National Technical University of Athens
School of Electrical and Computer Engineering
Division of Computer Science

Accelerating the Secure Boot Process in Modern Microcontrollers

Diploma Thesis

of

THRASYVOULOS F. ILIADIS

Supervisor: Dionusios Pnevmatikatos
Professor

Athens, March 2023

National Technical University of Athens
School of Electrical and Computer Engineering
Division of Computer Science

Accelerating the Secure Boot Process in Modern Microcontrollers

Diploma Thesis

of

THRASYVOULOS F. ILIADIS

Supervisor: Dionusios Pnevmatikatos
Professor

Approved by the examination committee on 16th March 2023.

(Signature)	*(Signature)*	*(Signature)*
.....	.....	.....
Dionusios Pnevmatikatos	Georgios Goumas	Nectarios Koziris
Professor	Associate Professor	Professor

Athens, March 2023

National Technical University of Athens
School of Electrical and Computer Engineering
Division of Computer Science

DISCLAIMER ON ACADEMIC ETHICS AND INTELLECTUAL PROPERTY RIGHTS

Being fully aware of the implications of copyright laws, I expressly state that this diploma thesis, as well as the electronic files and source codes developed or modified in the course of this thesis, are solely the product of my personal work and do not infringe any rights of intellectual property, personality and personal data of third parties, do not contain work / contributions of third parties for which the permission of the authors / beneficiaries is required and are not a product of partial or complete plagiarism, while the sources used are limited to the bibliographic references only and meet the rules of scientific citing. The points where I have used ideas, text, files and / or sources of other authors are clearly mentioned in the text with the appropriate citation and the relevant complete reference is included in the bibliographic references section. I fully, individually and personally undertake all legal and administrative consequences that may arise in the event that it is proven, in the course of time, that this thesis or part of it does not belong to me because it is a product of plagiarism.

(Signature)

. . . . . .
Thrasyvoulos F. Iliadis

16th March 2023

Abstract

In modern microcontrollers, the integrity of the system is guaranteed in the secure boot process. Specifically, every piece of the software is covered by the Secure Boot mechanism and is attested to not have been tampered with (maliciously or not). The secure boot step has been implemented to shield systems from various system attacks. However, in modern microcontrollers, this step is typically slow, thus degrading the clients' quality of service, while also being very safety critical in the automotive domain (e.g., might cause severe road accidents in cars). Even though prior works have tried to improve performance of the secure boot process by proposing probabilistic verification schemes or by running the secure boot step in the background (not in the critical path), they only utilize one core of the system. In contrast, modern architectures comprise multiple cores. which so far remain idle and are not used in the secure boot step.

In this work, we accelerate the verification step in the secure boot process in modern microcontrollers by leveraging all the available cores of the underlying hardware. We make two key contributions. First, we extensively characterize and evaluate various cryptographic algorithms, which are widely-used in different domains, in the verification step of the secure boot process. Second, we accelerate the secure boot process via two novel verification schemes: i) a parallel deterministic verification scheme and ii) a parallel probabilistic verification scheme. Both schemes are running on using all the available cores of the system. We evaluate our parallel verifications schemes in two use case scenarios: i) in a microcontroller manufactured by a lead company tailored for the automotive domain, and ii) the rocket RISC-V chip. We demonstrate that our deterministic parallel scheme provides 3.7x performance improvement in the verification step, and 1.4x performance improvement in the end-to-end secure boot process over the single core-baseline using 6 cores on the microcontroller designed for automotive scenarios. On the rocket RISC-V chip, our parallel deterministic and probabilistic verification schemes improve performance by 2.2x using 8 cores and by 1.8x cores using 14 cores over the single-core baseline scheme.

Keywords

Secure Boot, Parallelism, Hashing Algorithm, Automotive, TPM

Περίληψη

Στους σύγχρονους μικροελεγκτές, η ακεραιότητα του συστήματος διασφαλίζεται μέσω της διαδικασίας της Ασφαλούς Εκκίνησης (Secure Boot). Συγκεκριμένα, κάθε κομμάτι του λογισμικού καλύπτεται από τον μηχανισμό ασφαλούς εκκίνησης και πιστοποιείται ότι δεν έχει αλλοιωθεί (κακόβουλα ή μη). Η διαδικασία της Ασφαλούς Εκκίνησης έχει υλοποιηθεί με σκοπό τη θωράκιση των συστημάτων από διάφορες επιθέσεις. Ωστόσο, στους σύγχρονους μικροελεγκτές, αυτή η διαδικασία είναι συνήθως αργή, υποβαθμίζοντας έτσι την ποιότητα των συστημάτων, ενώ παραμένει κρίσιμη για την ασφάλεια στον τομέα της αυτοκινητοβιομηχανίας (π.χ. μπορεί να προκαλέσει σοβαρά τροχαία ατυχήματα σε αυτοκίνητα). Παρόλο που προηγούμενες εργασίες έχουν προσπαθήσει να βελτιώσουν την απόδοση της διαδικασίας Ασφαλούς Εκκίνησης προτείνοντας πιθανοτικά σχήματα επαλήθευσης ή εκτελώντας ελέγχους στο παρασκήνιο μετά την εκκίνηση, χρησιμοποιούν μόνο έναν πυρήνα του συστήματος. Αντίθετα, οι σύγχρονες αρχιτεκτονικές προσφέρουν πολλαπλούς πυρήνες, οι οποίοι παραμένουν αδρανείς και δεν χρησιμοποιούνται κατά τη διαδικασία της Ασφαλούς Εκκίνησης.

Στην παρούσα εργασία, επιταχύνουμε το βήμα επαλήθευσης στη διαδικασία Ασφαλούς Εκκίνησης σε σύγχρονους μικροελεγκτές αξιοποιώντας όλους τους διαθέσιμους πυρήνες του υλικού. Στην εργασία μας κάνουμε δύο βασικές συνεισφορές. Πρώτον, χαρακτηρίζουμε και αξιολογούμε εκτενώς διάφορους αλγορίθμους κατακερματισμού, οι οποίοι χρησιμοποιούνται ευρέως σε διάφορους τομείς, ωχ προς το βήμα επαλήθευσης στη διαδικασία Ασφαλούς Εκκίνησης. Δεύτερον, επιταχύνουμε τη διαδικασία μέσω δύο νέων σχημάτων επαλήθευσης: i) ένα παράλληλο ντετερμινιστικό και ii) ένα παράλληλο πιθανοτικό σχήμα επαλήθευσης. Και τα δύο σχήματα εκτελούνται με τη χρήση όλων των διαθέσιμων πυρήνων του συστήματος. Αξιολογούμε τα σχήματα παράλληλης επαλήθευσης σε δύο σενάρια χρήσης: i) σε έναν μικροελεγκτή που κατασκευάζεται από κορυφαία εταιρεία προσαρμοσμένο για τον τομέα της αυτοκινητοβιομηχανίας, και ii) ένα rocket RISC-V chip. Δείχνουμε ότι το ντετερμινιστικό παράλληλο σχήμα μας παρέχει 3,7x βελτίωση των επιδόσεων στο βήμα επαλήθευσης και 1,4x βελτίωση των επιδόσεων στο συνολικό χρόνο εκκίνησης σε σχέση με το βασικό μοντέλο με έναν πυρήνα, χρησιμοποιώντας 6 πυρήνες στον μικροελεγκτή για την αυτοκινητοβιομηχανία. Στο rocket RISC-V chip, τα δικά μας παράλληλα ντετερμινιστικά και πιθανοτικά σχήματα επαλήθευσης βελτιώνουν την απόδοση κατά 2,2x χρησιμοποιώντας 8 πυρήνες και κατά 1,8x χρησιμοποιώντας 14 πυρήνες, σε σχέση με το βασικό σχήμα του ενός πυρήνα.

Λέξεις Κλειδιά

Ασφαλής Εκκίνηση, Παραλληλισμός, Αλγόριθμοι Κατακερματισμού, Αυτοκίνηση, Εμπιστευμένη Υπολογιστική

to my family

Acknowledgements

I would like to wholeheartedly thank Mr Dionisios Pnevmatikatos for his continuous support and excellent communication throughout this whole thesis.

I would also like to especially thank Christina Giannoula for her advising and guidance, her feedback and all the support she gave me throughout this work, without which this thesis would not be complete. Also a special thank you to Nicolaos Papadopoulos who provided brilliant technical and research advice.

For the part of this work that was completed in Munich, I would like to thank all the colleagues and advisors in Huawei's TTE-DE TSP lab.

Finally, I am grateful to family and friends, for putting up with and supporting me, because this support got me through the hardest parts of this journey.

Athens, March 2023

Thrasyvoulos F. Iliadis

Table of Contents

List of Figures

List of Tables

Chapter 1

Περίληψη

1.1 Θεωρητικό Υπόβαθρο

1.1.1 Ασφάλεια και Εμπιστοσύνη

Τον τελευταίο μισό αιώνα, η αυτοκινητοβιομηχανία έχει κάνει σημαντική πρόοδο, με τα σύγχρονα αυτοκίνητα να διαθέτουν πάνω από 100 ηλεκτρονικές μονάδες ελέγχου (ECUs), που παρέχουν διάφορες υπηρεσίες [1]. Τα αυτοκίνητα γίνονται όλο και περισσότερο σαν ὑπολογιστές με ρόδες¨, με τη δυνατότητα να παρακολουθούν την εσωτερική τους κατάσταση και να αλληλεπιδρούν με τον εξωτερικό κόσμο μέσω επικοινωνίας αυτοκινήτου προς αυτοκίνητο (vehicle-to-vehicle) και αυτοκίνητου προς Ξ (vehicle-to-X) [2]. Ωστόσο, η βιομηχανία αντιμετωπίζει επίσης σημαντικές προκλήσεις ασφαλείας λόγω της αυξανόμενης εξάρτησης από ηλεκτρονικές συσκευές. Μερικά παραδείγματα συστημάτων σε ένα όχημα που βασίζονται άμεσα ή έμμεσα σε έναν ελεγκτή αποτελούν το σύστημα αντι-μπλοκαρίσματος τροχών (ABS), το σύστημα ελέγχου πρόσφυσης (TCS - Traction Control), τα συστήματα ψυχαγωγίας όπως ραδιόφωνα, συνδεσεις με κινητά τηλέφωνο, το διαδίκτυο ή το GPS, αλλά και έπειτα την αυτόνομη κίνηση, την όραση αυτοκινήτων κλπ. Με την ανατολή της εποχής του αυτόνομου αυτοκινήτου, η δυνατότητα γρήγορης και ασφαλούς επικοινωνίας με άλλα αυτοκίνητα ή συσκευές είναι ζωτικής σημασίας [1], οπότε πρέπει να αντιμετωπιστούν τα ζητήματα ασφαλείας όσο το δυνατόν γρηγορότερα.

Η αυτοκινητοβιομηχανία έχει γίνει μια επικερδής βιομηχανία για τους χάκερ καθώς τα αυτοκίνητα έχουν πλέον περισσότερες δυνατότητες υπολογιστικής επεξεργασίας. Αυτό σημαίνει ότι η επιφάνεια επίθεσης έχει επεκταθεί στον κυβερνοφυσικό τομέα, διευκολύνοντας τους επιτιθέμενους να παρενοχλούν τα οχήματα. Επιπλέον, οι προηγμένες εξελίξεις στο IT έχουν οδηγήσει σε ευπάθειες που μεταφέρονται στα αυτοκίνητα, διευρύνοντας σημαντικά την επιφάνεια επίθεσης [3]. Οι κακόβουλοι επιτιθέμενοι μπορούν να έχουν πρόσβαση στα οχήματα με διάφορους τρόπους, συμπεριλαμβανομένων των ρεπλαψ επιθέσεων στα κλειδιά ξεκλειδώματος αυτοκινήτου (fobs), της απομακρυσμένης πρόσβασης στα συστήματα ψυχαγωγίας του αυτοκινήτου, απομακρυσμένων προσπαθειών για πρόσβαση στο δίκτυο CAN του αυτοκινήτου και στην περίπτωση των αυτόνομων οχημάτων ακόμη περισσότερο έλεγχο.

Η ασφάλεια σε αυτά τα συστήματα έχει βαθιές ρίζες στην θεωρία της κρυπτογραφίας. Οι 4 ορισμοί-κλειδιά της κρυπτογραφίας που μας ενδιαφέρουν στη δουλειά μας είναι ακεραιότητα, αυθεντικότητα, εχεμύθεια και διαθεσιμότητα [4]. Η ακεραιότητα αναφέρεται στη διασφάλιση

ότι τα δεδομένα δεν έχουν παραβιαστεί ή αλλοιωθεί κατά τη διάρκεια της μετάδοσης από το ένα σημείο στο άλλο. Η αυθεντικότητα αναφέρεται στη διασφάλιση ότι η πηγή των δεδομένων είναι πράγματι αυτό που υποστηρίζει ότι είναι, η εχεμύθεια αναφέρεται στη διασφάλιση ότι τα δεδομένα δεν μπορούν να προσπελαστούν από μη εξουσιοδοτημένα μέρη, ενώ η διαθεσιμότητα αναφέρεται στην εγγύηση ότι τα δεδομένα μπορούν με σίγουρο τρόπο να προσπελαστούν από εξουσιοδοτημένα μέρη όταν απαιτείται. Αυτοί οι τέσσερις όροι είναι ουσιαστικοί για τη διασφάλιση της ασφάλειας των πληροφοριών που μεταδίδονται μέσω δικτύων επικοινωνίας.

Άλλη μια σημαντική ιδέα που αγκαλιάζει ολόκληρη τη δουλειά μας εδώ είναι η έννοια της ψηφιακής υπογραφής. Οι ψηφιακές υπογραφές εξασφαλίζουν τη γνησιότητα και την ακεραιότητα ψηφιακών δεδομένων χρησιμοποιώντας ένα ιδιωτικό κλειδί και έναν αλγόριθμο κατακερματισμού. Τα δεδομένα κατακερματίζονται για τη δημιουργία μιας μοναδικής αποτύπωσης, ενώ στη συνέχεια κρυπτογραφούνται με το ιδιωτικό κλειδί για τη δημιουργία μιας ψηφιακής υπογραφής. Ο παραλήπτης επαληθεύει την υπογραφή αποκρυπτογραφώντας τη ψηφιακή υπογραφή με το δημόσιο κλειδί και συγκρίνοντας τις αποτυπώσεις. Οι ψηφιακές υπογραφές χρησιμοποιούνται ευρέως στις ασφαλείς επικοινωνίες, στις συναλλαγές και στον έλεγχο λογισμικού για να παρέχουν ασφάλεια και εμπιστοσύνη. Στο πλαίσιο ασφαλούς εκκίνησης, χρησιμοποιούνται για την επαλήθευση της ακεραιότητας των εκάστοτε κομματιών λογισμικού.

Η εμπιστοσύνη και η εμπιστευμένη υπολογιστική (trust and trusted computing) είναι ένα πεδίο που σχετίζεται με τη διασφάλιση της ακεραιότητας του συστήματος. Η ρίζα της εμπιστοσύνης (root of trust) είναι μια ασφαλής οντότητα που εμπιστεύεται να εκτελέσει λειτουργίες που σχετίζονται με την ασφάλεια και να προστατεύσει ενάντια σε μη εξουσιοδοτημένη πρόσβαση ή αλλοίωση. Η αλυσίδα εμπιστοσύνης (chain of trust) περιγράφει τη διαδικασία θέσπισης εμπιστοσύνης σε όλα τα στοιχεία του συστήματος με σειριακό τρόπο, ξεκινώντας από μια ρίζα εμπιστοσύνης. Ένα παράδειγμα τέτοιας αλυσίδας εμπιστοσύνης βρίσκουμε στο σχήμα 1.1.

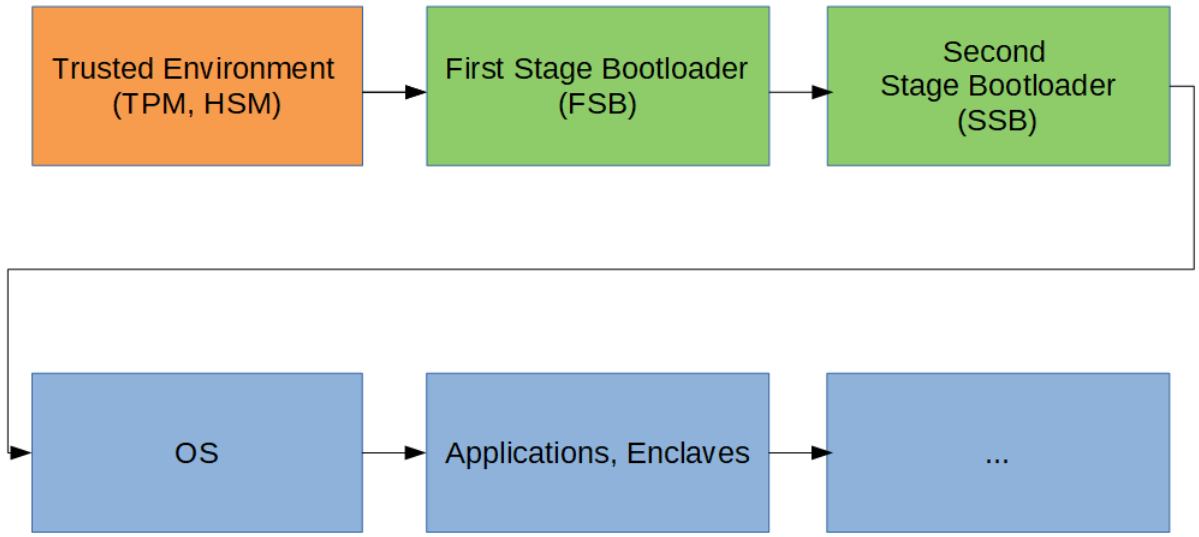

Figure 1.1. *Κάποια παραδοσιακά βήματα από τα οποία πιθανώς αποτελείται μία αλυσίδα εμπιστοσύνης σε ένα σύστημα, ξεκινώντας από μία ρίζα εμπιστοσύνης.*

Η Αξιόπιστη Πλατφόρμα (Trusted Platform Module, TPM) είναι ένα υλικού βάσης στοιχείο ασφαλείας που παρέχει ένα ασφαλές περιβάλλον εκτέλεσης και αποθηκεύει κρυπ-

τογραφημένα κλειδιά και άλλα ευαίσθητα δεδομένα. Είναι ο πιο συνηθισμένος τρόπος για τη δημιουργία μιας ρίζας εμπιστοσύνης που βασίζεται σε υλικό. Το TPM παρέχει αρκετά συγκεκριμένα χαρακτηριστικά, όπως ασφαλή αποθήκευση, ασφαλή εκκίνηση, απομακρυσμένη πιστοποίηση, σφραγισμένη αποθήκευση, δημιουργία και διαχείριση κλειδιών, και παραγωγή τυχαίων αριθμών. Μπορεί να χρησιμοποιηθεί για να διασφαλίσει την ακεραιότητα της διαδικασίας εκκίνησης, να προστατεύσει ευαίσθητα δεδομένα και να δημιουργήσει ένα αξιόπιστο κανάλι επικοινωνίας μεταξύ δύο συστημάτων. Πέρα από τα βασισμένα σε υλικό TPM, υπάρχουν επίσης εναλλακτικές βασισμένες σε λογισμικό που προσομοιάζουν τα πρότυπα ασφαλείας που προσφέρει ένα φυσικό TPM. Τα βασισμένα σε υλικό TPM έχουν αρκετά πλεονεκτήματα, όπως η παροχή υψηλού επιπέδου ασφάλειας, αλλά και κάποια μειονεκτήματα, όπως η πιθανότητα αποτυχίας του υλικού. Η ενσωμάτωση της εμπιστοσύνης σε ένα σύστημα προστατεύει από μια ποικιλία κυβερνοεπιθέσεων, συμπεριλαμβανομένων επιθέσεων στην αλυσίδα εφοδιασμού. Συνολικά, η χρήση αξιόπιστων υπολογιστικών πλατφορμών γίνεται όλο και πιο σημαντική στα σύγχρονα συστήματα, καθώς τα συστήματα με ενσωματωμένα TPM γίνονται η κανονικότητα ακόμη και στους προσωπικούς εμπορικούς υπολογιστές [5].

1.1.2 Ασφαλής Εκκίνηση

Περιγραφή

Η Ασφαλής Εκκίνηση είναι μια σημαντική λειτουργία ασφαλείας που διασφαλίζει την ακεραιότητα της διαδικασίας εκκίνησης επαληθεύοντας την ψηφιακή υπογραφή κάθε στοιχείου του φορτωτή εκκίνησης πριν από την εκτέλεσή του [6]. Αυτό εμποδίζει τον κακόβουλο και άλλο μη εξουσιοδοτημένο λογισμικό από το να φορτωθεί κατά τη διάρκεια της διαδικασίας εκκίνησης. Η Ασφαλής Εκκίνηση υλοποιείται από μια συνδυασμένη χρήση υλικού και λογισμικού στοιχείων, που θα μπορούσε να περιέχει το BIOS, το φορτωτή εκκίνησης ή και τομ πυρήνα του εκάστοτε λειτουργικού συστήματος.

Η Ασφαλής Εκκίνηση παρέχει υψηλό επίπεδο ασφαλείας για τη διαδικασία εκκίνησης και βοηθά στην προστασία του συστήματος από επιθέσεις που στοχεύουν στη διαδικασία εκκίνησης. Βεβαιώνεται ότι κατά τη διάρκεια της διαδικασίας εκκίνησης φορτώνεται μόνο αξιόπιστο λογισμικό και παρέχει μια ασφαλή βάση για το υπόλοιπο σύστημα. Η Ασφαλής Εκκίνηση γίνεται με τη συνδυαστική χρήση υλικού και λογισμικού, συμπεριλαμβανομένου του BIOS, του εκκινητή και του πυρήνα του λειτουργικού συστήματος.

Κατά τη διάρκεια της Ασφαλούς Εκκίνησης, το προ-υπολογισμένο ηαση κάθε συνιστώσας στη διαδικασία εκκίνησης αποθηκεύεται σε μια ασφαλή περιοχή αποθήκευσης που ονομάζεται Platform Configuration Registers (PCR) μέσα στο TPM της μητρικής πλακέτας του συστήματος. Το TPM είναι ένας αφιερωμένος κομμάτι υλικού που σχεδιάστηκε για να αποθηκεύει κρυπτογραφικά κλειδιά, να εκτελεί κρυπτογραφικές λειτουργίες και να διασφαλίζει την ακεραιότητα της διαδικασίας εκκίνησης, όπως είδαμε παραπάνω.

Υπάρχουν δύο τρόποι για την υλοποίηση της Ασφαλούς Εκκίνησης: Λειτουργία βασισμένη σε λογισμικό και λειτουργία βασισμένη σε υλικό. Η επαλήθευση βασισμένη σε λογισμικό περιλαμβάνει τη χρήση εργαλείων λογισμικού για την επαλήθευση της ακεραιότητας των στοιχείων εκκίνησης. Αυτό συνήθως γίνεται ελέγχοντας ψηφιακές υπογραφές, επαληθεύοντας checksums ή συγκρίνοντας τα στοιχεία με μια γνωστή καλή κατάσταση της μνήμης. Η επαλήθευση

βασισμένη σε λογισμικό είναι ευέλικτη και μπορεί να υλοποιηθεί σε μια ευρεία γκάμα πλατφορμών, αλλά είναι επίσης ευάλωτη σε επιθέσεις που μπορούν να καταστρέψουν το λογισμικό που χρησιμοποιείται για την επαλήθευση. Η επαλήθευση με βάση το υλικό περιλαμβάνει τη χρήση αποκλειστικών στοιχείων υλικού για τη διασφάλιση της ακεραιότητας των στοιχείων εκκίνησης. Αυτό συνήθως περιλαμβάνει τη χρήση μιας μονάδας αξιόπιστης πλατφόρμας για την αποθήκευση κρυπτογραφικών κλειδιών και την εκτέλεση κρυπτογραφικών πράξεων, καθώς και για τη μέτρηση και τη βεβαίωση της ακεραιότητας των στοιχείων εκκίνησης. Η επαλήθευση με βάση το υλικό είναι πιο ασφαλής από την επαλήθευση με βάση το λογισμικό, επειδή το υλικό είναι φυσικά απομονωμένο από το υπόλοιπο σύστημα, καθιστώντας πιο δύσκολο για τους επιτιθέμενους να παραβιάσουν κάποιο μέρος της διαδικασίας επαλήθευσης.

Τόσο η επαλήθευση με βάση το λογισμικό όσο και η επαλήθευση με βάση το υλικό έχουν τα πλεονεκτήματα και τα μειονεκτήματά τους. Η επαλήθευση με βάση το λογισμικό είναι πιο ευέλικτη και μπορεί να εφαρμοστεί σε μεγαλύτερο εύρος πλατφορμών, αλλά είναι επίσης πιο ευάλωτη σε επιθέσεις. Η επαλήθευση με βάση το υλικό είναι πιο ασφαλής, αλλά απαιτεί ειδικά εξαρτήματα υλικού και μπορεί να είναι πιο δύσκολο να υλοποιηθεί σε ορισμένες πλατφόρμες. Τελικά, η επιλογή μεταξύ επαλήθευσης βασισμένης σε λογισμικό και επαλήθευσης βασισμένης σε υλικό θα εξαρτηθεί από τις συγκεκριμένες απαιτήσεις του συστήματος και το επίπεδο ασφάλειας που απαιτείται.

Συνολικά, η Ασφαλής εκκίνηση παρέχει μια ασφαλή και αξιόπιστη βάση για το λειτουργικό σύστημα και συμβάλλει στην προστασία από κακόβουλο λογισμικό και άλλες απειλές ασφαλείας που μπορούν να θέσουν σε κίνδυνο το σύστημα. Καθώς τα συστήματα με ενσωματωμένα TPM γίνονται ο κανόνας ακόμη και μεταξύ των προσωπικών εμπορικών υπολογιστών, η χρήση αξιόπιστων υπολογιστικών πλατφορμών αποκτά ολοένα και μεγαλύτερη σημασία στα σύγχρονα συστήματα. Είναι σημαντικό να σημειωθεί ότι σε εφαρμογές όπου η ταχύτητα είναι το ζητούμενο, οι σχεδιαστές μπορεί να επιλέξουν να προτιμήσουν την ταχύτητα έναντι της ασφάλειας με διάφορους τρόπους.

Μοντέλα Ασφαλούς Εκκίνησης

Η ασφαλής εκκίνηση είναι ένα βασικό στοιχείο της ασφάλειας του συστήματος που περιλαμβάνει την επαλήθευση της ακεραιότητας των στοιχείων εκκίνησης κατά τη διαδικασία εκκίνησης. Υπάρχουν τρεις κύριοι τρόποι υλοποίησης της ασφαλούς εκκίνησης: Ντετερμινιστικό Secure Boot, Πιθανολογικό Secure Boot και Run-Then-Check.

Η ντετερμινιστική ασφαλής εκκίνηση περιλαμβάνει τη μέτρηση και την επαλήθευση ολόκληρης της περιοχής μνήμης για να εξασφαλιστεί η ακεραιότητά της. Η περιοχή μνήμης κατακερματίζεται και η τιμή κατακερματισμού ελέγχεται σε σχέση με το αποτέλεσμα που είναι αποθηκευμένο σε ασφαλή χώρο αποθήκευσης. Εάν οι κατακερματισμοί ταιριάζουν, η διαδικασία εκκίνησης συνεχίζεται - εάν όχι, η αποτυχία αντιμετωπίζεται αναλόγως. Τα σχήματα 1.2 έως 1.5 περιγράφουν μια τέτοια διαδικασία.

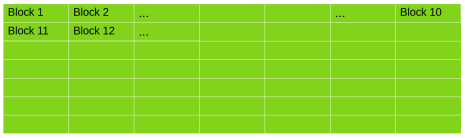

Figure 1.2. *Έλεγξε ολόκληρη τη μνήμη.*

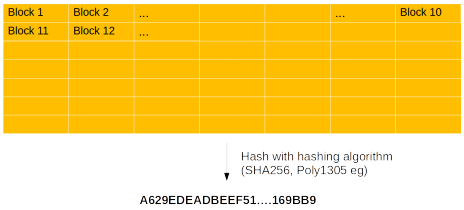

Figure 1.3. *Κατακερμάτισε ολόκληρο το χώρο και πάραξε μία υπογραφή.*

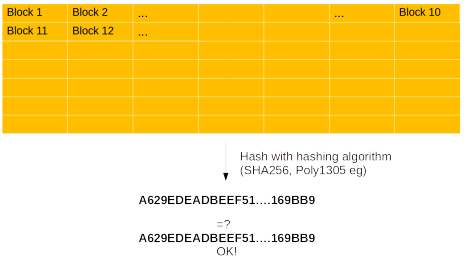

Figure 1.4. *Σύγκρινε την προκύπτουσα υπογραφή με την αποθηκευμένη στον ασφαλή χώρο.*

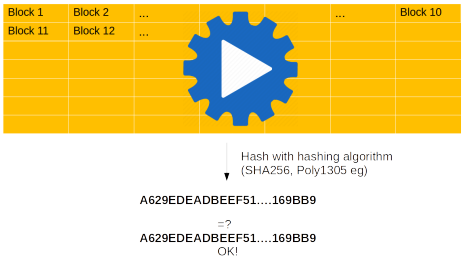

Figure 1.5. *Εάν ταιριάζουν το σύστημα εκκινεί, εάν όχι η αποτυχία χειρίζεται αναλόγως.*

Figure 1.6. *Στάδια Ντετερμινιστικού ελέγχου Ασφαλούς Εκκίνησης.*

Η πιθανολογική ασφαλής εκκίνηση χωρίζει την περιοχή μνήμης σε μικρότερα κομμάτια και επαληθεύει μόνο ένα υποσύνολο αυτών. Ο κατακερματισμός της σωστής κατάστασης αυτών των κομματιών υπολογίζεται και αποθηκεύεται στην ασφαλή μνήμη. Μετά την εκκίνηση της εφαρμογής, εκτελείται πλήρης σάρωση εικόνας στο παρασκήνιο, μειώνοντας την πιθανότητα να προκληθεί ζημιά από κακόβουλο λογισμικό. Ωστόσο, τα πιθανοτικά σχήματα προσφέρουν χαμηλότερο επίπεδο ασφάλειας από τα ντετερμινιστικά, καθώς υπάρχει πάντα η πιθανότητα κάποια αλλοίωση να έχει περάσει από τον έλεγχο. Στα σχήματα 1.7 έως 1.10 περιγράφεται μια τέτοια διαδικασία.

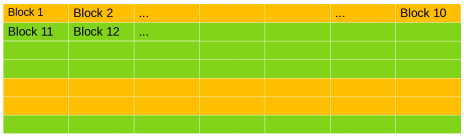

Figure 1.7. *Επίλεξε τυχαία κομμάτια μνήμης προς επαλήθευση.*

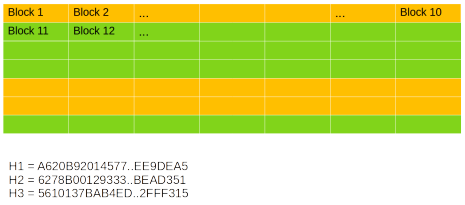

Figure 1.8. *Κατακερμάτισε το καθένα ξεχωριστά, παράγοντας πολλά hashes.*

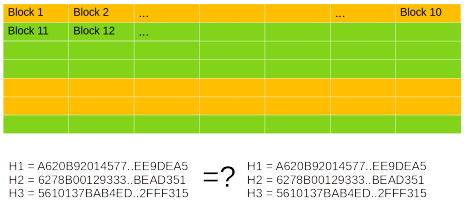

Figure 1.9. *Σύγκρινε το κάθε ένα από αυτά με το αντίστοιχο αποθηκευμένο, ή κατακερμάτισε ξανά τις υπογραφές μεταξύ τους για να παράξεις ένα τελικό hash.*

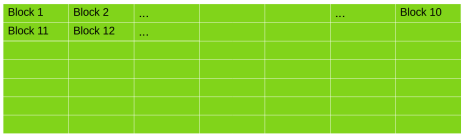

Figure 1.10. *Εάν ο έλεγχος επιτύχει, συνέχισε. Αλλιώς, χειρίσου την αποτυχία αναλόγως.*

Figure 1.11. *Στάδια Πιθανολογικού ελέγχου Ασφαλούς Εκκίνησης.*

Το Run-Then-Check αναβάλλει την ασφαλή ακολουθία εκκίνησης μέχρι να εκτελεστεί η εφαρμογή και εκτελείται συνήθως στο παρασκήνιο. Εάν βρεθεί κακόβουλος κώδικας σε αυτό το σημείο, η συσκευή πραγματοποιεί επαναφορά και αντιμετωπίζει την αποτυχία αναλόγως. Αυτά τα συστήματα βασίζονται στο μικρό χρονικό παράθυρο που δίνουν στον κώδικα του επιτιθέμενου για να εξισορροπήσουν την έλλειψη ασφάλειας. Ακόμη και στο χειρότερο σενάριο, ο χρόνος που έχει ο κακόβουλος δράστης για να εκτελέσει τις περαιτέρω επιθέσεις του είναι περιορισμένος, γεγονός που συνεπάγεται κάποια μορφή ασφάλειας. Στα σχήματα 1.12 έως 1.14 περιγράφεται μια τέτοια διαδικασία.

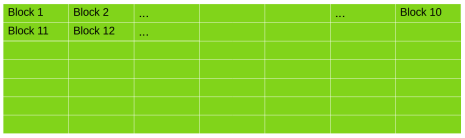

Figure 1.12. *Μην προχωρήσεις σε κάποιο έλεγχο πριν την εκκίνηση του συστήματος.*

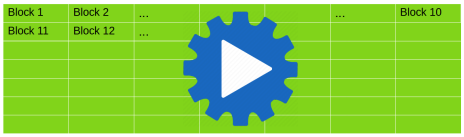

Figure 1.13. *Ξεκίνα το σύστημα ως έχει.*

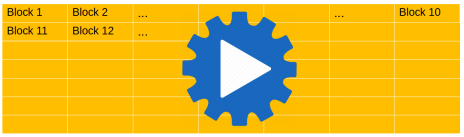

Figure 1.14. *Μόνον μετά την εκκίνηση τρέξε έναν πλήρη ντετερμινιστικό έλεγχο στο παρασκήνιο. Χειρίσου το αποτέλεσμα αναλόγως.*

Figure 1.15. *Στάδια Run-Then-Check ελέγχου Ασφαλούς Εκκίνησης.*

Κάθε σύστημα ασφαλούς εκκίνησης προσφέρει μια ισορροπία μεταξύ ασφάλειας και ταχύτητας και η επιλογή του συστήματος που θα χρησιμοποιηθεί εξαρτάται από τις βασικές απαιτήσεις του συστήματος. Ενώ η ντετερμινιστική ασφαλής εκκίνηση παρέχει το υψηλότερο επίπεδο ασφάλειας, είναι επίσης η πιο αργή. Τα πιθανοτικά σχήματα προσφέρουν χαμηλότερο επίπεδο ασφάλειας αλλά είναι ταχύτερα, ενώ το run-then-check παρέχει τον ταχύτερο χρόνο εκκίνησης αλλά το χαμηλότερο επίπεδο ασφάλειας.

1.2 Σχετική βιβλιογραφία

Η ασφαλής εκκίνηση είναι ένας βασικός μηχανισμός ασφαλείας που χρησιμοποιείται για να διασφαλίσει την ακεραιότητα ενός συστήματος κατά την εκκίνηση. Κατά τη διάρκεια της έρευνάς μας, εξετάζουμε ορισμένες από τις προηγούμενες εργασίες σχετικά με τα συστήματα ασφαλούς εκκίνησης για να αποκτήσουμε πληροφορίες σχετικά με τα δυνατά σημεία και τους περιορισμούς τους. Αρχικά εξετάζουμε ντετερμινιστικά σχήματα ασφαλούς εκκίνησης που παρέχουν μια αξιόπιστη διαδρομή για την αρχικοποίηση του συστήματος και την επαλήθευση της ακεραιότητας του λογισμικού του συστήματος. Στη συνέχεια, εξετάζουμε πιθανοτικά σχήματα ασφαλούς εκκίνησης που βασίζονται στη δημιουργία, αποθήκευση και επαλήθευση δακτυλικών αποτυπωμάτων από το χώρο μνήμης.

Η πιό κλασσική μορφή ντετερμινιστικού σχήματος που προτείνονται στη βιβλιογραφία είναι μια ασφαλής και αξιόπιστη αρχιτεκτονική εκκίνησης που χρησιμοποιεί ψηφιακές υπογραφές για την επαλήθευση της ακεραιότητας και της αυθεντικότητας των στοιχείων λογισμικού του συστήματος, συμπεριλαμβανομένου του λειτουργικού συστήματος, του υλικολογισμικού και άλλου λογισμικού επιπέδου συστήματος. Η αρχιτεκτονική διασφαλίζει ότι μόνο εξουσιοδοτημένο λογισμικό φορτώνεται στο σύστημα και μπορεί να ανιχνεύσει και να αποτρέψει την εκτέλεση κακόβουλου λογισμικού, συμπεριλαμβανομένων rootkits, κακόβουλου λογισμικού και άλλων μορφών μη εξουσιοδοτημένου κώδικα. Πρόκειται στην ουσία για την παραδοσιακή μορφή που παίρνει το secure boot εδώ και χρόνια στα συστήματά μας.

Αντίθετα, τα πιθανοτικά συστήματα ασφαλούς εκκίνησης βασίζονται στη δημιουργία και επαλήθευση δακτυλικών αποτυπωμάτων από το χώρο μνήμης. Το σχήμα εκκίνησης σε φέτες, για παράδειγμα, αντιμετωπίζει τη μνήμη ως ξεχωριστά λογικά μπλοκ μνήμης και τα τεμαχίζει για να πάρει κύτταρα μνήμης. Τα δακτυλικά αποτυπώματα δημιουργούνται ομοιόμορφα και αποθηκεύονται σε μη προστατευμένη μνήμη, ενώ τα κλειδιά αποθηκεύονται σε προστατευμένη μνήμη. Κατά τη φάση επαλήθευσης, ο αλγόριθμος επιλέγει τυχαία μια στήλη, υπολογίζει την υπογραφή και τη συγκρίνει με την αποθηκευμένη υπογραφή. Εάν η επαλήθευση είναι επιτυχής, ο αλγόριθμος ελέγχει ολόκληρη τη μνήμη στο παρασκήνιο ως σειρά εργασιών. Αν και πρόκειται για μια πιθανολογική προσέγγιση ασφαλούς εκκίνησης, δεν είναι αλάνθαστη, καθώς θα μπορούσε να εκτελεστεί ένα προσεκτικά διαμορφωμένο ή τυχαία επιλεγμένο ωφέλιμο φορτίο.

Η επιταχυνόμενη ασφαλής εκκίνηση για ενσωματωμένα συστήματα ασφαλείας πραγματικού χρόνου είναι ένα άλλο πιθανοτικό σχήμα που εξισορροπεί την ισχυρή απαίτηση για ανίχνευση αλλοίωσης δεδομένων με την ισχυρή απαίτηση για διαθεσιμότητα σε πραγματικό χρόνο. Μια πιθανολογική μέτρηση εκκίνησης εκτελείται στην πρώτη φάση για να επιτραπεί η γρήγορη εκκίνηση του συστήματος, ακολουθούμενη από μια πλήρη μέτρηση εκκίνησης για την επαλήθευση των αποτελεσμάτων της πρώτης φάσης και τη δημιουργία του νέου δειγματοληπτικού χώρου για τον επόμενο κύκλο εκκίνησης. Η προσέγγιση δύο φάσεων επιτρέπει στο σύστημα να είναι λειτουργικό σε ένα κλάσμα του χρόνου που απαιτείται για μια πλήρη μέτρηση εκκίνησης, ενώ παράλληλα παράγει υψηλή πιθανότητα ανίχνευσης παραποίησης δεδομένων.

Τέλος υπάρχουν δουλειές που προτείνουν τα μοντέλα Run-Then-Check. Η εργασία [7] για παράδειγμα προτείνει μια υλοποίηση ασφαλούς εκκίνησης για συστήματα ελέγχου των μηχανολογικών συστημάτων αυτοκινήτων που χρησιμοποιεί λειτουργίες ασφαλούς εκκίνησης

σε πρώτο πλάνο και σε υπόβαθρο για τον έλεγχο της ακεραιότητας. Το Foreground Secure Boot εξασφαλίζει ντετερμινιστική ακεραιότητα αλλά επιβραδύνει το σύστημα, ενώ το Background Secure Boot θυσιάζει την πρόληψη για την ταχύτητα. Για την εξισορρόπηση των πλεονεκτημάτων, οι συγγραφείς προτείνουν μια υβριδική προσέγγιση όπου ο πρώτος bootloader ελέγχεται σε λειτουργία foreground και η εφαρμογή εκτελείται χωρίς προηγούμενη επαλήθευση. Στη συνέχεια, ο έλεγχος ακεραιότητας εκτελείται στο παρασκήνιο και τα αποτελέσματα παραδίδονται από το HSM στον υπεύθυνο πυρήνα. Στα κρίσιμα μπλοκ μπορούν να ανατεθούν υψηλότερες προτεραιότητες για πρόσθετη ασφάλεια. Η έμφαση δίνεται στη γρήγορη αρχικοποίηση του συστήματος για ενσωματωμένες συσκευές που εργάζονται για παράδειγμα με συστήματα λειτουργίας κινητήρων.

Συνολικά, τα συστήματα ασφαλούς εκκίνησης έχουν εξελιχθεί ώστε να παρέχουν μια πρακτική και αποτελεσματική λύση για την ασφαλή και αξιόπιστη αρχικοποίηση των συστημάτων υπολογιστών. Τα ντετερμινιστικά σχήματα είναι πιο αξιόπιστα αλλά απαιτούν περισσότερους πόρους, ενώ τα πιθανοτικά και τα Run-Then-Check σχήματα είναι ταχύτερα αλλά λιγότερο αξιόπιστα. Ωστόσο, εξακολουθεί να υπάρχει ανάγκη για περισσότερη έρευνα για τη βελτίωση της αξιοπιστίας και της ασφάλειας των συστημάτων ασφαλούς εκκίνησης.

1.3 Επιταχύνοντας την Ασφαλή Εκκίνηση

Η συμβολή αυτής της εργασίας επικεντρώνεται στη βελτίωση της ασφαλούς διαδικασίας εκκίνησης με την επιτάχυνση του υπολογισμού κατακερματισμού. Η διαδικασία αυτή είναι χρονοβόρα και αποτελεί το σημαντικότερο εμπόδιο στη διαδικασία εκκίνησης για συστήματα χαμηλότερου επιπέδου, όπως οι μικροελεγκτές που χρησιμοποιούνται στην αυτοκινητοβιομηχανία και σε βιομηχανικά σενάρια. Προτείνουμε δύο πυλώνες για τη βελτίωση της διαδικασίας ασφαλούς εκκίνησης. Πρώτον, διερευνούμε ελαφρύτερους κρυπτογραφικούς αλγόριθμους κατακερματισμού που μπορούν να βελτιώσουν την απόδοση χωρίς να διακυβεύεται η ασφάλεια της διαδικασίας. Δεύτερον, παρατηρούμε ότι το μεγαλύτερο μέρος του χρόνου της διαδικασίας ασφαλούς εκκίνησης δαπανάται στην περίοδο του πραγματικού κατακερματισμού της μνήμης (memory hashing). Ως εκ τούτου, προτείνουμε την παραλληλοποίηση του βήματος κατακερματισμού για τη μείωση του χρόνου υπολογισμού.

Δοκιμάζουμε στην εργασία μας διάφορους αλγορίθμους κατακερματισμού, τόσο βιομηχανικού προτύπου όσο και τελευταίας τεχνολογίας. Ξεκινούμε με τον SHA256, ο οποίος χρησιμοποιείται ευρέως σε διάφορες εφαρμογές όπως το TLS, το SSL, το PGP, το IPsec και κρυπτονομίσματα όπως το Bitcoin. Στη συνέχεια, εξετάζουμε τον Poly1305, έναν αλγόριθμο κατακερματισμού που σχεδιάστηκε από τον D. J. Bernstein, ο οποίος παράγει μια χώνευση 16 byte ενός μηνύματος οποιουδήποτε μήκους χρησιμοποιώντας ένα nonce 16 byte και ένα κλειδί 32 byte. Είναι γνωστό ότι είναι ταχύτερος από τον MD5 και τον CBC-MAC-AES και εγγυάται τουλάχιστον την ασφάλεια του AES. Ο τρίτος αλγόριθμος είναι ο αλγόριθμος Chaskey, ο οποίος μπορεί να χρησιμοποιηθεί για τη διασφάλιση της ακεραιότητας μηνυμάτων, την πιστοποίηση χρηστών ή τη δημιουργία τυχαίων αριθμών. Είναι πολύ γρήγορος και συμπαγής και σχεδιάστηκε με γνώμονα την ενεργειακή αποδοτικότητα. Τέλος, δοκιμάζουμε και τον AES128-CMAC, έναν αλγόριθμο κρυπτογραφικού κώδικα αυθεντικοποίησης μηνυμάτων (MAC) που χρησιμοποιεί το Advanced Encryption Standard (AES) με κλειδί

128 bit για να παρέχει ακεραιότητα και αυθεντικότητα ενός μηνύματος. Χρησιμοποιούμε τον AES128-CMAC λόγω της επιτάχυνσης του υλικού του στην πλατφόρμα της αυτοκινητοβιομηχανίας όπου τον αξιολογούν και λόγω της χρήσης του σε προϋπάρχοντα μοντέλα ασφαλούς εκκίνησης εκεί.

Μετά τα πειράματα διαπιστώσαμε ότι ο Poly1305 και ο Chaskey παρέχουν τα καλύτερα αποτελέσματα όσον αφορά τις επιδόσεις ταχύτητας. Ενώ καταγράφουμε τα κέρδη απόδοσης και των τεσσάρων αλγορίθμων καθ' όλη τη διάρκεια των δοκιμών τους, σημειώνουμε ότι ο Poly1305 είναι λίγο πιο εύκολο να ενσωματωθεί σε υπάρχοντα έργα πραγματικής χρήσης, όπως αυτό στο οποίο βασίστηκε η έρευνά μας για την αυτοκινητοβιομηχανία. Ως εκ τούτου, προτείνουμε ότι μια λύση που διαθέτει τον Poly1305 μπορεί να έχει ευκολότερο χρόνο ενσωμάτωσης λόγω της καλύτερης υποδοχής της από την κοινότητα και της υιοθέτησης από τεχνολογικούς κολοσσούς όπως η Google και η CloudFlare.

Ο δεύτερος πυλώνας της πρότασης είναι ο παραλληλισμός του βήματος κατακερματισμού για τη μείωση του χρόνου υπολογισμού. Επικεντρωνόμαστε στο βήμα κατακερματισμού επειδή είναι το πιο χρονοβόρο μέρος της διαδικασίας ασφαλούς εκκίνησης. Ξεκινούμε εξηγώντας την τυπική διαδικασία ασφαλούς εκκίνησης, η οποία περιλαμβάνει την ενεργοποίηση και την αρχικοποίηση του υλικού, τη φόρτωση και την επικύρωση του φορτωτή εκκίνησης και μια αλυσίδα εκτέλεσης ελέγχων ακεραιότητας, όπως ο πυρήνας, το λειτουργικό σύστημα, τα προγράμματα οδήγησης συσκευών και οι εφαρμογές. Σημειώνεται ότι στα συστήματα χαμηλότερης τεχνολογίας, ο φόρτος εργασίας κατακερματισμού απαιτεί πολύ χρόνο, επειδή οι λιγότερο ισχυροί επεξεργαστές είναι συνήθως πιο αργοί. Ως εκ τούτου, θεωρούμε πως μια παράλληλη υλοποίηση κατακερματισμού μπορεί να βελτιώσει την απόδοση της διαδικασίας ασφαλούς εκκίνησης σε μεγάλο βαθμό. Μία εποπτεία της πρότασης μας περιγράφεται και στο Σχήμα 1.16, όπου στο επανω μέρος βλέπουμε μια σειριακή υλοποίηση ενώ ακριβώς από κάτω βλέπουμε πως θα λειτουργούσε ένας παράλληλος κατακερματισμός.

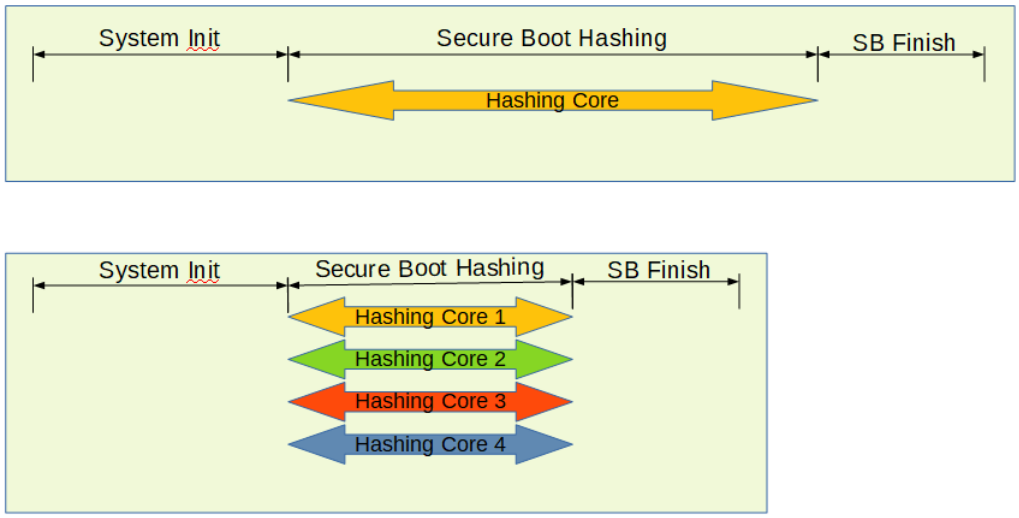

Figure 1.16. *Επάνω, η κλασσική σειριακή προσέγγιση του κατακερματισμού. Κάτω, η προτεινόμενη παράλληλη υλοποίηση του βήματος κατακερματισμού της μνήμης.*

Για να επεκτείνουμε την αλυσίδα εμπιστοσύνης εκτός του HSM, πρέπει να αντιμετωπίσουμε το ερώτημα αν η εκτέλεση κώδικα επαλήθευσης σε πυρήνες που δεν έχουν πιστοποιηθεί είναι ασφαλής. Το μοντέλο μας βασίζεται στην παραλληλοποίηση του τμήματος κατακερματισμού της διαδικασίας ασφαλούς εκκίνησης και επαληθεύουμε τον πρώτο φορτωτή εκκίνησης (First Bootloader) εντός του TPM πριν εκτελέσουμε οτιδήποτε εκτός αυτού. Αυτό διασφαλίζει ότι ο κώδικας για τον παράλληλη κατακερματισμό είναι άθικτος. Το σύστημά που προτείνουμε δεν επεμβαίνει στη διαδικασία ασφαλούς εκκίνησης πέρα από αυτό το βήμα.

Στα ντετερμινιστικά συστήματα, επαληθεύεται ολόκληρη η εικόνα μνήμης του επόμενου βήματος της διαδικασίας ασφαλούς εκκίνησης. Λόγω των μαθηματικών ιδιοτήτων των συναρτήσεων κατακερματισμού, είναι μαθηματικά απίθανο να υπογράψουμε μια αλλοιωμένη περιοχή μνήμης με την ίδια υπογραφή με την αντίστοιχη άθικτη.

Στα πιθανοτικά σχήματα, το επίπεδο ασφάλειας ποσοτικοποιείται με τη χρήση της μετρικής Detection Rate (Ποσοστό ανίχνευσης). Εκτελούμε προσομοιώσεις ασφαλούς εκκίνησης ενός συστήματος με ένα πιθανοτικό σχήμα σε ισχύ, αλλοιώνοντας λίγο την περιοχή μνήμης και βλέπουμε αν το σχήμα μπορεί να εντοπίσει τη διαφθορά. Τα ποσοστά ανίχνευσης σχετίζονται με τη συγκεκριμένη υλοποίηση του μοντέλου και δίνουν μια καλή ευρετική προσέγγιση για τα πρότυπα ασφαλείας που μπορεί να παρέχει.

Η παραλληλοποίηση του κατακερματισμού παράγει πολλαπλούς κατακερματισμούς, απαιτώντας διαφορετικές μεθόδους επαλήθευσης από την παραδοσιακή ασφαλή εκκίνηση. Παρουσιάζονται δύο μέθοδοι: συνδυασμός των κατακερματισμών σε έναν τελικό κατακερματισμό με χρήση ενός κύριου πυρήνα ή αποστολή όλων των διαφορετικών κατακερματισμών στο HSM και σύγκριση του καθενός με μια αποθηκευμένη τιμή. Η πρώτη προσέγγιση περιλαμβάνει τη συνένωση των κατακερματισμών και στη συνέχεια τον κατακερματισμό του προκύπτοντος αντικειμένου για τη δημιουργία ενός τελικού κατακερματισμού. Η δεύτερη προσέγγιση στέλνει όλους τους κατακερματισμούς στην TPM, είτε συγκεντρώνοντάς τους σε έναν πυρήνα είτε βάζοντας κάθε πυρήνα να στείλει απευθείας τον κατακερματισμό του. Η επιλεγμένη μέθοδος θα πρέπει επίσης να χρησιμοποιείται κατά την πρώτη εκκίνηση για τη δημιουργία του κατακερματισμού που είναι αποθηκευμένος στο HSM.

1.4 Πειράματα και Αποτελέσματα

Τα πειράματα που ακολουθούν εκτελέστηκαν σε δύο περιβάλλοντα. Από τη μία, χρησιμοποιήσαμε μία FPGA πλατφόρμα, το Xilinx U250 με 32 πυρήνες, για να εκτελέσουμε πειράματα στο πλαίσιο της RISC-V αρχιτεκτονικής. Από την άλλη, στο κομμάτι της δουλειάς που έγινε στο εργαστήριο της Huawei στο Μόναχο, χρησιμοποιήθηκε μια πλακέτα σχεδιασμένη για εφαρμογές στην αυτοκίνηση, με τουλάχιστον 6 πυρήνες, τα υπόλοιπα χαρακτηριστικά της οποίας θα πρέπει να παραμείνουν προστατευμένα.

1.4.1 RISC-V

Αρχικά, το RISC-V είναι μια αρχιτεκτονική συνόλου εντολών ανοικτού κώδικα που έχει γίνει όλο και πιο δημοφιλής στην αγορά τσιπ λόγω του εντελώς ανοικτού της χαρακτήρα. Παρακάτω συζητάμε γιατί επιλέξαμε το RISC-V για το πείραμά μας και παρέχουμε πληροφορίες

σχετικά με τις λειτουργίες και τους μετρητές επιδόσεων του RISC-V.

Επιλέξαμε το RISC-V για τους ακόλουθους λόγους: 1) είναι πλήρως ανοικτού κώδικα, 2) έχει προσθέσει πρωτόκολλα προσανατολισμένα στην ασφάλεια, όπως η προστασία φυσικής μνήμης (PMP), η οποία καθιστά δυνατή την απομόνωση της μνήμης, 3) υπάρχει μεγάλη ποικιλία επεκτεινόμενων πυρήνων ανοικτού κώδικα και άλλων προϊόντων.

Το RISC-V προσφέρει τρία επίπεδα προνομίων - Machine, Supervisor και User - το καθένα με ένα περιορισμένο σύνολο λειτουργιών. Η πιο προνομιούχα λειτουργία είναι η λειτουργία μηχανής (Machine Mode, M-Mode), ακολουθούμενη από τη λειτουργία επόπτη (Supervisor Mode, S-Mode) και τη λειτουργία χρήστη (User Mode, U-Mode).

Για να μετρήσουμε το χρόνο και τους κύκλους που χρειάζονται για να τρέξουν οι υλοποιήσεις μας, χρησιμοποιήσαμε τους καταχωρητές CYCLES και TIME, οι οποίοι είναι δύο από τους 32 διαφορετικούς μετρητές επιδόσεων που είναι προσβάσιμοι μέσω καταχωρητών μόνο για ανάγνωση. Πραγματοποιήσαμε τα πειράματά μας σε ένα Rocket Chip FPGA, το Xilinx U250, που διαθέτει 32 πυρήνες, χωρίς επιτάχυνση υλικού για οποιονδήποτε αλγόριθμο που χρησιμοποιήθηκε συγκεκριμένα. Περισσότερες πληροφορίες για το σύστημα δίνονται στο μέρος 5.1.

Αξιολογούμε δύο διαφορετικές προσεγγίσεις ασφαλούς εκκίνησης, την ντετερμινιστική και την πιθανολογική. Τα αποτελέσματα μετρούν τον χρόνο κατακερματισμού τριών αλγορίθμων κατακερματισμού: SHA256, Poly1305 και Chaskey, με τον AES128-CMAC να μην περιλαμβάνεται στον πειραματισμό μας. Μετράμε ένα ευρύ φάσμα μεγεθών μνήμης προς επαλήθευση, συγκεκριμένα 2/4/16/64/128 MB.

Αποτελέσματα Ντετερμινιστικών Μοντέλων

Τα αποτελέσματά μας δείχνουν πως οι Poly και Chaskey έχουν καλύτερες επιδόσεις από τον SHA256 για μικρότερα μεγέθη εικόνων και λιγότερους από 10 πυρήνες. Καθώς τα μεγέθη των εικόνων και των ελεγχόμενων κομματιών μνήμης αυξάνονται, ο Poly1305 παραμένει καλύτερος από τον SHA256 για μεγάλα μεγέθη και επίσης καλύτερος από τον Chaskey, ο οποίος δεν κλιμακώνεται καλά με περισσότερους πυρήνες και μεγάλα μεγέθη. Ο Poly1305 είναι ένας ισχυρός υποψήφιος για ντετερμινιστικές υλοποιήσεις ασφαλούς εκκίνησης πολλών πυρήνων, ξεπερνώντας τους αντιπάλους του κατά μέσο όρο σε όλα τα μεγέθη. Παρατηρούμε πως οι επιδόσεις των αλγορίθμων τείνουν να είναι ασυνεπείς σε όλους τους πυρήνες, ειδικά με τον Poly και τον SHA, γεγονός που πιθανώς να οφείλεται στον σχεδιασμό των αλγορίθμων, καθώς αυτή η ασυνέπεια δεν παρατηρήθηκε στον ίδιο βαθμό με τον Chaskey, ούτε σε αυτή τη μελέτη ούτε κατά τη διάρκεια των πειραμάτων πιθανοτικής ασφαλούς εκκίνησης που ακολούθησαν. Αποτελέσματα για τα Ντετερμινιστικά μοντέλα σε περιβάλλον RISC-V δίνονται στα σχήματα 1.17 έως 1.19, ενώ περισσότερες λεπτομέρειες δίνονται και στο μέρος 5.1.4.1.

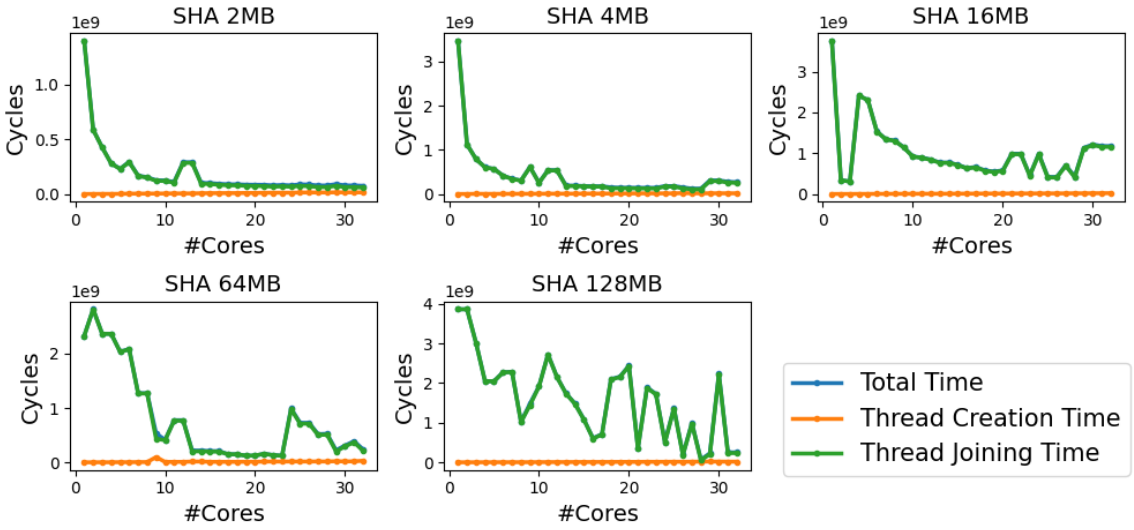

Figure 1.17. *Αποτελέσματα χρόνου κατακερματισμού με SHA256 για μεγέθη από 2 έως*

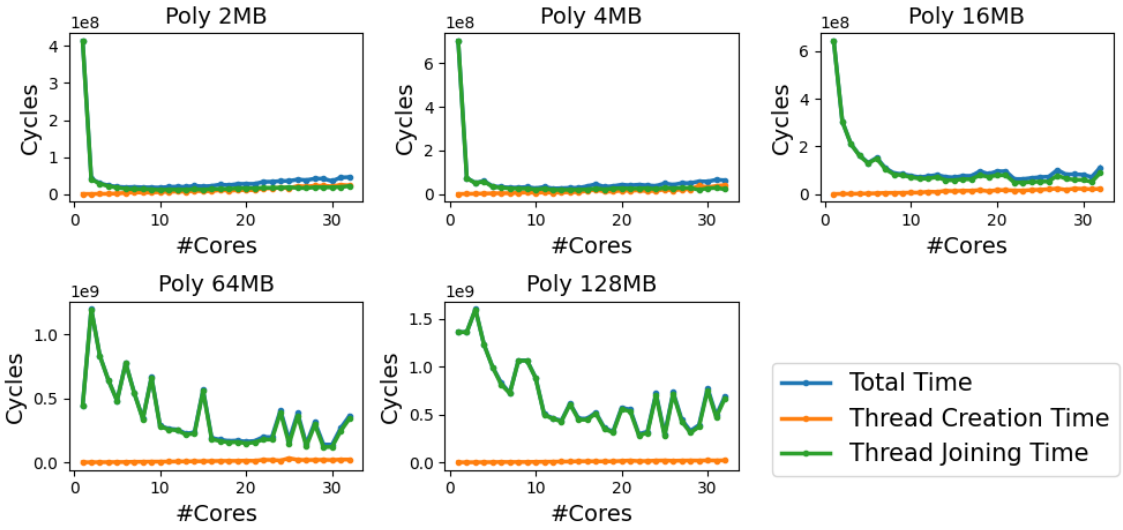

Figure 1.18. *Αποτελέσματα χρόνου κατακερματισμού με Poly1305 για μεγέθη από 2 έως*

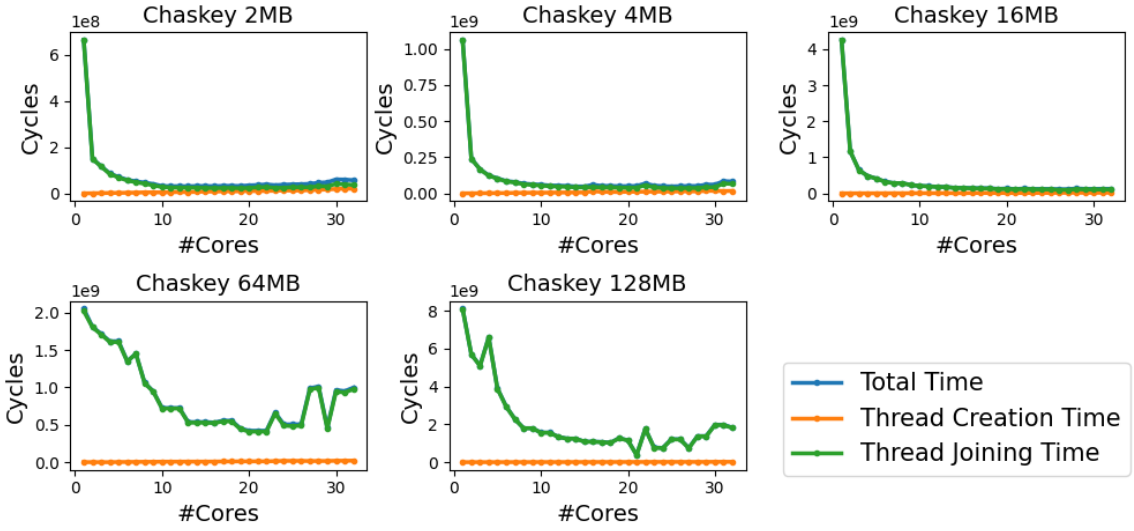

Figure 1.19. *Αποτελέσματα χρόνου κατακερματισμού με Chaskey για μεγέθη από 2 έως 128 MB.*

Αποτελέσματα Πιθανολογικών Μοντέλων

Στα πιθανοτικά σχήματα πολλαπλών πυρήνων, τα Chaskey και Poly1305 υπερτερούν έναντι του SHA256 τόσο σε υλοποιήσεις τόσο με έναν όσο και με πολλούς πυρήνες. Η χρήση άνω του ενός πυρήνων είναι επωφελής εφόσον ο αριθμός των ελεγχόμενων κομματιών είναι ίσος ή μικρότερος από τον αριθμό των πυρήνων. Η επιβάρυνση δημιουργίας thread είναι σημαντική μόνο για αρκετά μικρά μεγέθη που πρέπει να ελεγχθούν. Και οι τρεις αλγόριθμοι κλιμακώνουν καλά μέχρι γύρω στους 15-20 πυρήνες. Οι Poly1305 και Chaskey είναι καλύτεροι υποψήφιοι για πιθανοτικό κατακερματισμό πολλών πυρήνων από τον SHA256.

Υπάρχουν και κάποιες άλλες παρατηρήσεις που βρίσκουμε να ισχύουν για όλες τις υλοποιήσεις. Ο αριθμός των πυρήνων που χρησιμοποιούνται σχετίζεται στενά με τον αριθμό των τεμαχίων που ελέγχονται, ενώ τα κέρδη σε χρόνο γενικά σταματούν μετά τη χρήση περισσότερων πυρήνων από ότι κομμάτια. Όλες οι υλοποιήσεις σταματούν να παρουσιάζουν σοβαρά πλεονεκτήματα μετά από περίπου 20 πυρήνες λόγω της επιβάρυνσης της μνήμης και πιθανότατα και των αλγορίθμων που δεν είναι βελτιστοποιημένοι για παραλληλισμό. Ο χρόνος δημιουργίας νημάτων ακολουθεί την αναμενόμενη διαδρομή ενώ σε κάποιες περιπτώσεις εμφανίζονται υπεργραμμικές (super-linear) επιταχύνσεις, οι οποίες οφείλονταισε θέματα που σχετίζονται με την cache μνήμη και κυρίως το cache locality. Αποτελέσματα για τα Πιθανολογικά μοντέλα σε περιβάλλον RISC-V δίνονται στο μέρος 5.1.4.2

Παρακάτω, για τα σχήματα 1.20 έως 1.24, βλέπουμε τα αποτελέσματα για τον αλγόριθμο SHA256 χρησιμοποιόντας ένα πιθανολογικό μοντέλο όπως αναφέρθηκε πρότερα.

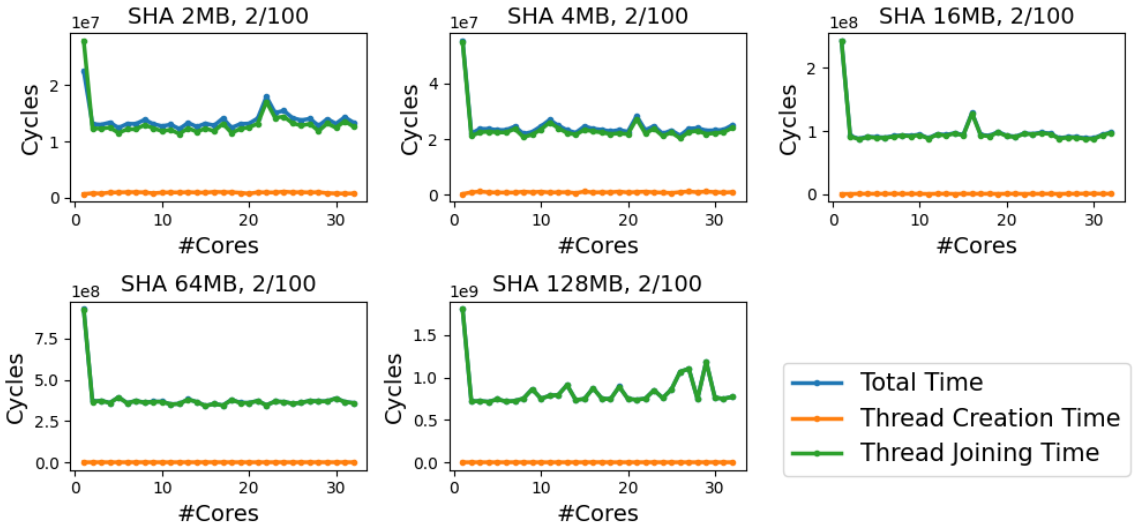

Figure 1.20. *Αποτελέσματα χρόνου κατακερματισμού με SHA256 ελέγχοντας 2/100 κομ-*

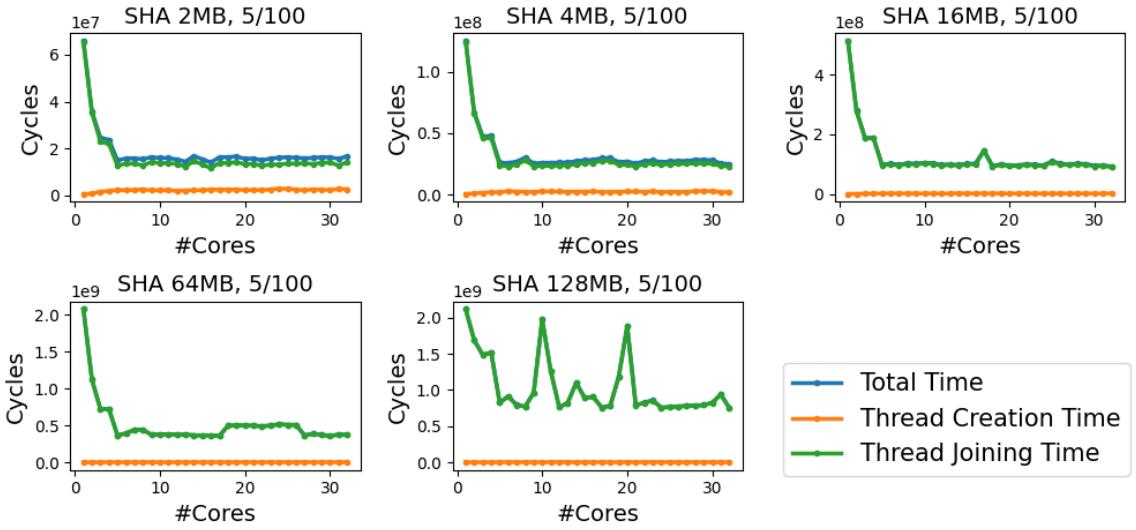

Figure 1.21. *Αποτελέσματα χρόνου κατακερματισμού με SHA256 ελέγχοντας 5/100 κομ-*

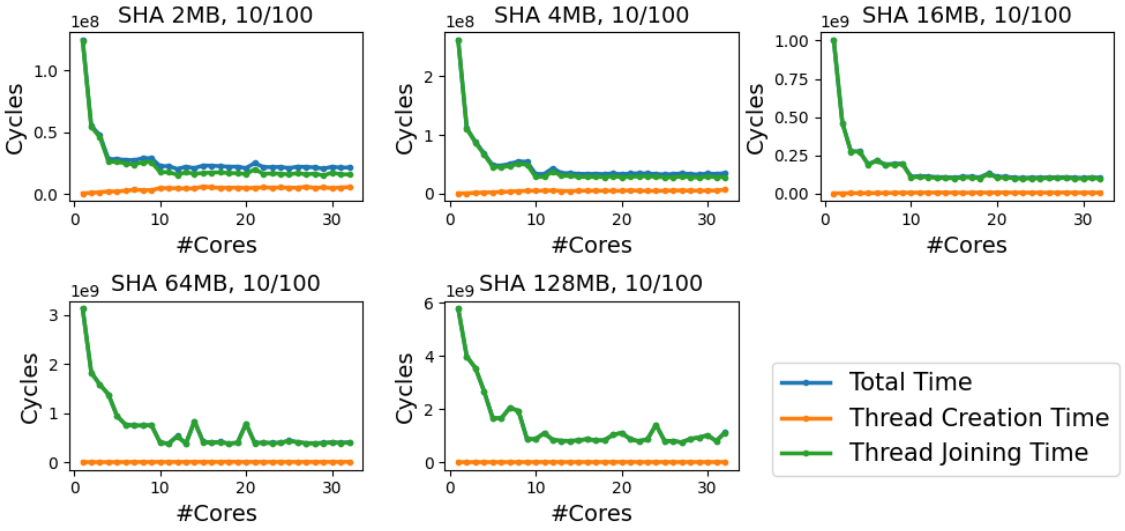

Figure 1.22. *Αποτελέσματα χρόνου κατακερματισμού με SHA256 ελέγχοντας 10/100 κομμάτια.*

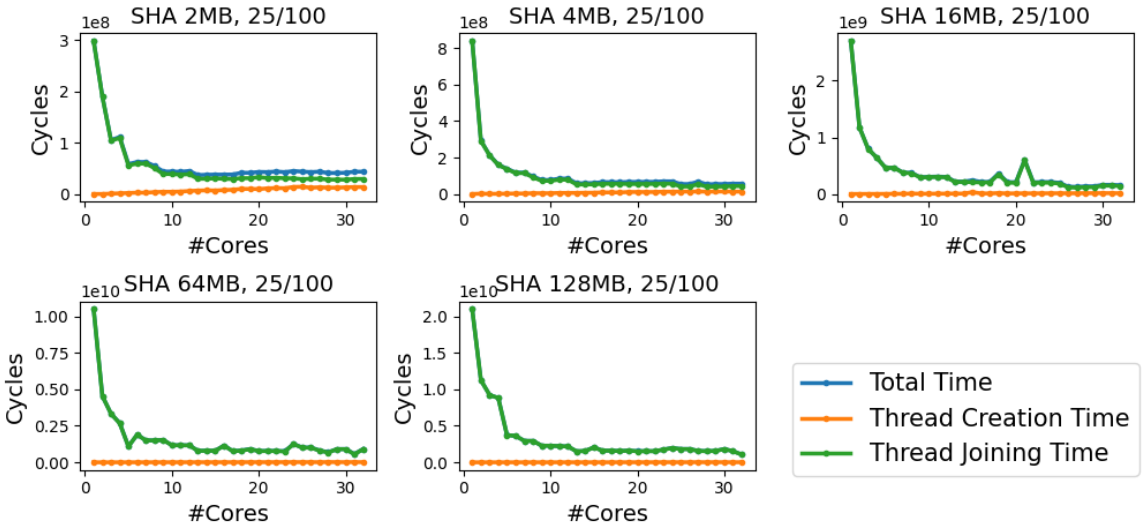

Figure 1.23. *Αποτελέσματα χρόνου κατακερματισμού με SHA256 ελέγχοντας 25/100 κομ-*

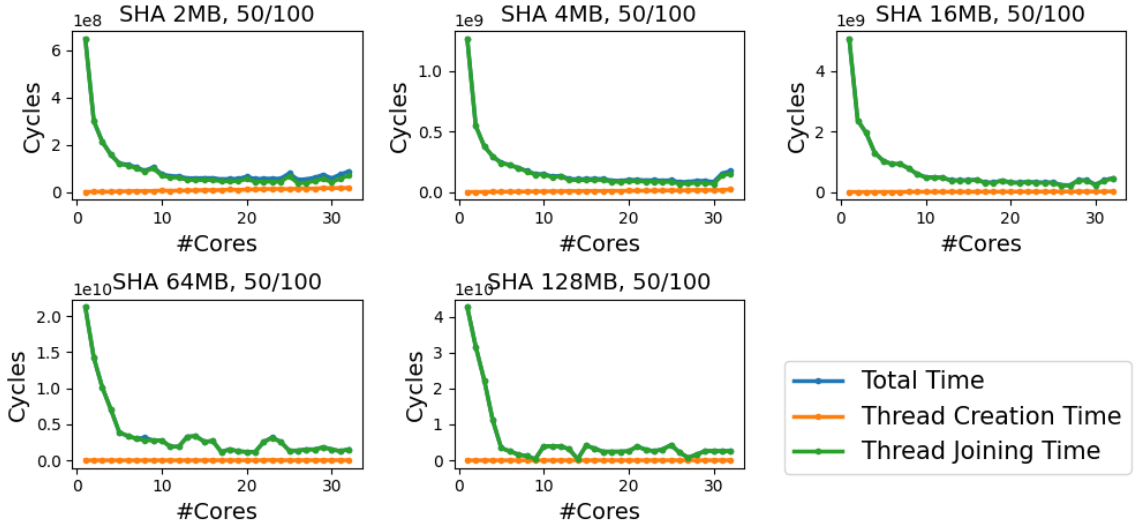

Figure 1.24. *Αποτελέσματα χρόνου κατακερματισμού με SHA256 ελέγχοντας 50/100 κομμάτια.*

Για τον αλγόριθμο SHA256 , βλέπουμε πως η κλιμακοσημότητα λειτουργεί όπως θα περιμέναμε, ενώ η βελτίωση σταματά μόλις ο αριθμός των πυρήνων ξεπεράσει τον αριθμό των κομματιών προς έλεγχο. Παρατηρούμε ότι από άποψη κύκλων ο SHA256 παραμένει αργότερος των υπόλοιπων αλγορίθμων, όπως είδαμε και στη συνέχεια.

Παρακάτω, για τα σχήματα 1.25 έως 1.29, βλέπουμε τα αποτελέσματα για τον αλγόριθμο Poly1305 χρησιμοποιόντας το ίδιο πιθανολογικό μοντέλο με πριν.

Poly1305

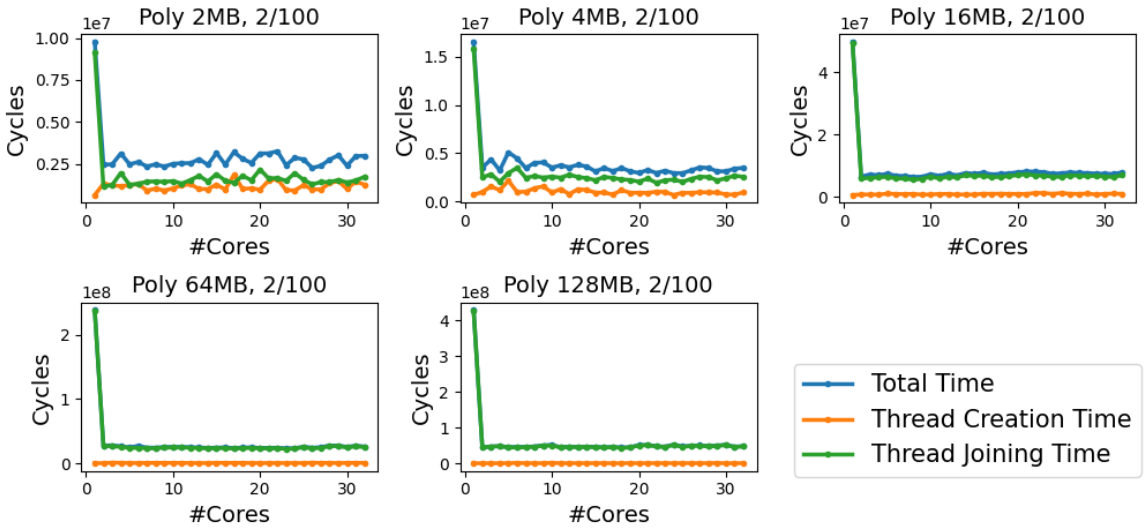

Figure 1.25. *Αποτελέσματα χρόνου κατακερματισμού με Poly1305 ελέγχοντας 2/100 κομ-*

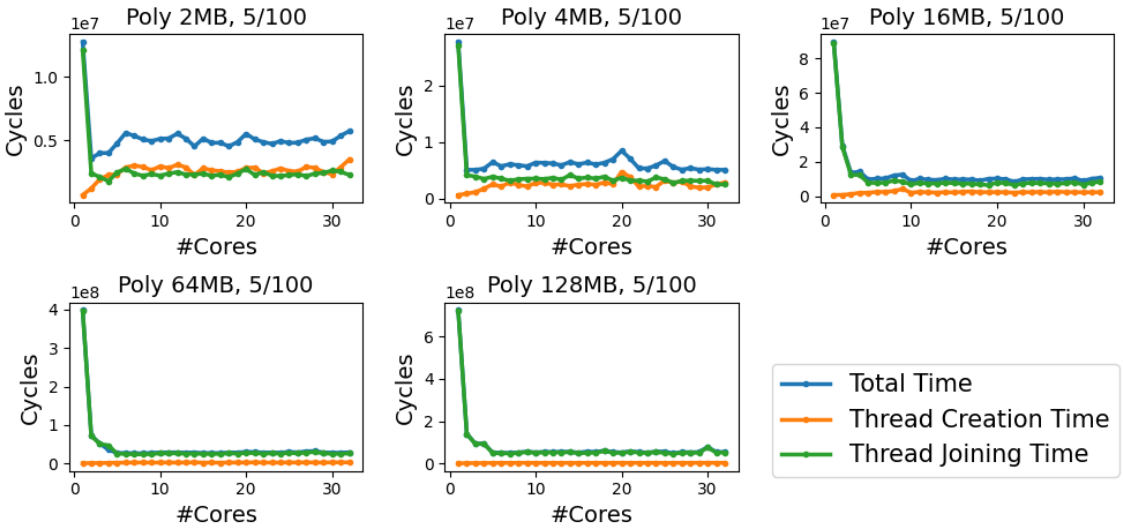

Figure 1.26. *Αποτελέσματα χρόνου κατακερματισμού με Poly1305 ελέγχοντας 5/100 κομ-*

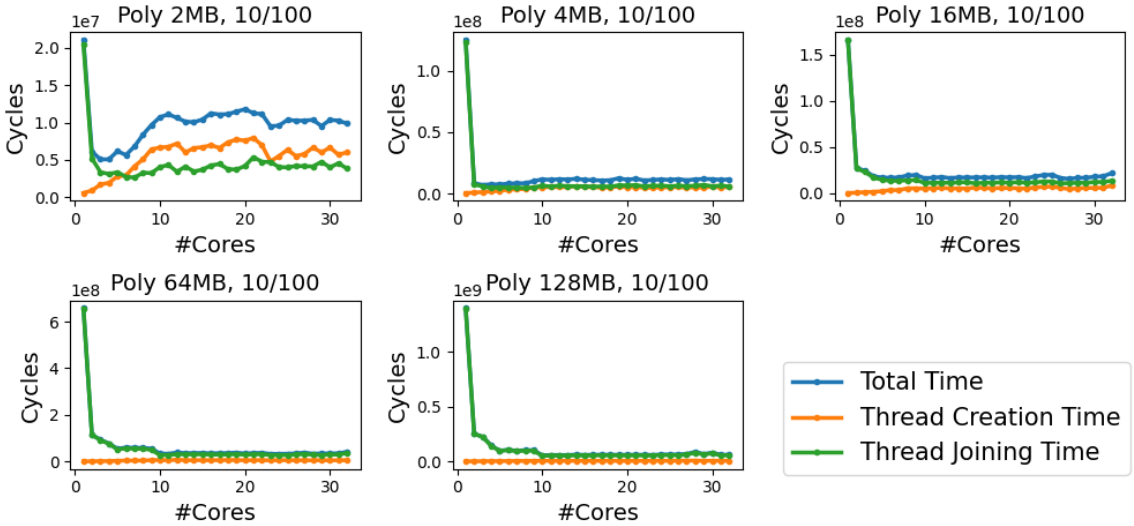

Figure 1.27. *Αποτελέσματα χρόνου κατακερματισμού με Poly1305 ελέγχοντας 10/100 κομμάτια.*

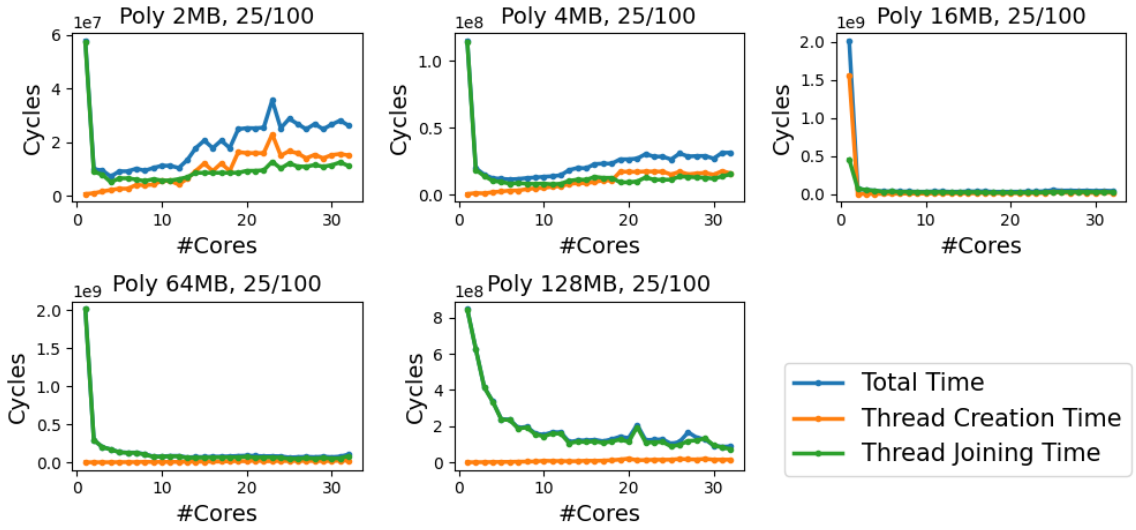

Figure 1.28. *Αποτελέσματα χρόνου κατακερματισμού με Poly1305 ελέγχοντας 25/100 κομ-*

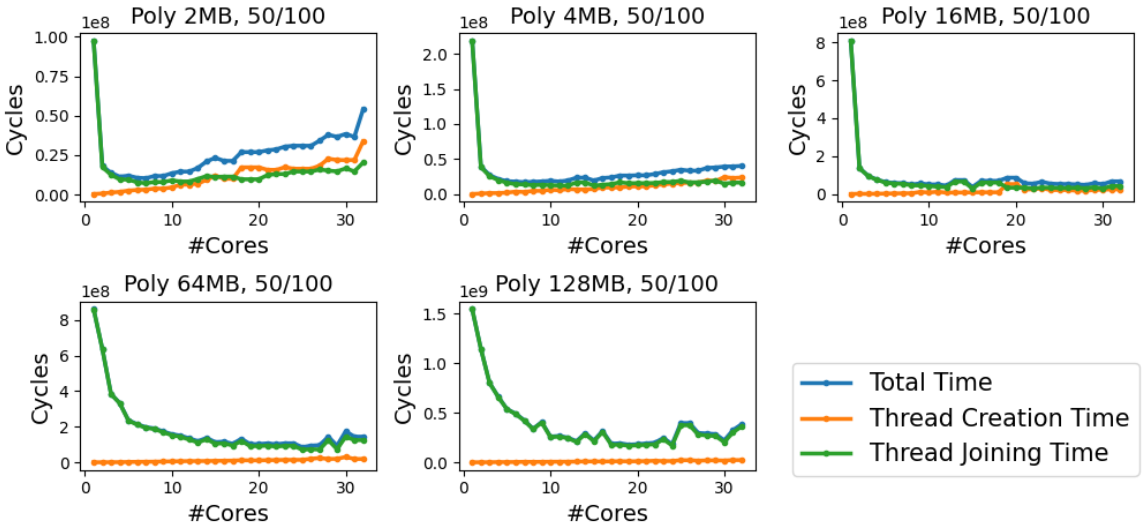

Figure 1.29. *Αποτελέσματα χρόνου κατακερματισμού με Poly1305 ελέγχοντας 50/100 κομμάτια.*

Για τον αλγόριθμο Poly1305 , στις περισσότερες περιπτώσεις συνεχίζουμε να έχουμε κέρδη από την νέα υλοποίηση. Ο Poly είναι πολύ ταχύτερος αλγόριθμος του SHA αντίστοιχα με τις υλοποιήσεις με ίσους πυρήνες, ακόμα και στη μονοπύρηνη περίπτωση. Ειδική παρατήρηση για τον Poly είναι πως οι διαφορές μεταξύ στην μονοπύρηνη και την πρώτη πολυ-πύρηνη λύση προκύπτουν ιδιαίτερα μεγάλες.

Παρακάτω, για τα σχήματα 1.30 έως 1.34, βλέπουμε τα αποτελέσματα για τον αλγόριθμο Chaskey χρησιμοποιόντας το ίδιο πιθανολογικό μοντέλο με πριν.

Chaskey

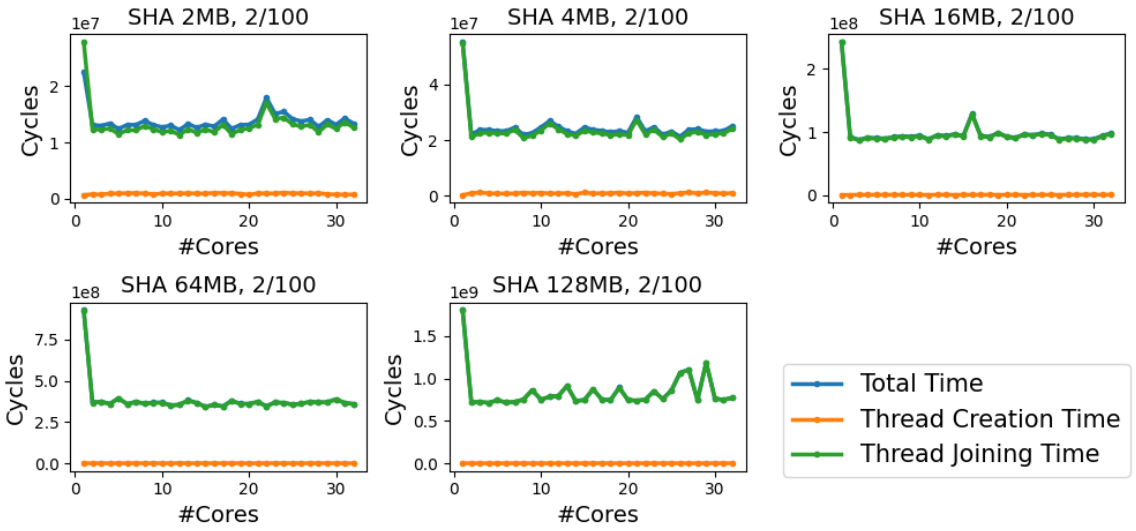

Figure 1.30. *Αποτελέσματα χρόνου κατακερματισμού με Chaskey ελέγχοντας 2/100 κομ-*

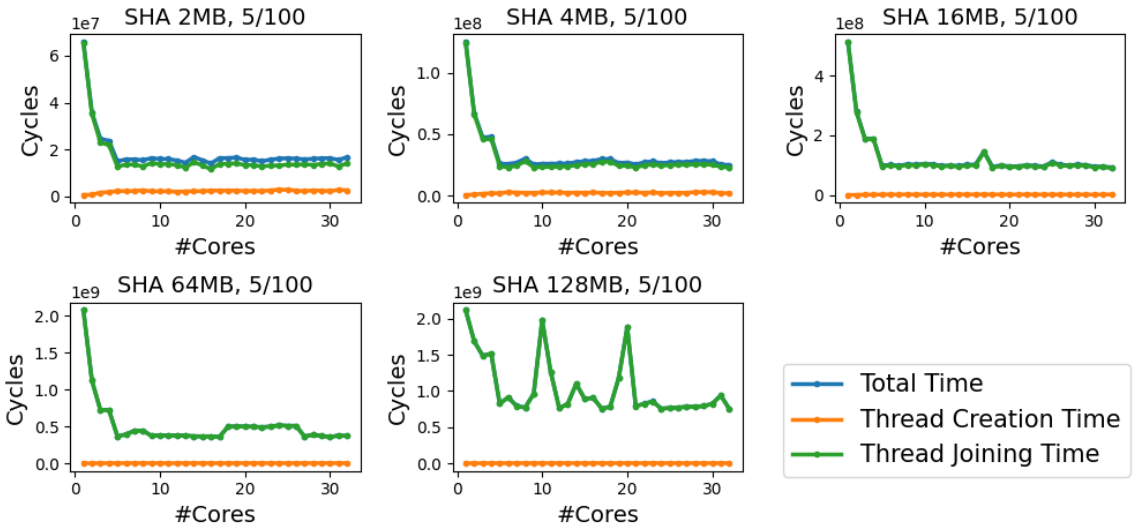

Figure 1.31. *Αποτελέσματα χρόνου κατακερματισμού με Chaskey ελέγχοντας 5/100 κομ-*

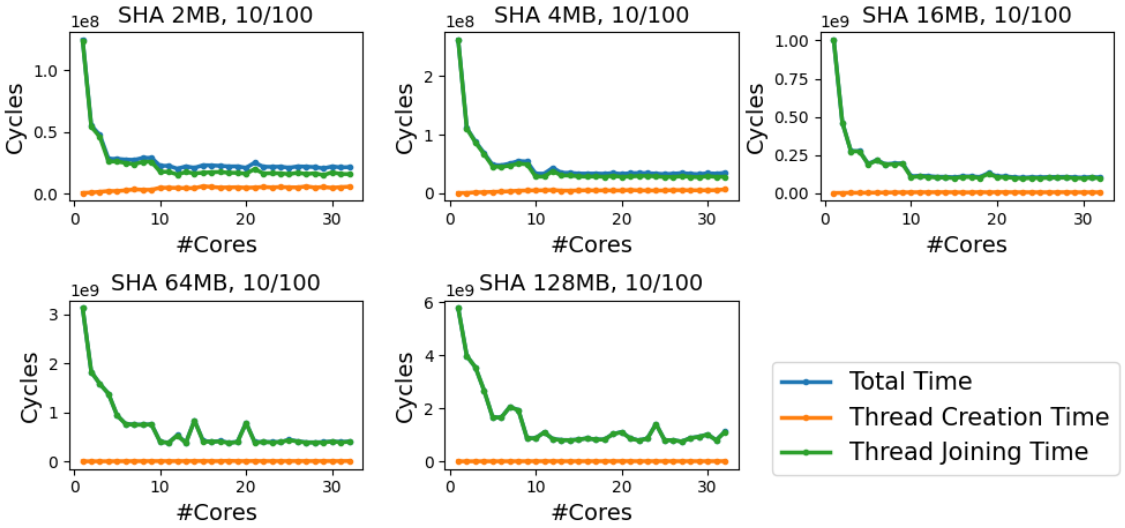

Figure 1.32. *Αποτελέσματα χρόνου κατακερματισμού με Chaskey ελέγχοντας 10/100 κομμάτια.*

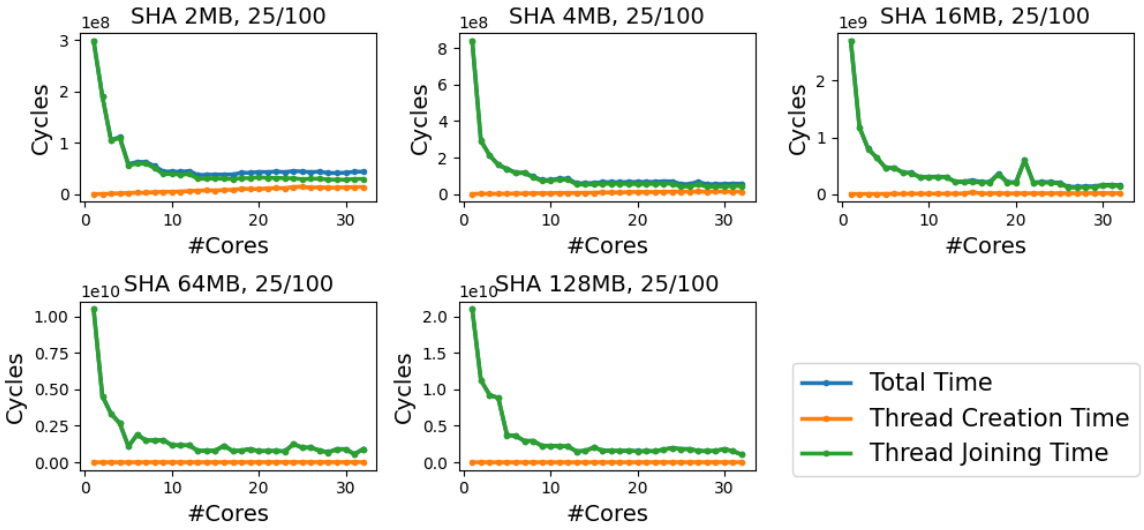

Figure 1.33. *Αποτελέσματα χρόνου κατακερματισμού με Chaskey ελέγχοντας 25/100 κομ-*

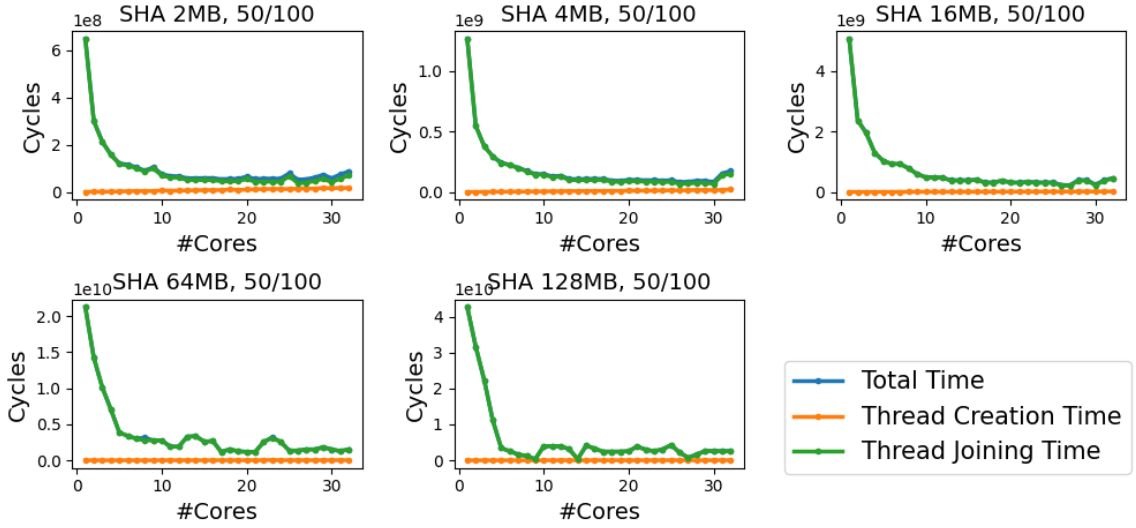

Figure 1.34. *Αποτελέσματα χρόνου κατακερματισμού με Chaskey ελέγχοντας 50/100 κομμάτια.*

Και με τον αλγόριθμο Chaskey προκύπτουν παρόμοια συμπεράσματα, ενώ μια ιδιαιτερότητα αυτών των πειραμάτων είναι πως τείναμε να παίρνουμε αρκετά πιο συνεπή αποτελέσματα με την θεωρητική οδό, συγκριτικά με τους άλλους δύο αλγορίθμους. Τα κέρδη του αλγόριθμου παραμένουν συγκρίσιμα και ουσιώδη τόσο για τα μικρά όσο και για τα μεγάλα μεγέθη που δοκιμάζονται.

Συμπεραίνουμε λοιπόν πως τόσο τα πιθανοτικά όσο και τα ντετερμινιστικά σχήματα μπορούν να επωφεληθούν από τη χρήση περισσότερων του ενός πυρήνων, ειδικά με προκαθορισμένα κομμάτια μνήμης που χωρίζονται σε διαφορετικούς πυρήνες κατά την εκτέλεση. Αυτά τα οφέλη ισχύουν ανεξάρτητα από το ποσοστό της ελεγχόμενης μνήμης και για διάφορα μεγέθη μνήμης στα πλαίσια των δοκιμών μας.

1.4.2 Πλατφόρμα Αυτοκίνησης

Το επόμενο κομμάτι της δουλειάς περιστρέφεται γύρω από τη χρήση μιας πλακέτας που περιέχει μια μονάδα ασφαλείας υλικού και 6 μονάδες επεξεργασίας. Λόγω εμπιστευτικότητας, δεν μπορούμε να μοιραστούμε περισσότερες λεπτομέρειες σχετικά με την πλακέτα. Ωστόσο, τα χαρακτηριστικά της πλακέτας είναι ευρέως διαθέσιμα σε πολλούς μικροελεγκτές που χρησιμοποιούνται στην αυτοκινητοβιομηχανία, ενώ δεν γίνεται χρήση κάποιου εξειδικευμένου κομματιού hardware πέρα από τα αναγραφόμενα. Η εργασία μας μπορεί να εφαρμοστεί σε οποιονδήποτε μικροελεγκτή διαθέτει περισσότερους από έναν πυρήνες και ένα αξιόπιστο περιβάλλον εκτέλεσης (trusted execution environment, TEE).

Η έρευνά μας επικεντρώνεται στην επιτάχυνση της διαδικασίας ασφαλούς εκκίνησης με τη χρήση τεσσάρων βασικών αλγορίθμων: SHA256, AES128-CMAC, Poly1305 και Chaskey. Διεξάγουμε πειράματα σε δύο διαφορετικές υλοποιήσεις των SHA256, AES128-CMAC και Poly1305. Παρόλα αυτά, οι διαφορές στις επιδόσεις ήταν αμελητέες, οπότε παρουσιάζουμε στην αξιολόγησή μας τα αποτελέσματα για μία υλοποίηση του καθενός.

Μετράμε την κλιμακωσιμότητα των αλγορίθμων μέχρι έξι πυρήνες και παρουσιάζουμε την επιτάχυνση κάθε αλγορίθμου σε σχέση με τις υλοποιήσεις τους σε έναν πυρήνα. Παρουσιάζουμε τα αποτελέσματα της επιτάχυνσης στους πίνακες 1.1, 1.2, 1.3 και 1.4 και στο σχήμα 1.35.

1-πυρήνα	2-πυρήνες	4-πυρήνες	6-πυρήνες
1	1.9451	3.8822	5.8323

Table 1.1. *SHA256 Speedup στο χρόνο κατακερματισμού με έως και 6 πυρήνες σχετικά με τη μονοπύρηνη υλοποίηση.*

1-πυρήνα	2-πυρήνες	4-πυρήνες	6-πυρήνες
1	1.9711	3.9412	5.7435

Table 1.2. *Poly1305 Speedup στο χρόνο κατακερματισμού με έως και 6 πυρήνες σχετικά με τη μονοπύρηνη υλοποίηση.*

1-πυρήνα	2-πυρήνες	4-πυρήνες	6-πυρήνες
1	1.9703	3.8530	5.6791

Table 1.3. *Chaskey Speedup στο χρόνο κατακερματισμού με έως και 6 πυρήνες σχετικά με τη μονοπύρηνη υλοποίηση.*

1-πυρήνα	2-πυρήνες	4-πυρήνες	6-πυρήνες
1	1.7307	3.072	3.12

Table 1.4. *AES128-CMAC Speedup στο χρόνο κατακερματισμού με έως και 6 πυρήνες σχετικά με τη μονοπύρηνη υλοποίηση.*

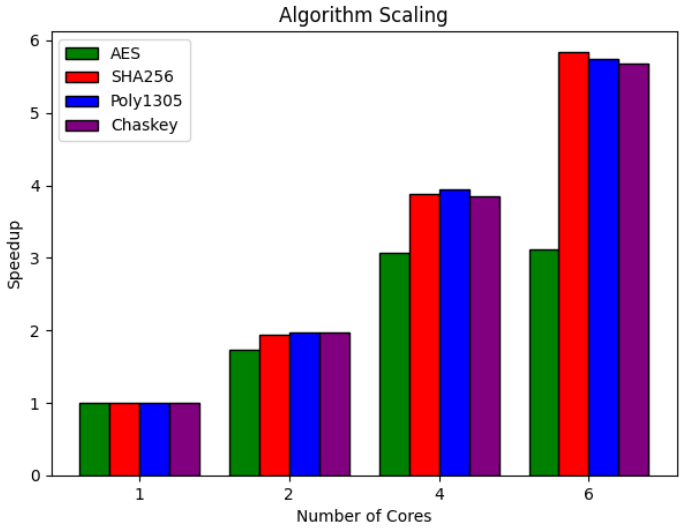

Figure 1.35. *Κλιμακωσιμότητα των διαφόρων αλγορίθμως σε σχέση με τις μονοπύρηνες υλοποιήσεις τους, μόνο για χρόνο κατακερματισμού.*

Έπειτα, συγκρίνουμε την υλοποίηση των αλγορίθμων με 6 πυρήνες, συμπεριλαμβανομένης της απόδοσης της βασικής περίπτωσης, η οποία είναι μια μέτρηση με επιτάχυνση υλικού AES128-CMAC με έναν πυρήνα που λαμβάνεται με τον πυρήνα εντός του TPM. Τα αποτελέσματα αυτής της σύγκρισης φαίνονται στο παρακάτω σχήμα 1.36. Τελικά καταφέρνουμε να πάρουμε speedups της τάξεως του 3.7 και 4.7 σε σύγκριση με την υπάρχουσα υλοποίηση - δηλαδή το AES128-CMAC με hardware acceleration - με τους αλγορίθμους Poly και Chaskey αντίστοιχα.

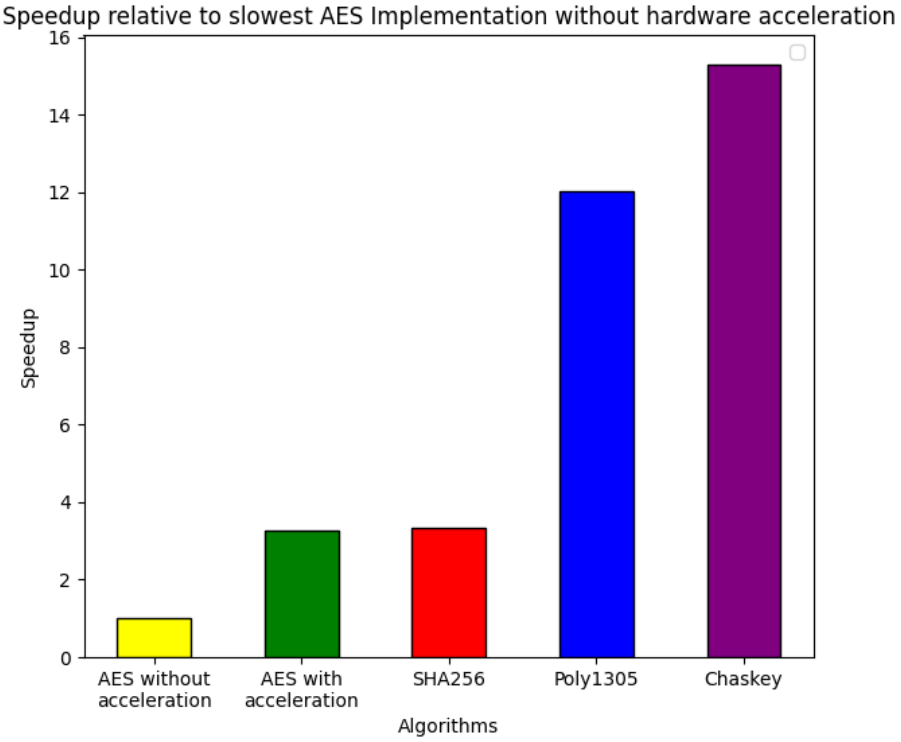

Figure 1.36. *Χρόνοι κατακερματισμού σε σύγκριση με την αργότερη υλοποίηση (AES-CMAC χωρίς hardware acceleration.*

Εκτός από τους χρόνους κατακερματισμού, εκτελέσαμε και πειράματα για να δούμε το κέρδος στον end-to-end χρόνο του συστήματος, δηλαδή από τη στιγμή που πατάμε το κουμπί της έναρξης στο υλικό έως τη στιγμή που το σύστημα είναι ικανό να επικοινωνήσει με άλλες συσκευές στο δίκτυο. Τα αποτελέσματα αυτών των πειραμάτων φαίνονται στο σχήμα 1.37. Εδώ βρήκαμε πως παίρνουμε επιταχύνσεις της τάξεως του 1.4 και 1.45 με τους Poly και Chaskey αντίστοιχα.

Δείξαμε τελικά πως υπάρχει χρόνος να κερδηθεί από το κομμάτι του κατακερματισμού κατά το Secure Boot, ενώ δείξαμε επίσης πως το κέρδος αυτό είναι σημαντικό και για τον ολικό χρόνο εκκίνησης του συστήματος. Πετύχαμε με τον αλγόριθμο Poly1305 speedups 3.7 και 1.4 για το χρόνο κατακερματισμού και end-to-end αντίστοιχα, και με τον αλγόριθμο Chaskey speedups 4.7 και 1.45 αντίστοιχα.

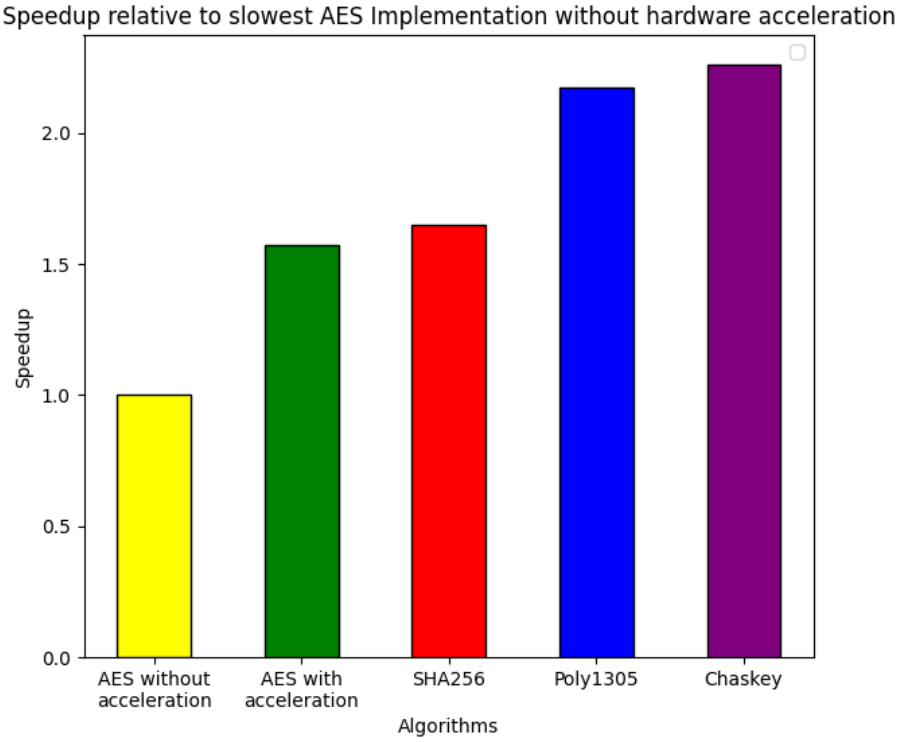

Figure 1.37. *Χρόνοι end-to-end σε σύγκριση με την αργότερη υλοποίηση (AES-CMAC χωρίς hardware acceleration.*

1.5 Συμπεράσματα και Μελλοντική Δουλειά

1.5.1 Συμπεράσματα

Στόχος της παρούσας εργασίας ήταν να προσδιοριστούν μέθοδοι για την επιτάχυνση της διαδικασίας ασφαλούς εκκίνησης, η οποία αποτελεί κρίσιμο στοιχείο για την ασφάλεια των ενσωματωμένων συστημάτων. Κατά τη διάρκεια της έρευνάς μας, κατέστη προφανές ότι υπάρχουν δύο κύριες οδοί για τη βελτίωση της ταχύτητας της ασφαλούς εκκίνησης: η επιλογή ενός πιο ελαφρού και γρήγορου κρυπτογραφικού αλγορίθμου και η παραλληλοποίηση του βήματος κατακερματισμού με πολλαπλούς πυρήνες.

Η έρευνά μας αποκάλυψε ότι το βήμα κατακερματισμού είναι από τα κύρια σημεία συμφόρησης στη διαδικασία ασφαλούς εκκίνησης. Με τη μείωση του χρόνου που απαιτείται για τον κατακερματισμό της μνήμης, μπορούν να επιτευχθούν σημαντικές βελτιώσεις στον συνολικό χρόνο.

Ξεκινήσαμε τη διερεύνησή μας εξετάζοντας δύο ευρέως χρησιμοποιούμενους αλγορίθμους κατακερματισμού: SHA256 και AES128-CMAC. Στη συνέχεια προχωρήσαμε στη διερεύνηση δύο σχετικά νεότερων αλγορίθμων: Poly1305 και Chaskey. Μετά από προσεκτική ανάλυση, διαπιστώσαμε ότι οι Poly1305 και Chaskey ήταν ιδιαίτερα υποσχόμενοι όσον αφορά την ταχύτητά τους υλοποιούμενοι σε πολλαπλούς πυρήνες.

Εκτός από την επιλογή ταχύτερων αλγορίθμων, αναπτύξαμε επίσης μια μέθοδο παραλληλισμού της διαδικασίας κατακερματισμού σε πολλαπλούς πυρήνες. Οι σύγχρονοι ενσωματωμένοι

μικροελεγκτές προσφέρουν μεγάλο αριθμό πυρήνων, ιδίως σε τομείς με υψηλή ζήτηση, όπως η αυτοκινητοβιομηχανία. Ωστόσο, κατά τη διάρκεια της διαδικασίας ασφαλούς εκκίνησης, οι περισσότεροι από αυτούς τους πυρήνες παραμένουν αδρανείς, χωρίς να συνεισφέρουν στον απαιτούμενο υπολογισμό. Η προτεινόμενη προσέγγισή μας περιλαμβάνει την εκχώρηση ενός τμήματος της μνήμης σε κάθε πυρήνα για κατακερματισμό και στη συνέχεια τη χρήση μίας από τις δύο μεθόδους επαλήθευσης των παραγόμενων κατακερματισμών: την παραγωγή ενός τελικού hash ή την αποστολή όλων σε ένα αξιόπιστο περιβάλλον.

Οι δοκιμές μας έδειξαν ότι το ντετερμινιστικό παράλληλο σχήμα μας παρέχει βελτίωση 3,7 φορές στο στάδιο του κατακερματισμού της μνήμης από μόνο του και 1,4 φορές στη διαδικασία ασφαλούς εκκίνησης end-to-end, σε σύγκριση με τη βασική περίπτωση του ενός πυρήνα. Στον μικροελεγκτή που έχει σχεδιαστεί για σενάρια αυτοκινητοβιομηχανίας, όταν χρησιμοποιούνται 6 πυρήνες η βελτίωση των επιδόσεων είναι ακόμη πιο σημαντική. Στο τσιπ RISC-V , τα παράλληλα ντετερμινιστικά και πιθανοτικά σχήματα επαλήθευσης βελτιώνουν την απόδοση κατά 2,2 χρησιμοποιώντας οκτώ πυρήνες και κατά 1,8 χρησιμοποιώντας 14 πυρήνες σε σχέση με το βασικό σχήμα ενός πυρήνα.

Συμπερασματικά, η παρούσα διπλωματική παρουσιάζει μια πολλά υποσχόμενη προσέγγιση για τη βελτίωση της ταχύτητας της διαδικασίας ασφαλούς εκκίνησης μέσω της επιλογής ταχύτερων κρυπτογραφικών αλγορίθμων και της παραλληλοποίησης του βήματος κατακερματισμού σε πολλαπλούς πυρήνες. Αν και υπάρχει ακόμη πολλή δουλειά να γίνει σε αυτόν τον τομέα, τα αποτελέσματα των δοκιμών μας καταδεικνύουν τις σημαντικές δυνατότητες για τη μείωση του χρόνου που απαιτείται για την ασφαλή εκκίνηση, ενισχύοντας έτσι την ασφάλεια των ενσωματωμένων συστημάτων.

1.5.2 Μελλοντική Δουλειά

Κοιτάζοντας μπροστά, υπάρχουν πολλοί δρόμοι για μελλοντικές εργασίες σε αυτόν τον τομέα. Ένας τομέας που αξίζει να διερευνηθεί είναι η αξιολόγηση νεότερων, ταχύτερων αλγορίθμων κατακερματισμού, ιδίως καθώς η επιτάχυνση υλικού για παλαιότερους αλγορίθμους γίνεται πιο διαδεδομένη. Με την επικείμενη εισαγωγή της κβαντικής πληροφορικής, η οποία μπορεί να απαιτήσει τη χρήση αλγορίθμων κατακερματισμού ανθεκτικών στην κβαντική πληροφορική, η έρευνα αυτή μπορεί να γίνει ακόμη πιο κρίσιμη.

Κατά τη χρήση πιθανολογικών μοντέλων επαλήθευσης της μνήμης, ο τεμαχισμός της μνήμης και ο διαμοιρασμός των κομματιών που προκύπτουν γίνονται με έναν αρκετά πρώιμο τρόπο, σύμφωνα με τον οποίο χωρίζουμε τη μνήμη απλά σε ίσα κομμάτια χωρίς να λαμβάνουμε υπόψιν για παράδειγμα το περιεχόμενο τους, τη χρησιμότητα ή την κρισιμότητα του περιεχομένου της μνήμης κλπ. Ο τρόπος λοιπόν με τον οποίο τεμαχίζουμε και διαμοιράζουμε χρήζει βελτιώσεων.

Μία ιδέα άξια αναφοράς είναι η ενσωμάτωση στο σύστημα τεμαχισμού τεχνικών δανεισμένων από τη μηχανική μάθηση, όπου το σύστημα θα μπορεί να ¨μάθει' τα κρίσιμα κομμάτια της μνήμης, να ¨θυμάται' κομμάτια που ελέγθηκαν πρόσφατα και να επιλέγει στην ουσία διαφορετικά κομμάτια μνήμης προς επαλήθευση σε κάθε κύκλο έναρξης, ώστε να εξισορροπείται το σύστημα. Με αυτόν τον τρόπο το τυχαίο στοιχείο καταργείται για χάριν μίας επαλήθευσης πιο ενημερωμένης για την κατάσταση του λογισμικού.

Οι πολυπύρηνες συσκευές παρουσιάζουν επίσης την ευκαιρία για έρευνα σχετικά με την επαλήθευση λογισμικού κατά παραγγελία, την ιεράρχηση των κρίσιμων στοιχείων και τη διενέργεια ελέγχων υποβάθρου. Όπως αναφέρεται σε κάποιες δουλειές, η κατα-παραγγελία επαλήθευση κομματιών λογισμικού καθώς αυτά ζητούνται από το run-time περιβάλλον είναι δυνατή και μπορεί να βοηθήσει στην επιτάχυνση της εκκίνησης. Ένα σύστημα όπως αυτό που περιγράψαμε παραπάνω θα είναι ικανό να αξιολογεί τις πιθανότητες να ζητηθεί στο άμεσο μέλλον συγκεκριμένο κομμάτι software και να προετοιμάζει τον έλεγχο σε τέτοιες περιπτώσεις.

Τέλος, δεδομένου ότι η απόδοση της ασφαλούς εκκίνησης εξαρτάται σε μεγάλο βαθμό από το υλικό και την εκάστοτε εφαρμογή, η δοκιμή των προτεινόμενων συστημάτων σε μια ποικιλία υλικού στον πραγματικό κόσμο, ιδίως στους τομείς της αυτοκινητοβιομηχανίας, του αυτοματισμού και της βιομηχανίας, θα παρείχε πολύτιμες πληροφορίες τόσο για τις επιδόσεις όσο και για τις απαιτήσεις των συγκεκριμένων εφαρμογών.

Chapter 2

Background

2.1 Cybersecurity in the Automotive Field

In the last 50 years the automotive industry has taken great leaps in terms of progress. What was once a diesel engine car equipped with just the bare minimum to be able to traverse some distances has come to become a highly developed and fine piece of engineering featuring sensors and instruments to measure and drive many different subsystems. Due to - in some degree - the astronomical progress in the field of computing and electrical engineering in these years, modern cars feature more than 100 electronic control units (ECUs) to provide functionality services that increase safety, comfort and efficient through intricate electrical systems [1]. One can only look at the amount of sensors a modern engine (combustion or electrical) needs to be able to efficiently run or the driving assistance subsystems to get a rough sense of the degree by which cars essentially become 'computers with wheels'.

Until the last few years cars had been getting better at monitoring their internal state, meaning they could detect the causes of malfunctions or prevent them entirely, notify the owners on maintenance needs and so on. But today cars are starting to look not only within them but also to the outside world [2]. From cars that help their human driver or can find the best route from point A to point B up to cars that can actually drive themselves this is a new and evolving area that brings a lot of possibilities and, of course, a lot of danger, as the industry seems to be moving towards the adoption of the software-defined car.

Another reason for vehicles to integrate ECUs is the application of vehicle-to-vehicle (for e.g. car-to-car interaction) and vehicle-to-X (for e.g. traffic coordination or law enforcement) communications. Especially in the dawn of the autonomous vehicle era, the ability to quickly efficiently and safely communicate with other cars or any type of device is crucial. It should be obvious at this point that the vehicle networks of the future will be rich in electronic devices, so it is only reasonable that we try to address the challenges these devices may face early on. One major challenge they are facing is of course security.

2.1.1 Cars as Computers

There were a lot of areas that could benefit from the integration of computation on a car, and the automotive industry has already recognised that. From the efficient tuning of a car all the way to infotainment computing is branching out.

- Point about embedded electronics being small and specific in their tasks - A mini list of example electronic systems on a car (ABS, EPS, Traction control, Infotainment, Engine running, Batteries e.g.) The following is a brief list of examples where computation is being used today:

- **ABS** Anti-lock Braking System. A subsystem helping with maximizing the brake force a car can output while maintaining traction, so a driver can still steer the car under sever braking.

- **Traction Control** A system controlling the power that is being transfered from the engine to the wheel to prevent wheelspin.

- **ESP/ESC** Electronic Stability Programme or Electronic Stability Control. A combination of ABS and Traction Control to smooth the driver's control inputs and increase overall car stability.

- **Engine Monitoring** Providing feedback on the stable and healthy operation of the engine.

- **Fuel Injection** An ECU is responsible for configuring the engine's fuel injection timing, allowing for smooth operation or even fine tuning of the car's performance by professionals.

- **Lane Assistance/Cruise Control** Some newer cars come equipped with sensors and software able to return the car to it's lane if needed and taking over from the driver under specific circumstances with cruise control.

- **Autonomous Driving/Car Vision** Advanced sensors and software that are able to drive the car without relying on driver inputs. The vehicle is able to "see" the road, the obstacles, the traffic signs and lights and requires from little to no assistance to be functional. A lot of engineering fields come into plaz with these cars such as Mechanical Engineering, Computer Vision, Machine Learning as well as other aspects of engineering, with Cyberphysical security being the one most tightly related to this work.

- **GPS** Global Positioning System. GPS have been close to a standard feature for most cars in the last decade.

- **Climate Control** Air conditioning and the ability to adapt to temperature changes in order to maintain the climate inside the cabin.

- **Infotainment** Cars start having a lot of infotainment inside of them with the usage of Radio, TV and interconnectivity between the car and the user's devices becoming more widespread.

Notice that none of the above mentioned systems relies on breakthroughs and innovation on the mechanical engineering field but rather on the broad usage of electronics on the car.

2.1.2 Cyberphysical Security

As cars start transforming into systems similar to computers, they bring along not only the capabilities that these systems provide but also their security risks. The window of opportunity for attackers to compromise vehicles went from requiring them to have physical access to the car's interior to them being able to hijack a vehicle remotely only using a laptop, as demonstrated with the famous "Jeep Hack" [3].

2.1.3 Attacking an Automotive System

Car hacking is unfortunately a thriving industry. With cars being entrusted with so much more computing capabilities as we've discussed, the attack surface has expanded from the physical into the cyberphysical domain. This makes it easier for attackers to tamper with vehicles as it lifts the necessity of physical access to the car - which in itself is a very risky procedure. But there is also another major problem. The major advancements in the IT industry have drawn the automotive one to adopt their progress. This in turn means that the vulnerabilities that a software may have when it is deployed in the context of an IT industry carry over to cars, expanding the exposed attack surface significantly. A malicious user that has experience tampering with a specific software does not necessarily care whether that software is deployed in a server, a laptop or an ECU. Keep in mind that these vulnerabilities need not apply on the software package level, but can also be targeted against specific processes, services or even lower software development levels.

The number of ways that a malicious actor can access a vehicle have risen as well. Where 15 years ago the only way to hijack a car was to compromise its physical security - e.g. violate the doors, steal the keys etc. - nowadays there are a multiple of avenues to go down, as an attacker could for example:

- Use a replay attack on the key fob that unlocks the car to gain access to the inside,
- Remotely access the car's infotainment systems (play loud noises while driving through the radio, access personal data stored on the system etc),
- Remotely try to gain access to the car's CAN bus, and virtually compromise the whole car if successful,
- In the case of autonomous vehicles, even more control can be forced on the car through a successful attack, because of the automation natively existing in the car

As car hacking is evolving, we can expect these kinds of attacks to expand in other areas and also to keep becoming sharper and more difficult to mitigate.

2.2 Cryptography Background

In William Stallings *Cryptography and Network Security Principles and Practice* [4] a very fitting definition is given as to what Computer Security is:

Computer Security: The protection afforded to an automated information system in order to attain the applicable objectives of preserving the integrity, availability, and

confidentiality of information system resources (includes hardware, software, firmware, information/data, and telecommunications).

It is becoming aparent, that the field of computer security and cryptography are so closely tied that it can be unreasonable to try to untangle them, at least in the context of computer science. The three concepts of *Confidentiality*, *Integrity* and *Availability* form what is known as *The CIA triad*, the reasoning behind the name being that these concepts embody the fundamental security objectives for a system to be seen as secure enough. In the context of our work another concept from cryptography is very handy to get familiar with, and that would be *Authenticity*. Among cryptographers another concept is sometimes considered to be a requirement for computer security, and that is *Accountability*, but it is not relevant in our work and we will not go deeper regarding it.

2.2.1 Integrity, Authenticity, Confidentiality and Availability

The definitions of these concepts is critical so that we avoid misconceptions. Here we try to define those, using Stalling's book [4] as a guideline.

Integrity

Integrity can be used in the context of either **Data Integrity** or **System Integrity**. The first part of it is concerned with guaranteeing that stored and transmitted data and programs are only changed in a specific and authorized manner. System Integrity on the other side provides the assurance that a system performs its intended function in an unimpaired manner and not through outside or unauthorized manipulation of the system.

Examples of where data integrity is important are large data servers, personal disks and as is our case whole firmware images or other sensitive OS related data that should not be tampered with. For system integrity that would encompass the several runtime mechanisms which try to assert the correct execution of programs or operating systems, like control flow integrity [8] or data flow integrity [9].

Authenticity

The property of being genuine and being able to be verified and trusted; confidence in the validity of a transmission, a message, or message originator. This means verifying that users are who they say they are and that each input arriving at the system came from a trusted source. In some cases, like message authenticity, authenticity also inherently *implies* integrity, especially when it has to do with Data Integrity. The reverse is not the case.

Confidentiality

Again, this sole definition covers two more specific concepts: **Data Confidentiality** and **Privacy**.

Data Confidentiality assures that private information is not made available to unauthorized entities, whereas Privacy assures that control what information related to them may

be collected and stored and by whom and to whom that information may be disclosed.

Data Confidentiality is especially relevant in the Internet, where a multiple of protocols, with the most familiar being TLS/SSL, are used to guarantee that data can travel throught the network with confidentiality, meaning that no unauthorized person can get a hold on them, and even if they do, they would not be able to understand or use the data (because it is encrypted for example).

Privacy refers to the ability of individuals to control access to their personal information and the level of disclosure they choose to make. It involves protecting sensitive information such as personally identifiable data, financial information, and communication from unauthorized access, use, or disclosure. This is achieved through various technical and legal measures such as encryption, firewalls, and data protection laws.

On the Privacy side, with the rise of online advertising and similar techniques, privacy is of utmost importance in defending personal information when surfing the Internet.

Availability

A more loosely connected concept to actual cryptography, **Availability** is the assurance that a system works promptly and that service is not denied to authorized users. Availability is of very high importance in systems that have cyber-physical characteristics, such as Automotives, Aeroplanes, Industrial Electronics and any other area where the continuous function of a system is of critical importance. Mostly on embedded systems availability is deemed a high priority, including in our own project as we will see later.

2.2.2 Digital Signatures

Digital signatures are a type of cryptographic mechanism used to ensure the authenticity and integrity of digital data. A digital signature is created using a combination of a private key and a hashing algorithm. The private key is kept secret and is only known by the signer, while the hashing algorithm is used to generate a unique fingerprint or hash of the data being signed.

To create a digital signature, the hashing algorithm is applied to the data, generating a hash value. This hash value is then encrypted with the private key of the signer, producing a unique digital signature. The resulting digital signature can be attached to the data and transmitted along with it.

When the data is received by the recipient, the recipient can verify the digital signature by applying the same hashing algorithm to the data, generating a hash value. The recipient can then use the public key corresponding to the private key used to create the digital signature to decrypt the digital signature and obtain the original hash value. If the hash value generated by the recipient matches the original hash value, the data is considered to be authentic and has not been modified since it was signed. If the hash values do not match, the data is considered to be compromised or tampered with.

Digital signatures are commonly used in various applications such as secure communications, electronic transactions, and software validation. They provide a way to authenticate

the identity of the signer and ensure the integrity of the data being signed, providing a high level of security and trustworthiness.

In the context of secure boot, digital signatures are used to verify the integrity of the boot components.

2.3 Trust and Trusted Computing

Another field relevant to our work is *Trust and Trusted Computing*. Trusted Computing builds upon the inherent features of cryptography with the goal of defending system integrity. Trust computing has many applications, among which are Secure Boot which we will see in detail later, but also Remote Attestation, Trusted Execution Environments (TEEs), Enclaves and so on.

2.3.1 Root of Trust

A root of trust is a security component in a computer system that serves as the foundation for establishing and verifying the authenticity of all other components. It is a secure entity, typically implemented as a hardware or software module, which is trusted to perform security-related functions and protect against unauthorized access or manipulation.

The root of trust is responsible for generating and managing cryptographic keys, establishing secure boot processes, verifying the authenticity of firmware and software updates, and enforcing access control policies. There exist a multitude of mechanisms, both hardware and software based, such as hardware security modules (HSMs), secure enclaves, and trusted platform modules (TPMs), that can be leveraged to ensure the integrity of the system from the moment it boots up.

2.3.2 Chain of Trust

A Chain of Trust is a term used to describe the process of establishing trust throughout system components in a sequential manner. The chain of trust starts with a root of trust, a secure component as described above. The root of trust is considered to be secure by definition, meaning that the model on which we base and evaluate the security of a system takes the integrity of the root of trust as a given.

In Figure 2.1 we show how trust typically flows in a system, starting from a root-of-trust and reaching the OS or beyond.

Once the root of trust has been established, it is responsible for verifying the integrity and authenticity of the next level of system components, which could be firmware, bootloaders, or other low-level system software. These components in turn verify the authenticity of the next level of components, such as device drivers or applications.

This process continues until all system components have been verified, creating a "chain" of trust that links all components back to the root of trust. Each component in the chain must be signed and validated by the previous trusted entity in the chain, ensuring that any malicious or unauthorized changes to the system are detected and prevented.

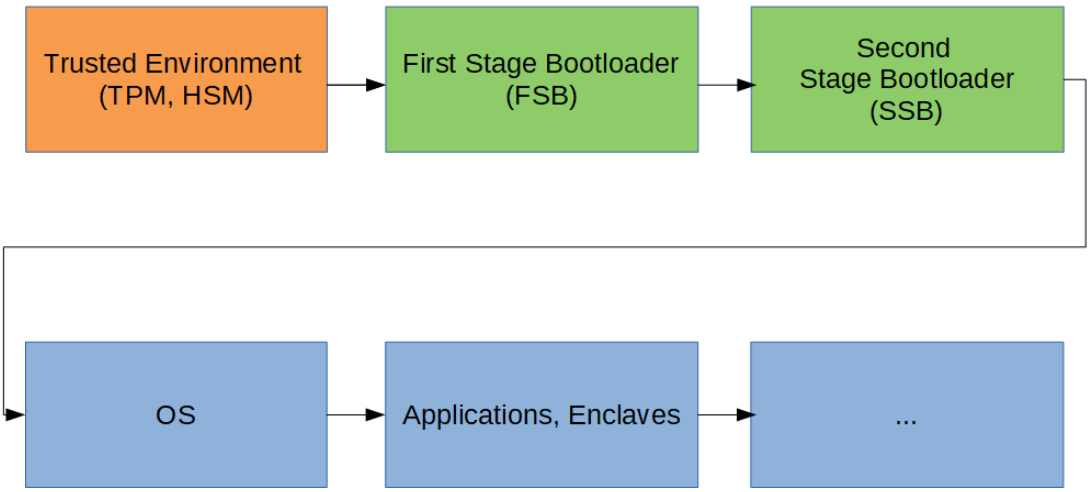

Figure 2.1. *A typical flow to create the chain-of-trust on a system.*

Embedding Trust within a system shields it from a variety of cyber attacks, including attacks on the supply chain - essentially not booting the firmware or software that is expected, modifying the existing software maliciously and so on.

2.3.3 Trusted Platform Modules (TPMs)

A Trusted Platform Module is a hardware-based security component that provides a secure execution environment and stores cryptographic keys and other sensitive data. It is the most common way to create a root of trust rooted in hardware. The TPM is a tamper-resistant chip that is integrated into the motherboard of a computer, and it enables a range of security-related functions, including secure boot, secure storage, and remote attestation.

The TPM offers several specific features to enhance the security of a computing environment, including:

- Secure storage: The TPM provides a secure storage space for sensitive data such as encryption keys, passwords, and digital certificates. This storage is isolated from the rest of the system, making it difficult for an attacker to access.
- Secure boot: The TPM can ensure the integrity of the boot process by verifying that the system has not been tampered with before loading the operating system. The way this works is the internal core of the TPM handles all the verification that needs to be done during secure booting, something that as we will see later, can become slow.
- Remote attestation: The TPM can provide a proof that a system is running in a trusted state to remote parties. This can help establish trust between two systems communicating over a network.
- Sealed storage: The TPM can "seal" data so that it can only be accessed in a trusted environment. This is useful for protecting sensitive data like encryption keys, so that

they can only be accessed when the system is in a trusted state.

- Key generation and management: The TPM can generate and manage encryption keys, making it easier to use encryption to protect sensitive data.
- Random number generation: The TPM has a built-in hardware random number generator, which can provide high-quality random numbers for cryptographic operations.

The TPM can be used to ensure the integrity of the boot process, protect sensitive data, and establish a trusted communication channel between two systems. It is widely used in enterprise environments to provide a strong foundation for security and is considered a critical component for building trusted computing platforms especially in modern systems, as systems with built in TPMs are becoming the norm even among personal commercial computers. [5]

2.3.4 Hardware and Software-based TPMs

Apart from the hardware-based TPMs, there also exist alternatives that utilize software features to emulate the security standards a physical TPM offers.

Hardware-based TPMs are physical chips that are integrated into a device. These chips provide a secure storage environment for cryptographic keys and perform cryptographic operations, such as digital signature verification and key generation. Hardware-based TPMs have several advantages, such as providing a high level of security due to their physical separation from the rest of the system, and being resistant to attacks such as tampering or software exploits. They are however more costly to deploy, making them therefore less widespread, especially on low end and embedded devices.

Software-based TPMs, on the other hand, are implemented using software that runs on the device's CPU. They use the device's hardware resources, such as the CPU and memory, to perform cryptographic operations. Software-based TPMs are typically used in virtualized environments or in devices that do not have a physical TPM. Some features like memory protection enable such solutions. While software-based TPMs are less secure than hardware-based TPMs due to the increased attack surface provided by the shared hardware resources, they provide a cost-effective alternative to hardware-based TPMs.

2.4 Secure Boot

Secure Boot is a feature that ensures the integrity of the boot process by verifying the digital signature of each component of the boot loader before it is executed. This helps prevent malware and other unauthorized software from being loaded during the boot process. Secure Boot is implemented by a combination of hardware and software components, including the BIOS, bootloader, and operating system kernel.

Secure Boot offers a high level of security for the boot process and helps protect the system from attacks that target the boot process. It ensures that only trusted software is loaded during the boot process and provides a secure foundation for the rest of the system. Secure Boot is being used more and more in the industry, and it is becoming a standard

feature in modern computer systems. A recent example of the broad application of Secure Boot is Microsoft's announcement that they will require the existence of a TPM and the enabling of Secure Boot to even be able to install Windows 11 from now on, essentially turning Secure Boot from optional to mandatory in the long run. [5]

In recent works and in applications where speed is of the essence - for example in Automotive - designers may choose to favor speed over security in a variety of ways as we will discuss later.

The Secure Boot process typically involves the following steps:

1. Power on and bootloader initialization: When the embedded system is powered on, the bootloader initializes the hardware and begins the boot process.

2. Bootloader validation: The bootloader is validated to ensure that it has not been tampered with or replaced by malicious software. This may involve checking the bootloader's digital signature, verifying its checksum, or comparing it to a known good image.

3. Operating system kernel validation: The bootloader loads the operating system kernel, which is validated to ensure that it has not been tampered with or replaced. This may involve checking the kernel's digital signature, verifying its checksum, or comparing it to a known good image.

4. Application and driver validation: The operating system loads additional applications and drivers as needed, and each one is validated to ensure that it has not been tampered with or replaced. Again this may involve the techniques of the previous steps

By performing these steps, Secure Boot provides a secure and trusted foundation for the operating system and helps protect against malware and other security threats that can compromise the system.

Where are the hashes stored?

During Secure Boot, the pre-calculated hash of each component in the boot process is stored in a secure storage area called the Platform Configuration Registers (PCR) within the Trusted Platform Module (TPM) on the system's motherboard. The TPM is a dedicated hardware component that is designed to store cryptographic keys, perform cryptographic operations, and ensure the integrity of the boot process.

Each component in the boot process has its own PCR in the TPM, and the pre-calculated hash of that component is stored in its corresponding PCR. During the boot process, the system firmware verifies the integrity of each component by comparing its pre-calculated hash with the current hash of the component. If the hashes match, the boot process continues. If the hashes don't match, the boot process is interrupted and the system may display an error message or take other appropriate actions to prevent unauthorized access to the system.

2.4.1 Software vs Hardware-based Verification

Secure Boot can be implemented using either software-based or hardware-based verification.

Software-based verification involves using software tools to verify the integrity of the boot components. This is typically done by checking digital signatures, verifying checksums, or comparing the components to a known good image. Software-based verification is flexible and can be implemented on a wide range of platforms, but it is also vulnerable to attacks that can compromise the software used to perform the verification.

Hardware-based verification involves using dedicated hardware components to ensure the integrity of the boot components. This typically involves using a Trusted Platform Module (TPM) to store cryptographic keys and perform cryptographic operations, as well as to measure and attest to the integrity of the boot components. Hardware-based verification is more secure than software-based verification because the hardware is physically isolated from the rest of the system, making it more difficult for attackers to compromise the verification process.

Both software-based and hardware-based verification have their advantages and disadvantages. Software-based verification is more flexible and can be implemented on a wider range of platforms, but it is also more vulnerable to attacks. Hardware-based verification is more secure, but it requires dedicated hardware components and may be more difficult to implement on some platforms. Ultimately, the choice between software-based and hardware-based verification will depend on the specific requirements of the system and the level of security that is needed.

2.4.2 Secure Boot Schemes

There exist three different ways of approaching secure booting a device. Each offers a balance between security and speed, and each applies in different projects depending on the underlying demands.

Deterministic Secure Boot

The most widespread approach to secure booting devices so far has been the Deterministic Secure Boot. During the booting process of a device, a memory area that is not yet trusted must be checked. Let's take the memory area in Figure 2.2 as an example. The trusted environment - which currently has control - must therefore measure and verify this entire memory to guarantee its integrity. This happens by taking the whole memory area to be checked and hashing it, producing a single hash value as the output, as shown in Figure 2.3. This hash value is then checked against the hash stored in our secure storage that has been produced from the hashing of the correct state of the software in question (Figure 2.4). If the hashes match, we can proceed with the booting process. If not, each application has to handle failure accordingly (Figure 2.5).

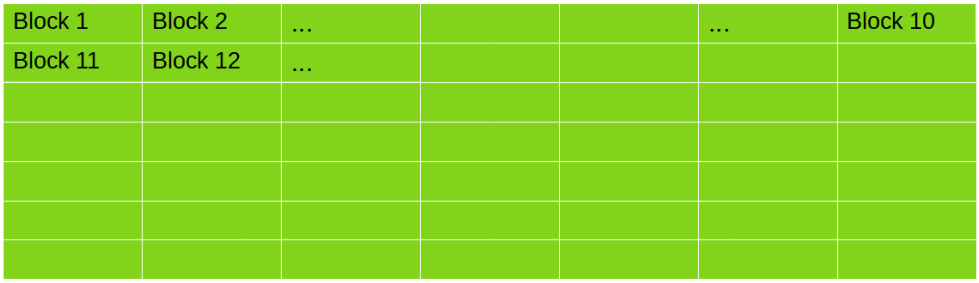

Figure 2.2. *Deterministic Check Step 1) Check the whole memory*

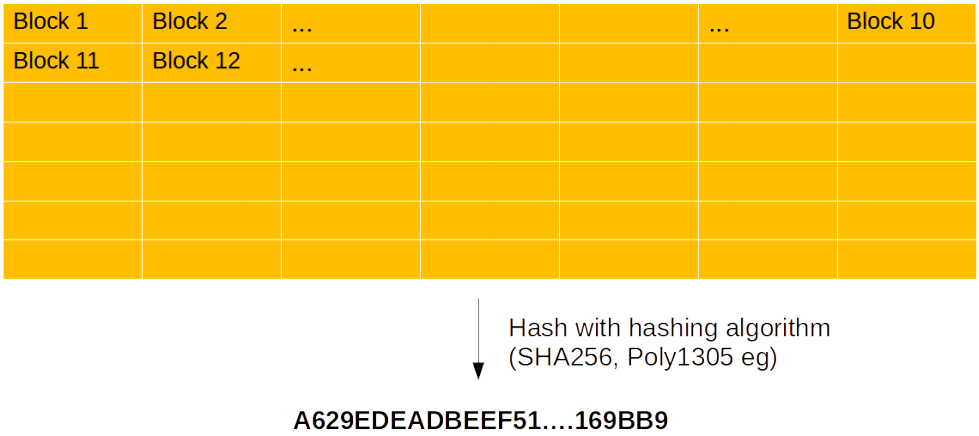

Figure 2.3. *Deterministic Check Step 2) Hash it all and produce a digest*

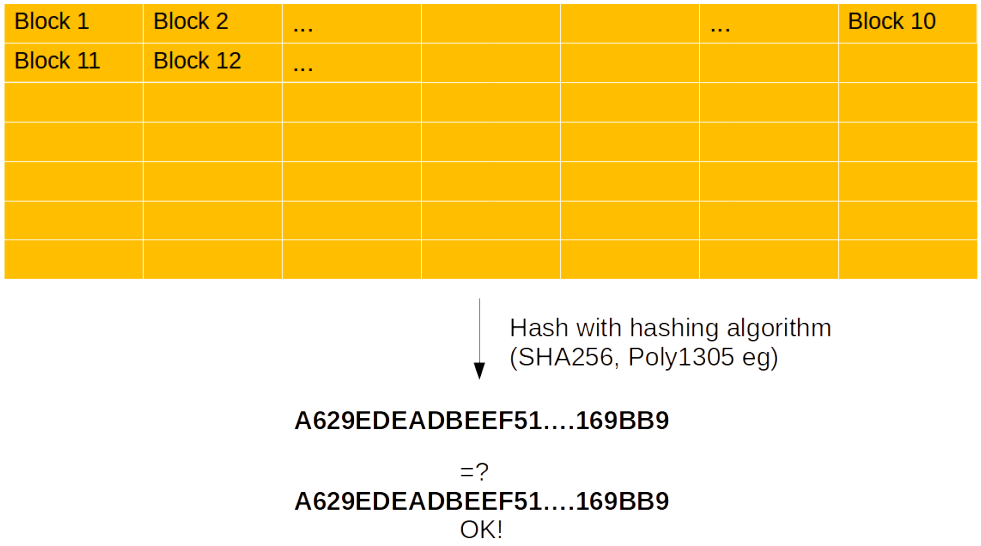

Figure 2.4. *Deterministic Check Step 3) Compare produced hash with stored hash from known state*

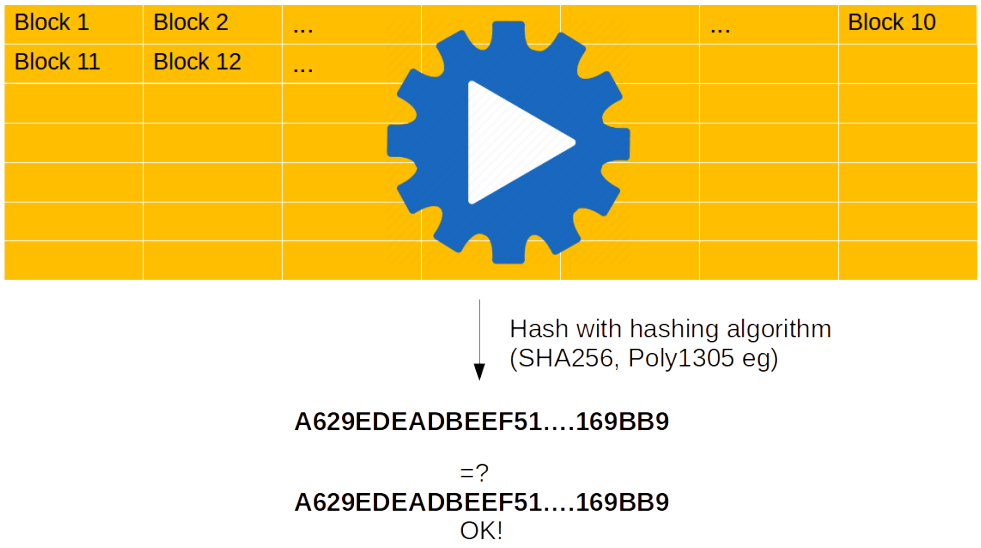

Figure 2.5. *Deterministic Check Step 4) If hashes match, continue with the next step. If not, handle failure accordingly.*

Probabilistic Secure Boot

In papers such as [10] and [11] probabilistic secure boot schemes are proposed. In those schemes, instead of verifying the whole memory area in question, we choose to split it into smaller pieces and only verify a subset of those. One such example is shown in Figure 2.6. Depending on the actual model, hashes of the correct state of these pieces are calculated and stored on secure memory either during manufacturing or at some point during which the system is in a secure state. In Figures 2.7 and 2.8 we show the calculation of the hash for every checked piece, and the comparison with the saved hashes. Depending on the result of the comparison (Figure 2.9), the scheme continues either by booting or by handling failure accordingly.

Probabilistic schemes offer a lower level of security than deterministic ones because there always exists a chance that some corruption has made it through the check. This can happen if the corruption lies in a region of the memory that has not been picked to be verified. Typically however, after the start of the application a full image scan is performed in the background, therefore decreasing the window of opportunity for any malware that has made it through to do actual damage.

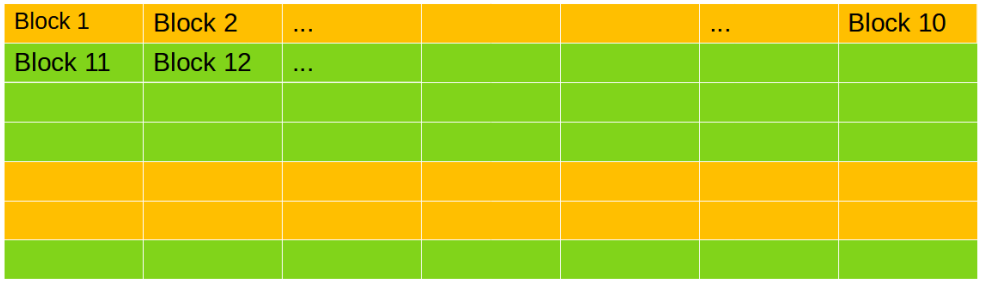

Figure 2.6. *Probabilistic Check Step 1) Choose random slices of memory to verify.*

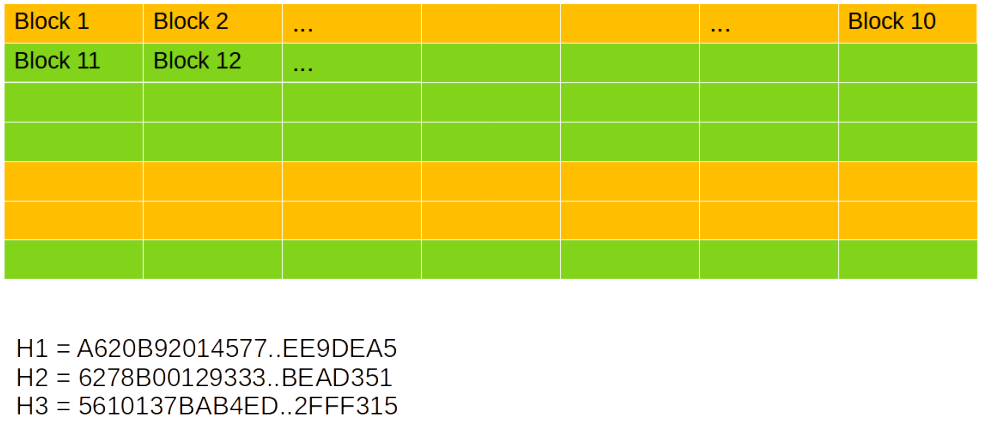

Figure 2.7. *Probabilistic Check Step 2) Hash each slice on its own, producing many digests.*

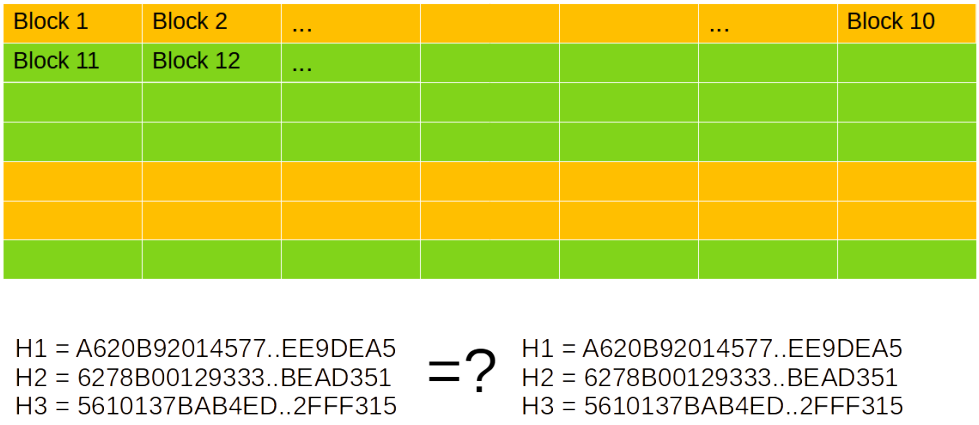

Figure 2.8. *Probabilistic Check Step 3) Compare every hash with its respective stored one, or hash the digests together to produce one final digest.*

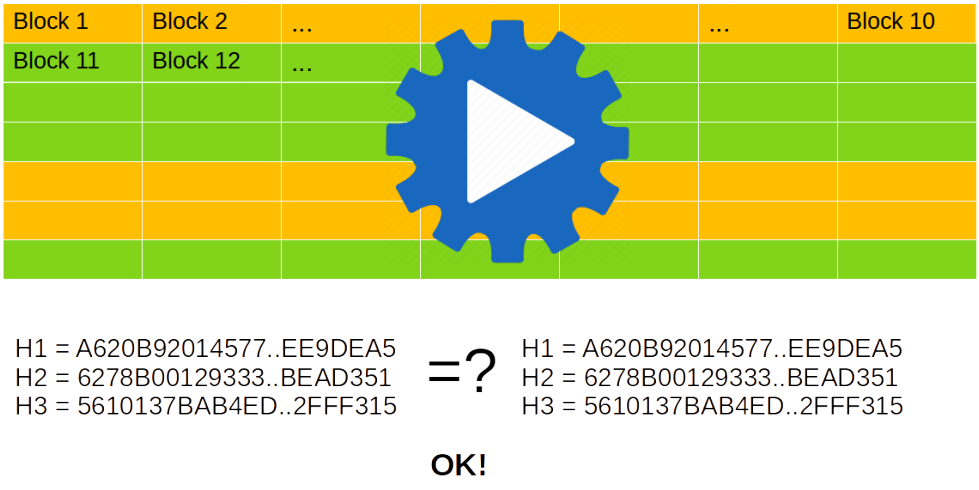

Figure 2.9. *Probabilistic Check Step 4) If the comparison succeeds, continue with the next step. If not, handle failure accordingly.*

Run-Then-Check

Other research [7] goes even further to have the system available as soon as possible. In these papers the secure boot sequence is postponed until after the application is running (Figures 2.10 and 2.11), and is performed typically in the background (Figure 2.12). If malicious code is found at this point, the device issues a reset and handles failure accordingly.

These schemes rely on the small time window they give the attacker's code to balance their lack of security. This essentially means that even in the worst case scenario where malicious code has found its way onto the system, due to the immediate check that follows the bootup, the time the malicious actor has to perform their further attacks is very limited, to the point where it can imply some form of security. System engineers can also utilize additional protection mechanisms, such as memory protection, from the point the system starts until the point the system is considered secured, in order to try and minimize the potential damage that can be done.

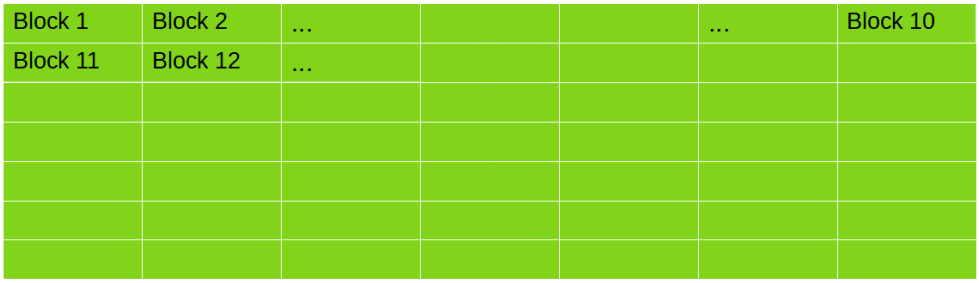

Figure 2.10. *Run-Then-Check Step 1) Do not perform any checks before starting the system.*

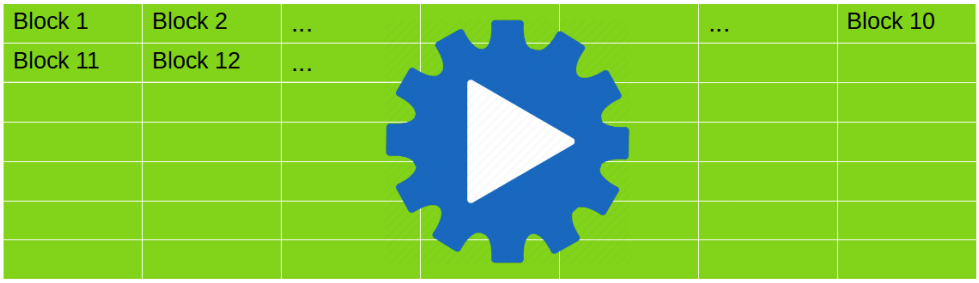

Figure 2.11. *Run-Then-Check Step 2) Start the system as is.*

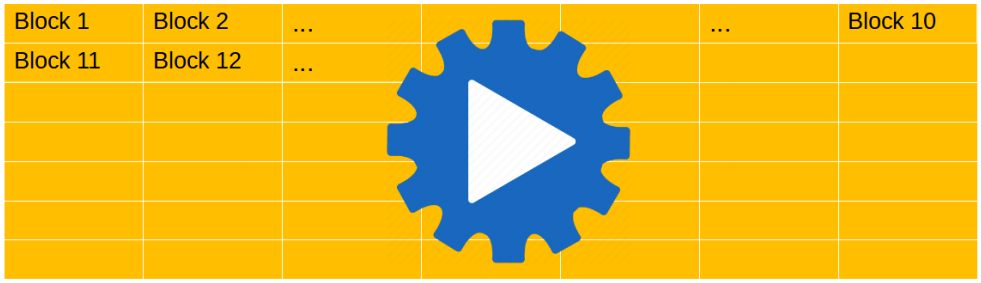

Figure 2.12. *Run-Then-Check Step 3) Only after start proceed to do a full measure in the background. Handle failures accordingly.*

Comparison

A high level comparison of the three main models is provided in Table 2.1. As expected, there is no better alternative in terms of offered security than using a full deterministic secure boot model that checks the whole image, which is guaranteed to find any existing alternations in the memory. In terms of speed however, it becomes apparent that probabilistic and run-then-check methods have the upper hand, due to the less computation time they demand. As for the complexity of the scheme, there is a bit more to take care of when writing probabilistic schemes, but this comes with additional customization options, such as how much of the memory to check, with how many cores etc.

	Safety	Speed	Complexity	Customization
Deterministic	✓✓✓	✓	✓	✓
Probabilistic	✓✓	✓✓	✓✓	✓✓✓
Run Then Check	✓	✓✓✓	✓	✓✓

Table 2.1. *Comparison between different approaches to Secure Booting.*

The existence of schemes that provide such a broad spectrum of balances between security, performance and complexity is due to the different nature of applications, some of which demand high availability where others have strong requirements for security.

Chapter 3

Prior Work

A lot of work has been put into fields related to our work. Those include Secure Boot schemes that try to squeeze time performance from the system, lightweight algorithms suited for small self-contained systems and general performance benchmarking method combinations.

We take a look firstly at other Secure Boot methods proposed in the bibliography, then we discuss different MACing and hashing schemes that could be useful and we mention other research that has helped us with our work.

3.1 Deterministic Schemes

3.1.1 A secure and reliable bootstrap architecture

The paper proposes a secure and reliable bootstrap architecture for computer systems that provides a trusted path for initializing the system and verifying the integrity of system software [6]. The architecture is designed to mitigate the risks associated with bootstrapping, which is a vulnerable and critical phase in the system startup process.

The proposed architecture employs a secure bootstrap loader that runs from read-only memory and provides a secure foundation for loading the system software stack. The bootstrap loader uses digital signatures to verify the integrity and authenticity of the system software components, including the operating system, firmware, and other system-level software. The bootstrap loader ensures that only authorized software is loaded into the system, and can detect and prevent the execution of malicious software, including rootkits, malware, and other forms of unauthorized code.

The paper discusses the design and implementation of the bootstrap architecture, including the use of cryptographic techniques such as hash functions, digital signatures, and public-key infrastructure. The performance and security characteristics of the architecture are evaluated and shown to be highly secure and reliable, with the ability to resist a variety of attacks, including physical tampering, software attacks, and supply chain attacks.

Overall, the proposed architecture provides a practical and effective solution for secure and reliable bootstrap initialization of computer systems, particularly in high-security and mission-critical environments. What is essentially described in this work is a very early secure boot sequence.

3.2 Probabilistic Schemes

3.2.1 Sliced Secure Boot

The sliced boot schema as proposed in [10], relies on the creation, storage and verification of fingerprints from the memory space. In the paper, the memory is being treated like separate "logical" memory blocks and these blocks are further sliced to get "memory cells". A memory block therefore consists of many memory cells in this context. The creation of the fingerprints as well as the selection of the fingerprints to be validated is ensured to be uniform and unbiased leading to each cell being protected by the same number of fingerprints. We can think of the process split in 3 phases: Setup, Verify and Monitor Setup Phase: Memory is considered as a matrix of d blocks (rows) each b cells wide (columns) and generates a fingerprint for each column-wise slice of memory, giving b $\times$ d memory cells. A CBC-MAC function builds each fingerprint with an ECU-specific secret key and the memory cells from the slice of memory. The HSM stores the key in protected memory and the fingerprints in unprotected memory. Verify Phase: "At each ignition cycle, the algorithm randomly selects one column, computes the signature, and compares it with the stored signature. When the microcontroller starts up, the HSM holds the main core in reset until it completes its initialization phase. While the main core is in reset, the HSM can be granted authoritative control and maintain compatibility with the safety concept."

Monitor Phase: After the successful probabilistic check, the algorithm checks the entire memory in the background as a series of tasks.

While a full memory check will be done after boot either way, this still is a probabilistic secure boot approach. This means that there exists a chance that a carefully crafted payload - or one at random - could execute. Several things need to be true for this to happen. Firstly, no continuous piece of the compromised code can be more in size than the size of a memory block, as at least one cell from each block is guaranteed to be checked. Secondly, the attack must be carefully planned to use jump commands to link the different code pieces. Thirdly, the fingerprints that was randomly selected at startup must not contain any compromised cells.

In the worst-case scenario where all of the above are true, the escape rate of malware is about 1.5% on the first boot cycle. With slightly better configurations this quickly drops to 0.03-0.2%. Bear in mind that with every consequent boot cycle the probability that a compromised check will be contained in the checked fingerprint increases.

Another weak point of the design is the relative inability to capture small changes in the code. Critical bit-wise changes on the code have a higher chance of being left out of the checks, and that is why the paper suggests manual handling of critical memory space by the user.

3.2.2 Accelerated Secure Boot for Real-Time Embedded Safety Systems

The University of Michigan-Dearborn, USA, and Rhein-Waal University of Applied Science-Kleve, Germany, published in 2017 this research [11]. They propose a dual-phase secure boot algorithm that balances the strong requirement for data tamper detection with

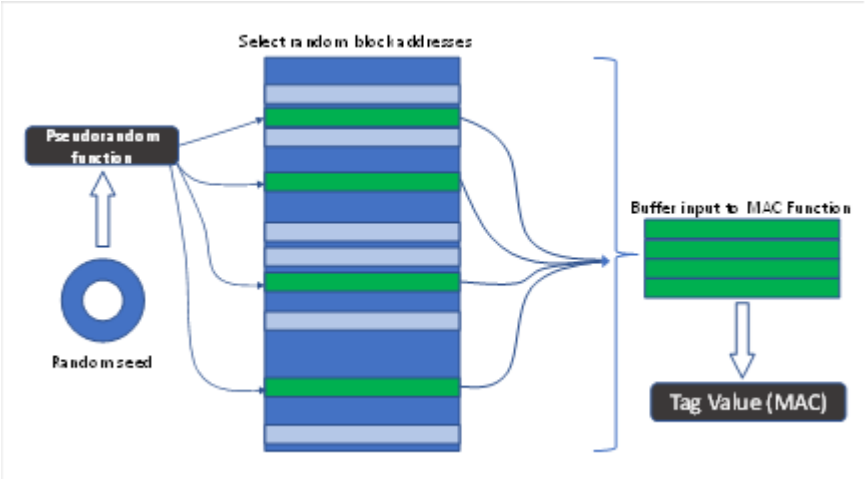

Figure 3.1. *The sampling scheme proposed. Only some blocks are randomly selected to be used in the verification.*

the strong requirement for real-time availability. A probabilistic boot measurement is executed in the first phase to allow the system to be quickly booted. This is followed by a full boot measurement to verify the first-phase results and generate the new sampled space for the next boot cycle. The dual-phase approach allows the system to be operational within a fraction of the time needed for a full boot measurement while producing a high detection probability of data tampering. They propose two efficient schemes of the dual-phase approach along with calibratable parameters to achieve the desired tamper detection probability. They evaluate the tampering detection accuracy within a simulation environment. Then they build a real system to evaluate the real-time performance using an automotive embedded microcontroller with a built-in Hardware Security Module (HSM).

An AES CTR DRBG module is used to create randomly sampled addresses belonging to a user provided image. Sampled data corresponding to the sampled addresses is then used to create a CMAC tag of the sampled image. The generated tag can then be used to verify whether or not modifications have been made to the hex image, with a certain degree of probability which depends on the size of the hex image, the modifications done to it and the chosen scheme for the procedure. This scheme is called "Random Block Sampling" (RBS) and illustrated on the below diagram. It showed high detection probability but only if the number of tampered bytes were large enough. This might not be suitable for applications of high criticality where tolerance for undetected block tampering is lower than what RBS can protect. To cope with low number of modified data blocks, they extend algorithm to perform a double sampling approach. The first level of sampling is done at the full firmware image level by dividing it into smaller blocks. The second level of sampling is done in each of these small blocks by randomly sampling within a few more bytes. If the full boot measurement fails, the HSM switches off the sampled boot measurement until the system has at least been fully booted successfully once. The attacker must now replace their altered blocks with a valid image to ensure a full download before they can repeat their experiments to find unamplified blocks.

3.3 Run-Then-Check Schemes

3.3.1 Secure Boot Implementation for Hard Real-Time Powertrain System

The paper uses two different modes for checking the integrity – Foreground Secure Boot and Background Secure Boot. [7] Foreground Secure Boot forces the CPU to wait until the verification check of the bootloader is completed. This is done before handling control to the application to continue. This sequential boot flow allows for security and deterministic integrity of the software but lacks performance in terms of speed and availability.

On the other hand, the Background Secure Boot allows the verifying of the bootloader to be performed in parallel with the application initialization. This approach allows for instant availability of the device and only later detects and acts against tampered code pieces, sacrificing prevention of compromised code running for the benefits of speed. This paper's interests are more oriented towards embedded devices that work closely with Engine Operation systems, and thus give high priority to fast system initialization after starts.

With these in mind, the paper's authors suggest a hybrid approach to Secure Boot that uses both foreground and background booting. While the bootloader will be checked in foreground mode, the application can run without prior verification. Next, the secure boot code will initiate the integrity check on all application memory blocks, but in background mode. The result of the check is to be delivered to the CPI from the HSM after each application memory block has verified. To further fine tune the system, it is suggested that the user manually appoints priorities to application blocks, with critical blocks enjoying higher priorities.

As with other similar schemes, the major drawback of this approach is that some parts of this code may be executed in the state of uncompleted integrity check, meaning that if malicious code is planted on the system, then it will certainly run until the full verification result comes in. This is why a strategy for integrity failure of executed code needs to be thoroughly planned if this scheme is to be used.

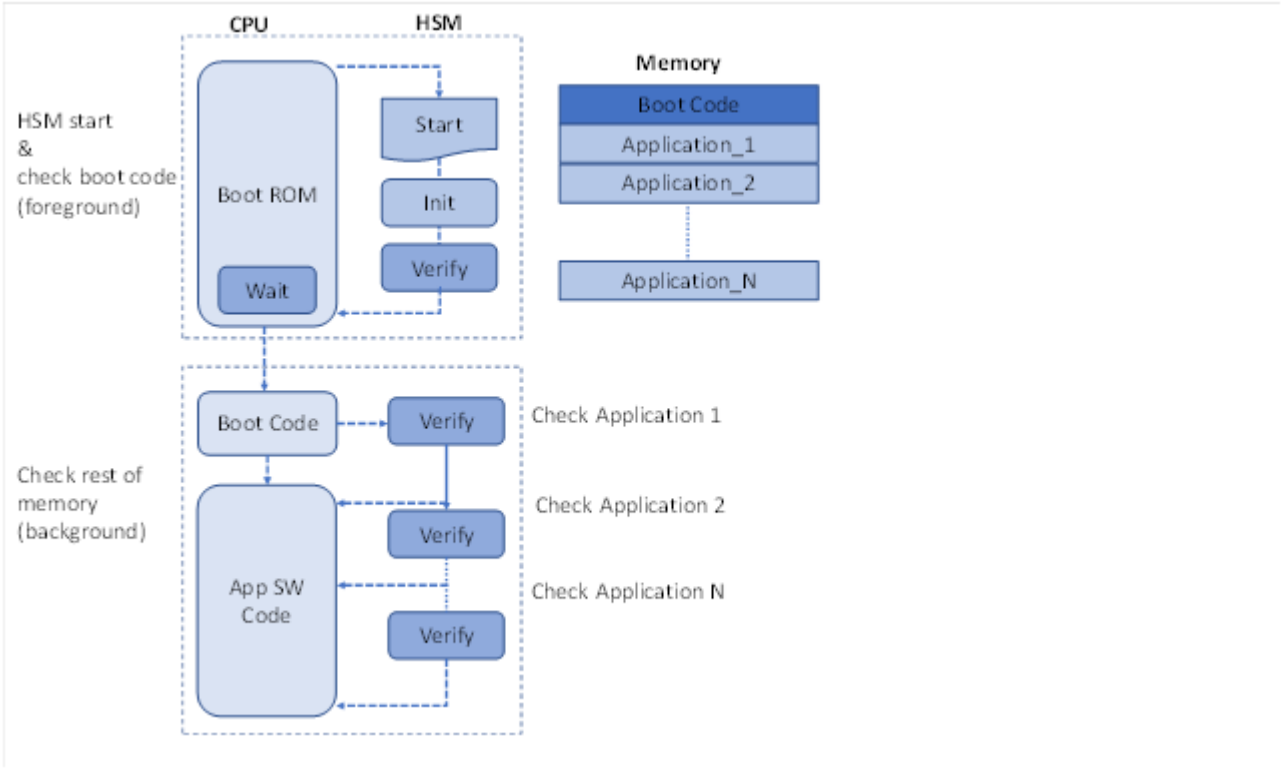

Figure 3.2. *Hybrid approach of using Foreground Secure Boot on the bootloader and Background Secure Boot on the Application.*

Chapter 4

Accelerating the Secure Boot

Our contribution is based on two pillars. First we looked for lightweight cryptographical hashing algorithms that could provide more performance without endangering the security of the process. We tested both algorithms widely used in the industry and academia so far as well as cutting edge ones. Second, we made the key observation that most of the secure booting process time is spent on the actual hashing period. Therefore there was a lot of performance to be gained by accelerating the hashing computation. In this work we parallelize this hashing step to produce lower times.

4.1 Hashing Algorithms

4.1.1 SHA256

SHA256 is a hashing algorithm that uses 32-bit words to perform its calculations. Alongside the rest of the implementation from its family, SHA384 and SHA512, SHA256 is widely used in various applications. Some of its most known applications include TLS and SSL, the SSH protocol, PGP as well as IPsec. Lately some cryptocurrencies used SHA256 for proof-of-works and to verify transactions, inlcuding most notably Bitcoin. SHA256 remains an approved algorithm according to NIST [12]. We used a widely available implementation from [13].

4.1.2 Poly1305

Poly1305 is a state-of-the-art hashing algorithm designed by D. J. Bernstein. [14] Poly produces a 16-byte digest of a message of any length using a 16-byte nonce and a 32-byte key. It is guaranteed to be at least as secure as AES and has very high speed. According to Bernstein and the measurements provided, Poly1305 is faster than both MD5 and CBC-MAC-AES. As per its author, there exists some form of inherent parallelizability for Poly1305, but no more info is provided yet [14]. Poly1305-ChaCha20 was adopted by Google to be used in TLS for both its speed and security. It is standardized in RFC 7539, RFC 7905 and replaced in June 2018 by RFC 8439. In 2015, it was also integrated in Cloudflare. For Poly1305 we used the offered implementation found here [15]

4.1.3 Chaskey

Chaskey is a hashing algorithm that can be used both to ensure message integrity as well as to authenticate users or even create random numbers. We use it as a MAC in our work. According to its creators, Chaskey is *"Currently deployed in commercial products by Tier 1 automotive suppliers and in major industrial control systems"*, It is very fast and compact, and was also designed with energy-efficiency in mind. As far as security, the 8-round variation remains unbroken for several years now, giving the 12-round variation a comfortable security margin. [16]. We used the implementation offered here [**?**].

4.1.4 AES128-CMAC

AES128-CMAC is a cryptographic message authentication code (MAC) algorithm that uses the Advanced Encryption Standard (AES) with a 128-bit key to provide integrity and authenticity of a message. One of the reasons we used AES128-CMAC was the existence of hardware acceleration in the automotive platform where we were evaluating, plus its usage in pre existing secure booting models there. Where used, we extracted and used the implementations offered inside the MbedTLS library found here [17].

A note on real-case scenarios: In this work we get our best results in terms of speed performance from Poly1305 and Chaskey. While we document the performance gains of both algorithms throughout our testing, our experience shows that Poly1305 is a little easier to integrate into existing real use-case projects, such as the one that our automotive research was based on. Because of its relatively better reception from the community, the large amount of applications it has already and the adoption from tech giants such as Google and CloudFlare, we feel that a solution featuring Poly1305 instead of Chaskey may have an easier time getting integrated.

4.2 Paralellization of the hashing

A typical secure boot process consists of a few standard steps:

- Power-on and hardware initialization: The system powers on, and the firmware (BIOS or UEFI) initializes the hardware components and performs a self-test to ensure they are working correctly.
- Bootloader loading and validation: The firmware loads the bootloader from a trusted storage device such as a read-only memory (ROM) or a secure boot partition on a hard drive. The firmware is either considered secure by definition or validated cryptographicly before that. The bootloader is then validated using cryptographic techniques to ensure it has not been modified or tampered with.
- The chain of performing integrity checks continues as long as the developer wishes with:
  - the kernel

– the operating system
– device drivers
– applications and so on

A variation of these steps is already shown in Figure 1.1, which shows the Flow of Trust through a system.

Especially in lower end systems, such as microcontrollers used in automotive or industrial scenarios, intermittent computing systems or other highly specialized controllers, most of the time of the secure boot process is spent actually hashing the memory. The less powerful processors that these embedded devices are usually equipped with are slower, thus making the hashing workload take loads of time. Therefore there is a lot of performance to be gained by targeting this specific computation step.

Our proposal aims to reduce the time it takes to perform all the checks. In our automotive work we exclude the bootloader check from that group, and choose to keep the bootloader check completely inside the platform's TPM because of the project's security requirements. What we describe next can be applied to every integrity check during secure boot.

Instead of hashing the whole image to be verified sequentially with one core, we propose the following. We split the image into equal parts according to the amount of available cores. We then assign each core to one of those memory chunks. Each core must hash their own part using the same algorithm, and at the end the results are gathered.

In the implementation of the idea, we are using an abundance of cores, ideally as many as there are available in each system (although we saw that some algorithms scale poorly). As each core manages a small chunk of memory and calculates a hash at the end, we need to gather all these hashes somewhere. A possibility for that is to have one core be the "master" and the rest of the cores be the "slaves".

The "master" core would be responsible for the following:

- Initialize the rest of the cores
- Compute your own hash for you chunk
- Wait until every core is ready
- Gather all the hashes and verify with the HSM

The "slave" core on the contrary only needs to use the address and the size of the chunk it has to verify and do the hashing. This hashing process is independent of any other work done in other cores, so no synchronization is needed during it between them. After completing, the core needs to report to the "master" core and communicate its result through some mechanism (could be shared memory, signals, pipes or any other implementation depending on the system).

Following are two pseudo-algorithms that describe how the we implemented this process in the automotive part of this work. Due to confidentiality policies the actual code cannot be shared. Algorithm 1 shows the procedure followed by the "master" core, while Algorithm 2 follows the "slave" cores.

µ 4.1: *An example of the process on "master" core*

ί: Initialize Core 0.
ί: Initialize rest of the used cores.
Hash(Memory Address, Memory Size, Cores, Algorithm, Result)
Get result from hashing.
while C **do**ores have still not reported result.
Busy Wait core 0.
end while
′E: All cores returned valid result.
Verify result with HSM.

µ 4.2: *An example of the process on "slave" core*

ί: Initialize self
Hash(Memory Address, Memory Size, Cores, Algorithm, Result)
Get result from hashing.
′E: Result is valid.
Notify master core and send result.

4.2.1 Deterministic and Probabilistic

This approach is not limited to the traditional full image deterministic checks. While applicable in the standard variation described above, it may be the case that parallel secure booting may have the most to give within the context of probabilistic secure booting schemes. In Figure 4.1 we see a possible splitting of a memory region between the available cores. It is not mandatory that the number of cores and the number of pieces match.

In Figure 4.2 we see a memory split for a probabilistic secure booting scheme. In such a model, only some parts of memory are chosen to be verified. This results in a plethora of scattered memory areas that need to be hashed. With our proposal, it is possible to assign every such piece onto a responsible core and have all these independent workloads happen simultaneously. In Figure 4.2, each piece is given to a different core, but again, it is not mandatory for each core to only be responsible for one single chunk. To the contrary, if the number of chunks exceed the number of cores, then by definition some cores will have more work to do.

Alternative schemes also exist, in Figure 4.3 we show such an alternative scheme where the chunks of memory that will be used for verification are comprised of smaller randomly picked pieces throughout the memory space, and not necessarily from continuous segments.

4.3 Security

Add figure showing the FBL being verified before the APP hashing starts. The question of whether running verification code in non-attested cores - the "slave" cores used for hashing on the next step - arises naturally. How can we extend the chain-of-trust outside the HSM whilst using an untrusted execution environment?

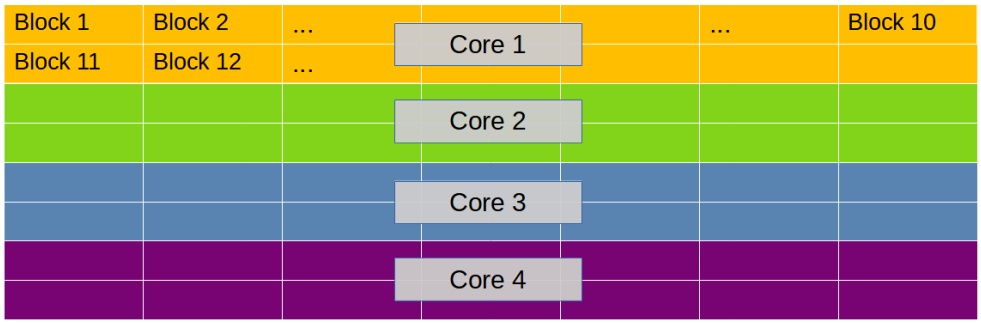

Figure 4.1. *Parallel Deterministic Secure Booting. The image is split into equal parts, and each core is responsible for hashing one of them. The hashes are later gathered and handled as described in the Verification section.*

To address this question, we must first note two things.

Firstly, throughout this work, we consider the idea of "Secure Boot" as secure by default. Secure Boot has been around for more than 20 years and is to this day widely used in the industry, becoming the standard for building trust in your system from Power-On [18] [19] [20]. Although there have been attacks capable of bypassing secure boot on systems that use it [21], these types of attacks most regularly exploit some outside component of the system, either some poisoned hardware or some side-channel attack to compromise it, and do not target the secure boot process as such. Therefore we can consider it as "sufficiently safe" for our purposes.

Secondly, our model is based on parallelizing the hashing part of the secure boot process. The code responsible for orchestrating this work resides inside the system's first bootloader (FBL). This bootloader is verified inside the TPM before it runs. In our case, when wokring on the actual hardware designed for automotive, this verification is handled by the internal core of the HSM provided by the platform. By verifying the FBL before running anything outside the TPM we can guarantee that the code that will run the parallel hashing is indeed intact.

Our model does not further meddle with the secure boot process, that is after ending the parallel hashing step and the handling of the resulting digests. Every line of code that runs up to that point has already been verified, because it resides inside the first bootloader.

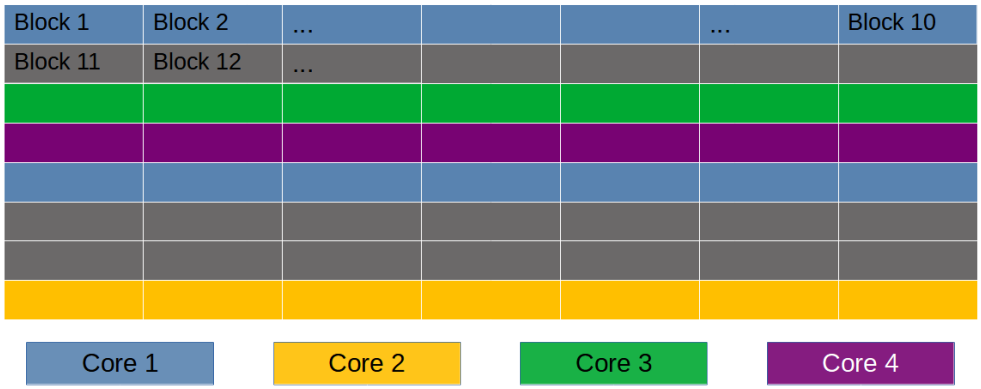

Figure 4.2. *Parallel Probabilistic Secure Booting. Some parts of the image are chosen for verification and split among different processors.*

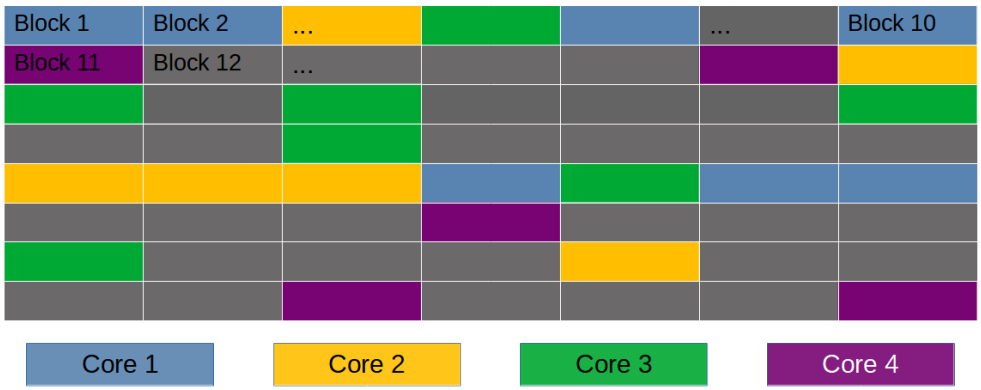

Figure 4.3. *Alternative Parallel Probabilistic Secure Booting. Some works have proposed a random approach at choosing the memory bits that constitute a memory region. Such approaches are also supported by the multicore hashing proposal, as every core hashes the corresponding resulting chunk of memory.*

4.3.1 Deterministic Schemes Security

In Deterministic approaches we verify the whole memory image of the next step in the secure boot process. It is therefore - due to the mathematical attributes of the hashing functions themselves - guaranteed that if a memory corruption exists within that range, the hashing function will produce a completely different result than the one stored in secure storage. As far as the deterministic secure boot process is concerned, it is impossible to sign a tampered memory area with the same signature its intact counterpart would have.

4.3.2 Probabilistic Schemes Security

There are several ways to quantify the security level offered by a probabilistic scheme. In our work we use a metric called the Detection Rate. The Detection rate has already been used as a metric in previous works [11]. To figure this rate, we run a large number of simulations of secure booting a system with the according probabilistic scheme in place, with the difference that we corrupt the memory range a bit. After each boot, we see whether the scheme was able to pinpoint the corruption, and count this as a hit. By measuring how many hits over the total booting sequences we get and dividing we come out with the detection rate of the model.

Detection rates in probabilistic schemes are analogous to the size of the corruption in memory, with bigger corruptions being easier to model. The Detection Rate is more closely related to the specific permutation of the variables of the specific model implementation (e.g. the size of memory, the size of the cells, the percentage of memory the implementation checks etc.) rather than the general probabilistic scheme itself. Nevertheless, it gives a good heuristic on the safety standards a specific iteration of such a scheme can provide.

4.4 Verification

Due to the nature of parallelizing the hashing, we end up with more than one hash. In traditional secure boot, the product of the hashing is one unique digest, which is sent to the HSM to be compared with a known hash.

The need arises to utilize all the produced hashes in order to verify the state of the software in question. That can be done in two different ways:

- Combining the resulting hashes into one final hash and comparing that with the stored value.
- Sending all the different digests to the HSM and comparing each one with a respective stored one.

4.4.1 Combining the Hashes

The resulting hashes are equal to the amount of chunks checked. With this approach, one of the used cores (a master core) is to gather all the hashes, concatenate them into one longer string and hash the resulting object. The final hash can then be sent to the HSM.

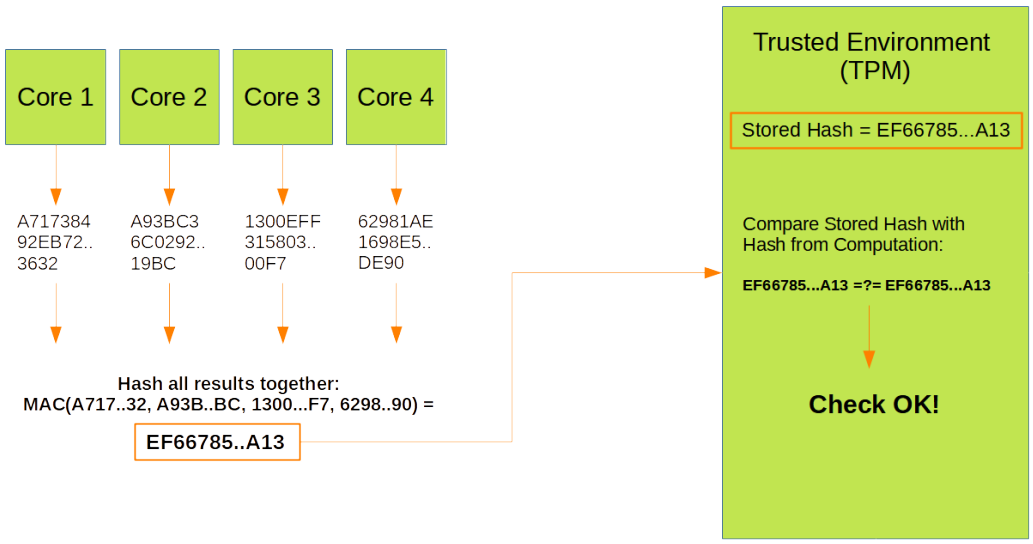

Figure 4.4. *Concatenating the hashes together, then hashing them again to produce a final digest. The product is sent to the HSM to be compared with the stored value.*

Concatenating the digests and then hashing again is not the only way to combine the results. Some other form of corelation may be used, for example XORing the hashes or doing some more sophisticated computation on them, or even using the concatenation as the final product. However, whichever process is chosen should also be performed on the first boot of the system, so as to create the hash stored in the HSM.

In Figure 4.4 we describe such a process using 4 cores.

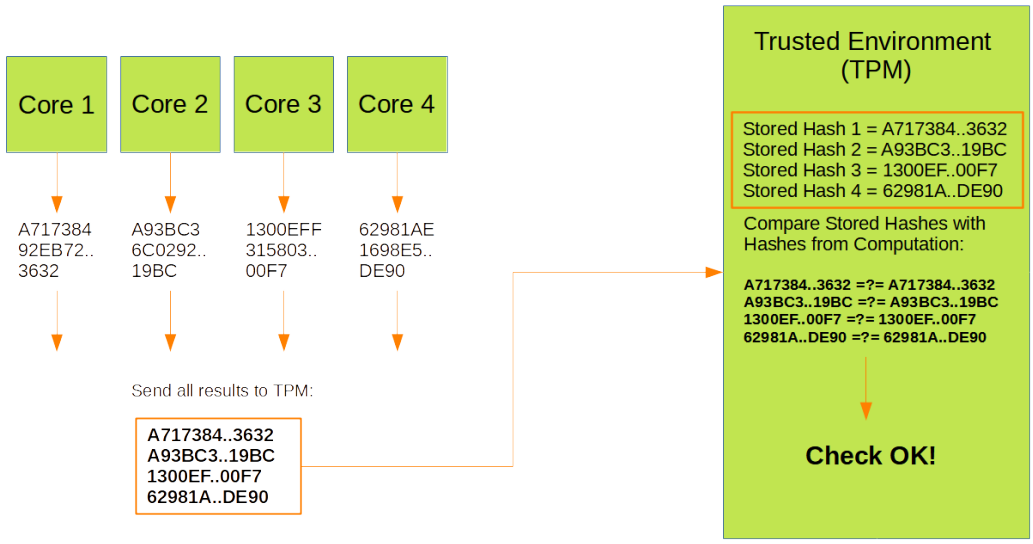

Figure 4.5. *Send all the produced hashes to the trusted environment. This requires hashes of the same software pieces to already exist in the TPM.*

4.4.2 Sending all the Hashes

After all cores are done with the hashing of their piece, all the hashes are to be sent to the TPM. Depending on the actual hardware limitations and extra security, it may be possible that only one core can communicate with the TPM. If that is the case, the digests are gathered on that core and then sent all at once to the trusted environment. Otherwise, it may be faster for each core that finishes to send their hash directly, so the comparison can immediately start.

In Figure 4.5 we describe such a process using 4 cores.

Chapter 5

Performance

5.1 RISC-V

5.1.1 About RISC-V

RISC-V ("risk-five") is an open instruction set architecture, with its strong point being that it is completely open source. As such, it has reshaped the landscape of the chip market, especially in the last few years, with big organizations like the European Union embracing it as a very strong contender for the chip industry [22]

With the liberty to test on our own hardware for this part of the experiment, we wanted to broaden the use case of the parallel secure boot idea that we are proposing. Some of the reasons for picking the RISC-V architecture for this next part of our work are the following:

- It is open-source, the whole codebase is publicly available and free for any usage. We want to keep our implementation open to the public so we strive to keep every component of our implementation open-source.
- RISC-V has added security-oriented primitives that make memory isolation possible, which is a prerequisite for having TPMs. Most notable among those primitives is Physical Memory Protection (PMP).
- As we will discuss later, finding hardware with dedicated TPMs or HSMs can be very tricky. One alternative is to use on-the-fly dynamically created trusted execution environments, such as Keystone enclaves [23]. Keystone is a framework that does exactly that and runs on RISC-V processors.
- As RISC-V is booming at the moment, there are a wide variety of expanding open-source cores and other products. To broaden the scope of our proposed implementation, it makes sense to use RISC-V as a basis, and create a product that can possibly work on any RISC-V architecture.

RISC-V Modes

RISC-V Architecture offers three privilege levels - Machine, Supervisor, and User. Each of these levels is limited to an appropriate set of functionality. The most privileged mode

is the Machine Mode (M-Mode), followed by the Supervisor Mode (S-Mode) with the User Mode (U-Mode) being last.

In RISC-V, the analog of what we consider a thread in mainstream computing systems would be a *hart*, or a Hardware Thread. At every time, a hart is running in some specific privilege level (M, U or S) that strictly defines the operations it may and may not perform. In this sense, privilege modes are used to guarantee protection between different entities on the software stack. Not all RISC-V based systems have to implement all these modes however, although implementing at least the M-Mode is required.

RISC-V Performance Counters

To get as accurate results as possible we utilized built in hardware performance counters from the RISC-V architecture set. RISC-V offers up to 32 different performance counters that are accessible via read-only registers. Three of those counters are standardized as CYCLES, TIME and INSTRET and offer respectively cycle count, real time clock and instructions retired.

To measure the time and the cycles it takes for our implementations to run, we used the CYCLES and TIME registers. Because of our program running on S-MODE, it is not allowed by default to read M-MODE registers. As per the RISC-V Privileged Architecture Documentation: *"The counter-enable register "mcounteren" is a 32-bit register that controls the availability of the hardware performance-monitoring counters to the next-lowest privileged mode."* In order to read from them, we needed to modify the according "scounteren" and "mcounteren" entries in the device tree.

5.1.2 Board Information

This part of our evaluation was deployed on a Rocket Chip FPGA, the Xilinx U250. [24] The platform featured 32 cores. For these experiments, no hardware acceleration for any algorithm was specifically used, in order for our results to be applicable in as broader a scope of microcontrollers and applications as possible.

5.1.3 Comparison Points

We evaluate on top of two different secure booting approaches, the deterministic and probabilistic ones.

All the results from this part of the evaluation measure the hashing time, as in the time it takes to perform the hashing of the image step in the secure boot process.

For every experiment we show in the same diagram the total cycles it took to complete the hashing, but we also show how much of that time was spent in the creation of the threads and how much time was spent waiting for these threads to join back.

For the Deterministic approach, we study the performance of three hashing algorithms:

- SHA256
- Poly1305

- Chaskey

AES128-CMAC was left out of our experimenting after our early results showed it lacked a lot in terms of performance compared to even the slowest of the aforementioned algorithms, SHA256. AES128-CMAC is still used in the industry in specific applications but mainly because of the existence of hardware acceleration, and not because of its inherent speed. In our results we measure a wide range of memory sizes to be verified, specifically 2/4/16/64/128 MB. This is based on the observation that secure boot measures images on many different steps, and the images in question can range from a few MBs (e.g. a bootloader) up to some hundred megabytes or more (e.g. a smaller or larger OS, an RTOS or similar).

For the Probabilistic approach, we study the performance of the same three algorithms in a range of schemes. We take measurements for checking different sizes of memory for the same reasons described above.

We also take into consideration different percentages of memory to be verified, 2/5/10/25/50%, to showcase the different implementations that an application may put to use. An application with stricter security requirements may not be happy with checking only 5% of memory and may choose to go over that limit - accordingly very high demands on time performance may see this percentage drop significantly.

Lastly we try to quantify the offered level of security of these probabilistic permutations by using the "Detection Rate" metric, as described in Chapter 3.

5.1.4 Results

Deterministic Secure Boot

In this experiment we hashed images of varying sizes with single and multi-core implementations. When using more than one core, the image was split into parts equal to the amount of used hardware cores and distributed amongst them.

In all following graphs exist 3 lines - a blue one for Total Time taken, an orange one for the Thread Creation Time, the amount of the total time that was spent for creating the hashes and splitting the work between them, and a green one for the Thread Joining Time, or the time from the moment the first thread started hashing until the moment the last thread finished its work and rejoined. We group the results by the algorithm used.

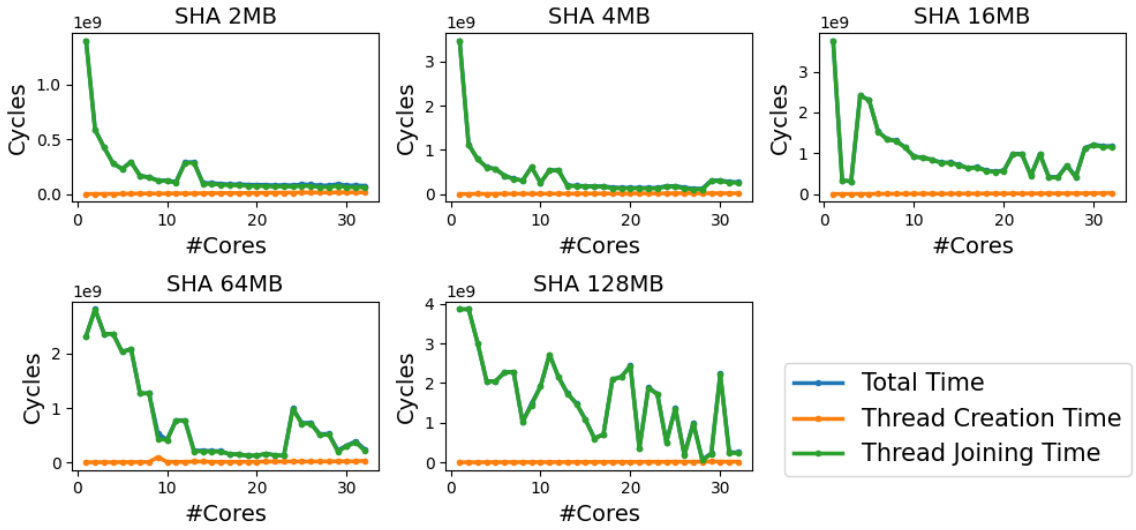

Figure 5.1. *Results for hashing time (in cycles) using SHA256 for ranging sizes from 2*

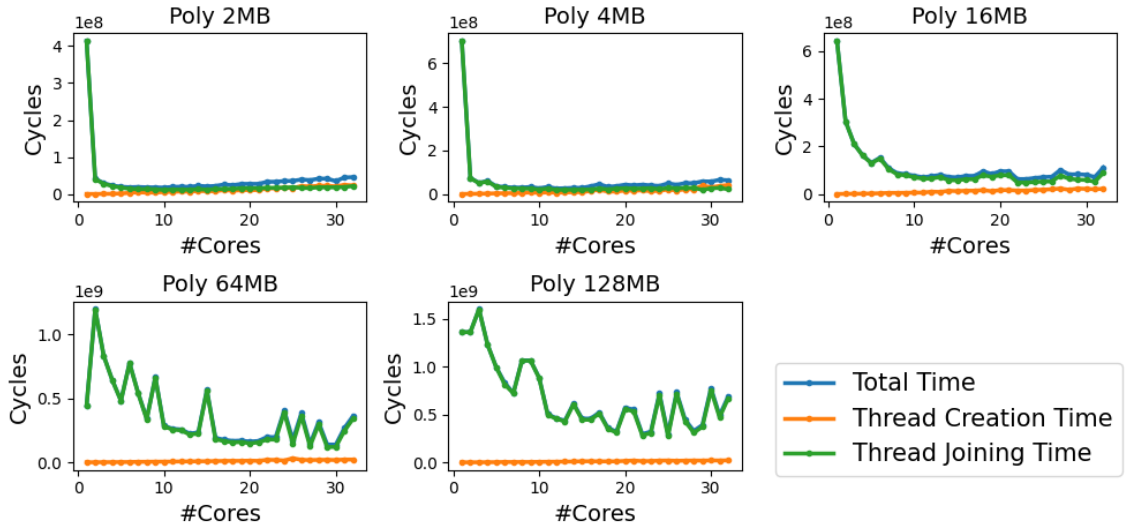

Figure 5.2. *Results for hashing time (in cycles) using Poly1305 for ranging sizes from 2*

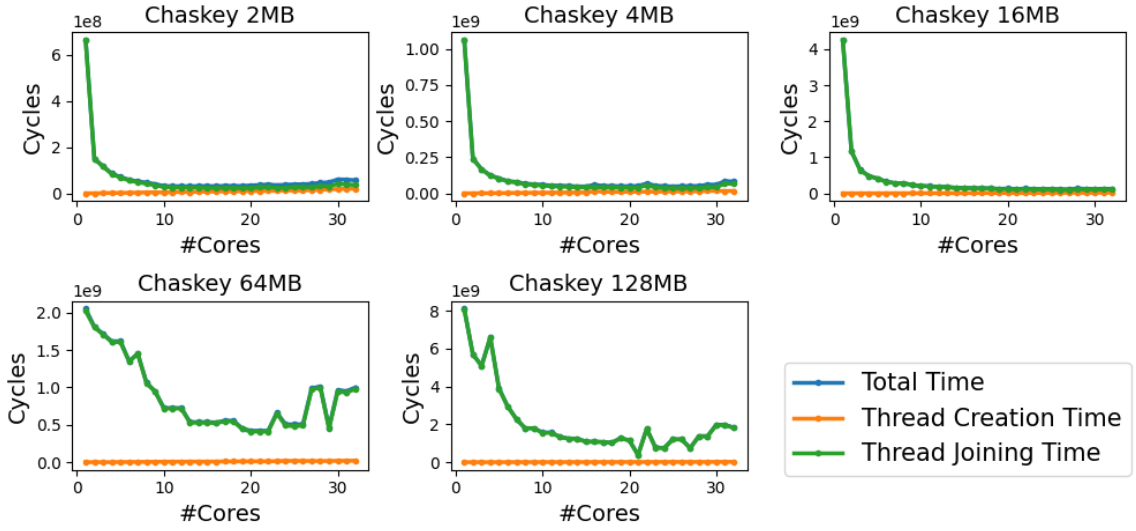

Figure 5.3. *Results for hashing time (in cycles) using Chaskey for ranging sizes from 2 to 128 MB.*

Conclusions

In graphs 5.1, 5.2 and 5.3 we see the performance of hashing different memory sizes, using from 1 up to 32 processors.

We find that for smaller image sizes, Poly and Chaskey outperform SHA256, especially when using fewer than 10 cores. As however image sizes and pieces checked grow, both Poly1305 and Chaskey lose a bit of performance. Poly1305 is still better than SHA256 in large sizes, but is now also a better choice than Chaskey, which does not scale well with more cores and bigger sizes.

We show that Poly1305 is a well rounded algorithm that outperforms its opponents on average across sizes, making it therefore a very strong candidate for multi-core deterministic secure boot implementations.

Our results also show that algorithms tend to be inconsistent across cores, especially Poly and SHA. While some of these peaks and dips that occur can be attributed to measurement errors during the experiments, we also think that this phenomena also gives us insight into the algorithm design itself, since such thing was not (at least not at the same scale) observed with Chaskey, neither here nor during the probabilistic secure boot experiments that follow.

Probabilistic Secure Boot

For these experiments we slice the memory into 100 pieces. We then randomly select a number of those pieces to be verified. We run tests checking 2, 5, 10, 25 and 50 pieces out of 100 total, for the various memory sizes we previously used.

In all following graphs exist 3 lines - a blue one for Total Time taken, an orange one for the Thread Creation Time, the amount of the total time that was spent for creating the hashes and splitting the work between them, and a green one for the Thread Joining Time, or the time from the moment the first thread started hashing until the moment the last thread finished its work and rejoined. We group the results by the percentage of memory that was checked.

SHA256

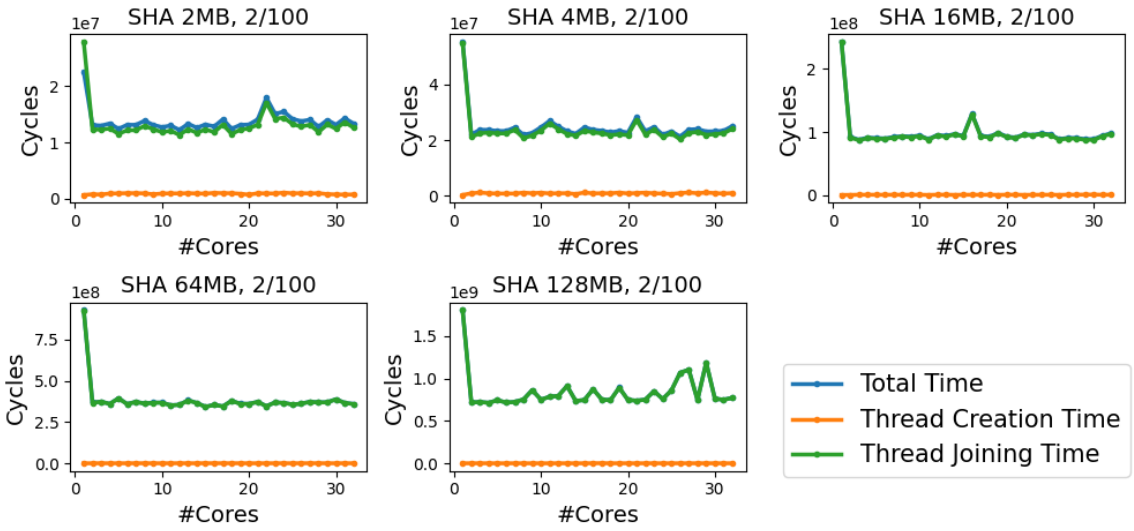

Figure 5.4. *SHA256 results for checking 2 out of 100 pieces of memory, for different*

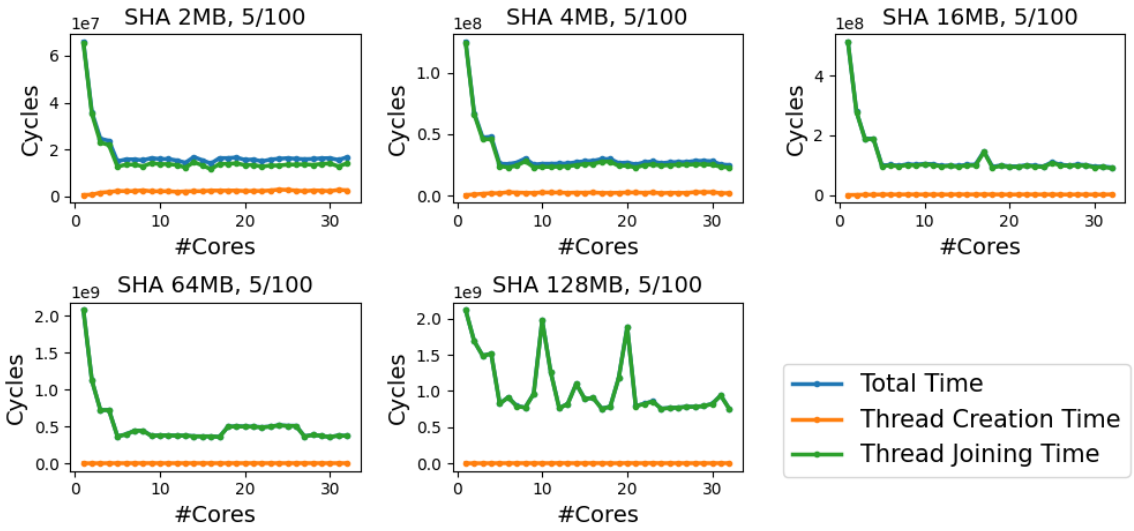

Figure 5.5. *SHA256 results for checking 5 out of 100 pieces of memory, for different*

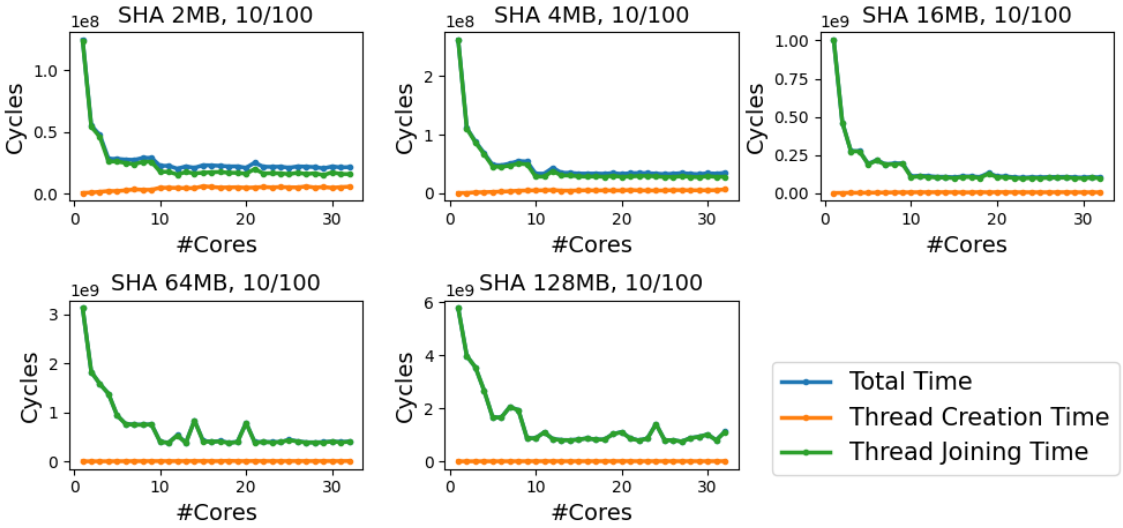

Figure 5.6. *SHA256 results for checking 10 out of 100 pieces of memory, for different memory sizes.*

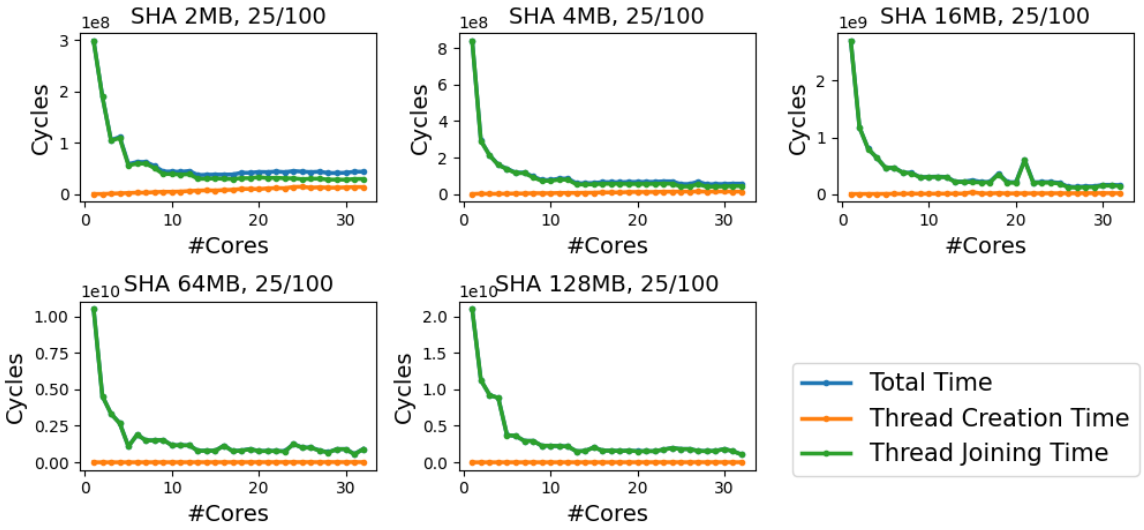

Figure 5.7. *SHA256 results for checking 25 out of 100 pieces of memory, for different*

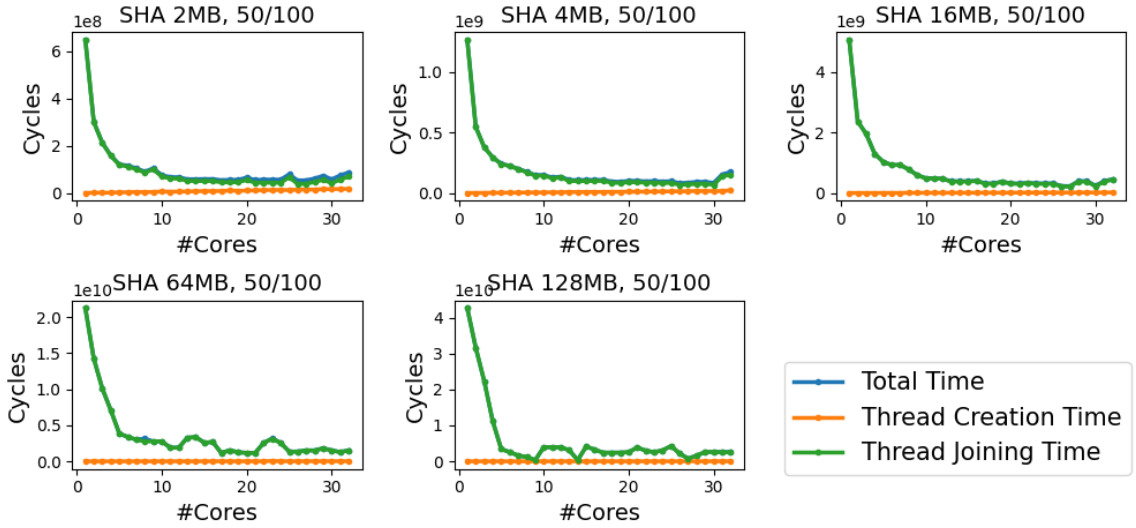

Figure 5.8. *SHA256 results for checking 50 out of 100 pieces of memory, for different memory sizes.*

In figures 5.4 to 5.8 , we see the performance in cycles for checking 2, 5, 10, 25 and 50 pieces out of the total, by giving whole pieces to the cores to hash.

For the SHA results, scaling works as intended, and the performance gain stops when the cores used are the same with the pieces checked. The cycle count is bigger compared to the other implementations, both at single and at multi-core implementations, making SHA256 the slowest of the three algorithms.

An observation stemming from the results is that they are somewhat inconsistent throughout experiments. In some cases, a few dips and peaks occur for specific core values. We attribute those to the occasional measurement error, as well as to the SHA256 algorithm itself. These inconsistencies also become larger the bigger the memory size gets, pointing to cache locality issues and the algorithmic design of SHA256. However, the more meaningful conclusions of our experiments come from the observation of the generic tendency of the performance, and not so much from the specific value pairs of cores - cycles, since those can also be prone to measurement errors.

In some cases we produce super-linear speedups. This could be explained through the cache locality effect. Superlinear speedup might come from the fact that not only the number of processors increases, but also amount of cache. By increasing this amount of cache, it may be easier for other processors to get hold of their data from some cache level because of an earlier load into said cache.

Poly1305

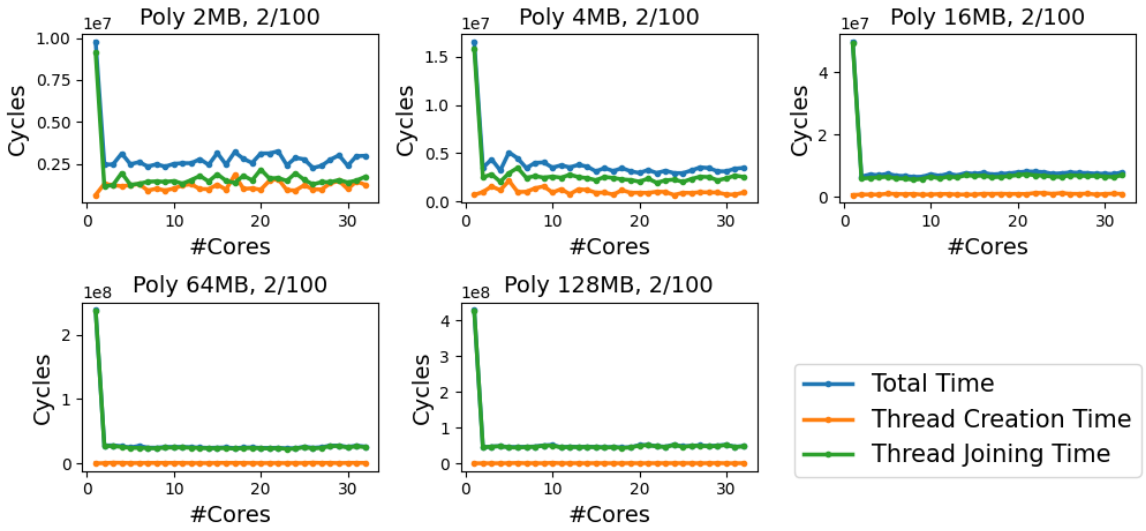

Figure 5.9. *Poly1305 results for checking 2 out of 100 pieces of memory, for different*

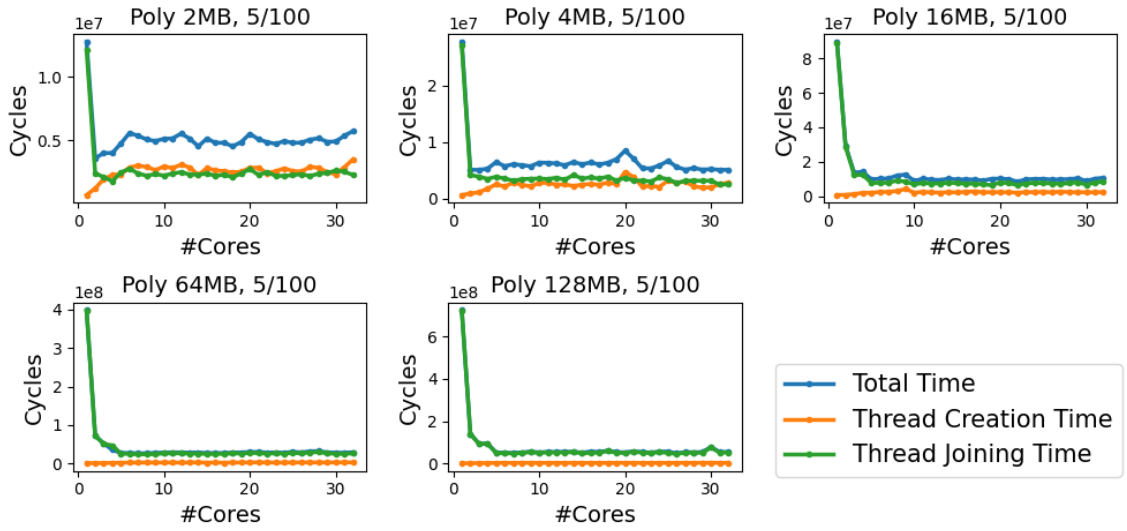

Figure 5.10. *Poly1305 results for checking 5 out of 100 pieces of memory, for different*

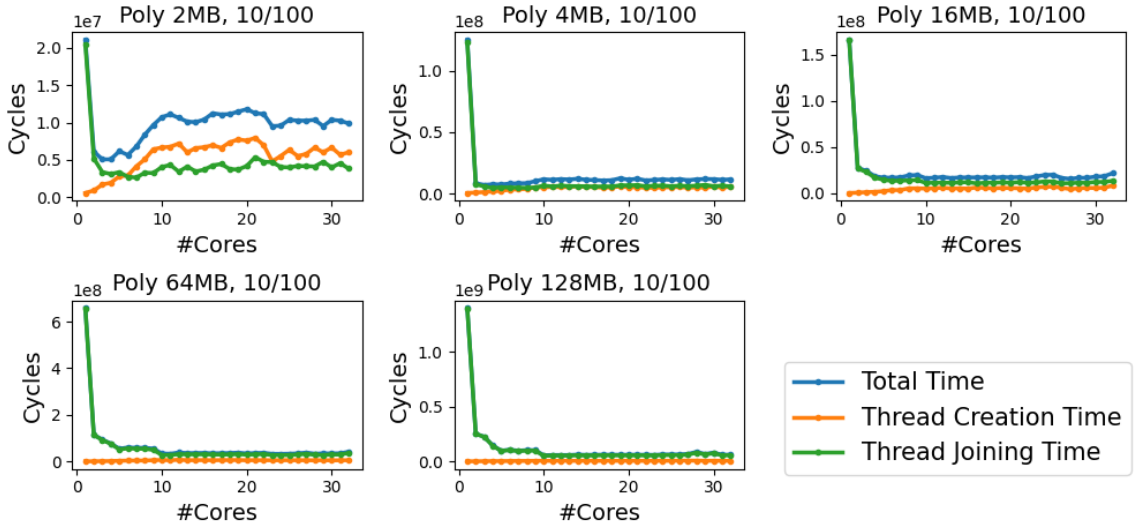

Figure 5.11. *Poly1305 results for checking 10 out of 100 pieces of memory, for different memory sizes.*

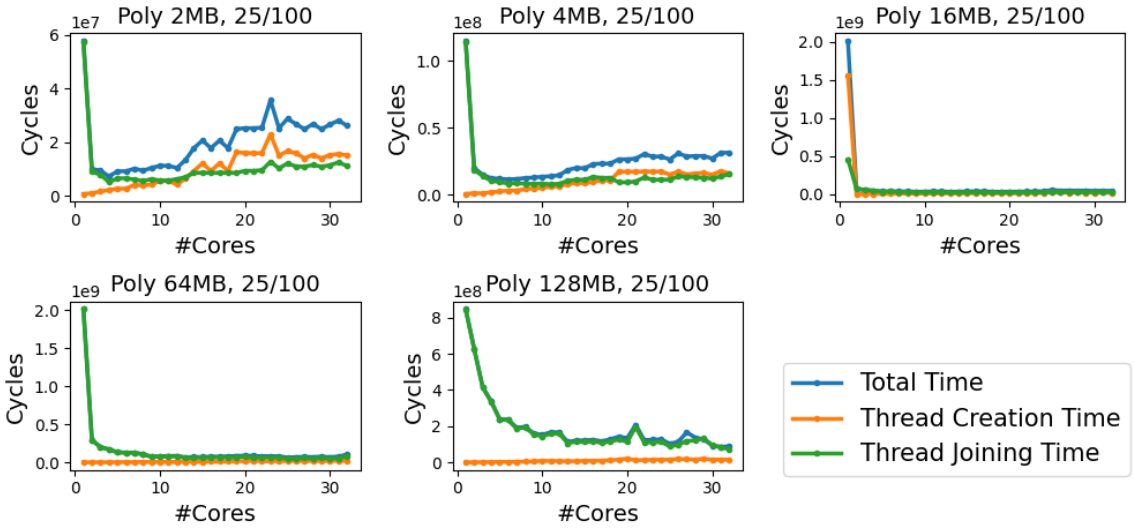

Figure 5.12. *Poly1305 results for checking 25 out of 100 pieces of memory, for different*

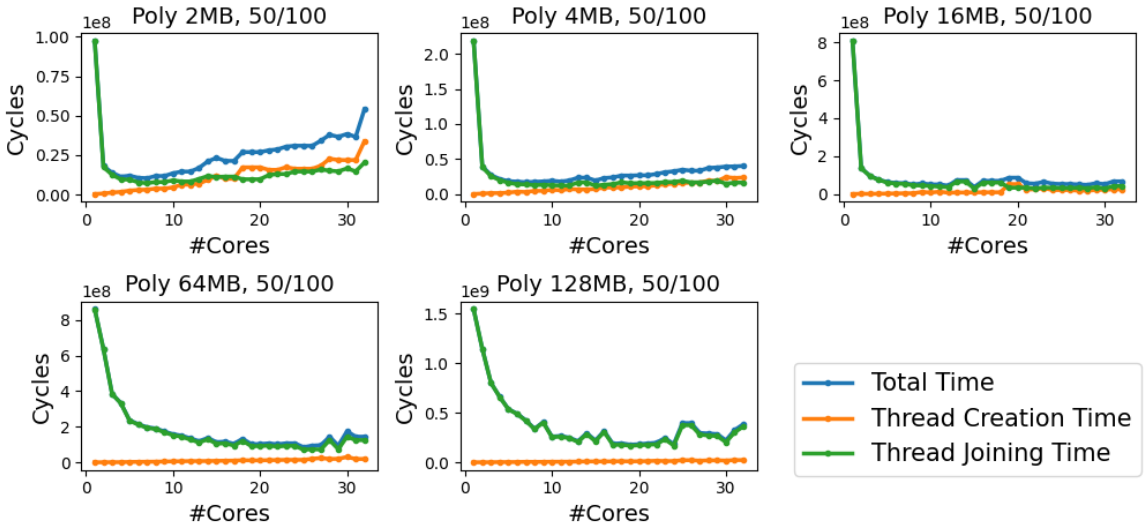

Figure 5.13. *Poly1305 results for checking 50 out of 100 pieces of memory, for different memory sizes.*

In Figures 5.9 to 5.14 we see the performance in cycles for using Poly1305 to check 2, 5, 10, 25 and 50 pieces of memory, as explained earlier.

Again, just like when using SHA, in most cases we tend to keep getting performance out of the scheme using up to the same number of cores as the number of pieces we check. Poly is much faster than SHA in all implementations, even in the single-core ones. Poly1305's performance

Poly1305 has another feature - the differences from the single-core to the multi-core implementations are big. We explain this difference in the conclusions, while we suggest that it is the product of caching locality and the inherent nature of the algorithm.

Chaskey

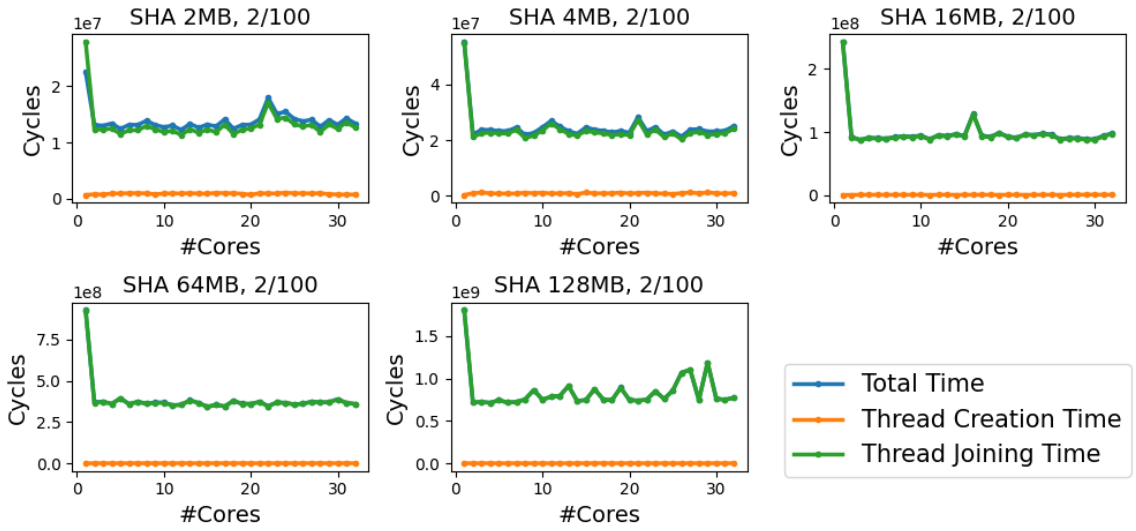

Figure 5.14. *SHA256 results for checking 2 out of 100 pieces of memory, for different*

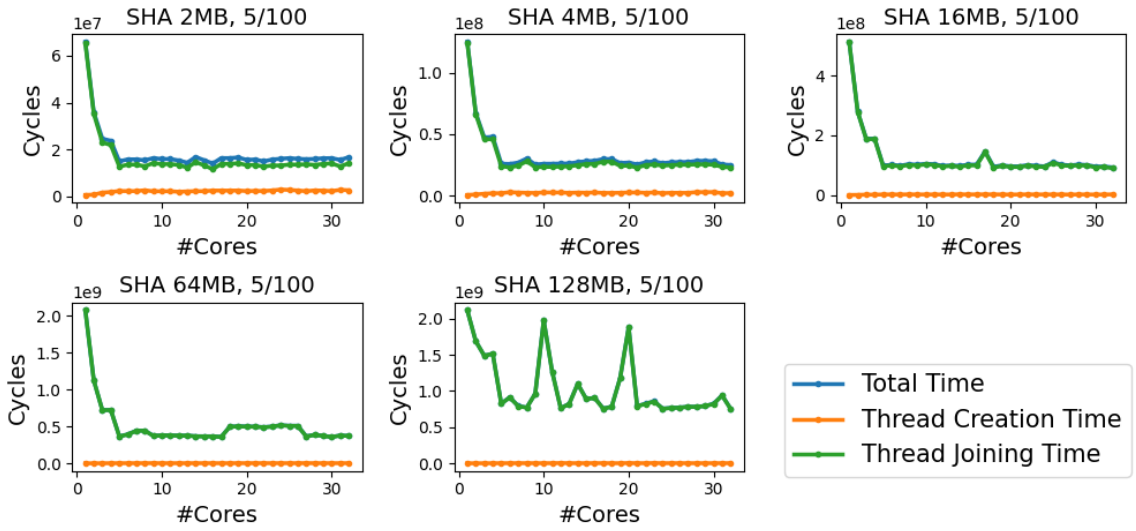

Figure 5.15. *SHA256 results for checking 5 out of 100 pieces of memory, for different*

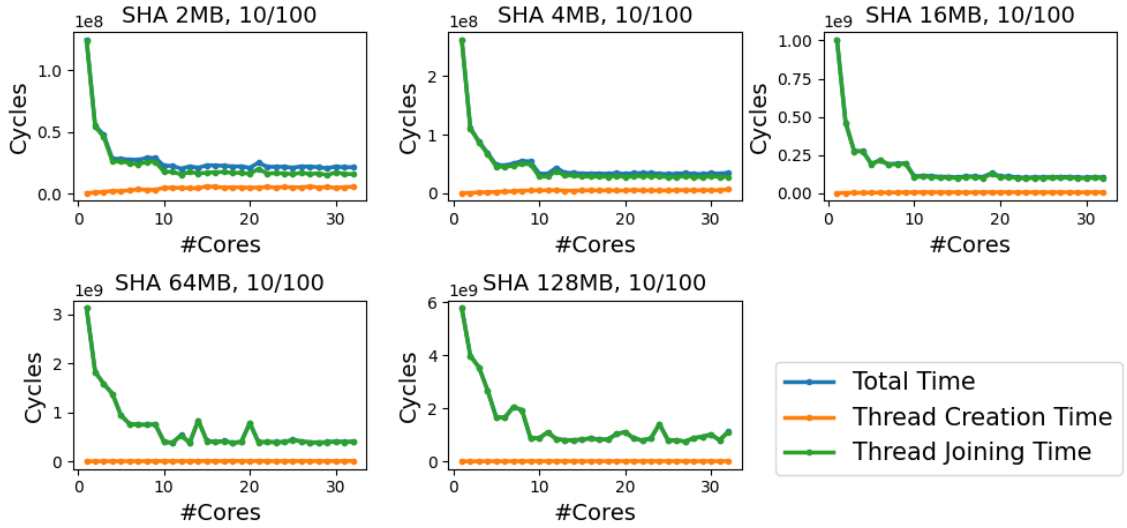

Figure 5.16. *SHA256 results for checking 10 out of 100 pieces of memory, for different memory sizes.*

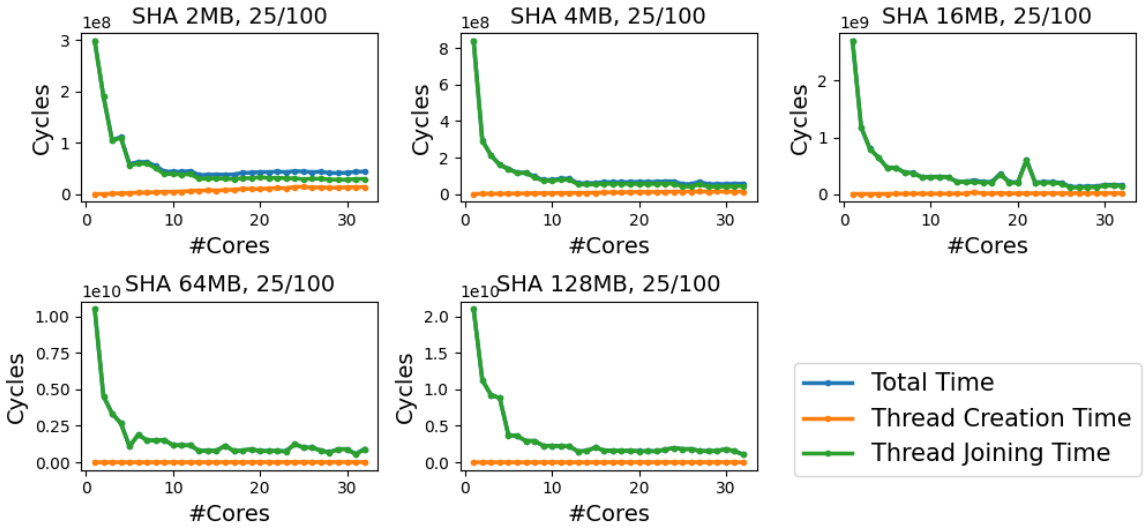

Figure 5.17. *SHA256 results for checking 25 out of 100 pieces of memory, for different*

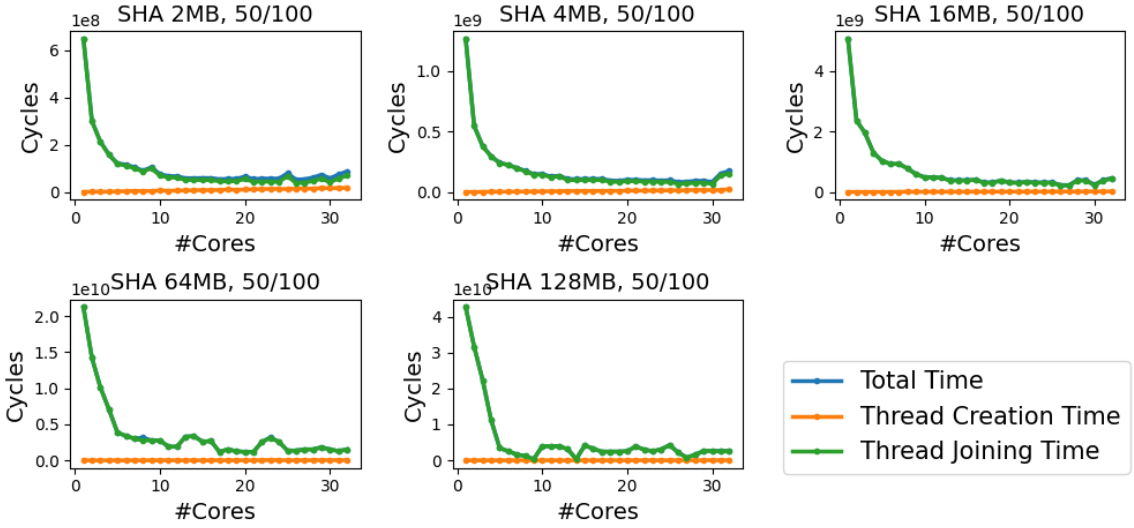

Figure 5.18. *SHA256 results for checking 50 out of 100 pieces of memory, for different memory sizes.*

With Chaskey, the same observations from before apply. We tend to keep getting performance out of the scheme as long as we are using up to the same number of cores as the number of pieces we check.

With Chaskey the results were a bit more consistent than other algorithms. Also, the graphs show very clearly the patterns the scheme follows, such as how the scaling is beneficial up to the point where amount of checked pieces is equal to cores, and the general shape of the lines is also in accordance to theory. It also shows that the performance gains are still relevant when checking larger groups of memory pieces, e.g. up to 50.

Conclusions

With these experiments, we find that in multi-core probabilistic schemes Chaskey and then Poly1305 outperform the more standard SHA256 algorithm in both single-core and multi-core implementations. We find that the benefits of using more than one core apply for as long as we keep our model checking at most a number of pieces equal to or less than the number of cores used. We also find that the overhead of thread creation is only relevant and should be taken into consideration only when dealing with smaller memory sizes, e.g a few megabytes.

We show that all three algorithms scale well as expected in each experiment, but only up to around 15 to 20 cores, and we explain this below. Finally, we also find that at some points we get super-linear speedups from parallelization, and we attribute this mainly to cache locality issues and hashing algorithm design.

With all that in mind, we conclude that Poly1305 and Chaskey are better candidates for a multi-core probabilistic hashing scheme than SHA256.

There are some other observations that apply throughout all algorithm implementations we used in our experiment.

During the probabilistic scheme tests we see there is a close relation between the number of cores used and the number of pieces checked. This becomes apparent if we look at how the experiment works. There are two possible scenarios when running a single measurement - either we have more cores than pieces to check or we have more pieces to check than cores. If the number of cores is bigger than the number of pieces to check, then we split the pieces only among the needed cores while the rest of them are not working. On the other hand, when the pieces outnumber the cores, we cannot split the pieces themselves in half, because that hashing would result in a different hash than the stored one. We therefore must hash each piece in full. That means that some cores will have more than one piece to hash, and that computation must happen sequentially. With this in mind, we now see why the time gains generally stop after we start using more cores than the pieces we need to verify.

Another observation is that all implementations stop showing serious advantages after the usage of roughly 20 cores. For the experiments that only check less than 20 pieces of memory this makes sense, as checking more than 20 would mean that at least one core would have to do more work than all the others, therefore adding a lot of time in the computation. But even for the workloads where there should theoretically be improvement (e.g. checking 25 or 50 pieces of memory), the results show otherwise. This is a behaviour that has troubled us, and we think it can be explained by two main reasons. One, the memory overhead from using more than 20 cores on our FPGA is probably adding a lot of time when adding more cores to the implementation. Two, the hashing algorithms themselves are not optimized around parallelization inherently, and their algorithmic inworks are probably not keeping up with more than 20 cores.

As for the thread creation time (the orange line in the graphs), we can see that it follows the expected route. It is also noticeable that for smaller sizes and for fewer pieces checked the total time is split in a more balanced way between thread creation and actual hashing times. Similarly, for bigger tests the thread joining time tends to be almost identical to

the Total Time.

Another thing is the super-linear speedup that occur in some experiments, especially from the single-core to the dual-core implementation. This we suggest happens due to cache related issues, cache locality and the growth of cache space the more cores we use. When using more than one cores, more of the data in cache may be used simultaneously, therefore not needing to load data from main memory, which takes a lot of time - and cycles. We attribute part of this super-linear speedups to this.

With these experiments we show that probabilistic schemes have a lot time to gain by utilizing more than one cores. Especially with schemes with a similar approach as ours, where memory pieces are predetermined and split to different cores at runtime, parallelization can have an important impact. We show that these benefits apply regardless of the percentage of memory checked and for a variety of memory sizes.

5.2 Automotive Platform

5.2.1 Board Information

The core of this thesis' idea revolves around the fact that the board that we used includes 2 important components:

- A Hardware Security Module and
- 6 Processing Units

Due to confidentiality issues, we cannot disclose a lot more information on the board that was used. These features are widely available in many microcontrollers used in automotive, with differences in count, power, endurance and quality of components. Our work can be used with any microcontroller that features an more than one core and a trusted execution environment.

5.2.2 Comparisson Points

On this platform our research focused on accelerating the deterministic secure booting process. The specific project this work was part of had very strict security requirements, so probabilistic and run-then-check schemes were not considered.

We take into account four main algorithms and conduct our experiments with them:

- SHA256
- AES128-CMAC
- Poly1305
- Chaskey

For SHA256, AES128-CMAC and Poly1305 we have tested two different implementations. The differences in performance and resource management in each case were minuscule, we therefore decide to showcase results for one implementation each in our evaluation.

5.2.3 Hashing Only Results

We measure the scalability of the algorithms up to six cores. We present the speedup of each hashing algorithm, SHA256, Poly1305, Chaskey and AES128-CMAC, over their single core implementations. In Tables 5.1, 5.2, 5.3 and 5.4 we see the according speedup results. In Figure 5.19, we show all the speedups together.

1-core	2-cores	4-cores	6-cores
1	1.9451	3.8822	5.8323

Table 5.1. *SHA256 Hashing Time Speedup with up to 6 cores over single-core implementation.*

1-core	2-cores	4-cores	6-cores
1	1.9711	3.9412	5.7435

Table 5.2. *Poly1305 hashing time speedup with up to 6 cores over single-core implementation.*

1-core	2-cores	4-cores	6-cores
1	1.9703	3.8530	5.6791

Table 5.3. *Chaskey hashing time speedup with up to 6 cores over single-core implementation.*

1-core	2-cores	4-cores	6-cores
1	1.7307	3.072	3.12

Table 5.4. *AES128-CMAC hashing time speedup with up to 6 cores over single-core implementation.*

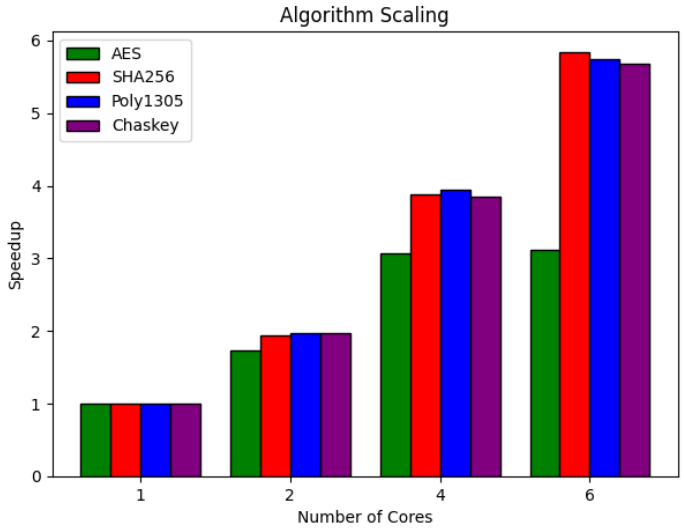

Figure 5.19. *Scaling of different algorithms relative to their single core implementations over hashing time.*

Comparison of Implementations

All the algorithms scale effectively up to 6 cores as we saw before. Therefore the most efficient implementation of each algorithm is the one using all six cores. We now compare the 6-core implementation of the algorithms. We also include in our comparison the performance of the baseline case. For our purposes, this would be a single-core AES128-CMAC hardware accelerated measurement taken with the hardware acceleration provided for the in-TPM core. The speedups are shown in figure 5.20.

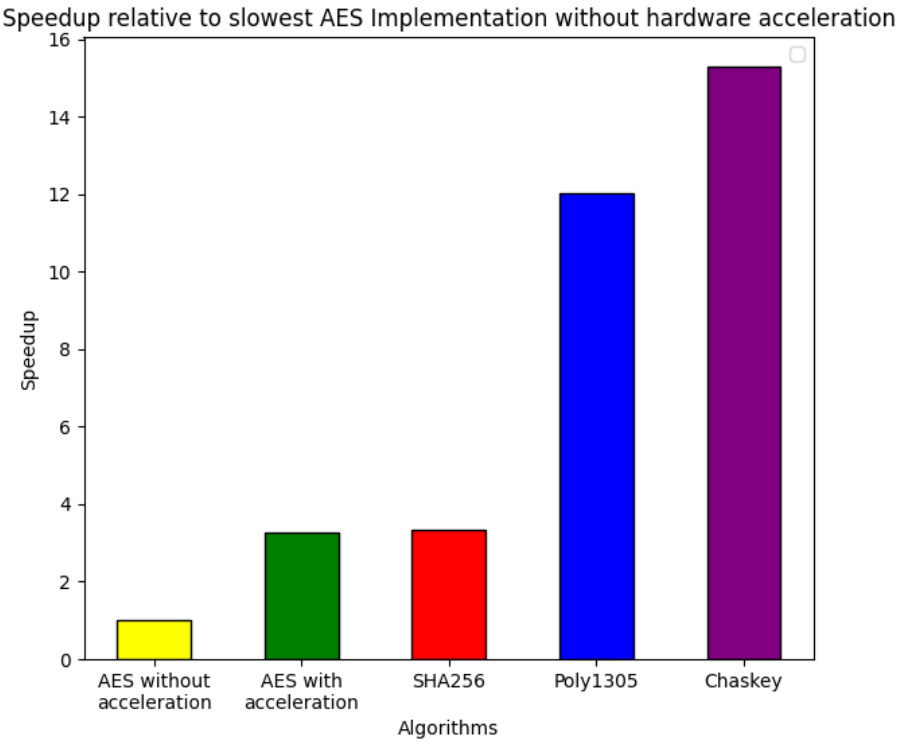

Figure 5.20. *Hashing-only speedups compared to the slowest implementation (AES without hardware acceleration).*

Conclusions

In Figure 5.19, we showed that most algorithms scale efficiently up to 6 cores. That fact on its own means that there is performance to be gained, even in the cases where for any reason the actual hashing algorithm of the application may not change to a faster one, yet unutilized hardware cores exist on the platform.

In Figure 5.20, we show the speedup of all implementations over the slowest one, which is the 6-core AES128-CMAC. AES128-CMAC remains very slow without hardware acceleration. The pre existing scheme used the on-chip accelerator with a single core and used AES. This implementation is the green bar on Figure 5.20. We show in Figure 5.20 that using 6-cores with SHA256, Poly1305 and Chaskey, and without using any acceleration whatsoever, we gain speedups of **3.7x** and **4.7x** with Poly1305 and Chaskey accordingly over the pre existing implementation.

5.2.4 End to End Results

In this experiment we measure the whole process from the power-on of the platform up to a specific moment in time when the platform is considered ready to be deployed. We run tests where the hashing step of this process uses from 1 up to 32 cores, and measure the whole time this takes. In Figure 5.21, we showcase our results.

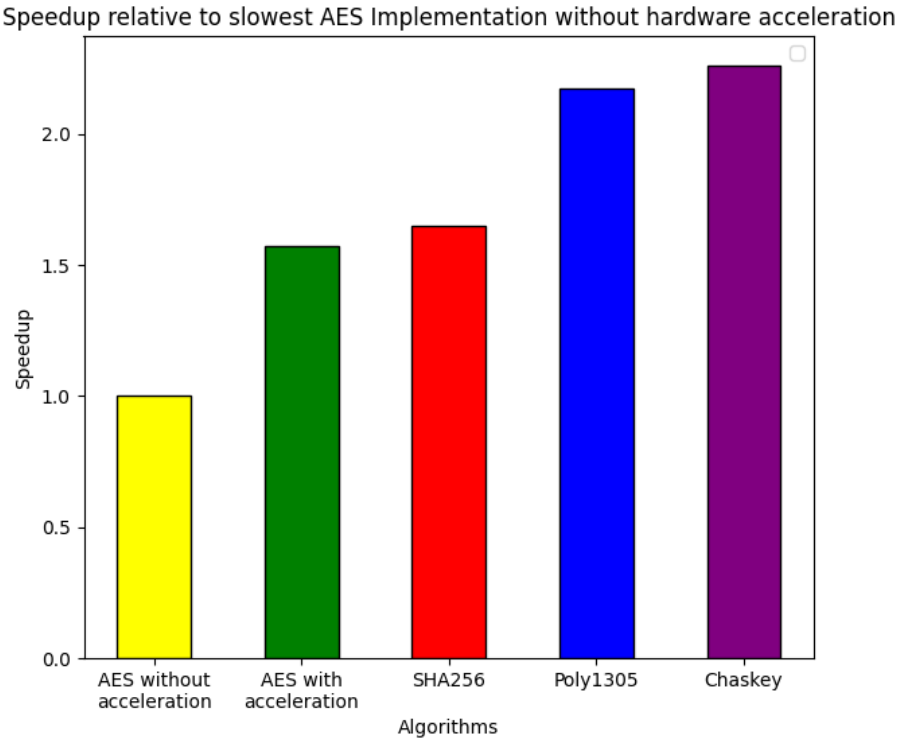

Figure 5.21. *The end-to-end speedup over the slowest of all implementations, AES128-CMAC with 6-cores (yellow bar). The green bar represents the existing implementation using hardware accelerated single-core AES128-CMAC.*

How we Measure End-To-End

This solution was created keeping in mind that it has to improve the speed performance of the booting of an embedded device. The end-to-end performance of said device must of course be kept confidential. However, we can describe the amount of performance gained relatively to the original booting time.

The end-to-end start up of the device comprises of the familiar stages that every embedded device goes through when booting: - Power On Sequence, may include: - Power ON - Reset processor until good power supply is available - Hardware Specific Initialization, in our case HSM related initialization This process takes a certain amount of time, which tends to be quite standard throughout booting sequences. The specific time of this startup should remain sequential, but is also not of much worth in our scenario.

After the initialization, the secure boot takes place. The HSM - which is now ready to run - starts by verifying the First Bootloader (FBL), in the classic secure boot way. Inside

the HSM protected memory lies code responsible for hashing the FBL and comparing the result with the stored hash. If the FBL verification is successful, we can be certain that the loaded bootloader is in a good known state.

With the FBL in a non-tampered state, we can begin our proposed part of the process. The code responsible for checking the image in multiple cores lies inside the bootloader, so that means that we can be certain that the code has not been modified. Apart from that, we can also be certain that no other code has run on the system so far outside the HSM verified bootloader. This guarantees the security of both the process so far and the code that is going to run the parallel hashing scheme. For more on the security of the scheme see the coresponding section "Scheme Security".

Finally, after the verification of the application image has completed, there exists a time period between the end of the verification and the system being able to send the first CAN message. This time is also sufficiently invariable between boots and can be considered fixed. This marks the end of the time slot we consider end-to-end, meaning from power on until the system is ready to send/receive a CAN message.

In Fig. 5.22 we show the basic stages any embedded device that implements secure boot roughly goes through. There are two fairly standard time zones at the beginning and at the end of the process, and there is the target zone in the middle which our implementation tries to compress. In Fig. 5.23 we show our proposed process and where the time is actually saved from.

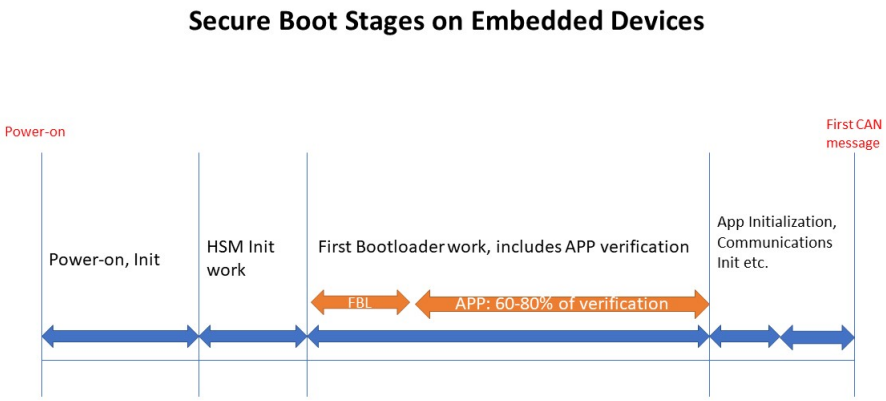

Figure 5.22. *Secure Boot stages of an embedded device.*

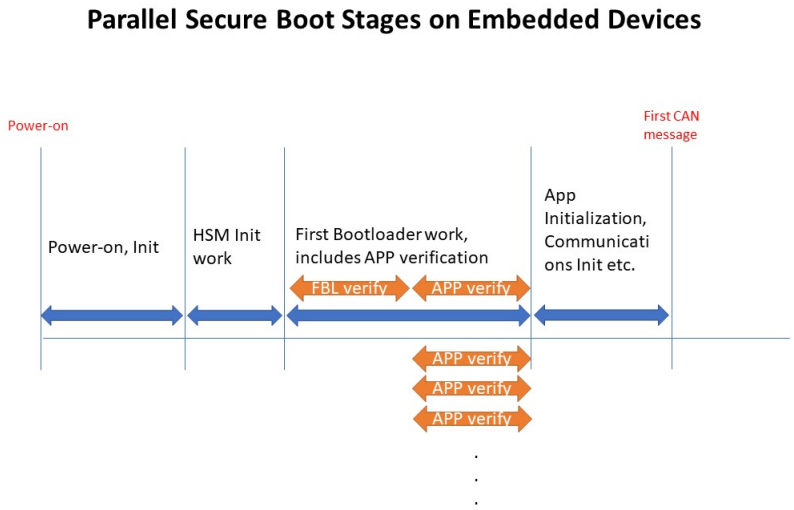

Figure 5.23. *Secure Boot stages of an embedded device incorporating parallel hashing of the application image.*

Conclusions

In figure 5.21 we show that we manage to extract a **1.45x** speedup over the existing implementation by using Chaskey, and a **1.4x** speedup by using Poly1305. We show that by parallelizing only the hashing step during the secure boot procedure we can gain significant performance benefits, utilizing cores that would otherwise be idling and not compromising any security guarantees.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

With this thesis, the main goal was to find ways to accelerate the secure booting process. Throughout our work, it has become apparent that there is performance to be gained in terms of speed through two main avenues - choosing a more lightweight and therefore faster cryptographic algorithm and splliting the hashing step among multiple cores. One key observation that made those clear was the fact that most of the time during the secure boot process was spent on the hashing step. That means that if we managed to reduce the time taken for the process to actually hash the memory it needed we would also manage significant time reductions over the whole.

Firstly, we researched into hashing algorithms. We picked two that were widely used already in the industry and academia and had real applications, SHA256 and AES128-CMAC. Then we looked through the more cutting-edge algorithms. After some initial research we concluded to continue testing two of them, Poly1305 and Chaskey. The two later offer a great deal of speed in our multicore implementations.

Secondly, we split the hashing of the image into multiple cores. Modern embedded microcontrollers offer a multitude of cores, especially in demanding fields such as the Automotive one. So far however, during the secure boot process, most of these cores stay idle not taking part in the computations that happen. We manage to change that, using additional cores by giving them a part of memory to hash. We then propose two ways to verify the produced hashes, either by producing a final digest or by sending them all to the trusted environment.

We demonstrate that our deterministic parallel scheme provides 3.7x performance improvement in the verification step, and 1.4x performance improvement in the end-to-end secure boot process over the single core-baseline using 6 cores on the microcontroller designed for automotive scenarios. On the rocket RISC-V chip, our parallel deterministic and probabilistic verification schemes improve performance by 2.2x using 8 cores and by 1.8x cores using 14 cores over the single-core baseline scheme.

6.2 Future Work

With this thesis a lot of future work possibilities open up. Firstly, one avenue worth exploring would be the evaluation of more hashing algorithms, as newer, faster algorithms come out and hardware acceleration for older ones becomes more widespread. This research may become even more critical with the future introduction of quantum computing that may demand the usage of quantum resistant hashing algorithms. Secondly, the memory slicing and random verification process of the probabilistic secure booting scheme described may be further enhanced, with more complex ways of picking and verifying memory areas. Other than that, with multicore devices some research may go into on-demand verification of software as it is needed, with priorities given to more critical components and background checks for others. Lastly, since secure boot performance is highly hardware and application dependent, it would be beneficial to test the schemes proposed in this work on other real hardware, specifically on hardware geared towards the automotive, automation and industrial domains, to gain some insight not only on the performance of the scheme, but also on the demands of each separate application.

Bibliography

[1] Steffen Sanwald, Liron Kaneti, Marc Stöttinger και Martin Böhner. *Secure Boot Revisited: Challenges for Secure Implementations in the Automotive Domain*. *SAE International Journal of Transportation Cybersecurity and Privacy*, 2(2):11–02–02–0008, 2020.

[2] Robert N. Charette. *This car runs on code*, 2021.

[3] Andy Greenberg. *Hackers remotely kill a Jeep on the highway-with me in it*, 2015.

[4] William Stallings. *Cryptography and Network Security Principles and Practice*. Pearson, 2017.

[5] Tom Warren. *Why Windows 11 is forcing everyone to use TPM chips*, 2021.

[6] William A Arbaugh, David J Farber και Jonathan M Smith. *A secure and reliable bootstrap architecture*. *Proceedings. 1997 IEEE Symposium on Security and Privacy (Cat. No. 97CB36097)*, σελίδες 65–71. IEEE, 1997.

[7] Daehyun Kim, Eunho Shin, Jin Seo Park, KyungSu Lee, Kok Cheng Gui και Klaus Scheibert. *Secure Boot Implementation for Hard Real-Time Powertrain System*. SAE Technical Paper 2017-01-1656, SAE International, Warrendale, PA, 2017. ISSN: 0148-7191, 2688-3627.

[8] Martín Abadi, Mihai Budiu, Úlfar Erlingsson και Jay Ligatti. *Control-Flow Integrity Principles, Implementations, and Applications*. *ACM Trans. Inf. Syst. Secur.*, 13(1), 2009.

[9] Miguel Castro, Manuel Costa και Tim Harris. *Securing software by enforcing data-flow integrity*. *Proceedings of the 7th symposium on Operating systems design and implementation*, σελίδες 147–160, 2006.

[10] Robert Kaster, Di Ma, Ashish Behl και Bartosz Bakalarczyk. *Sliced secure boot*. 2021.

[11] Ahmad M. K. Nasser, Wonder Gumise και Di Ma. *Accelerated Secure Boot for Real-Time Embedded Safety Systems*. *SAE International Journal of Transportation Cybersecurity and Privacy*, 2(1):35–48, 2019. Number: 11-02-01-0003.

[12] NIST. *Hash Functions*, 2021.

[13] Ilia Levin. *SHA256 ilvn*, 2022.

[14] D. J. Bernstein. *A state-of-the-art message-authentication code*.

[15] Andrew Moon. *poly1305-donna*, 2016.

[16] Nicky Mouha. *Chaskey*.

[17] *Mbed-TLS*.

[18] Microsoft. *Windows 11 and Secure Boot*.

[19] Ubuntu Wiki. *UEFI SecureBoot*.

[20] Intel. *Intel on Secure Boot*.

[21] Nisha Jacob, Johann Heyszl, Andreas Zankl, Carsten Rolfes και Georg Sigl. *How to break secure boot on fpga socs through malicious hardware*. *International Conference on Cryptographic Hardware and Embedded Systems*, σελίδες 425–442. Springer, 2017.

[22] Agam Shah. *Europe to Dish out €270 Million to Build RISC-V Hardware and Software*, 2022.

[23] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović και Dawn Song. *Keystone: An open framework for architecting trusted execution environments*. *Proceedings of the Fifteenth European Conference on Computer Systems*, σελίδες 1–16, 2020.

[24] *Xilinx U250*.

List of Abbreviations

SB	Secure Boot
TPM	Trusted Platform Module
HSM	Hardware Security Module
ECU	Electronic Control Unit
BPF	Band Pass Filter
OS	Operating Systems
FSB	First Stage Bootloader
SSB	Second Stage Bootloader