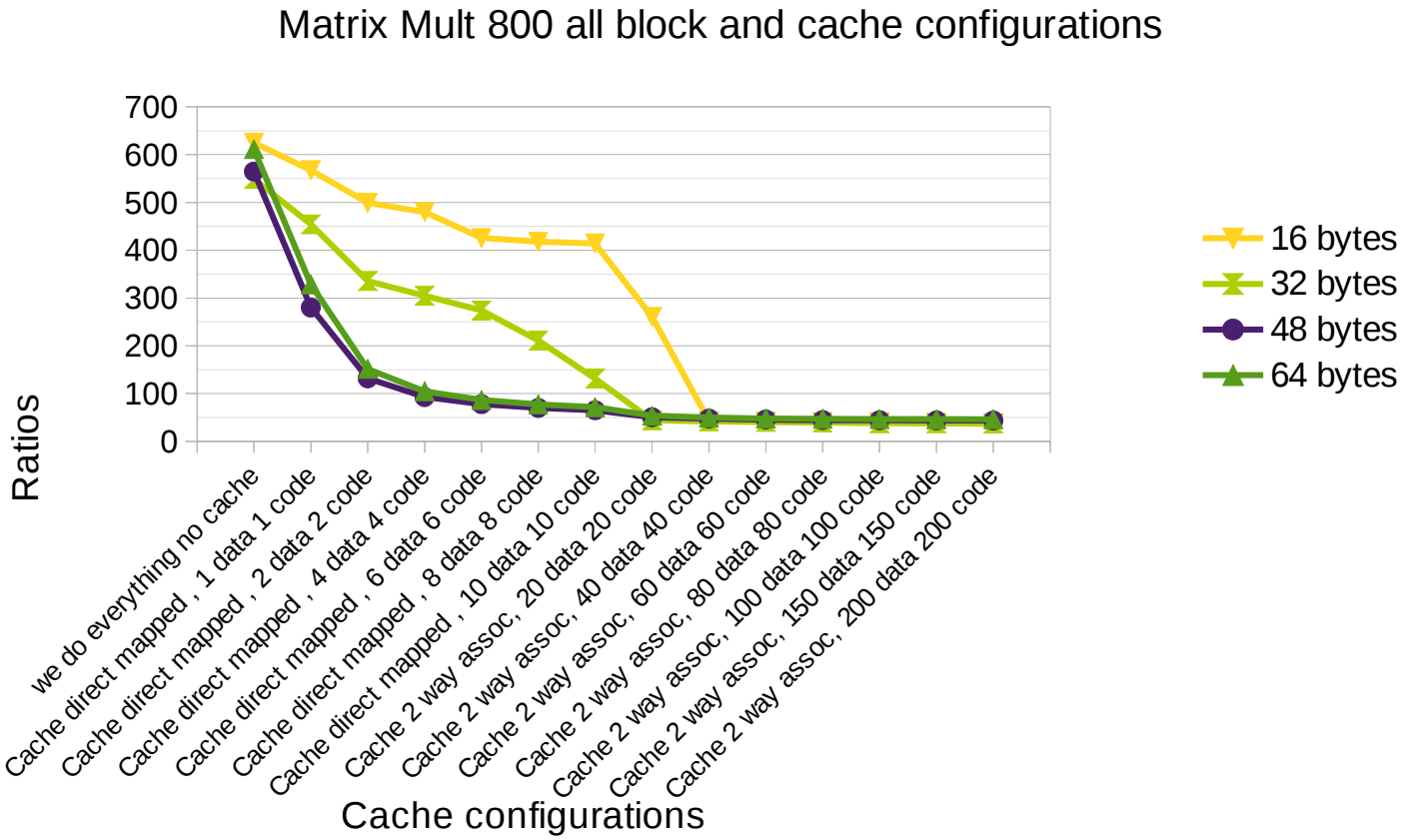
Το τρίτο πρόβλημα με το οποίο ασχολούμαστε είναι αυτό του πολλαπλασιασμού τετραγωνικών πινάκων [29].
Επιλέξαμε την κλασσική υλοποίηση, η οποία δεδομένων δύο πινάκων μεγέθους N^2 έκαστος, διατρέχει τον πρώτο κατά γραμμή και τον δεύτερο κατά στήλη, και αθροίζοντας τα γινόμενα των αριθμών, παράγει τα στοιχεία του αποτελέσματος του γινομένου των δύο πινάκων. Η συνολικά πολυπλοκότητα είναι $O(N^3)$.
Ο πολλαπλασιασμός πινάκων για ένα πρόβλημα το οποίο αγγίζει συχνότατα τη μνήμη, και μάλιστα σε σημεία που δεν ευνοούν καθόλου την cache (αφού ο δεύτερος πίνακας διατρέχεται κατά στήλη). Εμείς γι’ αυτό τον λόγο το συμπεριλάβαμε στα benchmarks, και θέσαμε N=800.

Ακολουθεί το baseline για τον πολλαπλασιασμό πινάκων, στο οποίο βλέπουμε τα εξής:
α) Υπάρχει μια ελάχιστη διαφορά μεταξύ της εκτέλεσης που δεν έχει MACs με αυτή που έχει αλλά δεν κάνει κάτι γι’ αυτά. Η διαφορά πάντως είναι στατιστικά ασήμαντη.
β) Εδώ το baseline είναι αρκετά υψηλό (~16-21 φορές πιο αργό), λόγω της φύσης του προβλήματος. Η προσπέλαση στα δεδομένα γίνεται αρκετά “σκόρπια”, με αποτέλεσμα να πληρώνουμε και περισσότερο χρόνο. Παρ’όλα αυτά, είναι πεποίθησή μας πως με φυσική υλοποίηση της ασφαλούς CPU χρόνος αυτός θα ελαχιστοποιούνταν.



Σχήμα 6-8. Το baseline του πολλαπλασιασμού πινάκων.

Ακολουθεί και το γράφημα χρόνου εκτέλεσης, όταν ενεργοποιούμε την επαλήθευση των MACs.



Σχήμα 6-9. Το γράφημα χρόνου εκτέλεσης για τον πολλαπλασιασμό πινάκων.

Η εικόνα μοιάζει με τα προηγούμενα δυο προβλήματα αλλά έχει και κάποια ιδιαίτερα χαρακτηριστικά.
Βλέπουμε πως χωρίς cache η απόδοση είναι απογοητευτική (~550-630 φορές πιο αργό από το μη ασφαλές πρόγραμμα), και με την cache να αυξάνεται, τόσο αυξάνεται και η απόδοση. Όπως και πριν μεγαλύτερα blocks δίνουν εν γένει και καλύτερα αποτελέσματα, εκτός αν έχουμε κακή, "σκόρπια" χρήση των δεδομένων ώστε να πληρώνουμε πολύ (υπολογισμός MAC) για μικρό όφελος. Αυτό συμβαίνει στο τέλος για μεγάλες τιμές cache, όπου τα μικρά μεγέθη block τα καταφέρνουν καλύτερα από τα μεγαλύτερα. Κάτι παρόμοιο βλέπουμε και στην αρχή (όταν δεν έχουμε cache), που το block των χρήσιμων 32 bytes τα πηγαίνει καλύτερα από όλα τα υπόλοιπα. Το block των 16 bytes όμως, λόγω των συχνών αλλαγών block στον κώδικα (και επικύρωση αυτών), δεν τα πάει το ίδιο καλά.
Το τελευταίο που πρέπει να τονίσουμε στη συγκεκριμένη εκτέλεση είναι πως, ακόμα και στα μεγαλύτερα μεγέθη cache, δεν αγγίζουμε το baseline. Παραμένουμε στις 36-46 φορές πιο αργά από τον μη ασφαλή κώδικα, ενώ το baseline είναι ~16-21. Βλέπουμε λοιπόν, πως σε προβλήματα που κάνουν κακή χρήση της cache, δεν μπορούμε να αποφύγουμε σημαντικό αριθμό επικυρώσεων MAC, ακόμα κι αν βάλουμε μεγάλη cache. Εδώ αυτό συμβαίνει διότι στον δεύτερο πίνακα που πολλαπλασιάζουμε, προσπελαύνουμε ένα block δεδομένων, και έπειτα ζητάμε το επόμενο στοιχείο στην ίδια στήλη. Λόγω του τρόπου οργάνωσης των πινάκων στη μνήμη, πλέον ζητάμε διαφορετικό block, άρα (μιας και δύσκολα θα βρίσκεται στην cache), επιβάλλουμε MAC verification. Αυτό συνεπάγεται μεγαλύτερους χρόνους εκτέλεσης.

6.6: Υπολογισμός Ορίζουσας Πίνακα

Το τέταρτο και τελευταίο πρόβλημα που χρησιμοποιήσαμε ως μετρική είναι αυτό του υπολογισμού της ορίζουσας ενός πίνακα [30].
Όπως και στα προηγούμενα, δεν επιλέξαμε την πιο αποδοτική λύση, αλλά μια απλή.
Η ορίζουσα λοιπόν ενός τετραγωνικού πίνακα δίνεται από τον εξής κανόνα:
α) Αν ο πίνακας είναι 2x2, τότε η ορίζουσα είναι $a_{11}*a_{22}-a_{12}*a_{21}$ όπου a_{ij} το στοιχείο στην γραμμή i και στήλη j του πίνακα.
β) Αν ο πίνακας είναι NxN, επιλέγουμε μία γραμμή ή στήλη (εμείς επιλέξαμε την πρώτη γραμμή), και για κάθε της στοιχείο υπολογίζουμε

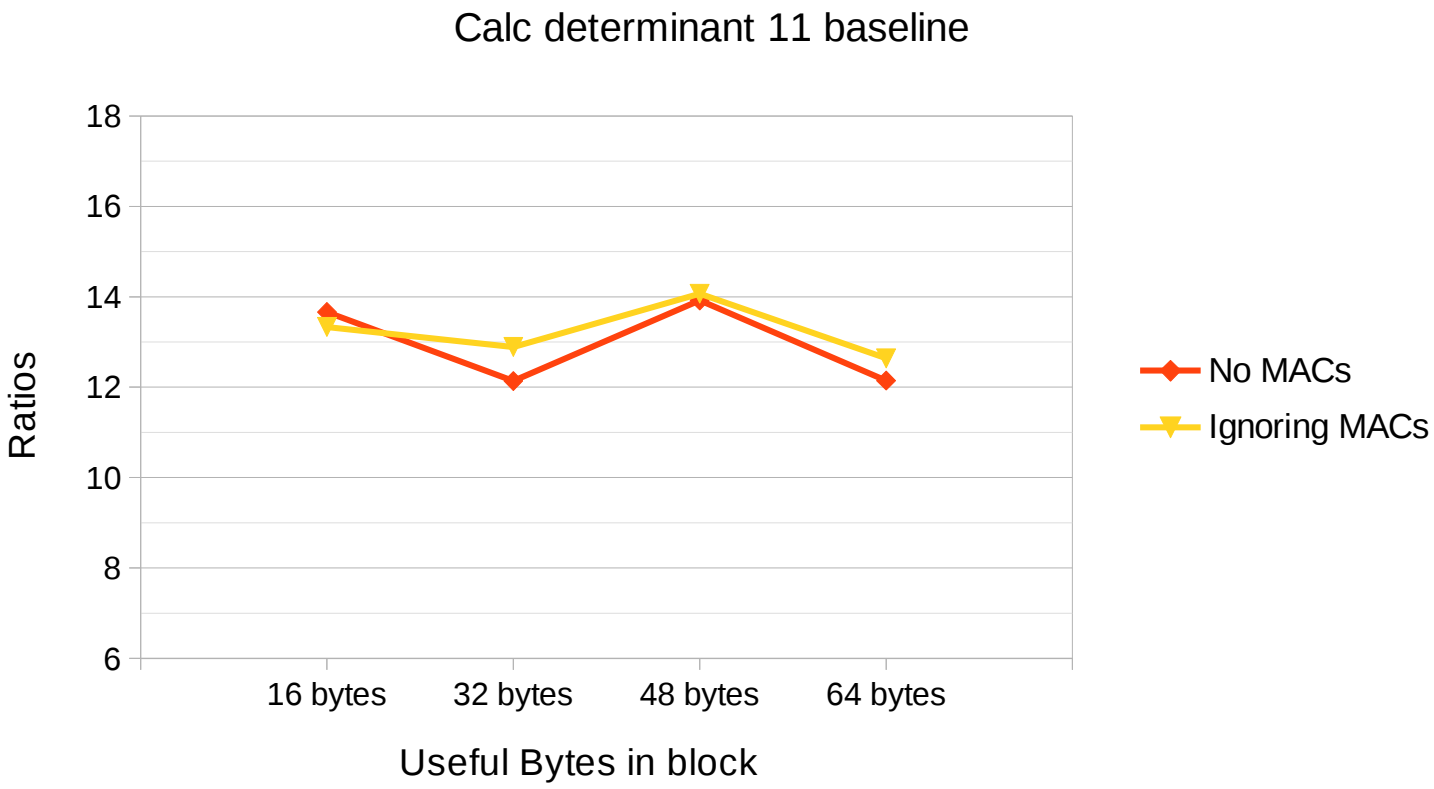
i) Το $(-1)^{i+j}$

ii) Την ορίζουσα του πίνακα (αναδρομικά) που προκύπτει σβήνοντας τη γραμμή i και τη στήλη j.

Αν πολλαπλασιάσουμε τα δύο αυτά στοιχεία μαζί και με το a_{ij} , προκύπτει η υπο-ορίζουσα για το στοιχείο στη θέση i,j. Αν προσθέσουμε τις υπο-ορίζουσες για όλα τα στοιχεία της γραμμής ή στήλης που επιλέξαμε, έχουμε την ορίζουσα του πίνακα.

Η ορίζουσα όπως βλέπουμε είναι ένα πρόβλημα το οποίο συνδυάζει αναδρομική κλήση συναρτήσεων και προσπέλαση στη μνήμη, και γι'αυτό το επιλέξαμε. Για να συμπεριλάβουμε και αυτές τις δύο ιδιότητες, για κάθε υπο-ορίζουσα τη στιγμή που σβήνουμε την γραμμή i και τη στήλη j, δεν την υπολογίζουμε πάνω στον ίδιο NxN πίνακα, αλλά αντιγράφουμε τον (N-1)x(N-1) πίνακα και την υπολογίζουμε στον αντιγραμμένο. Έτσι

η χρησιμοποίηση της μνήμης είναι σαφώς μεγαλύτερη και συχνότερη, αλλά ακριβώς αυτό θέλουμε να μετρήσουμε.
Θέσαμε το μέγεθος του πίνακα να είναι N=11.

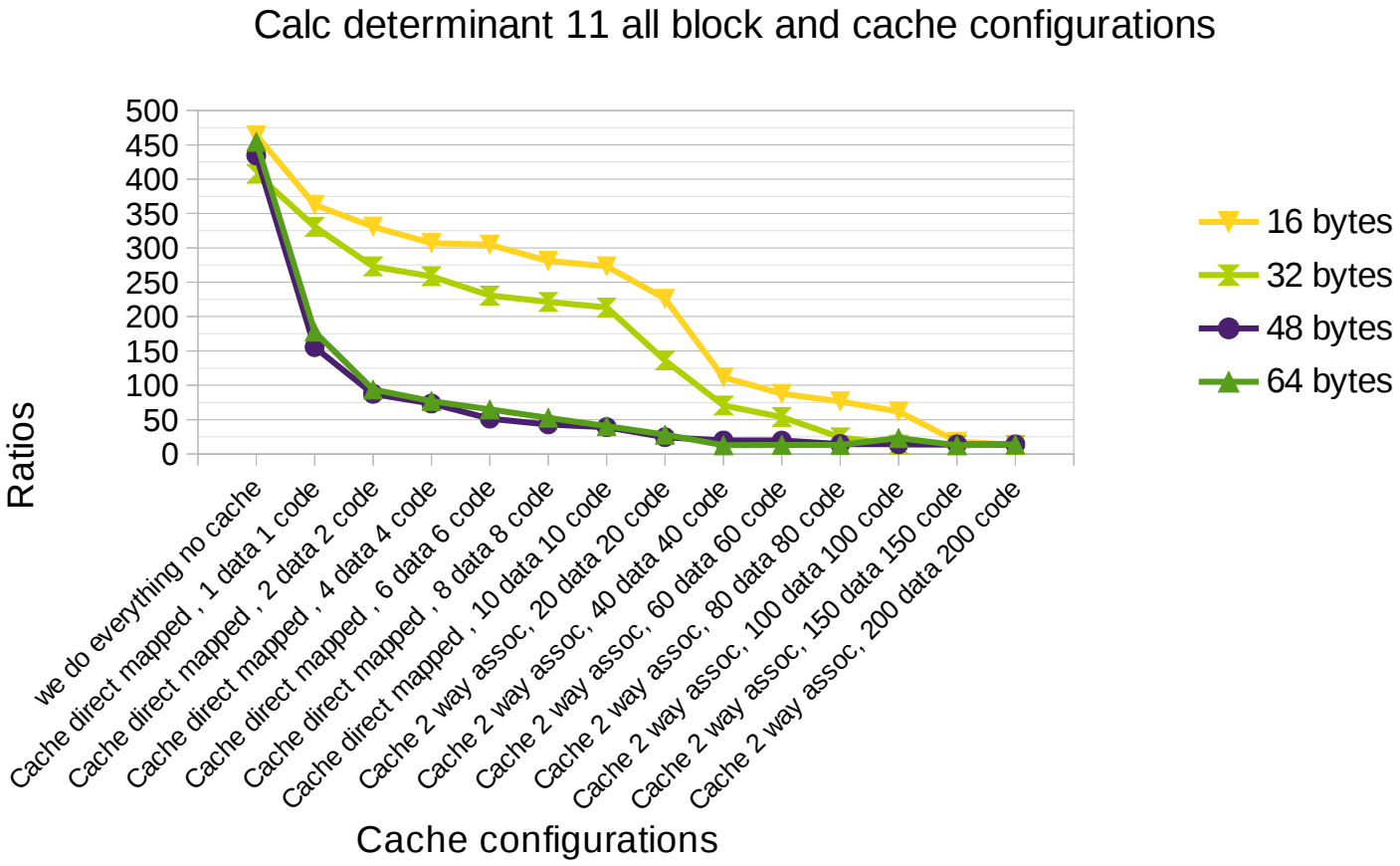


Σχήμα 6-10. Το baseline του υπολογισμού ορίζουσας.

Δεν υπάρχει κάτι ιδιαίτερο και πολύ διαφορετικό να σχολιάσουμε πάνω στο συγκεκριμένο γράφημα. Το baseline του υπολογισμού ορίζουσας βρίσκεται ανάμεσα στους πύργους του Ανόι και στον πολλαπλασιασμό πινάκων, με τιμές να κυμαίνονται από 12 ως και 14 φορές πιο αργά από το μη ασφαλές πρόγραμμα.

Ακολουθούν οι μετρήσεις όταν ενεργοποιούμε τα MACs. Στο γράφημα αυτό βλέπουμε να έχουμε γι άλλη μια φορά άσχημη απόδοση όταν δεν έχουμε cache (~400-470 φορές πιο αργά από το μη ασφαλές πρόγραμμα), αλλά σταδιακά η καθυστέρηση πέφτει και όπως και στα 2 πρώτα benchmarks, στα μεγάλα μεγέθη cache αγγίζουμε το baseline με τα MACs να μην μας καθυστερούν καθόλου.
Επίσης βλέπουμε πως για μικρά μεγέθη block, αργούμε να φτάσουμε στο baseline. Το πρόβλημα όμως του υπολογισμού ορίζουσας έχει ένα συνδυασμό των ιδιοτήτων του πολλαπλασιασμού πινάκων (το οποίο δεν φτάνει στο baseline), και των πύργων του Ανόι (το οποίο φτάνει). Έτσι λοιπόν εξηγείται η αργή σύγκλιση.
Όπως και στα προηγούμενα benchmarks, βλέπουμε τα blocks των οποίων τα χρήσιμα bytes έχουν μέγεθος 48 και 64 να δίνουν “μάχη” στα διάφορα μεγέθη cache, με το block μεγέθους

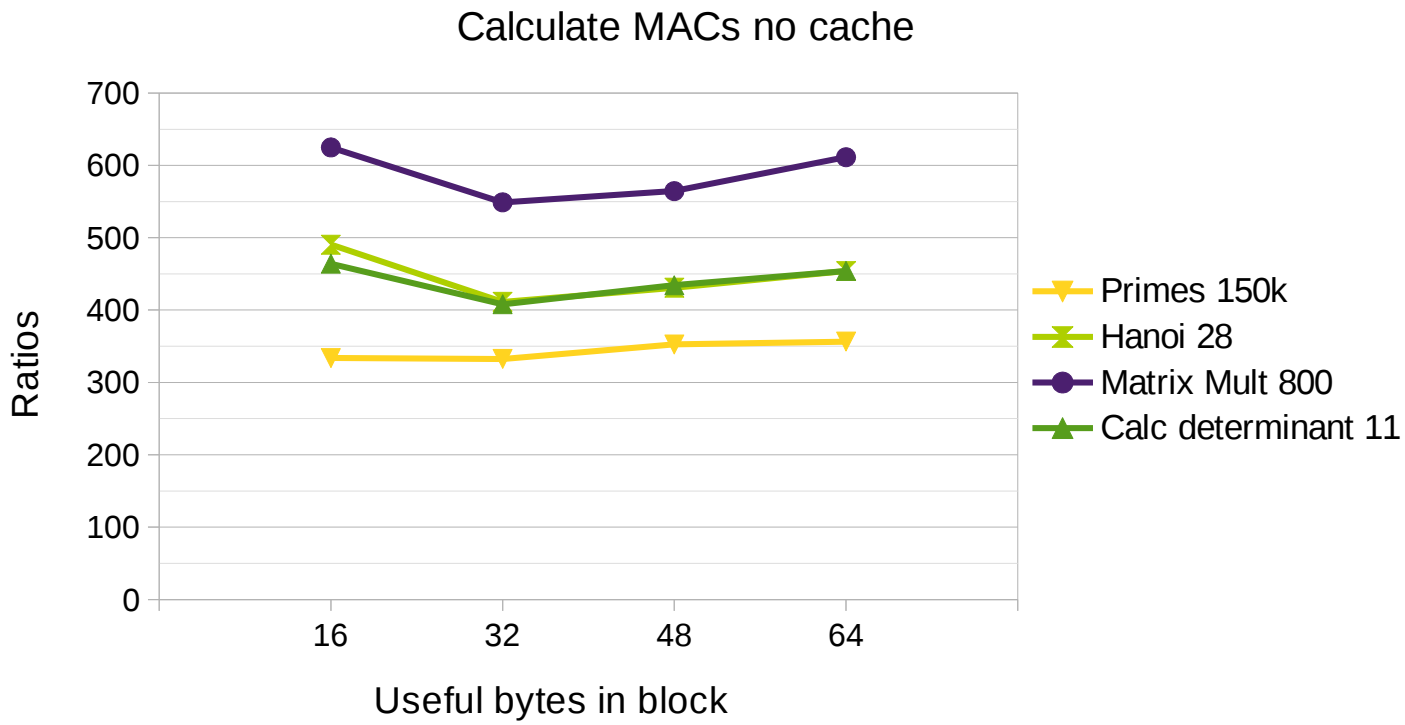
48 να προκύπτει κάποιες φορές καλύτερο. Αυτό όπως είπαμε είναι λόγω δύο αντικρουόμενων παραγόντων: Από τη μια το μεγάλο block επιτρέπει πολλές εντολές κώδικα να τρέξουν χωρίς επαλήθευση MAC, αλλά από την άλλη η χρησιμοποίηση λίγων δεδομένων από το heap, το stack ή το data segment ανά block, οδηγεί σε υπολογισμό μεγάλων MACs για μικρή αναλογική προσπέλαση δεδομένων που πραγματικά χρειαζόμαστε.



Σχήμα 6-11. Το γράφημα χρόνου εκτέλεσης για τον υπολογισμό ορίζουσας.

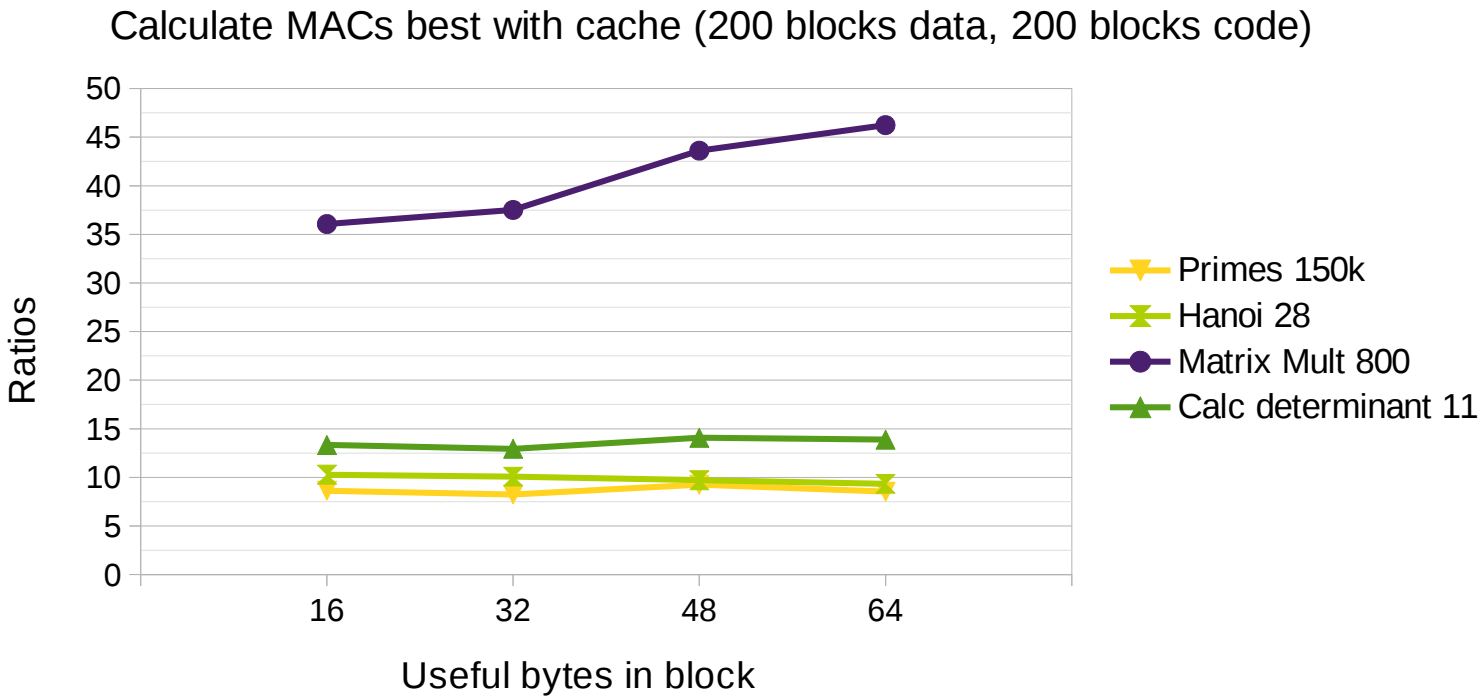
6.7: Συγκεντρωτικά Αποτελέσματα

Στην παράγραφο αυτή θα παρουσιάσουμε τα αποτελέσματά μας συγκεντρωμένα με τις στήλες να αντιπροσωπεύουν τα χρήσιμα bytes στο κάθε block, όπως επίσης θα παρουσιάσουμε και τις βέλτιστες τιμές τις οποίες επιτυγχάνουμε, για λόγους εποπτείας. Τα αποτελέσματα είναι τα ίδια με πριν, απλά συγκεντρωμένα διαφορετικά.
Ξεκινάμε με το γράφημα που παρουσιάζει τους χρόνους εκτέλεσης χωρίς ασφαλή cache.



Σχήμα 6-12. Οι καθυστερήσεις χωρίς χρήση ασφαλούς cache.

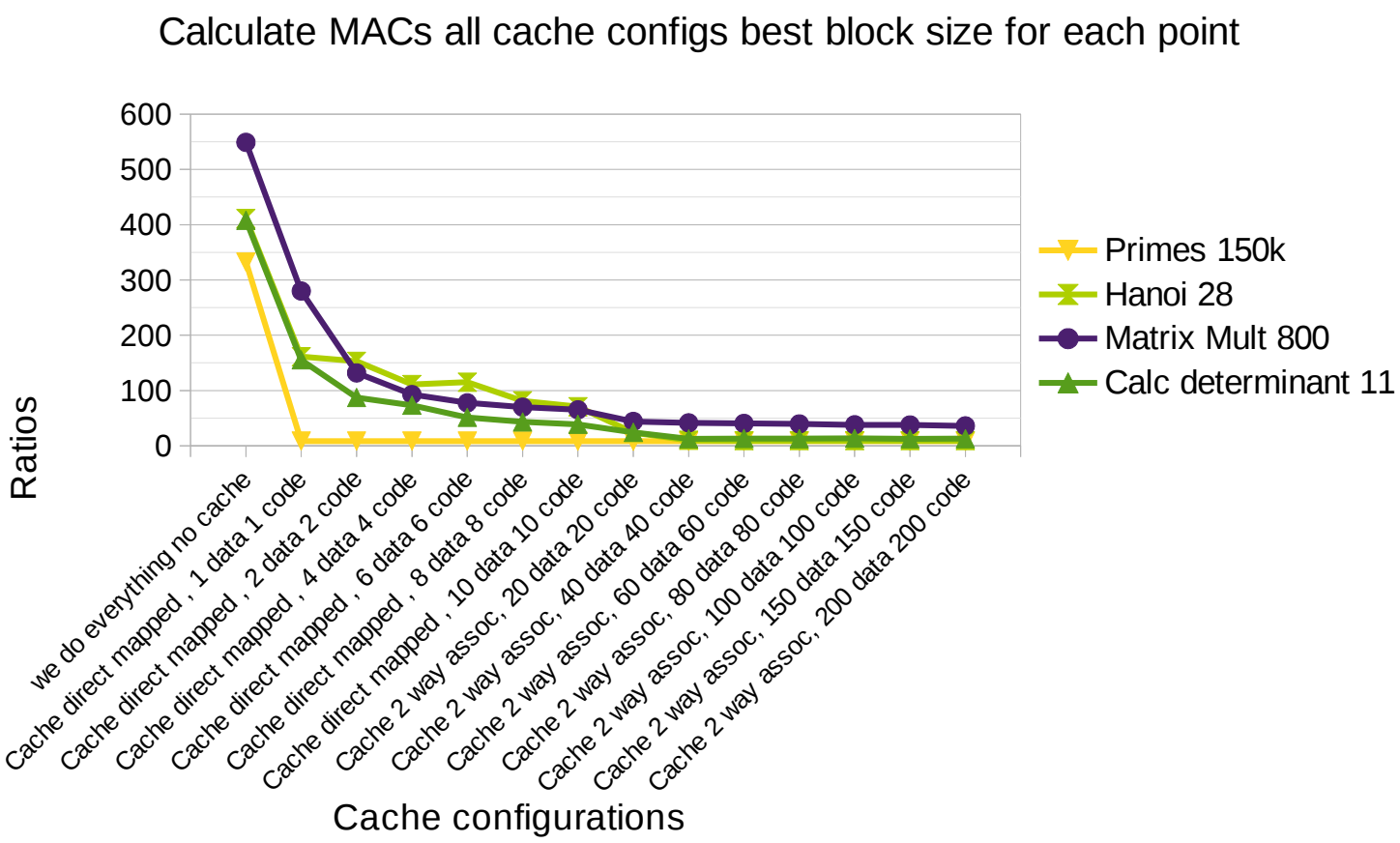
Είναι προφανές πως τα νούμερα είναι πολύ μεγάλα.
Ακολουθεί το γράφημα που παρουσιάζει τις καλύτερες τιμές που πετυχαίνουμε με τη χρήση cache (άρα για 200 blocks στον κώδικα και 200 στα δεδομένα).



Σχήμα 6-13. Οι καθυστερήσεις με χρήση του μεγαλύτερου μεγέθους ασφαλούς cache.

Εδώ τα νούμερα (εκτός του πολλαπλασιασμού πινάκων) αγγίζουν το baseline. Στον πολλαπλασιασμό πινάκων τα μικρότερα blocks ευνοούνται με το μεγάλο μέγεθος cache, μιας και στα μεγαλύτερα blocks υπολογισμός των MACs παίρνει πολύ, ενώ ο αλγόριθμος χρειάζεται μόνο ένα μικρό κομμάτι από κάθε block.

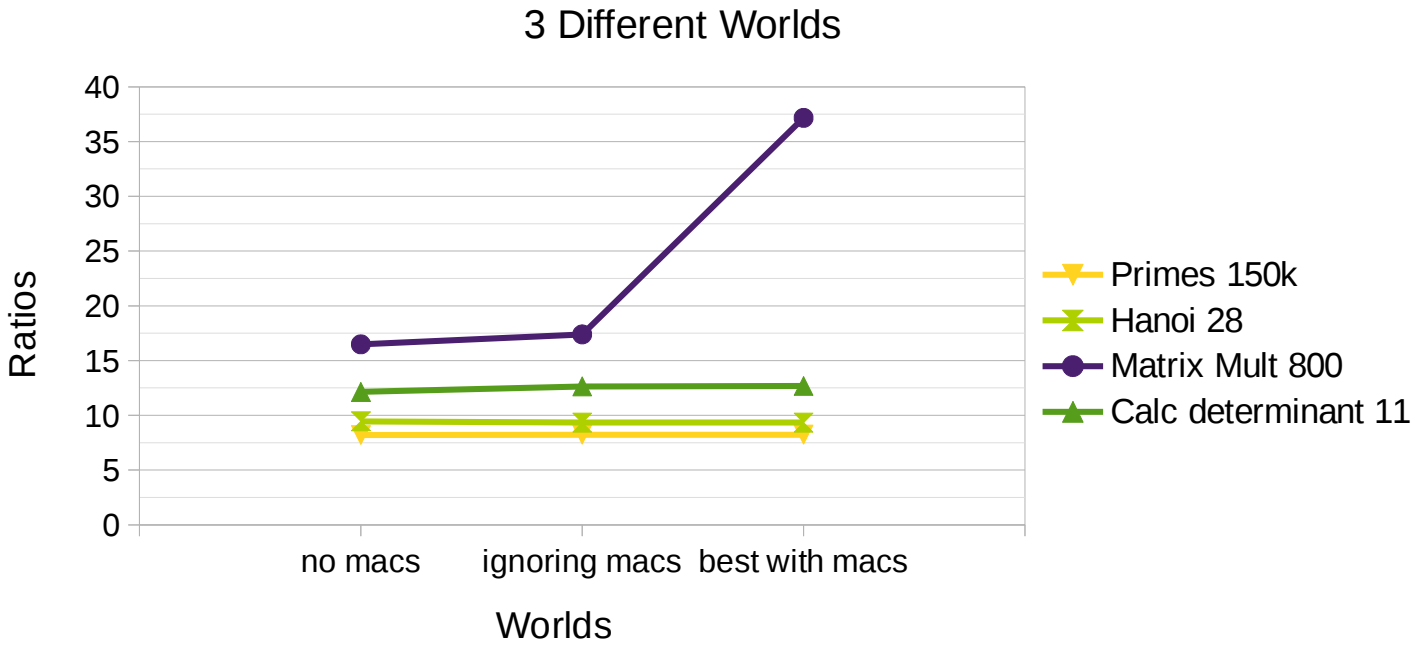
Το ακόλουθο γράφημα μας παρουσιάζει το βέλτιστο αποτέλεσμα (ως προς τα block sizes) για διάφορα μεγέθη cache ανά πρόβλημα.



Σχήμα 6-14. Η καλύτερη τιμή πάνω στα χρήσιμα bytes των blocks, για κάθε μέγεθος cache

Τέλος, παρουσιάζουμε το καλύτερο αποτέλεσμα το οποίο μπορούμε να πάρουμε (κοιτάζοντας όλους τους συνδυασμούς blocks και cache), για τα 4 benchmarks, και συγκρίνουμε με το καλύτερο baseline για το καθένα (κάτω από το οποίο είναι αδύνατον να πέσουμε χρησιμοποιώντας λογισμικό).

Όπως βλέπουμε, πέραν του πολλαπλασιασμού πινάκων (ο οποίος κάνει κακή χρήση cache), το καλύτερο αποτέλεσμα που παίρνουμε χρησιμοποιώντας MACs είναι σχεδόν ίδιο με το αποτέλεσμα χωρίς αυτά.



Σχήμα 6-15. Σύγκριση του καλύτερου χρόνου baseline με τον καλύτερο χρόνο με τα MACs

6.8: Σχολιασμός αποτελεσμάτων

Ανακεφαλαιώνοντας, αναφέρουμε και πάλι τα βασικά σημεία και συμπεράσματα.
Τα γραφήματα κατ'αρχάς καταδεικνύουν ότι απλώς και μόνο εκτελώντας τα MACs, η συνολική καθυστέρηση είναι απαγορευτική.
Όταν όμως εισάγουμε την ασφαλή cache, η τελική καθυστέρηση πέφτει πάρα πολύ, ως το baseline (εκτός του πολλαπλασιασμού πινάκων) που είναι και το όριο το οποίο μας θέτει το λογισμικό.
Τα μικρά block sizes αργούν κάπως να φτάσουν σε χαμηλές τιμές, ενώ τα μεγαλύτερα φτάνουν πιο γρήγορα. Αυτό είναι λογικό μιας και τα μεγάλα block sizes περιέχουν πολλά δεδομένα τα οποία μπορούν μετά να χρησιμοποιηθούν χωρίς επικύρωση του αντίστοιχου MAC, αφού την κάνουμε την πρώτη φορά. Μετά από ένα σημείο όμως, αν δεν χρησιμοποιούμε όλα τα bytes ενός μεγάλου block αλλά μόνο ένα μικρό κομμάτι τους, πληρώνουμε πολύ ακριβά την επικύρωση του μεγάλου block και ο συνολικός χρόνος μπορεί να προκύψει μεγαλύτερος.

Η μέγιστη cache που χρησιμοποιήσαμε (200 blocks για κώδικα και 200 για δεδομένα) είναι μια ρεαλιστική επιλογή. Αν υπολογίσουμε το μέγεθός της cache στην περίπτωση αυτή, έχουμε:

*Max Cache Size= 200 * code_block_size + 200 * data_block_size = 2* 200* (64+32+16) = 44800 bytes*

Ένα τυπικό μέγεθος L1 cache σε πραγματικούς επεξεργαστές είναι 64kb (32kb για κώδικα και 32 kb για δεδομένα), άρα πετυχαίνουμε πολύ καλή επίδοση με λιγότερη cache από την πιο μικρή σε μέγεθος cache (την L1 δηλαδή) που υπάρχει στους πραγματικούς επεξεργαστές.

Ο χρόνος baseline είναι ο χρόνος στον οποίο δεν επικυρώνουμε MACs και δεν διασπάμε blocks κώδικα. Κάτω από αυτόν τον χρόνο δεν μπορούμε να πέσουμε επειδή χρησιμοποιούμε λογισμικό. Αν διαθέταμε ειδικό hardware, τότε ο χρόνος αυτός εκτιμούμε θα κατακρημνιζόταν και θα πλησίαζε τον χρόνο του μη ασφαλούς προγράμματος.

Τέλος, πέραν του πολλαπλασιασμού πινάκων ο οποίος κάνει κακή χρήση της cache, βλέπουμε πως ο καλύτερος χρόνος με τα MACs δεν διαφέρει στην πράξη από τον καλύτερο χρόνο χωρίς αυτά. Αυτό είναι και το πιο αξιόλογο αποτέλεσμα της παρούσας διπλωματικής, ότι δηλαδή με την απλή προσθήκη της ασφαλούς cache το όριο μεταξύ χρησιμοποίησης MACs και μη χρησιμοποίησης αυτών γίνεται δυσδιάκριτο, και το μόνο που απομένει για τη μέγιστη ταχύτητα είναι η υλοποίηση της ασφαλούς CPU σε hardware.

Κεφάλαιο 7: Μελλοντική έρευνα

7.1: Υλοποίηση σε QEMU

Το άμεσο προφανές βήμα για την προσομοίωση του ασφαλούς σχήματος είναι να υλοποιήσουμε το ίδιο το hardware σε λογισμικό, μέσω ενός emulator. Ο QEMU (Quick Emulator) είναι μια ανοιχτού κώδικα λύση, η οποία προσφέρεται στο να προσομοιώνει CPUs (υπαρκτές ή φανταστικές). Ο QEMU μπορεί να κάνει χρήση επεκτάσεων του επεξεργαστή (όπως το KVM) που κατορθώνουν να εκτελούν το πρόγραμμα μέσω αυτού σε σχεδόν native ταχύτητα.
Μπορούμε λοιπόν να γράψουμε την ασφαλή CPU ως ένα νέο hardware σε QEMU. Αυτή η προσέγγιση θα δίνει ακόμα πιστότερες τιμές (ως προς την σωστή προσέγγιση μιας αληθινής ασφαλούς CPU) στα benchmarks, σε σχέση με την υλοποίηση της παρούσας διπλωματικής. Φυσικά αυτό θα απαιτήσει και αρκετό χρόνο τόσο με την εξοικείωση με τον QEMU, όσο και την συγγραφή της ασφαλούς CPU με έναν τρόπο με τον οποίον να την καταλαβαίνει ο QEMU.

Η υλοποίηση σε QEMU δίνει τα εξής πλεονεκτήματα:
α) Μπορούμε να μετρήσουμε με αρκετά μεγάλη ακρίβεια τον χρόνο που παίρνουν οι λειτουργίες που είναι υλοποιημένες σε hardware (πχ getters/setters ή το άλμα πάνω από τα keyshares+MACs στον κώδικα). Αυτό είτε μπορεί να γίνει real-time, είτε ακόμα καλύτερα να μετρηθεί το πλήθος των “κλήσεων” του hardware, και offline να βρεθεί ο χρόνος που παίρνει κάθε μία κλήση, και να πολλαπλασιάσουμε τα νούμερα. Από τη στιγμή που θα βρούμε τον χρόνο που παίρνει το hardware, μπορούμε να εκτιμήσουμε και πόσο θα παίρνει σε υλοποίηση σε κύλωμα. Ο υπόλοιπος χρόνος θα είναι χρόνος που παίρνει η εκτέλεση, και θα είναι υπολογισμένος σωστά.
β) Δεν χρειάζονται patches στον κώδικα ή ειδικές συναρτήσεις που υλοποιούν το hardware. Όλα θα τα κάνει ο QEMU.
γ) Δεν χρειάζεται να βάζουμε περιορισμούς στα benchmarks που θέλουμε να προσομοιώσουμε. Ο QEMU μπορεί real-time να μεταφράζει τις διευθύνσεις προσπέλασης με τις “ασφαλείς”, άρα μπορούμε να τρέξουμε και εξωτερικές βιβλιοθήκες (όπως η libC) ή και τον ίδιο τον Linux Kernel, μιας και ο QEMU αυτόματα θα έχει ασφαλίσει όλη την μνήμη χωρίς να χρειαστεί να κάνουμε κάτι.
δ) Μπορούμε να χρησιμοποιήσουμε “εξωτικές” εντολές της αρχιτεκτονικής, οι οποίες αγγίζουν τη μνήμη σε μεγάλα κομμάτια, χωρίς φόβο ότι θα χαλάσουν τα keyshares ή τα MACs. Την σωστή προσαρμογή τους θα την αναλαμβάνει ο QEMU, αφού τον προγραμματίσουμε κατάλληλα.

7.2: Υλοποίηση σε πραγματικό hardware

Όπως είναι λογικό, το τελευταίο βήμα για να έχουμε την απόλυτα σωστή εκτίμηση για το πόσο παίρνει μια ασφαλής CPU να τρέξει ένα πρόγραμμα, είναι να έχουμε την ασφαλή CPU μπροστά μας ως φυσική υλοποίηση. Μια τέτοια κατασκευή θα μπορούσε να γίνει είτε

σε ARM είτε σε x86 (με padded ειδικές εντολές για συμπλήρωση σταθερού μεγέθους block στη δεύτερη περίπτωση), και αυτόματα να ασφαλίζει όλη τη μνήμη του συστήματος πριν αρχίζει να τρέχει κάτι μέσα της.
Βεβαίως η υλοποίηση σε hardware χρειάζεται πρώτα από όλα χρήματα εκτός από χρόνο, μιας και πρέπει να κατασκευαστούν οι ειδικές πλακέτες σε κατάλληλο εργαστήριο.
Εκτίμησή μας είναι ότι οι επιπρόσθετες καθυστερήσεις που εμείς αντιμετωπίσαμε θα είναι ελάχιστες αν είναι υλοποιημένες σωστά σε κύκλωμα, και η γενική καθυστέρηση δεν θα είναι πιο μεγάλη από 2 φορές την πραγματική εκτέλεση του προγράμματος.

Βιβλιογραφία

[1] Lipton Richard, Ostrovsky Rafail, Zikas Vassilis: “Provably Secure Virus Detection: Using The Observer Effect Against Malware”. ICALP 2016, http://drops.dagstuhl.de/opus/volltexte/2016/6311/

[2] Malware. https://en.wikipedia.org/wiki/Malware

[3] Antivirus Software. https://en.wikipedia.org/wiki/Antivirus_software

[4] Buffer overflow attacks. https://en.wikipedia.org/wiki/Buffer_overflow

[5] Code injection attacks. https://en.wikipedia.org/wiki/Code_injection

[6] Race Condition. https://en.wikipedia.org/wiki/Race_condition

[7] Privilege Escalation. https://en.wikipedia.org/wiki/Privilege_escalation

[8] Row Hammer attack. https://en.wikipedia.org/wiki/Row_hammer

[9] Cryptographic Hash Function. https://en.wikipedia.org/wiki/Cryptographic_hash_function

[10] Message Authentication Code. https://en.wikipedia.org/wiki/Message_authentication_code

[11] Symmetric Cryptography. https://en.wikipedia.org/wiki/Symmetric-key_algorithm

[12] Block Cipher Mode of Operation. https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation

[13] Public-key Cryptography. https://en.wikipedia.org/wiki/Public-key_cryptography

[14] AES Algorithm (Advanced Encryption Standard) https://en.wikipedia.org/wiki/Advanced_Encryption_Standard

[15] CBC-MAC scheme. https://en.wikipedia.org/wiki/CBC-MAC

[16] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. 1998. “StackGuard: Automatic Adaptive Detection and Prevention of Buffer-overflow Attacks”. https://dl.acm.org/citation.cfm?id=1267549.1267554

[17] Buffer Overflow Protection. https://en.wikipedia.org/wiki/Buffer_overflow_protection

[18] Data Execution Prevention. https://en.wikipedia.org/wiki/Executable_space_protection

[19] Address Space Layout Randomization. https://en.wikipedia.org/wiki/Address_space_layout_randomization

[20] Error Correcting Memory. https://en.wikipedia.org/wiki/ECC_memory

[21] Protecting physical memory from radiation. https://en.wikipedia.org/wiki/Radiation_hardening

[22] Time-of-check-to-time-of-use. https://en.wikipedia.org/wiki/Time_of_check_to_time_of_use

[23] Elliptic Curve Cryptography. https://www.globalsign.com/en/blog/elliptic-curve-cryptography/

[24] Github code of diploma thesis. https://github.com/code-injection-detection/gcc-Linux-Implementation

[25] Cache types explained. http://www.computerweekly.com/feature/Write-through-write-around-write-back-Cache-explained

[26] x86 Calling Conventions. https://en.wikipedia.org/wiki/X86_calling_conventions

[27] Tower of Hanoi. https://en.wikipedia.org/wiki/Tower_of_Hanoi
[28] Tower of Hanoi Play. http://www.zylla.wipos.p.lodz.pl/games/hano10e.html
[29] Matrix Multiplication. https://en.wikipedia.org/wiki/Matrix_multiplication
[30] Matrix Determinant. https://en.wikipedia.org/wiki/Determinant