Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Επιστήμης Υπολογιστών

Aègis: Ασφαλής Επιλεκτική Εκφόρτωση Υπολογισμού από Ενσωματωμένες Συσκευές

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Μαρία-Αργυρώ Λάζου

Επιβλέπων: Γεώργιος Γκούμας
Καθηγητής ΕΜΠ

Αθήνα, Ιούλιος 2026

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών &
Μηχανικών Υπολογιστών
Τομέας Επιστήμης Υπολογιστών

Aègis: Ασφαλής Επιλεκτική Εκφόρτωση Υπολογισμού από Ενσωματωμένες Συσκευές

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Μαρία-Αργυρώ Λάζου

Επιβλέπων: Γεώργιος Γκούμας
Καθηγητής, ΕΜΠ

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την Δευτέρα 6 Ιουλίου 2026.

.....	.....	.....
Γεώργιος Γκούμας Καθηγητής, ΕΜΠ	Νίκος Βασιλάκης Αναπληρωτής Καθηγητής, Brown University	Διονύσιος Πνευματικάτος Καθηγητής, ΕΜΠ

Αθήνα, Ιούλιος 2026

.....
Μαρία-Αργυρώ Λάζου
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

1 Περίληψη

Οι ενσωματωμένες εφαρμογές έχουν καταστεί ολοένα και πιο διαδεδομένες, υποστηρίζοντας ένα ευρύ φάσμα ευφυών συστημάτων όπως συσκευές IoT, αυτόνομες πλατφόρμες και μικροελεγκτές. Οι εφαρμογές αυτές συχνά επεξεργάζονται δεδομένα ευαίσθητα ως προς την ιδιωτικότητα, τα οποία συλλέγονται από αισθητήρες ή εισάγονται από τον χρήστη, στο πλαίσιο σύνθετων ροών επεξεργασίας που υποστηρίζουν έλεγχο σε πραγματικό χρόνο, αναλυτική επεξεργασία και λήψη αποφάσεων. Ωστόσο, λόγω των περιορισμένων υπολογιστικών τους πόρων, η εκτέλεση εντατικών, υπολογιστικά απαιτητικών εργασιών υποβαθμίζει σημαντικά την απόδοσή τους. Η απευθείας μεταφορά τμημάτων του υπολογισμού σε παρόχους υποδομών cloud ενέχει σημαντικούς κινδύνους ασφάλειας, καθώς ο ίδιος ο απομακρυσμένος εξυπηρετητής ενδέχεται να έχει παραβιαστεί. Τα Trusted Execution Environments (TEEs) παρέχουν προστασία έναντι κακόβουλου λογισμικού με αυξημένα δικαιώματα, συμπεριλαμβανομένου του λειτουργικού συστήματος ή του υπερδιαχειριστή, καθώς και έναντι μη εξουσιοδοτημένης επιθεώρησης ή αλλοίωσης της μνήμης, προσφέροντας ισχυρές εγγυήσεις ασφάλειας για εφαρμογές που εκτείνονται τόσο σε τοπικά όσο και σε απομακρυσμένα περιβάλλοντα εκτέλεσης.

Το *Aégis* υιοθετεί ένα υβριδικό μοντέλο χρονοπρογραμματισμού, το οποίο κλιμακώνει δυναμικά τμήματα της εφαρμογής προς TEEs κατά τον χρόνο εκτέλεσης, αυξάνοντας τη ρυθμοαπόδοση ενώ ταυτόχρονα διατηρεί την εμπιστευτικότητα των δεδομένων. Το σύστημα επιλεκτικά μεταφέρει και κατανέμει συναρτήσεις κρίσιμες για την απόδοση σε διαθέσιμους κόμβους enclave, ακολουθώντας διαμερισμό προγράμματος καθοδηγούμενο από τον μεταγλωττιστή και αξιοποιώντας, όπου είναι εφικτό, παραλληλισμό σε επίπεδο εργασιών. Για την υποστήριξη αυτού του μοντέλου εκτέλεσης, το *Aégis* ενσωματώνει ένα ελαφρύ περιβάλλον εκτέλεσης εντός TEE, το οποίο ενορχηστρώνει την απομακρυσμένη εκτέλεση λειτουργώντας υπό τους αυστηρούς περιορισμούς μνήμης των περιβαλλόντων enclave.

Σε ένα ευρύ σύνολο ενσωματωμένων (embedded) workloads, το *Aégis* επιτυγχάνει διάμεση επιτάχυνση (median speedup) ίση με $4.7 \times 10^3 \times$ στην περίπτωση ενός μεμονωμένου server, καθώς και επιπλέον βελτίωση της τάξης του ενός μεγέθους (order-of-magnitude) υπό κατανεμημένη εκτέλεση σε πολλαπλούς enclave nodes.

Λέξεις-Κλειδιά

Trusted Execution Environments (TEEs), Ασφαλές Offloading Υπολογισμού, Ενσωματωμένα Συστήματα, Κατανεμημένη Εκτέλεση σε Enclaves, Μετασχηματισμός Προγραμμάτων, Παραλληλισμός Εργασιών, Συστήματα Εκτέλεσης

2 Abstract

Embedded applications have become increasingly popular, powering a wide range of intelligent systems such as IoT devices, autonomus platforms and microcontrollers. These applications often process privacy-sensitive data collected from sensors or user input, as part of complex workflows that support real-time control, analytics and decision-making. However, due to their restricted resources, executing intensive, compute-bound tasks degregates their performance. Outsourcing parts of computation directly to cloud vendors, poses significant security risks, since the server itself may be compromised. Trusted Execution Environments (TEEs) protect against malicious priviledges software including the operating system or hypervisor, as well as unauthorized memory inspection and tampering, offering security guarantees for applications that span on- and off-premise components. *Aégis* adopts a hybrid scheduling model that dynamically scales application components to TEEs at runtime, increasing throughput while preserving data confidentiality. It selectively offloads and distributes performance-critical functions across available enclave nodes following compiler-driven program partitioning, exploiting task-level parallelism whenever possible. To support this execution model, *Aégis* incorporates a lightweight TEE-embedded runtime that orchestrates remote execution while operating within the strict memory constraints of enclave environments. Across a diverse set of embedded workloads, *Aégis* achieves a median speedup of $4.7 \times 10^3\times$ in the single-server setting, and up to an order-of-magnitude additional improvement under distributed execution across multiple enclave nodes.

Keywords

Trusted Execution Environments (TEEs), Secure Computation Offloading, Embedded Systems, Distributed Enclave Execution, Program Transformation, Task Parallelism, Runtime Systems

3 Ευχαριστίες

Στο σημείο αυτό, θα ήθελα να εκφράσω τις ειλικρινείς μου ευχαριστίες προς τον Καθηγητή Γεώργιο Γκούμα, ο οποίος αποτέλεσε σημαντική πηγή έμπνευσης καθ' όλη τη διάρκεια των προπτυχιακών μου σπουδών. Ήταν εκείνος που με εισήγαγε στον χώρο των υπολογιστικών συστημάτων και με ενθάρρυνε να ακολουθήσω μεταπτυχιακές σπουδές και να εμβαθύνω περαιτέρω στο συγκεκριμένο επιστημονικό πεδίο.

Θα ήθελα επίσης να ευχαριστήσω θερμά τον Καθηγητή Νίκο Βασιλάκη, ο οποίος μου έδωσε τη μοναδική ευκαιρία να πραγματοποιήσω μέρος της ερευνητικής μου δραστηριότητας για τη διπλωματική εργασία στο Brown University, καθώς και να συμμετάσχω σε επιπλέον ερευνητικές εργασίες που διαμόρφωσαν ουσιαστικά την αντίληψή μου για την έρευνα σε μεταπτυχιακό επίπεδο και συνέβαλαν καθοριστικά στη διαμόρφωση της ακαδημαϊκής μου πορείας.

Τέλος, θα ήθελα να ευχαριστήσω τους φίλους μου και τους συναδέλφους μου στο εργαστήριο για όλες τις στιγμές που μοιραστήκαμε, οι οποίες έκαναν τα χρόνια μου στο ΕΜΠ πιο ευχάριστα και λιγότερο αγχωτικά.

Πάνω απ' όλα, θα ήθελα να ευχαριστήσω τους γονείς μου, Αφροδίτη και Ασημάκη, οι οποίοι με στήριξαν αδιάκοπα και χωρίς όρους σε κάθε στάδιο της ζωής μου. Από πολύ νωρίς μου ενστάλαξαν την αξία της σκληρής δουλειάς, της επιμονής και της διαρκούς προσπάθειας για την επίτευξη ουσιαστικών στόχων. Οι θυσίες, η καθοδήγηση και η εμπιστοσύνη που έδειξαν προς το πρόσωπό μου υπήρξαν καθοριστικές στη διαμόρφωση αυτού που είμαι σήμερα, και σε αυτούς οφείλω τη βαθύτερη ευγνωμοσύνη μου.

Περιεχόμενα

4 Εισαγωγή

Η υπολογιστική αιχμής (edge computing) γίνεται ολοένα και πιο διαδεδομένη λόγω της ραγδαίας εξάπλωσης των συσκευών του Διαδικτύου των Πραγμάτων (IoT) και της ανάγκης για προβλέψεις σε πραγματικό χρόνο και έξυπνο αυτοματισμό. Προς αυτή την κατεύθυνση, η τεχνητή νοημοσύνη διαδραματίζει καθοριστικό ρόλο, επιτρέποντας την ακριβή εξαγωγή συμπερασμάτων μέσω εκπαιδευμένων μοντέλων που εκτελούνται απευθείας στις συσκευές αιχμής. Συσκευές όπως φορετές συσκευές υγείας ή φυσικής κατάστασης, γυροσκόπια, κάμερες ασφαλείας και συσκευές καταγραφής ήχου βασίζονται σε αλυσίδες επεξεργασίας σημάτων, οι οποίες μετατρέπουν θορυβώδη και μη δομημένα δεδομένα αισθητήρων σε χρήσιμα χαρακτηριστικά, τα οποία στη συνέχεια τροφοδοτούν αλγορίθμους μηχανικής μάθησης για τη λήψη αποφάσεων. Οι συσκευές αυτές συλλέγουν και αναλύουν συνεχώς φυσιολογικά και περιβαλλοντικά σήματα, όπως καρδιακό ρυθμό, επίπεδα ινσουλίνης, μοτίβα αναπνοής, τροχιές κίνησης, καρέ εικόνας και ακουστικά σήματα, προκειμένου να τροφοδοτήσουν τις βασικές υπολογιστικές τους διεργασίες. Σε αυτό το πλαίσιο, τα δεδομένα που επεξεργάζονται είναι ιδιαίτερα ευαίσθητα και θα πρέπει να παραμένουν προσβάσιμα αποκλειστικά στον χρήστη. Ταυτόχρονα, η υψηλή υπολογιστική απαίτηση ορισμένων σταδίων της επεξεργασίας καθιστά ολόκληρη την αλυσίδα αναποτελεσματική για συσκευές με τόσο περιορισμένους πόρους. Το προκύπτον σημείο συμφόρησης οδηγεί σε αυξημένους χρόνους απόκρισης, υπερβολική κατανάλωση ενέργειας ή ακόμη και σε μείωση της συχνότητας δειγματοληψίας του υπό παρακολούθηση συστήματος, ώστε να περιοριστεί το μέγεθος της εισόδου, γεγονός που υποβαθμίζει την ακρίβεια των προβλέψεων [1].

Παρόλο που δημόσιοι πάροχοι υπολογιστικού νέφους, όπως οι AWS, Azure, IBM και Google Cloud, προσφέρουν σημαντικούς υπολογιστικούς πόρους, η ανάθεση της εκτέλεσης σε αυτούς μεταφέρει ευαίσθητους υπολογισμούς σε ένα μη έμπιστο περιβάλλον, δημιουργώντας έναν θεμελιώδη συμβιβασμό μεταξύ απόδοσης και εμπιστευτικότητας των δεδομένων [2]. Επιπλέον, ορισμένες λειτουργίες—όπως η συνεχής συλλογή δεδομένων από αισθητήρες, τα οποία παράγονται συνήθως ως ροή δεδομένων, καθώς και η εφαρμογή ελαφριών σταδίων φιλτραρίσματος—πρέπει εκ φύσεως να εκτελούνται στη συσκευή αιχμής.

Κατά συνέπεια, απαιτείται μια υβριδική προσέγγιση που να καθορίζει τη βέλτιστη θέση εκτέλεσης κάθε εργασίας της υπολογιστικής αλυσίδας, σε συνδυασμό με ένα απομονωμένο περιβάλλον εκτέλεσης (sandbox) που διασφαλίζει την ασφαλή και απομονωμένη εκτέλεση ευαίσθητων υπολογισμών. Το *Aégis* υιοθετεί αυτή την προσέγγιση, μεταφέροντας επιλεκτικά τις συναρτήσεις που αποτελούν σημεία συμφόρησης από ενσωματωμένες συσκευές σε ασφαλείς θύλακες εκτέλεσης (secure enclaves). Ο σχεδιασμός του βασίζεται στην παρατήρηση ότι οι περισσότερες εφαρμογές ενσωματωμένων συστημάτων ακολουθούν ένα επαναλαμβανόμενο πρότυπο εκτέλεσης βασισμένο σε βρόχους, το οποίο εμφανίζεται σε διαφορετικούς τύπους εφαρμογών με μικρές μόνο διαφοροποιήσεις στη ροή ελέγχου, στους χρονικούς περιορισμούς και στη λεπτομέρεια της ροής δεδομένων.

Το *Aégis* αποτελείται από έναν μεταγλωττιστή DSL, ο οποίος εκφράζει τις κατάλληλες σημασιολογικές έννοιες της επιλεκτικής εκφόρτωσης και πραγματοποιεί μετασχηματισμούς του προγράμματος ώστε να διαχωρίσει την εφαρμογή σε εγγενείς και απομακρυσμένα εκτελούμενες εργασίες. Παρέχει επίσης βασικούς μηχανισμούς χρόνου εκτέλεσης για την ασφαλή απομακρυσμένη εκτέλεση, συμπεριλαμβανομένου ενός προσαρμοσμένου διερμηνευτή QJS, ενισχυμένου με μηχανισμούς κρυπτογράφησης και επικοινωνίας που επιτρέπουν τη μεταφορά της κατάστασης εκτέλεσης μέσω ασφαλούς καναλιού μετά την ολοκλήρωση της διαδικασίας χειραψίας (handshake). Επιπλέον, το *Aégis* περιλαμβάνει μια μηχανή ανάλυσης που εντοπίζει τα εξαρτώμενα αρχεία και τις βιβλιοθήκες που πρέπει να μεταδοθούν στο περιβάλλον του θύλακα, καθώς και ένα περιβάλλον εκτέλεσης στην πλευρά του πελάτη, το οποίο διαχειρίζεται την υποβολή αιτημάτων και την ανάκτηση αποτελεσμάτων μέσω ενός μοντέλου εκτέλεσης βασισμένου

σε ουρές εργασιών, επιτρέποντας ταυτόχρονη εκφόρτωση όποτε αυτό είναι εφικτό.

Στην πλευρά του εξυπηρετητή, ένα προσαρμοσμένο περιβάλλον εκτέλεσης QJS ενσωματώνεται στον ασφαλή θύλακα για την αξιολόγηση των εισερχόμενων αιτημάτων, συνοδευόμενο από ένα ενδιάμεσο επίπεδο λογισμικού (middleware) π ου διαμεσολαβεί την επικοινωνία μεταξύ του θύλακα και του μη έμπιστου εξυπηρετητή. Τέλος, ένας χρονοπρογραμματιστής παρακολουθεί τις ενεργές συνεδρίες πελάτη–θύλακα και αναθέτει διαθέσιμους κόμβους θύλακα με πολιτική round-robin κατόπιν αιτήματος.

Συνοπτικά, η εργασία αυτή συνεισφέρει τα εξής:

- Μια δομική μελέτη και ταξινόμηση εφαρμογών ενσωματωμένων συστημάτων που βασίζονται σε βρόχους, η οποία εντοπίζει πρότυπα εκτέλεσης κατάλληλα για ασφαλή εκφόρτωση υπολογισμού.
- Ένα ελαφρύ περιβάλλον εκτέλεσης μέσα σε ασφαλείς θύλακες, τ ο οποίο επιτρέπει διαφανή απομακρυσμένη εκτέλεση, διατηρώντας παράλληλα την εμπιστευτικότητα των δεδομένων και λειτουργώντας εντός αυστηρών περιορισμών πόρων.
- Ένα πλαίσιο επιλεκτικής εκφόρτωσης υπολογισμού που κατανέμει δυναμικά τις συναρτήσεις που αποτελούν σημεία συμφόρησης σε κόμβους ασφαλών θυλάκων, υποστηρίζοντας μερική απομακρυσμένη εκτέλεση και βελτιώνοντας τη συνολική απόδοση.
- Έναν μεταγλωττιστή γλώσσας ειδικού σκοπού (DSL), ο οποίος μετασχηματίζει αυτόματα επισημασμένες δομές βρόχων σε ασύγχρονες απομακρυσμένα εκτελέσιμες εργασίες και ενορχηστρώνει την κατανεμημένη εκτέλεσή τους σε κόμβους ασφαλών θυλάκων.

Ο πηγαίος κώδικας θα είναι δημόσια διαθέσιμος στο Github μετά την υποβολή της εργασίας.

5 Παράδειγμα

Έστω η ακόλουθη ρουτίνα επεξεργασίας ΗΚΓ (ECG), η οποία διαβάζει δεδομένα καρδιακού σήματος και επιχειρεί να ανιχνεύσει εάν ένας ασθενής παρουσιάζει ενδείξεις αρρυθμίας.

```
1   const arrModel = await tf.loadLayersModel('file://./arr_model_v4/model.json');
2
3   function ECGPipeline() {
4       for (let i = 0; i < ITERATIONS; i++) {
5           const ecgData = readECG();
6           const result = predictArrhythmia(ecgData, arrModel);
7           console.log("Arrhythmia detection:", result);
8       }
9   }
```

Η συνάρτηση `readECG()` (γραμμή 5) συλλέγει παρτίδες μετρήσεων του καρδιακού σήματος από έναν αισθητήρα με σταθερή συχνότητα δειγματοληψίας. Τα συλλεχθέντα δεδομένα μεταβιβάζονται στη συνέχεια στη συνάρτηση `predictArrhythmia()` (γραμμή 6), η οποία εκτελεί ένα ελαφρύ στάδιο προεπεξεργασίας (π.χ. κανονικοποίηση) πριν πραγματοποιήσει εξαγωγή συμπερασμάτων χρησιμοποιώντας ένα προεκπαιδευμένο μοντέλο TensorFlow (γραμμή 1).

Παρότι η συλλογή του σήματος και η προεπεξεργασία είναι σχετικά χαμηλού υπολογιστικού κόστους, το στάδιο της εξαγωγής συμπερασμάτων κυριαρχεί στον συνολικό χρόνο εκτέλεσης της αλυσίδας επεξεργασίας. Σε ενσωματωμένες συσκευές με περιορισμένους πόρους, η επαναλαμβανόμενη εκτέλεση του συγκεκριμένου υπολογισμού μπορεί γρήγορα να εξελιχθεί σε σημείο συμφόρησης, περιορίζοντας τη ρυθμαπόδοση της εφαρμογής παρακολούθησης. Στην πράξη, εάν η ρουτίνα πρόβλεψης καταστεί υπερβολικά αργή, το σύστημα ενδέχεται να αναγκαστεί να μειώσει τη συχνότητα δειγματοληψίας του ΗΚΓ ώστε να ανταποκριθεί στις απαιτήσεις επεξεργασίας. Ωστόσο, η μείωση της συχνότητας δειγματοληψίας μπορεί να οδηγήσει σε απώλεια σημαντικών χαρακτηριστικών του σήματος και να επηρεάσει αρνητικά την ακρίβεια ανίχνευσης αρρυθμιών. Ως αποτέλεσμα, η `predictArrhythmia()` αποτελεί έναν φυσικό υποψήφιο για απομακρυσμένη εκτέλεση.

Για να εκφράσουν αυτή την πρόθεση, οι προγραμματιστές επισημάνουν τον βρόχο με μια οδηγία εκφόρτωσης (γραμμή 4), η οποία προσδιορίζει την υπολογιστικά απαιτητική συνάρτηση (γραμμή 5) και δηλώνει ότι δεν υπάρχουν εξαρτήσεις μεταξύ διαδοχικών επαναλήψεων του βρόχου (γραμμή 6), επιτρέποντας έτσι περαιτέρω παραλληλισμό.

```
1   const arrModel = await tf.loadLayersModel('file://./arr_model_v4/model.json');
2
3   function ECGPipeline() {
4     /* @offload
5        -- fname predictArrhythmia
6        --parallel
7     */
8     for (let i = 0; i < ITERATIONS; i++) {
9         const ecgData = readECG();
10        const result = predictArrhythmia(ecgData, arrModel);
11        console.log("Arrhythmia detection:", result);
12    }
13  }
```

Δεδομένου ότι τα ιατρικά δεδομένα είναι ιδιαίτερα ευαίσθητα, τα στοιχεία των ασθενών θα μπορούσαν να εκτεθούν σε περίπτωση παραβίασης του απομακρυσμένου εξυπηρετητή που αναλαμβάνει

την επεξεργασία τους. Συνεπώς, απαιτείται ένα επιπλέον επίπεδο ασφάλειας στην πλευρά του εξυπηρετητή, ώστε να διασφαλίζεται η εμπιστευτικότητα κατά την απομακρυσμένη εκτέλεση.

Το *Aégis* αντιμετωπίζει αυτή την πρόκληση μεταφέροντας τον υπολογισμό σε έναν αποκλειστικά αφιερωμένο στον πελάτη ασφαλή θύλακα εκτέλεσης, αξιοποιώντας συγκεκριμένα την τεχνολογία Intel SGX.

Το αξιόπιστο αυτό περιβάλλον εκτέλεσης παρέχει ισχυρές εγγυήσεις απομόνωσης, διασφαλίζοντας ότι τα ευαίσθητα δεδομένα ΗΚΓ παραμένουν προστατευμένα ακόμη και όταν το υποκείμενο σύστημα δεν είναι έμπιστο.

Ο μεταγλωττιστής DSL μετασχηματίζει αυτόματα τον επισημασμένο κώδικα σε ένα ασύγχρονο μοντέλο εκτέλεσης που υποστηρίζει ασφαλή απομακρυσμένη κλήση.

```
1  const arrModel = await tf.loadLayersModel('file://./arr_model_v4/model.json');
2
3  async function ECGPipeline() {
4      const ecgData = readECG();
5      // __remote__
6      const result = await predictArrhythmia(ecgData, arrModel);
7      console.log("Arrhythmia detection:", result);
8  }
9
10 scaleout_traffic(ECGPipeline, [], ITERATIONS);
```

Συγκεκριμένα, ο μεταγλωττιστής απομονώνει το σώμα του βρόχου και το ενσωματώνει σε μια ασύγχρονη ρουτίνα (γραμμή 3), ενώ παράλληλα επισημαίνει την κλήση της συνάρτησης εξαγωγής συμπερασμάτων ως απομακρυσμένη λειτουργία (γραμμή 5). Η κλήση της `scaleout_traffic()` (γραμμή 10) ενορχηστρώνει την ταυτόχρονη εκτέλεση πολλαπλών επαναλήψεων της `ECGPipeline()`, κατανέμοντας τις εργασίες στους διαθέσιμους κόμβους ασφαλών θυλάκων. Ο μετασχηματισμός αυτός διαχωρίζει ουσιαστικά τον υπολογισμό σε δύο μέρη: ένα τοπικό τμήμα υπεύθυνο για τη συλλογή δεδομένων και τη ροή ελέγχου, και ένα απομακρυσμένο τμήμα που εκτελεί την εξαγωγή συμπερασμάτων εντός του ασφαλούς θύλακα.

Η ανάπτυξη (unrolling) του βρόχου με αυτόν τον τρόπο εξυπηρετεί δύο βασικούς σκοπούς. Πρώτον, ευθυγραμμίζει το πρόγραμμα με το μοντέλο εκτέλεσης του ασφαλούς θύλακα, όπου μεμονωμένες εργασίες μπορούν να κληθούν και να εκτελεστούν με ασφάλεια απομακρυσμένα. Δεύτερον, επειδή τα δείγματα ΗΚΓ επεξεργάζονται ανεξάρτητα μεταξύ τους, πολλαπλές επαναλήψεις μπορούν να προγραμματιστούν ταυτόχρονα σε διαφορετικούς κόμβους ασφαλών θυλάκων, βελτιώνοντας σημαντικά τη συνολική ρυθμαπόδοση.

6 Κατηγοριοποίηση Βρόχων Εκτέλεσης σε Ενσωματωμένα Συστήματα

Τα ενσωματωμένα συστήματα οργανώνονται δομικά γύρω από έναν επίμονο βρόχο ελέγχου, ο οποίος συνήθως εκφράζεται ως `while(1)` ή με κάποια ισοδύναμη κατασκευή που εκτελεί συνεχώς τις λειτουργίες της δειγματοληψίας, του υπολογισμού και της ενεργοποίησης. Το αρχιτεκτονικό αυτό πρότυπο δεν αποτελεί μια τυχαία επιλογή, αλλά ένα θεμελιώδες χαρακτηριστικό των ενσωματωμένων συστημάτων. Όπως περιγράφεται στο *Programming Embedded Systems* [3], το λογισμικό (firmware) των ενσωματωμένων συστημάτων εκτελεί συνήθως τη βασική του λειτουργικότητα μέσα σε έναν άπειρο βρόχο, ο οποίος μετά την αρχικοποίηση επαναλαμβάνει διαρκώς τα στάδια της δειγματοληψίας, της επεξεργασίας και της ενεργοποίησης. Παρόμοιες αρχές συναντώνται και στη βιβλιογραφία των συστημάτων πραγματικού χρόνου, όπου ένας σταθερής διάρκειας «*κύριος κύκλος*» (*major cycle*)

εξασφαλίζει προβλέψιμο χρονισμό της εκτέλεσης [4]. Ο άπειρος βρόχος αποτελεί επομένως μια δομική αναλλοίωτη ιδιότητα του λογισμικού ενσωματωμένων συστημάτων, καθώς καθορίζει τόσο τη συνεχή λειτουργία όσο και τη χρονική δομή του συστήματος.

Με βάση αυτή την παρατήρηση, προτείνουμε μια ταξινόμηση των δομών βρόχων εκτέλεσης των ενσωματωμένων εφαρμογών, η οποία βασίζεται στις ακόλουθες τρεις ιδιότητες:

- χρονικές εγγυήσεις,
- βαθμός λεπτομέρειας της ροής δεδομένων,
- κρισιμότητα της ροής ελέγχου.

Η ταξινόμηση αυτή παρέχει μια συστηματική βάση για την κατάτμηση του υπολογισμού μεταξύ των συσκευών αιχμής και απομακρυσμένων περιβαλλόντων εκτέλεσης. Ειδικότερα, διαχωρίζει τις εργασίες που απαιτούν αυστηρά περιορισμένους χρόνους απόκρισης από εκείνες που μπορούν να ανεχθούν καθυστέρηση δικτύου και χρονική μη ντετερμινιστικότητα. Ως αποτέλεσμα, καθίσταται δυνατός ο εντοπισμός των υπολογιστικών τμημάτων που μπορούν να επωφεληθούν από την εκφόρτωση υπολογισμού χωρίς να παραβιάζονται οι περιορισμοί ορθότητας του συνολικού συστήματος.

6.1 Βρόχοι Ελέγχου Αυστηρού Πραγματικού Χρόνου

Τα ενσωματωμένα συστήματα αυστηρού πραγματικού χρόνου υλοποιούν στενά συνδεδεμένες αλυσίδες δειγματοληψίας, επεξεργασίας και ενεργοποίησης, όπου κάθε επανάληψη του βρόχου πρέπει να ολοκληρώνεται εντός αυστηρών χρονικών ορίων. Μια τυπική δομή έχει ως εξής:

```
while (1) {
  sensor = read();
  state = process(sensor);
  decision = control(state);
  actuate(decision);
}
```

Τα συστήματα αυτά λειτουργούν υπό αυστηρές ντετερμινιστικές προθεσμίες και χρησιμοποιούνται συνήθως σε περιβάλλοντα κρίσιμα ως προς την ασφάλεια. Η αλυσίδα ελέγχου περιλαμβάνει συνήθως ελάχιστη προσωρινή αποθήκευση δεδομένων και **δεν μπορεί** να ανεχθεί την απρόβλεπτη καθυστέρηση που εισάγει η επικοινωνία μέσω δικτύου ή η απομακρυσμένη εκτέλεση.

Χαρακτηριστικά παραδείγματα αποτελούν τα συστήματα πέδησης αυτοκινήτων, οι ελεγκτές αερόσακων, το λογισμικό ελέγχου κινητήρων στις ηλεκτρονικές μονάδες ελέγχου (ECUs), καθώς και τα συστήματα ελέγχου πτήσης στην αεροναυπηγική. Τα συστήματα αυτά διέπονται από αυστηρά πρότυπα λειτουργικής ασφάλειας και πιστοποίησης, όπως το ISO 26262 για τα αυτοκινητικά συστήματα και το DO-178C για τα αεροπορικά συστήματα, τα οποία επιβάλλουν αυστηρές απαιτήσεις ως προς τη ντετερμινιστική χρονική συμπεριφορά και την ορθότητα [5, 6].

Τέτοιου είδους εφαρμογές **δεν είναι κατάλληλες για απομακρυσμένη εκφόρτωση υπολογισμού**.

6.2 Βρόχοι Ροής Δεδομένων Χαλαρού Πραγματικού Χρόνου

Πολλές σύγχρονες εφαρμογές του Διαδικτύου των Πραγμάτων επεξεργάζονται συνεχώς ροές δεδομένων από αισθητήρες, ανεχόμενες ταυτόχρονα μέτριες αποκλίσεις στον χρόνο εκτέλεσης. Οι εφαρμογές αυτές ακολουθούν συχνά το ακόλουθο πρότυπο:

```
while True:
    frame = read_sensor()
    features = preprocess(frame)
    inference = model(features)
    local_actuation(inference)
```

Σε αντίθεση με τα συστήματα αυστηρού πραγματικού χρόνου, οι εφαρμογές αυτές **μπορούν** να ανεχθούν περιορισμένο χρονικό διασκορπισμό (jitter) ή περιστασιακή απώλεια πλαισίων χωρίς να διακυβεύεται η ορθότητα της λειτουργίας τους. Ως αποτέλεσμα, τα υπολογιστικά απαιτητικά στάδια—και ιδιαίτερα η εξαγωγή συμπερασμάτων μέσω μοντέλων μηχανικής μάθησης— μπορούν να μεταφερθούν σε ισχυρότερους υπολογιστικούς πόρους, μια προσέγγιση που έχει μελετηθεί εκτενώς στα συστήματα υπολογιστικής αιχμής [7].

Ενδεικτικά παραδείγματα αποτελούν οι αλυσίδες ανάλυσης βίντεο που βασίζονται σε πλαίσια υπολογιστικής όρασης όπως το OpenCV [8], τα συστήματα παρακολούθησης EEG και ECG που χρησιμοποιούνται σε διεπαφές εγκεφάλου–υπολογιστή και στη βιοϊατρική επεξεργασία σημάτων [9, 10], οι εφαρμογές αναγνώρισης ήχου που βασίζονται σε μοντέλα βαθιάς μάθησης [11], καθώς και οι φορετές συσκευές παρακολούθησης της υγείας που συνδυάζουν αισθητήρες με απομακρυσμένη ανάλυση [7].

6.3 Βρόχοι Επαναληπτικής Επεξεργασίας Κατά Παρτίδες

Ορισμένες εφαρμογές ανάλυσης σε ενσωματωμένα συστήματα συγκεντρώνουν μετρήσεις αισθητήρων μέσα σε ένα προκαθορισμένο χρονικό παράθυρο και στη συνέχεια τις επεξεργάζονται ως μία παρτίδα.

```
while True:
    batch = collect_window(T)
    results = process(batch)
    upload(results)
```

Οι βρόχοι αυτού του τύπου είναι ιδιαίτερα συνηθισμένοι σε εφαρμογές επεξεργασίας σημάτων και ανάλυσης δεδομένων στην υπολογιστική αιχμής, όπου εφαρμόζονται τεχνικές ομαδοποίησης σε χρονικά παράθυρα και εξαγωγής χαρακτηριστικών από χρονοσειρές [12]. Εφόσον η επεξεργασία πραγματοποιείται σε συγκεντρωμένα δεδομένα αντί για μεμονωμένα δείγματα, οι εφαρμογές αυτές διαθέτουν συνήθως χαλαρότερους χρονικούς περιορισμούς και είναι ιδιαίτερα κατάλληλες για απομακρυσμένη εκτέλεση.

Αντιπροσωπευτικά παραδείγματα περιλαμβάνουν συστήματα ανάλυσης ιατρικών εικόνων βασισμένα στο MONAI [13], συστήματα ενεργειακής παρακολούθησης που συγκεντρώνουν και αναλύουν δεδομένα κατανάλωσης, όπως το Emoncms [14], καθώς και πλατφόρμες προγνωστικής συντήρησης σε βιομηχανικά περιβάλλοντα IoT, οι οποίες βασίζονται στην κατά παρτίδες επεξεργασία ιστορικών δεδομένων αισθητήρων [15].

6.4 Βρόχοι Εποπτείας και Ενορχήστρωσης

Μια τελευταία κατηγορία βρόχων είναι υπεύθυνη για την παρακολούθηση και τον συντονισμό του συστήματος και όχι για την άμεση επεξεργασία δεδομένων αισθητήρων.

```
while True:
    monitor_system()
    schedule_tasks()
    sync_with_cloud()
```

Οι βρόχοι αυτοί λειτουργούν στο επίπεδο ελέγχου (control plane) και συνήθως διαχειρίζονται σχετικά μικρό όγκο δεδομένων από αισθητήρες. Συντονίζουν τη συμπεριφορά της συσκευής, προγραμματίζουν τις εργασίες και συγχρονίζουν την κατάσταση του συστήματος με απομακρυσμένες υπηρεσίες, σύμφωνα με αρχές που έχουν μελετηθεί εκτενώς στα κατανεμημένα συστήματα και στο υπολογιστικό νέφος [16, 17].

Παραδείγματα αποτελούν τα συστήματα διαχείρισης στόλων συσκευών, οι ελεγκτές ενημερώσεων λογισμικού εξ αποστάσεως (Over-The-Air, OTA) που υλοποιούνται με πλαίσια όπως το Eclipse hawkBit [18], καθώς και οι υπηρεσίες σχεδιασμού διαδρομών οχημάτων που βασίζονται σε μηχανές όπως το GraphHopper [19]. Επειδή οι λειτουργίες αυτές αλληλεπιδρούν εκ φύσεως με απομακρυσμένες υποδομές, αποτελούν επίσης κατάλληλους υποψηφίους για επιλεκτική εκφόρτωση υπολογισμού.

6.5 Εποχές Εκτέλεσης

Παρόλο που οι εφαρμογές ενσωματωμένων συστημάτων εκφράζονται συνήθως μέσω άπειρων βρόχων ελέγχου [3], στην πράξη η εκτέλεσή τους εξελίσσεται σε διακριτούς κύκλους δειγματοληψίας και επεξεργασίας. Κάθε κύκλος αντιστοιχεί σε ένα δείγμα αισθητήρα, ένα πλαίσιο εισόδου ή μία παρτίδα συλλεγμένων μετρήσεων.

Το *Aégis* αξιοποιεί αυτή τη δομική ιδιότητα, κατατμήνοντας τη συνεχή εκτέλεση σε *πεπερασμένα παράθυρα επαναλήψεων*, τα οποία ονομάζονται *εποχές εκτέλεσης* (*execution epochs*). Μια εποχή εκτέλεσης αντιπροσωπεύει μια πεπερασμένη ακολουθία επαναλήψεων που αποσπάται από τον άπειρο βρόχο ελέγχου της εφαρμογής.

Η αφαίρεση αυτή επιτρέπει στον μεταγλωττιστή να αντιμετωπίζει κάθε επανάληψη ως μία ανεξάρτητη υπολογιστική μονάδα, η οποία μπορεί να προγραμματιστεί είτε τοπικά είτε απομακρυσμένα μέσα σε έναν ασφαλή θύλακα εκτέλεσης. Όταν οι επαναλήψεις του βρόχου δεν παρουσιάζουν εξαρτήσεις μεταξύ τους (*loop-carried dependencies*), πολλαπλές επαναλήψεις της ίδιας εποχής μπορούν να εκτελεστούν ταυτόχρονα σε διαφορετικούς κόμβους ασφαλών θυλάκων [16].

Με τον τρόπο αυτό, η διάσπαση των άπειρων βρόχων ελέγχου σε πεπερασμένες εποχές εκτέλεσης επιτρέπει στο *Aégis* να ευθυγραμμίζει τους φόρτους εργασίας των ενσωματωμένων συστημάτων με το μοντέλο εκτέλεσης που βασίζεται σε ανεξάρτητες εργασίες και χρησιμοποιείται στα αξιόπιστα περιβάλλοντα εκτέλεσης και στις υποδομές υπολογιστικού νέφους, καθιστώντας δυνατή την ασφαλή και κλιμακούμενη απομακρυσμένη εκτέλεση.

7 Υπόβαθρο

7.1 Υπολογιστικό Νέφος

Το υπολογιστικό νέφος (cloud computing) έχει αναδειχθεί ως ένα θεμελιώδες υπολογιστικό παράδειγμα για την παροχή κλιμακούμενων και διαθέσιμων κατ' απαίτηση υπολογιστικών πόρων μέσω του διαδικτύου [17]. Αξιοποιώντας μεγάλης κλίμακας κέντρα δεδομένων, πάροχοι υπηρεσιών νέφους όπως οι Amazon Web Services (AWS) [20], Microsoft Azure [21], Google Cloud [22] και IBM Cloud [23] προσφέρουν ουσιαστικά απεριόριστες δυνατότητες επεξεργασίας, αποθήκευσης και δικτύωσης. Το μοντέλο αυτό επιτρέπει στις εφαρμογές να μεταφέρουν υπολογιστικά απαιτητικές εργασίες από συσκευές με περιορισμένους πόρους σε ισχυρούς απομακρυσμένους εξυπηρετητές, βελτιώνοντας τόσο την απόδοση όσο και την κλιμακωσιμότητα. Παράλληλα, το υπολογιστικό νέφος υποστηρίζει ευέλικτα μοντέλα ανάπτυξης εφαρμογών, ελαστική διάθεση πόρων και μοντέλα χρέωσης ανάλογα με τη χρήση,

γεγονός που το καθιστά ιδιαίτερα δημοφιλές σε ένα ευρύ φάσμα εφαρμογών, από την ανάλυση δεδομένων έως τη μηχανική μάθηση και τα κατανεμημένα συστήματα.

7.2 Αξιόπιστα Περιβάλλοντα Εκτέλεσης (Trusted Execution Environments – TEEs)

Τα Αξιόπιστα Περιβάλλοντα Εκτέλεσης (Trusted Execution Environments – TEEs) παρέχουν μηχανισμούς απομόνωσης βασισμένους στο υλικό (hardware), οι οποίοι επιτρέπουν την ασφαλή εκτέλεση κώδικα και την προστασία ευαίσθητων δεδομένων ακόμη και όταν το λειτουργικό σύστημα ή ο υπερεπόπτης (hypervisor) έχουν παραβιαστεί. Μεταξύ των σημαντικότερων τεχνολογιών TEE συγκαταλέγονται το Intel SGX, το οποίο προσφέρει απομόνωση μνήμης μέσω ασφαλών θυλάκων εκτέλεσης (enclaves), το ARM TrustZone, το οποίο διαχωρίζει την εκτέλεση σε ασφαλή και μη ασφαλή κόσμο, καθώς και νεότερες τεχνολογίες όπως το Intel TDX, οι οποίες επεκτείνουν τις ίδιες εγγυήσεις σε εικονικοποιημένα περιβάλλοντα. Μέσω κρυπτογράφησης της μνήμης, μηχανισμών προστασίας ακεραιότητας και αυστηρών ελέγχων πρόσβασης, τα TEEs διασφαλίζουν την εμπιστευτικότητα και την ακεραιότητα των υπολογισμών, αποτρέποντας μη εξουσιοδοτημένη πρόσβαση ή αλλοίωση από προνομιούχα επίπεδα λογισμικού. Οι ιδιότητες αυτές καθιστούν τα TEEs μια ιδιαίτερα ισχυρή βάση για την ασφαλή εκφόρτωση υπολογισμού σε μη έμπιστες υποδομές υπολογιστικού νέφους.

Intel SGX Ένας ασφαλής θύλακας εκτέλεσης (enclave) αποτελεί μια προστατευμένη περιοχή εκτέλεσης στη μνήμη που παρέχεται από το Intel SGX [24] και απομονώνει τον κώδικα και τα δεδομένα από το υπόλοιπο σύστημα, συμπεριλαμβανομένου του λειτουργικού συστήματος και του υπερεπόπτη. Οι θύλακες εκτέλεσης λειτουργούν εντός του επεξεργαστή, ενώ κάθε πρόσβαση στη μνήμη που τους αντιστοιχεί κρυπτογραφείται και προστατεύεται ως προς την ακεραιότητα με διαφανή τρόπο από το υλικό του SGX. Έτσι διασφαλίζεται ότι τα δεδομένα που βρίσκονται στο εσωτερικό του θύλακα δεν μπορούν να αναγνωστούν ή να τροποποιηθούν από οποιαδήποτε εξωτερική διεργασία.

Η επικοινωνία μεταξύ του ασφαλούς θύλακα και της μη έμπιστης εφαρμογής πραγματοποιείται μέσω σαφώς καθορισμένων διεπαφών:

- **ECALLs (Enclave Calls):** Πρόκειται για κλήσεις που πραγματοποιούνται από τη μη έμπιστη εφαρμογή προς τον ασφαλή θύλακα. Μέσω αυτών η εφαρμογή ζητά την εκτέλεση προστατευμένων λειτουργιών ή υπολογισμών.
- **OCALLs (Outside Calls):** Πρόκειται για κλήσεις που εκτελούνται από τον ασφαλή θύλακα προς τη μη έμπιστη εφαρμογή, συνήθως για λειτουργίες που δεν είναι δυνατό να εκτελεστούν εντός του θύλακα, όπως λειτουργίες εισόδου/εξόδου ή επικοινωνία μέσω δικτύου.

Οι βασικές εγγυήσεις ασφαλείας που παρέχουν οι ασφαλείς θύλακες είναι οι εξής:

- **Εμπιστευτικότητα:** Τα δεδομένα που βρίσκονται εντός του θύλακα είναι κρυπτογραφημένα στη μνήμη και δεν μπορούν να προσπελαστούν από οποιαδήποτε εξωτερική οντότητα.
- **Ακεραιότητα:** Οποιαδήποτε προσπάθεια αλλοίωσης του κώδικα ή των δεδομένων του θύλακα ανιχνεύεται μέσω μηχανισμών ελέγχου που υλοποιούνται σε επίπεδο υλικού.
- **Απομονωμένη εκτέλεση:** Ο κώδικας που εκτελείται μέσα στον θύλακα λειτουργεί σε απομονωμένο περιβάλλον εκτέλεσης του επεξεργαστή, ανεξάρτητα από το λειτουργικό σύστημα ή άλλες εφαρμογές.

Επιπλέον, το Intel SGX υποστηρίζει έναν μηχανισμό απομακρυσμένης πιστοποίησης (remote attestation) [24], ο οποίος επιτρέπει σε έναν απομακρυσμένο φορέα επαλήθευσης (verifier) να επιβεβαιώσει ότι ένα συγκεκριμένο πρόγραμμα εκτελείται πράγματι μέσα σε έναν γνήσιο ασφαλή θύλακα σε μια αξιόπιστη πλατφόρμα. Η διαδικασία ξεκινά με τη δημιουργία από τον θύλακα μιας κρυπτογραφικής μέτρησης (measurement), γνωστής ως `MRENCLAVE`, η οποία ταυτοποιεί μοναδικά τον κώδικα και την αρχική κατάσταση του θύλακα. Στη συνέχεια, ο θύλακας δημιουργεί ένα *quote*, δηλαδή μια υπογεγραμμένη δομή δεδομένων που περιλαμβάνει τη μέτρηση αυτή μαζί με επιπλέον μεταδεδομένα. Το *quote* υπογράφεται χρησιμοποιώντας ένα ειδικό κλειδί πιστοποίησης που παρέχεται από την Intel για τη συγκεκριμένη πλατφόρμα, διασφαλίζοντας τη γνησιότητά του.

Το *quote* αποστέλλεται στον φορέα επαλήθευσης, συνήθως μέσω της υπηρεσίας Intel Attestation Service (IAS) ή μέσω των Data Center Attestation Primitives (DCAP) στις νεότερες εγκαταστάσεις. Ο φορέας επαλήθευσης ελέγχει την εγκυρότητα της υπογραφής ώστε να επιβεβαιώσει ότι αυτή προέρχεται από αυθεντική πλατφόρμα με υποστήριξη SGX και συγκρίνει τη μέτρηση του θύλακα με την αναμενόμενη τιμή. Εφόσον η διαδικασία ολοκληρωθεί επιτυχώς, αποκτά τη βεβαιότητα ότι ο επιθυμητός κώδικας εκτελείται μέσα σε έναν αυθεντικό ασφαλή θύλακα και ότι δεν έχει υποστεί αλλοίωση. Με τον τρόπο αυτό καθίσταται δυνατή η ασφαλής παροχή μυστικών πληροφοριών, όπως κρυπτογραφικών κλειδιών, απευθείας στον θύλακα, δημιουργώντας ένα αξιόπιστο κανάλι εκτέλεσης ακόμη και μέσω μη έμπιστων δικτύων.

7.3 Εναλλακτικές Προσεγγίσεις

Πέρα από τους ασφαλείς θύλακες που βασίζονται σε υλικό, όπως το Intel SGX, υπάρχουν και άλλες τεχνικές για την προστασία υπολογισμών και ευαίσθητων δεδομένων σε μη έμπιστα συστήματα:

- **Oblivious RAM (ORAM):** Μια τεχνική που αποκρύπτει τα πρότυπα πρόσβασης στη μνήμη, αποτρέποντας τη διαρροή πληροφοριών μέσω του τρόπου με τον οποίο προσπελαύνονται τα δεδομένα [25, 26, 27].
- **Ομομορφική Κρυπτογράφηση (Homomorphic Encryption):** Επιτρέπει την εκτέλεση υπολογισμών απευθείας πάνω σε κρυπτογραφημένα δεδομένα χωρίς προηγούμενη αποκρυπτογράφησή τους, διατηρώντας την εμπιστευτικότητα των πληροφοριών [28, 29, 30].
- **Ασφαλής Πολυμερής Υπολογισμός (Secure Multi-Party Computation – MPC):** Επιτρέπει σε πολλαπλά μέρη να υπολογίζουν από κοινού μια συνάρτηση πάνω στις εισόδους τους χωρίς να αποκαλύπτουν τα ιδιωτικά τους δεδομένα [31, 32, 33].
- **Trusted Platform Modules (TPM):** Κρυπτογραφικές μονάδες βασισμένες σε υλικό που παρέχουν ασφαλή αποθήκευση κλειδιών και δυνατότητες πιστοποίησης της πλατφόρμας [34, 35, 36].

Παρόλο που οι παραπάνω προσεγγίσεις παρέχουν ισχυρές εγγυήσεις ασφαλείας, συχνά συνοδεύονται από σημαντικά μεγαλύτερο υπολογιστικό ή επικοινωνιακό κόστος σε σύγκριση με τους ασφαλείς θύλακες του SGX. Οι θύλακες εκτέλεσης προσφέρουν έναν ιδιαίτερα πρακτικό συμβιβασμό, επιτρέποντας την αποδοτική και απομονωμένη εκτέλεση κώδικα, διατηρώντας ταυτόχρονα την εμπιστευτικότητα και την ακεραιότητα των δεδομένων.

8 Μοντέλο Απειλών

Με βάση τους μηχανισμούς των Αξιόπιστων Περιβαλλόντων Εκτέλεσης (TEEs) και του Intel SGX που παρουσιάστηκαν προηγουμένως, υιοθετούμε το ακόλουθο μοντέλο απειλών. Θεωρούμε ότι το περιβάλλον εκτέλεσης στον απομακρυσμένο εξυπηρετητή—συμπεριλαμβανομένων του λειτουργικού συστήματος, του υπερεπόπτη (hypervisor) και κάθε άλλου προνομιούχου λογισμικού— δεν είναι έμπιστο. Τα στοιχεία αυτά ενδέχεται να ενεργούν κακόβουλα και να επιχειρούν να επιθεωρήσουν, να αλλοιώσουν ή να παρεμποδίσουν την εκτέλεση των εργασιών που έχουν εκφορτωθεί. Ειδικότερα, θεωρούμε ότι ο αντίπαλος μπορεί να παρακολουθεί όλη την επικοινωνία μέσω του δικτύου, να ελέγχει τον χρονοπρογραμματισμό των εργασιών και να αποκτά πρόσβαση σε κάθε περιοχή μνήμης που βρίσκεται εκτός του ασφαλούς θύλακα.

Για την αντιμετώπιση αυτών των απειλών βασιζόμαστε σε Αξιόπιστα Περιβάλλοντα Εκτέλεσης, και συγκεκριμένα στους ασφαλείς θύλακες του Intel SGX, οι οποίοι παρέχουν απομονωμένη εκτέλεση και προστατεύουν την εμπιστευτικότητα και την ακεραιότητα τόσο του κώδικα όσο και των δεδομένων που βρίσκονται στο εσωτερικό τους. Η επικοινωνία μεταξύ πελάτη και ασφαλούς θύλακα προστατεύεται μέσω απομακρυσμένης πιστοποίησης (remote attestation) και κρυπτογραφημένων καναλιών επικοινωνίας, διασφαλίζοντας ότι τα ευαίσθητα δεδομένα είναι προσβάσιμα αποκλειστικά από αξιόπιστες παρουσίες του θύλακα.

Υποθέτουμε ότι ο επεξεργαστής και το υλικό του Intel SGX είναι αξιόπιστα και ότι οι κρυπτογραφικοί μηχανισμοί που χρησιμοποιούνται για την ασφαλή επικοινωνία έχουν υλοποιηθεί ορθά. Δεν εξετάζουμε φυσικές επιθέσεις, όπως επιθέσεις πλευρικού καναλιού (π.χ. μέσω χρονισμού της κρυφής μνήμης ή ανάλυσης κατανάλωσης ισχύος), επιθέσεις άρνησης εξυπηρέτησης (Denial-of-Service) ή επιθέσεις επαναφοράς προηγούμενης κατάστασης (rollback attacks), καθώς αυτές βρίσκονται εκτός του αντικειμένου της παρούσας εργασίας.

9 Προκλήσεις

9.1 Κόστος και βαθμός λεπτομέρειας της εκφόρτωσης υπολογισμού

Η επιλεκτική εκφόρτωση υπολογισμού δεν είναι πάντοτε ωφέλιμη. Διαισθητικά, η μεταφορά ενός υπολογισμού σε απομακρυσμένο περιβάλλον είναι συμφέρουσα μόνο όταν το συνολικό κόστος της απομακρυσμένης εκτέλεσης είναι μικρότερο από τον χρόνο τοπικής εκτέλεσης. Αυτό μπορεί να εκφραστεί ως:

$$T_{\text{offload}} = T_{\text{enc}} + T_{\text{net}} + T_{\text{remote}} < T_{\text{local}}$$

όπου το T_{enc} αντιστοιχεί στο κόστος κρυπτογράφησης και σειριοποίησης των δεδομένων, το T_{net} εκφράζει την καθυστέρηση και τον χρόνο μετάδοσης μέσω του δικτύου, ενώ το T_{remote} είναι ο χρόνος εκτέλεσης μέσα στον απομακρυσμένο ασφαλή θύλακα.

Επιπλέον, ο καθορισμός του κατάλληλου βαθμός λεπτομέρειας της εκφόρτωσης αποτελεί μια ιδιαίτερα απαιτητική πρόκληση. Σε πολλές πραγματικές εφαρμογές, οι υπολογιστικές ροές οργανώνονται γύρω από συναρτήσεις *black-box*, οι οποίες συνήθως εισάγονται από εξωτερικές βιβλιοθήκες και ενσωματώνουν ένα πλήρες στάδιο επεξεργασίας, όπως εξαγωγή συμπερασμάτων, μετασχηματισμό σημάτων ή εξαγωγή χαρακτηριστικών. Οι συναρτήσεις αυτές καλούνται μέσα σε ευρύτερες δομές ελέγχου και συχνά κυριαρχούν στο συνολικό κόστος εκτέλεσης της εφαρμογής.

Ωστόσο, δεν είναι πάντοτε προφανές εάν θα πρέπει να εκφορτωθεί η ίδια η εισαγόμενη συνάρτηση ή η περιβάλλουσα συνάρτηση (*wrapper*) που την καλεί. Η δημιουργία προφίλ εκτέλεσης (profiling) στο επίπεδο της συνάρτησης-περιβλήματος μπορεί να οδηγήσει σε παραπλανητικά συμπεράσματα, καθώς συνδυάζει τόσο τις υπολογιστικά απαιτητικές λειτουργίες όσο και βοηθητική λογική χαμηλού κόστους, όπως η προετοιμασία των δεδομένων ή η διαχείριση της ροής ελέγχου. Ως αποτέλεσμα, ενδέχεται να υπερεκτιμά το πραγματικό υπολογιστικό έργο που μεταφέρεται και να υποεκτιμά το επικοινωνιακό κόστος που συνεπάγεται η μεταφορά πρόσθετων δεδομένων.

Η πρόκληση αυτή αναδεικνύει την ανάγκη για λεπτομερέστερη ανάλυση, ικανή να απομονώνει τα πραγματικά υπολογιστικά σημεία συμφόρησης μέσα σε μια συνάρτηση. Εντοπίζοντας το ελάχιστο υπολογιστικά απαιτητικό τμήμα του κώδικα, το σύστημα μπορεί να περιορίσει τις άσκοπες μεταφορές δεδομένων, διατηρώντας παράλληλα το μεγαλύτερο μέρος του οφέλους ως προς την απόδοση. Για τον σκοπό αυτό, το *Aégis* έχει σχεδιαστεί ώστε να υποστηρίζει εκφόρτωση υπολογισμού λεπτής βαθμός λεπτομέρειας, καθοδηγούμενη από τις επισημάνσεις του προγραμματιστή, επιτρέποντας την επιλεκτική εξαγωγή και απομακρυσμένη εκτέλεση μόνο των πλέον υπολογιστικά κρίσιμων τμημάτων αντί ολόκληρων συναρτήσεων-περιβλημάτων. Η σχεδιαστική αυτή επιλογή βελτιώνει τον λόγο επικοινωνίας προς υπολογισμό και οδηγεί σε αποδοτικότερες και περισσότερο κλιμακούμενες αποφάσεις εκφόρτωσης.

9.2 Περιορισμοί μνήμης των TEE

Τα Αξιόπιστα Περιβάλλοντα Εκτέλεσης (TEEs) επιβάλλουν αυστηρούς περιορισμούς στους διαθέσιμους πόρους, κυρίως ως προς τη χωρητικότητα της μνήμης, γεγονός που περιορίζει το μέγεθος και την πολυπλοκότητα των υπολογισμών που μπορούν να εκτελεστούν απομακρυσμένα. Ωστόσο, στην πράξη ο περιορισμός αυτός αποδεικνύεται συχνά λιγότερο αυστηρός από όσο φαίνεται. Οι εφαρμογές ενσωματωμένων συστημάτων σχεδιάζονται εξαρχής ώστε να λειτουργούν υπό αυστηρούς περιορισμούς τόσο σε μνήμη όσο και σε υπολογιστική ισχύ, ενώ συνήθως επεξεργάζονται δεδομένα αισθητήρων που καταφθάνουν σε μορφή συνεχών ροών ή περιορισμένων χρονικών παραθύρων και όχι ως μεγάλα μονολιθικά σύνολα δεδομένων. Ως αποτέλεσμα, οι επιμέρους μονάδες υπολογισμού είναι εκ φύσεως μικρές και αυτάρκεις. Επιπλέον, η εκφόρτωση πραγματοποιείται στο επίπεδο των εποχών εκτέλεσης (execution epochs) ή των επιμέρους εργασιών κάθε επανάληψης, γεγονός που περιορίζει ακόμη περισσότερο το αποτύπωμα μνήμης κάθε απομακρυσμένης εκτέλεσης. Συνολικά, οι ιδιότητες αυτές εναρμονίζονται ιδιαίτερα καλά με το περιορισμένο περιβάλλον των TEEs, μετριάζοντας τις επιπτώσεις της περιορισμένης χωρητικότητας μνήμης.

9.3 Προγραμματισμός εφαρμογών Intel SGX

Παρόλο που το Intel SGX παρέχει ισχυρές εγγυήσεις απομόνωσης, η ανάπτυξη εφαρμογών SGX και η εκφόρτωση υπολογισμού σε ασφαλείς θύλακες απαιτούν από τον προγραμματιστή να διαχειριστεί μια αρκετά πολύπλοκη διαδικασία ανάπτυξης, όπως περιγράφεται παρακάτω.

Πρώτον, ο προγραμματιστής πρέπει να **ορίσει ρητά τις διεπαφές του ασφαλούς θύλακα**. Αυτό απαιτεί τη συγγραφή ενός αρχείου **EDL** (Enclave Definition Language), στο οποίο δηλώνονται όλες οι ECALLs (συναρτήσεις που καλούνται από τη μη έμπιστη εφαρμογή προς τον θύλακα) και οι OCALLs (συναρτήσεις που καλούνται από τον θύλακα προς το μη έμπιστο περιβάλλον εκτέλεσης). Μετά από κάθε τροποποίηση του αρχείου EDL, πρέπει να εκτελεστεί η εργαλειοθήκη του SGX ώστε να παραχθεί αυτόματα ο αντίστοιχος αξιόπιστος και μη αξιόπιστος ενδιάμεσος κώδικας (proxy code) που υλοποιεί τις διεπαφές αυτές.

Δεύτερον, ο προγραμματιστής πρέπει να **διαχωρίσει ολόκληρη την εφαρμογή σε αξιόπιστα και μη αξιόπιστα τμήματα και να συντηρεί δύο ανεξάρτητες βάσεις κώδικα**. Το τμήμα που

εκτελείται μέσα στον ασφαλή θύλακα υπόκειται στους περιορισμούς του SGX: δεν επιτρέπεται να εκτελεί κλήσεις συστήματος, να προσπελαύνει απευθείας το σύστημα αρχείων ή το δίκτυο, ούτε να δεσμεύει μνήμη χωρίς περιορισμούς. Επιτρέπεται μόνο ένα περιορισμένο σύνολο λειτουργιών και κάθε αλληλεπίδραση με το εξωτερικό περιβάλλον πρέπει να πραγματοποιείται μέσω OCALLs προς το μη έμπιστο τμήμα της εφαρμογής.

Τρίτον, κάθε ανταλλαγή δεδομένων διαμέσου των ορίων του ασφαλούς θύλακα πρέπει να γίνεται με χειροκίνητη προσαρμογή (marshalling). Τα ορίσματα των ECALLs και οι επιστρεφόμενες τιμές πρέπει να σειριοποιούνται σε απλούς πίνακες μνήμης πριν εισέλθουν ή εξέλθουν από τον θύλακα, να αντιγράφονται μεταξύ των δύο περιοχών μνήμης και στη συνέχεια να αποσειριοποιούνται στην άλλη πλευρά. Ο προγραμματιστής οφείλει να διαχειρίζεται προσεκτικά την ιδιοκτησία της μνήμης, τον κύκλο ζωής των buffers και τη διάταξη των δεδομένων, ώστε να αποφεύγονται διαρροές μνήμης, αλλοιώσεις ή μη ορισμένη συμπεριφορά.

Τέταρτον, ο μηχανισμός απομακρυσμένης πιστοποίησης και ασφαλούς επικοινωνίας πρέπει να υλοποιηθεί από τον ίδιο τον προγραμματιστή. Προκειμένου να αποδειχθεί ότι ο απομακρυσμένος κώδικας εκτελείται πράγματι μέσα σε έναν αυθεντικό ασφαλή θύλακα, ο προγραμματιστής πρέπει να υλοποιήσει τη διαδικασία remote attestation (π.χ. μέσω του Intel Attestation Service ή του DCAP) και να επαληθεύσει την ταυτότητα και τη μέτρηση του θύλακα. Μετά την ολοκλήρωση της πιστοποίησης, πρέπει να δημιουργηθεί μια ασφαλής συνεδρία επικοινωνίας, η οποία περιλαμβάνει ανταλλαγή κλειδιών και εγκαθίδρυση κρυπτογραφημένων καναλιών που παρέχουν εμπιστευτικότητα, ακεραιότητα και προστασία έναντι επιθέσεων επανάληψης (replay) ή αλλοίωσης.

Τέλος, το σύστημα πρέπει να **μεταγλωττιστεί και να αναπτυχθεί**. Ο κώδικας του ασφαλούς θύλακα μεταγλωττίζεται σε δυαδικό αρχείο enclave, υπογράφεται ψηφιακά και συσκευάζεται μαζί με τη μη έμπιστη εφαρμογή υποδοχής. Ο προγραμματιστής είναι υπεύθυνος για την εγκατάσταση και των δύο τμημάτων στον εξυπηρετητή, τη ρύθμιση του περιβάλλοντος εκτέλεσης (οδηγοί SGX, βιβλιοθήκες και λοιπές εξαρτήσεις), καθώς και για τη διασφάλιση ότι οι παρουσίες των ασφαλών θυλάκων αρχικοποιούνται σωστά και είναι διαθέσιμες για απομακρυσμένη εκτέλεση.

Συνοψίζοντας, η ανάπτυξη εφαρμογών με Intel SGX απαιτεί από τον προγραμματιστή να ορίζει ρητά τις διεπαφές του ασφαλούς θύλακα μέσω αρχείων EDL, να διαχωρίζει τον κώδικα μεταξύ εφαρμογής υποδοχής και θύλακα, να διαχειρίζεται τη μεταφορά δεδομένων διαμέσου των ορίων του θύλακα και να αναλαμβάνει ολόκληρο τον κύκλο ζωής και τη διαδικασία ανάπτυξης των enclaves. Όταν το SGX χρησιμοποιείται ως βάση για κατανεμημένη εκφόρτωση υπολογισμού, οι ευθύνες αυτές επιβαρύνονται ακόμη περισσότερο από την ανάγκη ενσωμάτωσης δικτυακής επικοινωνίας, σειριοποίησης αιτημάτων, διαχείρισης εξαρτήσεων και ασφαλούς επικοινωνίας. Κατά συνέπεια, η λογική της εφαρμογής συνδέεται στενά με τη χαμηλού επιπέδου υποδομή του SGX, αυξάνοντας την πολυπλοκότητα της ανάπτυξης και μειώνοντας τη φορητότητα του λογισμικού. Το γεγονός αυτό αναδεικνύει την ανάγκη για αφαιρέσεις υψηλότερου επιπέδου, οι οποίες επιτρέπουν στον προγραμματιστή να δηλώνει *τι* πρέπει να εκφορτωθεί, αποκρύπτοντας παράλληλα τις λεπτομέρειες του *πώς* υλοποιείται η ασφαλής απομακρυσμένη εκτέλεση.

10 Σχεδιασμός του Συστήματος

10.1 Στόχοι Σχεδιασμού

Ο σχεδιασμός του *Aégis* καθοδηγείται από τις προκλήσεις που παρουσιάστηκαν στις προηγούμενες ενότητες, καθώς και από την ανάγκη παροχής ισχυρών εγγυήσεων εμπιστευτικότητας των δεδομένων και τον περιορισμό της επιβάρυνσης που συνεπάγεται η απομακρυσμένη εκτέλεση. Με βάση τους

περιορισμούς αυτούς, το σύστημα σχεδιάστηκε ώστε να ικανοποιεί τους ακόλουθους στόχους:

- **Απόδοση:** Βελτίωση της ρυθμαπόδοσης των εφαρμογών μέσω της εκφόρτωσης υπολογιστικά απαιτητικών εργασιών και της αξιοποίησης παραλληλισμού σε πολλαπλούς κόμβους ασφαλών θυλάκων.
- **Κλιμακωσιμότητα:** Υποστήριξη ταυτόχρονης εκτέλεσης σε πολλαπλούς ασφαλείς θύλακες, επιτρέποντας στο σύστημα να κλιμακώνεται ανάλογα με τους διαθέσιμους ασφαλείς υπολογιστικούς πόρους.
- **Ασφάλεια:** Διασφάλιση της προστασίας των ευαίσθητων δεδομένων κατά την εκφόρτωση υπολογισμού μέσω της αξιοποίησης Αξιόπιστων Περιβαλλόντων Εκτέλεσης και ασφαλών μηχανισμών επικοινωνίας.
- **Διαφάνεια:** Ελαχιστοποίηση της προσπάθειας που απαιτείται από τον προγραμματιστή μέσω ενός υψηλού επιπέδου μοντέλου προγραμματισμού, το οποίο αποκρύπτει τις λεπτομέρειες χαμηλού επιπέδου της απομακρυσμένης εκτέλεσης.

10.2 Επισκόπηση του Συστήματος

Το Σχήμα 1 παρουσιάζει την αρχιτεκτονική υψηλού επιπέδου του συστήματος.

Ο προγραμματιστής παρέχει τον επισημασμένο πηγαίο κώδικα στον μεταγλωττιστή DSL του *Aégis* , ο οποίος τον μετασχηματίζει σε μια ισοδύναμη ροή εκτέλεσης που διαχωρίζει το πρόγραμμα σε τοπικές και απομακρυσμένες εργασίες. Κατά τη διάρκεια του μετασχηματισμού, ο μεταγλωττιστής επισημαίνει τις συναρτήσεις που πρόκειται να εκφορτωθούν ως *remote* και μετατρέπει τις κλήσεις τους σε ασύγχρονες. Στη συνέχεια, ο μηχανισμός διανομής (dispatcher) περικλείει τις επισημασμένες συναρτήσεις σε αντικείμενα διαμεσολάβησης (proxies), αποκρύπτοντας τις λεπτομέρειες της απομακρυσμένης κλήσης και επιτρέποντας την επίκλησή τους σαν να επρόκειτο για τοπικές συναρτήσεις.

Το περιβάλλον εκτέλεσης QuickJS στην πλευρά του πελάτη εκτελεί το μετασχηματισμένο πρόγραμμα τοπικά μέχρι να συναντήσει μια οδηγία *remote*. Στο σημείο αυτό, συλλέγει τα απαραίτητα δεδομένα εισόδου μαζί με όλα τα απαιτούμενα εξαρτώμενα αρχεία και βιβλιοθήκες, δημιουργώντας ένα αίτημα απομακρυσμένης εκτέλεσης. Το αίτημα αυτό κρυπτογραφείται και μεταδίδεται μέσω ειδικού ασφαλούς καναλιού σε έναν προκαθορισμένο εξυπηρετητή.

Μετά την παραλαβή του αιτήματος, ένας χειριστής αιτημάτων (request handler) στο μη έμπιστο τμήμα του εξυπηρετητή προωθεί το κρυπτογραφημένο φορτίο στον ασφαλή θύλακα. Στο εσωτερικό του αξιόπιστου περιβάλλοντος εκτέλεσης, ένα εξειδικευμένο περιβάλλον εκτέλεσης αποκρυπτογραφεί το αίτημα και εκτελεί απομονωμένα τη ζητούμενη συνάρτηση. Μετά την ολοκλήρωση της εκτέλεσης, τα αποτελέσματα κρυπτογραφούνται μέσα στον θύλακα και επιστρέφονται με ασφάλεια στον πελάτη, όπου επανενσωματώνονται στη ροή εκτέλεσης της εφαρμογής.

10.2.1 DSL και Μετασχηματισμοί Προγραμμάτων

Ο μεταγλωττιστής DSL του *Aégis* πραγματοποιεί έναν μετασχηματισμό πηγαίου προς πηγαίο κώδικα (*source-to-source transformation*), ο οποίος μετατρέπει επισημασμένους υπολογισμούς που βασίζονται σε βρόχους σε ένα κατατμημένο μοντέλο εκτέλεσης κατάλληλο για ασφαλή εκφόρτωση υπολογισμού. Ο μετασχηματισμός καθοδηγείται από επισημάνσεις που παρέχονται από τον προγραμματιστή και προσδιορίζουν τις υπολογιστικά απαιτητικές συναρτήσεις μέσα στο σώμα των βρόχων ως υποψήφιες για απομακρυσμένη εκτέλεση.

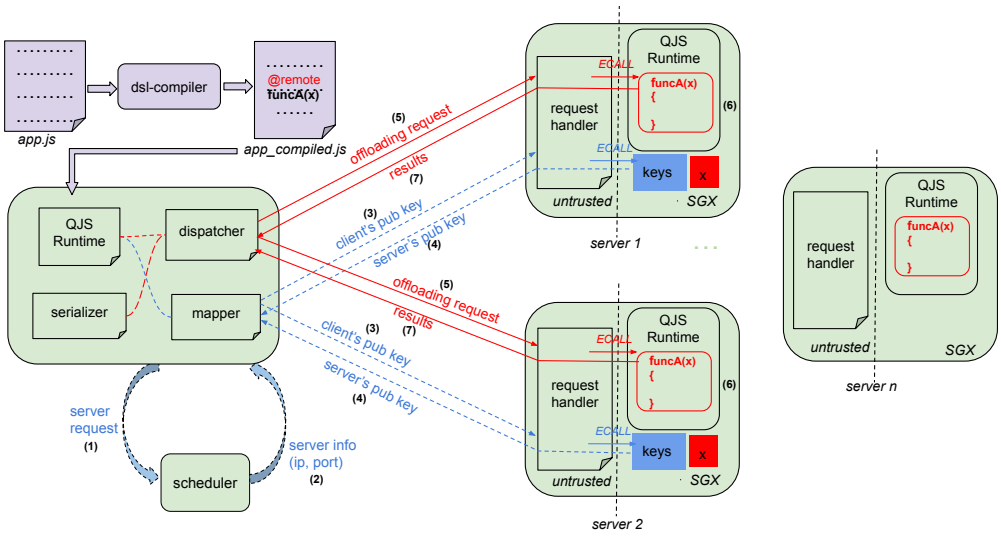

Figure 1: Επισκόπηση του *Aégis* . Ο μεταγλωττιστής DSL μετασχηματίζει τον επισημασμένο κώδικα σε ένα κατατμημένο μοντέλο εκτέλεσης που βασίζεται σε απομακρυσμένες κλήσεις μέσω proxies. Το περιβάλλον εκτέλεσης στην πλευρά του πελάτη εκφορτώνει εργασίες σε εξυπηρετητές που υποστηρίζονται από ασφαλείς θύλακες εκτέλεσης, όπου οι υπολογισμοί πραγματοποιούνται με ασφάλεια και τα αποτελέσματα επιστρέφονται στον πελάτη.

Σε υψηλό επίπεδο, ο μεταγλωττιστής στοχεύει σε κανονικούς (*canonical*) βρόχους `for` με στατικά γνωστό αριθμό επαναλήψεων. Για κάθε τέτοιο βρόχο, εξάγει το σώμα του και το μεταφέρει σε μια ξεχωριστή ασύγχρονη συνάρτηση. Ο αρχικός βρόχος αντικαθίσταται στη συνέχεια από μια κλήση προς το περιβάλλον εκτέλεσης, το οποίο αναλαμβάνει τον συντονισμό πολλαπλών εκτελέσεων της παραγόμενης συνάρτησης.

Στο εσωτερικό της νέας συνάρτησης, οι κλήσεις προς τις επισημασμένες ρουτίνες μετασχηματίζονται σε ασύγχρονες κλήσεις μέσω proxy, με την εισαγωγή εκφράσεων `await`. Τα proxies αποκρύπτουν τις λεπτομέρειες της απομακρυσμένης εκτέλεσης, επιτρέποντας στις εκφορτωμένες συναρτήσεις να καλούνται διαφανώς σαν να ήταν τοπικές. Παράλληλα, η περιβάλλουσα συνάρτηση επισημαίνεται ως `async`, ώστε να διατηρείται η ορθή σημασιολογία της ροής ελέγχου.

Ο μετασχηματισμός αυτός διασπά έναν σειριακό βρόχο σε ένα σύνολο ανεξάρτητων μονάδων εκτέλεσης, οι οποίες μπορούν να προγραμματιστούν αυτόνομα. Ανάλογα με τις επισημάνσεις που έχει ορίσει ο προγραμματιστής, ο μεταγλωττιστής εφαρμόζει έναν από τους τρεις ακόλουθους τρόπους εκτέλεσης.

Παράλληλη λειτουργία (`--parallel`). Η λειτουργία αυτή χρησιμοποιείται όταν οι επαναλήψεις του βρόχου είναι ανεξάρτητες και επεξεργάζονται δεδομένα με την ίδια δομή εισόδου. Ο μεταγλωττιστής αφαιρεί τον βρόχο και αναθέτει τον έλεγχο των επαναλήψεων στο περιβάλλον εκτέλεσης μέσω της συνάρτησης `scaleout_traffic`, η οποία αποστέλλει τα αιτήματα ταυτόχρονα σε πολλαπλούς κόμβους ασφαλών θυλάκων. Κάθε απομακρυσμένη εκτέλεση αντιστοιχεί στην ίδια υπολογιστική εργασία που εφαρμόζεται σε διαφορετική επανάληψη του βρόχου, αξιοποιώντας έτσι *παραλληλισμό σε επίπεδο εργασιών* (*task-level parallelism*), γνωστό και ως *embarrassingly parallel execution*.

Λειτουργία κατά παρτίδες (`--batch`). Η λειτουργία αυτή εφαρμόζεται όταν κάθε επανάληψη επεξεργάζεται διαφορετικά δεδομένα εισόδου που εξαρτώνται από τον δείκτη του βρόχου. Ο μεταγλωττιστής

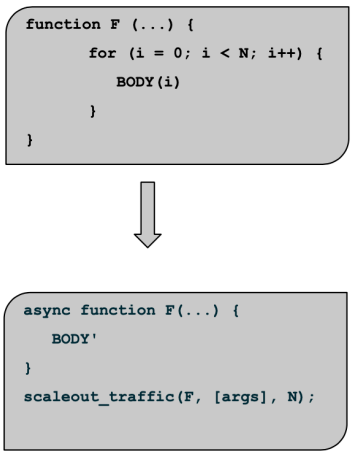

Figure 2: Μετασχηματισμός παράλληλης εκτέλεσης (*–parallel*)

εξάγει το σώμα του βρόχου σε μια παραμετρική ασύγχρονη συνάρτηση και μετατρέπει τον χώρο επαναλήψεων σε μια ρητά καθορισμένη λίστα εργασιών. Οι μεταβλητές που μεταβάλλονται ανά επανάληψη, καθώς και όλες οι τοπικές μεταβλητές που απαιτούνται για την εκτέλεση, εντοπίζονται και περνούν ως ρητές παράμετροι της συνάρτησης. Στη συνέχεια, το περιβάλλον εκτέλεσης κατανέμει τις εργασίες στους διαθέσιμους κόμβους μέσω της συνάρτησης `scaleout_map`.

Με τη ρητή απαρίθμηση των εισόδων, η λειτουργία αυτή αξιοποιεί *παραλληλισμό σε επίπεδο δεδομένων* (*data-level parallelism*), όπου κάθε εργασία επεξεργάζεται διαφορετικό τμήμα του χώρου εισόδου. Έτσι καθίσταται δυνατή η αποδοτική κατανεμημένη εκτέλεση πάνω σε ετερογενή δεδομένα, διατηρώντας παράλληλα τη σημασιολογία κάθε επανάληψης.

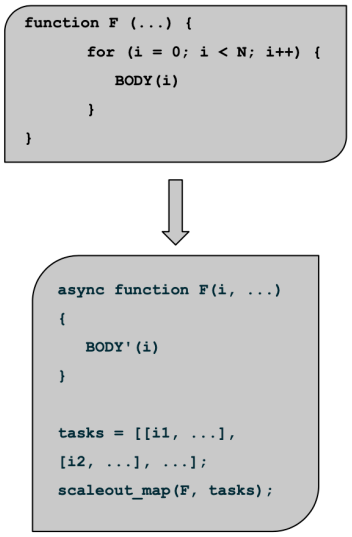

Figure 3: Μετασχηματισμός κατά παρτίδες (*–batch*)

Προεπιλεγμένη λειτουργία (χωρίς επισημάνσεις). Όταν δεν έχει καθοριστεί κάποιος τρόπος εκτέλεσης, ο μεταγλωττιστής διατηρεί συντηρητικά την αρχική ροή ελέγχου και περικλείει τον υπολογισμό μέσω της συνάρτησης `create_traffic`, διασφαλίζοντας την πλήρως σειριακή σημασιολογία της

εκτέλεσης.

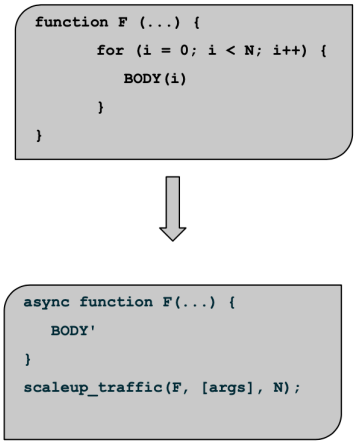

Figure 4: Προεπιλεγμένος μετασχηματισμός (χωρίς επισημάνσεις)

Περιορισμοί Ορθότητας Για να διασφαλιστεί η ορθότητα του μετασχηματισμού, ο μεταγλωττιστής υποθέτει ένα σύνολο δομικών περιορισμών στους επισημασμένους βρόχους. Συγκεκριμένα, απαιτείται: (i) κανονική μορφή του βρόχου, (ii) στατικά γνωστό πλήθος επαναλήψεων, (iii) απουσία εντολών που διακόπτουν τη ροή του βρόχου, όπως οι `break` και `continue`, καθώς και (iv) απουσία εξαρτήσεων μεταξύ διαδοχικών επαναλήψεων (*loop-carried dependencies*). Οι περιορισμοί αυτοί εγγυώνται ότι κάθε επανάληψη μπορεί να εκτελεστεί ανεξάρτητα και με ασφάλεια.

Πέρα από τον μετασχηματισμό της ροής ελέγχου, ο μεταγλωττιστής πραγματοποιεί επίσης στατική ανάλυση εξαρτήσεων, προκειμένου να εντοπίσει όλες τις συναρτήσεις και τις βιβλιοθήκες που απαιτούνται για την απομακρυσμένη εκτέλεση. Για τον σκοπό αυτό κατασκευάζεται και διατρέχεται ένα γράφημα κλήσεων (call graph), ώστε να επιλυθούν αναδρομικά όλες οι μεταβατικές εξαρτήσεις κάθε εκφορτωμένης συνάρτησης. Το προκύπτον σύνολο εξαρτήσεων σειριοποιείται και συσκευάζεται μαζί με τον μετασχηματισμένο κώδικα, σχηματίζοντας μια πλήρως αυτάρκη μονάδα εκτέλεσης. Για τη μείωση του κόστους κατά τον χρόνο εκτέλεσης, τα μεταδεδομένα αυτά αποθηκεύονται τοπικά σε προσωρινή μνήμη (cache) και μπορούν προαιρετικά να μεταγλωττιστούν εκ των προτέρων σε byte-code, επιτρέποντας στο σύστημα να μεταδίδει συμπαγείς αναπαραστάσεις αντί για ολόκληρο τον πηγαίο κώδικα κατά τις απομακρυσμένες εκτελέσεις.

10.2.2 Μηχανισμός Εκφόρτωσης Υπολογισμού

Ο Μηχανισμός Εκφόρτωσης Υπολογισμού (Offloading Engine) είναι υπεύθυνος για τον συντονισμό των υπολογιστικών εργασιών τόσο στους τοπικούς όσο και στους απομακρυσμένους κόμβους. Όταν πραγματοποιείται κλήση μιας συνάρτησης, ο μηχανισμός παρεμβάλλεται στη διαδικασία εκτέλεσης και αποφασίζει εάν η συνάρτηση θα εκτελεστεί τοπικά ή θα εκφορτωθεί σε έναν απομακρυσμένο κόμβο εκτέλεσης.

Για τις εργασίες που πρόκειται να εκφορτωθούν, δημιουργεί ένα αίτημα το οποίο περιλαμβάνει τα σειριοποιημένα ορίσματα της συνάρτησης, όλες τις απαραίτητες εξαρτήσεις, καθώς και τις αντίστοιχες δηλώσεις εισαγωγής (*import statements*), εξασφαλίζοντας ότι ο απομακρυσμένος κόμβος διαθέτει πλήρες περιβάλλον εκτέλεσης.

Για την υποστήριξη της διαδικασίας αυτής, ο μηχανισμός διαβάζει αρχικά ένα αρχείο μεταδεδομένων, το οποίο έχει παραχθεί από τον μεταγλωττιστή και αναγνωρίζεται μέσω κρυπτογραφικού κατακερματισμού (hash). Το αρχείο αυτό περιέχει προϋπολογισμένες πληροφορίες σχετικά με τις συναρτήσεις, τις επισημάνσεις εκφόρτωσης, τις εξαρτήσεις τους και τον αντίστοιχο πηγαίο κώδικα. Τα μεταδεδομένα χρησιμοποιούνται για την αρχικοποίηση εσωτερικών δομών, όπως το σύνολο των εκφορτωμένων συναρτήσεων (offloads set), ο χάρτης εξαρτήσεων μεταξύ συναρτήσεων και βιβλιοθηκών, καθώς και το ακατέργαστο κείμενο του πηγαίου κώδικα, επιτρέποντας την αποδοτική προετοιμασία των εργασιών χωρίς να απαιτείται ανάλυση της ροής του προγράμματος κατά τον χρόνο εκτέλεσης.

Σε κάθε αίτημα αποδίδεται ένας μοναδικός αριθμός ακολουθίας (nonce), ώστε να γίνεται αποδεκτή μόνο η αντίστοιχη απάντηση, αποτρέποντας την αποδοχή απαντήσεων εκτός σειράς ή επιθέσεις επανάληψης (replay attacks).

Για την υποστήριξη ασύγχρονης εκτέλεσης και παραλληλισμού, οι κλήσεις συναρτήσεων τυλίγονται σε αντικείμενα `Promise`, επιτρέποντας στον τοπικό κώδικα να συνεχίσει την εκτέλεσή του όσο οι απομακρυσμένες εργασίες βρίσκονται σε εξέλιξη. Όταν ένας απομακρυσμένος κόμβος ολοκληρώσει μια εργασία, ο μηχανισμός επαληθεύει την απάντηση και επιλύει (*resolve*) το αντίστοιχο promise, επιστρέφοντας διαφανώς το αποτέλεσμα στην αρχική κλήση της συνάρτησης.

Μέσω αυτού του μηχανισμού, το Offloading Engine διαχειρίζεται αποτελεσματικά την ουρά εργασιών, την εκτέλεση και την επιστροφή αποτελεσμάτων σε ένα κατανεμημένο περιβάλλον, ενώ παράλληλα επιτρέπει τη δυναμική κλιμάκωση της εκτέλεσης σε πολλαπλούς κόμβους.

10.2.3 Χειραψίες (Handshakes) και Αποκλειστικές Συνδέσεις

Πριν από οποιαδήποτε εκφόρτωση υπολογισμού, ο πελάτης εγκαθιστά ασφαλείς συνδέσεις με κάθε εξυπηρετητή που του έχει ανατεθεί. Η διαδικασία αυτή ξεκινά με μια χειραψία (handshake) για κάθε κόμβο, κατά την οποία ο πελάτης και ο εξυπηρετητής ανταλλάσσουν δημόσια κλειδιά και παράγουν ένα κοινό συμμετρικό κλειδί, μοναδικό για τη συγκεκριμένη συνεδρία.

Τα κλειδιά αυτά χρησιμοποιούνται για την κρυπτογράφηση όλης της επόμενης επικοινωνίας, εξασφαλίζοντας εμπιστευτικότητα από άκρο σε άκρο. Κάθε πελάτης διατηρεί μια αποκλειστική σύνδεση TCP με κάθε εξυπηρετητή, η οποία παραμένει ενεργή καθ' όλη τη διάρκεια πολλαπλών εκφορτώσεων εργασιών, μειώνοντας το κόστος δημιουργίας νέων συνδέσεων και βελτιώνοντας τη συνολική απόδοση.

Σε αντίθεση με τους πλήρεις μηχανισμούς απομακρυσμένης πιστοποίησης του SGX (όπως οι IAS και DCAP), ο συγκεκριμένος μηχανισμός χειραψίας δεν πραγματοποιεί κρυπτογραφική επαλήθευση της ταυτότητας του ασφαλούς θύλακα. Αντίθετα, δημιουργεί ένα ασφαλές κανάλι επικοινωνίας μέσω ανταλλαγής κλειδιών, θεωρώντας δεδομένη την αξιοπιστία της αρχικής εγκατάστασης του enclave.

Μετά την εγκαθίδρυση της συνεδρίας, το περιβάλλον εκτέλεσης στην πλευρά του εξυπηρετητή, το οποίο εκτελείται μέσα στον ασφαλή θύλακα, μπορεί να εκτελέσει με ασφάλεια τα scripts που αποστέλλει ο πελάτης, μαζί με τις ήδη φορτωμένες εξαρτήσεις τους, και να επιστρέψει τα αποτελέσματα σε κρυπτογραφημένη μορφή.

Ο μηχανισμός αυτός συμπληρώνει τη λειτουργία του Offloading Engine. Συγκεκριμένα, κάθε αίτημα συνοδεύεται από έναν μοναδικό nonce για την αποτροπή επιθέσεων επανάληψης, ενώ οι μόνιμες συνδέσεις επιτρέπουν στον πελάτη να προγραμματίζει αποτελεσματικά πολλαπλές εργασίες σε διαφορετικούς κόμβους, διατηρώντας ασφαλή και σωστά συγχρονισμένη επικοινωνία.

Οι συνδέσεις τερματίζονται μόνο όταν ο πελάτης ολοκληρώσει την εργασία του ή απελευθερώσει ρητά τους δεσμευμένους εξυπηρετητές. Τότε τα κλειδιά της συνεδρίας διαγράφονται και οι αντίστοιχες συνδέσεις κλείνουν.

10.2.4 Προσαρμοσμένος Διερμηνέας QJS

Ο διερμηνευτής έχει επεκταθεί ώστε να ενσωματώνει μηχανισμούς δικτυακής επικοινωνίας, κρυπτογράφησης και διαχείρισης κατανεμημένων εργασιών, επιτρέποντας ασφαλή και αποδοτική αλληλεπίδραση μεταξύ πελατών και εξυπηρετητών.

Ένα ολοκληρωμένο επίπεδο δικτύωσης επιτρέπει στα scripts να δημιουργούν και να διαχειρίζονται συνδέσεις TCP, να εξυπηρετούν εισερχόμενες και εξερχόμενες συνδέσεις και να μεταδίδουν δεδομένα μεταξύ απομακρυσμένων κόμβων. Μεγάλα φορτία δεδομένων αποστέλλονται και λαμβάνονται τμηματικά, ώστε να διασφαλίζεται αξιόπιστη επικοινωνία χωρίς υπέρβαση των ορίων των buffers.

Οι επεκτάσεις κρυπτογράφησης περιλαμβάνουν ασφαλή αποθήκευση κλειδιών, διαχείριση δημόσιων και ιδιωτικών κλειδιών και παραγωγή κλειδιών συμμετρικής κρυπτογράφησης ανά συνεδρία. Το σύστημα υποστηρίζει τόσο συμμετρική όσο και ασύμμετρη κρυπτογράφηση, ενώ η ταυτόχρονη πρόσβαση στην κρυπτογραφική κατάσταση προστατεύεται μέσω mutex, αποτρέποντας συνθήκες ανταγωνισμού (race conditions) κατά την παράλληλη επικοινωνία.

Η υποστήριξη ταυτόχρονης εκτέλεσης επεκτείνεται και στους κοινόχρηστους πόρους του συστήματος, επιτρέποντας την ασφαλή εκτέλεση πολλαπλών ασύγχρονων εργασιών εκφόρτωσης χωρίς να τίθενται σε κίνδυνο η ακεραιότητα ή η ασφάλεια των δεδομένων.

Παράλληλα, τα modules και τα scripts παρακολουθούνται και σειριοποιούνται, επιτρέποντας τη δυναμική εκφόρτωση και απομακρυσμένη εκτέλεσή τους, διατηρώντας ταυτόχρονα συνεπή κατάσταση μεταξύ των κόμβων.

Όλες αυτές οι δυνατότητες είναι πλήρως ενσωματωμένες στο περιβάλλον JavaScript και είναι διαθέσιμες μέσω των τυπικών διεπαφών του διερμηνευτή. Η ασφάλεια αποτελεί βασικό στοιχείο του σχεδιασμού: κάθε κόμβος διαθέτει μοναδικό αναγνωριστικό, όλη η επικοινωνία πραγματοποιείται σε κρυπτογραφημένη μορφή, οι αριθμοί ακολουθίας και οι nonce αποτρέπουν επιθέσεις επανάληψης, ενώ η χρήση mutex διασφαλίζει ορθή σειρά των μηνυμάτων και ασφαλή πρόσβαση στα κοινόχρηστα δεδομένα.

10.3 Στοιχεία της Πλευράς του Διακομιστή

10.3.1 Έμπιστη Εφαρμογή (Trusted App)

Η Έμπιστη Εφαρμογή εκτελείται μέσα στον ασφαλή θύλακα και παρέχει ένα ασφαλές και απομονωμένο περιβάλλον για την εκτέλεση των scripts που αποστέλλονται από τον πελάτη.

Χρησιμοποιεί έναν προσαρμοσμένο και ελαφρύ διερμηνευτή QuickJS, από τον οποίο έχουν αφαιρεθεί πολλές μη απαραίτητες λειτουργίες της JavaScript, ώστε να περιοριστεί το αποτύπωμα μνήμης και να υποστηριχθούν κυρίως οι βασικές λειτουργίες λογικής που συναντώνται στις εφαρμογές ενσωματωμένων συστημάτων.

Τα scripts του πελάτη αποστέλλονται κρυπτογραφημένα και αποκρυπτογραφούνται αποκλειστικά μέσα στον ασφαλή θύλακα χρησιμοποιώντας συμμετρικά κλειδιά AES ανά συνεδρία. Μετά την αποκρυπτογράφηση, ο κώδικας εκτελείται απευθείας από συμβολοσειρά μέσα σε ένα καθολικό περιβάλλον QuickJS (`quickjs_enclave_ctx`), το οποίο διατηρείται μεταξύ διαδοχικών αιτημάτων του ίδιου πελάτη.

Οι εξαρτήσεις που απαιτούνται από τα scripts, όπως προφορτωμένα modules ή βοηθητικές συναρτήσεις, φορτώνονται και αποθηκεύονται στην προστατευμένη μνήμη του enclave μόνο κατά την πρώτη χρήση τους. Οι εξαρτήσεις αυτές διατηρούνται σε καθολική κατάσταση και δεν διαγράφονται μεταξύ διαδοχικών εκτελέσεων, επιτρέποντας την επαναχρησιμοποίησή τους και μειώνοντας σημαντικά το κόστος αρχικοποίησης.

Μόνο όταν ένας πελάτης αποσυνδεθεί, το περιβάλλον εκτέλεσης επαναφέρεται πλήρως, διαγράφοντας τόσο τις αποθηκευμένες εξαρτήσεις όσο και τα κλειδιά της συνεδρίας.

Τα αποτελέσματα της εκτέλεσης συλλέγονται σε έναν buffer απαντήσεων, κρυπτογραφούνται εκ νέου μέσα στον ασφαλή θύλακα και επιστρέφονται στον πελάτη, διατηρώντας την εμπιστευτικότητα από άκρο σε άκρο.

Ο ασφαλής θύλακας εκθέτει ένα περιορισμένο σύνολο διεπαφών ECALL για την αρχικοποίηση, την ανταλλαγή κλειδιών, την εκτέλεση κώδικα και τη διαχείριση συνεδριών, ενώ χρησιμοποιεί OCALL για μη ευαίσθητες λειτουργίες όπως η καταγραφή συμβάντων, η δικτυακή επικοινωνία και οι λειτουργίες εισόδου/εξόδου.

Η σχεδίαση αυτή παρέχει ασφαλή και προσωρινή εκτέλεση με δυνατότητα επαναχρησιμοποίησης της κατάστασης του περιβάλλοντος εκτέλεσης, βελτιώνοντας σημαντικά την απόδοση.

10.3.2 Μη Έμπιστη Εφαρμογή (Untrusted App)

Η Μη Έμπιστη Εφαρμογή λειτουργεί ως ενδιάμεσο επίπεδο μεταξύ των δικτυακών πελατών και του ασφαλούς θύλακα, αναλαμβάνοντας τη διαχείριση της επικοινωνίας και του κύκλου ζωής των συνεδριών.

Τα εισερχόμενα αιτήματα λαμβάνονται αρχικά από το επίπεδο δικτύου και προωθούνται ως ακατέργαστες ακολουθίες bytes προς τον ασφαλή θύλακα, όπου αποκρυπτογραφούνται και επεξεργάζονται με ασφάλεια. Κάθε μήνυμα προηγείται από ένα πεδίο μήκους, επιτρέποντας στην εφαρμογή να γνωρίζει το μέγεθός του πριν το προωθήσει στον θύλακα.

Η εφαρμογή αναλαμβάνει επίσης τη διαδικασία ασφαλούς χειραψίας, ανταλλάσσοντας δημόσια κλειδιά με τους πελάτες και παράγοντας κοινά κλειδιά συνεδρίας για την κρυπτογράφηση της επικοινωνίας.

Παράλληλα, αρχικοποιεί το περιβάλλον JavaScript για κάθε νέο πελάτη, διατηρώντας ωστόσο την απαραίτητη καθολική κατάσταση στη μνήμη ώστε να βελτιστοποιείται η απόδοση.

Όταν ένας πελάτης αποσυνδεθεί, η εφαρμογή εξασφαλίζει ότι όλα τα ευαίσθητα δεδομένα μέσα στον ασφαλή θύλακα διαγράφονται, κλείνει την αντίστοιχη σύνδεση δικτύου και ενημερώνει τον χρονοπρογραμματιστή ώστε να απελευθερωθούν οι δεσμευμένοι πόροι.

Με τον τρόπο αυτό, ακόμη και τα μη έμπιστα τμήματα του συστήματος διαχειρίζονται με ασφάλεια τα ευαίσθητα δεδομένα, ενώ υποστηρίζουν αποτελεσματική και ασφαλή εξυπηρέτηση πολλαπλών πελατών.

10.4 Χρονοπρογραμματιστής

Ο χρονοπρογραμματιστής (Scheduler) είναι υπεύθυνος για την ελεγχόμενη και αποδοτική κατανομή των εξυπηρετητών στους πελάτες.

Διατηρεί μια λίστα όλων των καταχωρημένων εξυπηρετητών, η οποία περιλαμβάνει τις διευθύνσεις δικτύου τους καθώς και την κατάσταση διαθεσιμότητάς τους, και εκχωρεί εξυπηρετητές στους πελάτες με πολιτική κυκλικής εναλλαγής (round-robin), επιτυγχάνοντας ισορροπημένη κατανομή του φόρτου.

Όταν φθάνει ένα νέο αίτημα, ο χρονοπρογραμματιστής διατρέχει τη λίστα ξεκινώντας από τη θέση της τελευταίας ανάθεσης και επιλέγει τον ζητούμενο αριθμό διαθέσιμων εξυπηρετητών. Εάν οι διαθέσιμοι εξυπηρετητές είναι λιγότεροι από όσους ζητήθηκαν, επιστρέφει τον μέγιστο δυνατό αριθμό μαζί με κατάλληλο ενημερωτικό μήνυμα.

Tanto οι πελάτες όσο και οι εξυπηρετητές υποστηρίζουν πρωτόγονα για ασφαλή αντιστοίχιση και δημιουργία συνδέσεων με τους ασφαλείς θύλακες, εξασφαλίζοντας ότι κάθε ανάθεση συμμορφώνεται με τις απαιτήσεις του πρωτοκόλλου.

Επιπλέον, ο χρονοπρογραμματιστής παρακολουθεί όλες τις ενεργές συνεδρίες πελατών και απελευθερώνει τους δεσμευμένους εξυπηρετητές όταν κάποιος πελάτης αποσυνδεθεί, διατηρώντας συνεπή κατάσταση και αποφεύγοντας συγκρούσεις στη χρήση των πόρων.

Η ελαφριά αυτή σχεδίαση παρέχει έναν αξιόπιστο μηχανισμό διαχείρισης της κατανεμημένης εκτέλεσης, διατηρώντας παράλληλα πολύ χαμηλή επιβάρυνση λειτουργίας.

11 Αξιολόγηση

11.1 Σουίτα Δοκιμών Αξιολόγησης (Benchmark Suite)

Το σύνολο δοκιμών που χρησιμοποιείται στην παρούσα εργασία έχει σχεδιαστεί ώστε να αξιολογεί υπολογιστικά φορτία τα οποία είναι αντιπροσωπευτικά εφαρμογών ενσωματωμένων συστημάτων, κινητών συσκευών και ρομποτικής.

Δεν υπάρχουν πλήρως ανοικτού κώδικα έργα που να αντιστοιχούν ακριβώς στα συγκεκριμένα φορτία εργασίας. Αντίθετα, διατίθενται μόνο σύνολα δεδομένων, βιβλιοθήκες ή επιμέρους αλγοριθμικά στοιχεία που αποτυπώνουν ορισμένες από τις αντίστοιχες λειτουργίες. Με βάση αυτούς τους διαθέσιμους πόρους και αντλώντας έμπνευση από πραγματικές εφαρμογές και επιχειρησιακά πρότυπα, κατασκευάσαμε συνθετικά φορτία εργασίας τα οποία αναπαριστούν τα βασικά υπολογιστικά και μνημονικά χαρακτηριστικά των αντίστοιχων πεδίων εφαρμογής.

Παρότι τα benchmarks είναι συνθετικά, βασίζονται σε πραγματικά υπολογιστικά φορτία σύμφωνα με τις ακόλουθες αρχές:

1. **Λειτουργικός ρεαλισμός:** Κάθε benchmark αναπαριστά τα κυρίαρχα πρότυπα υπολογισμού, προσπέλασης μνήμης και παραλληλισμού που συναντώνται σε πραγματικά συστήματα.

2. **Παραμετροποίηση:** Μεταβλητές όπως το μέγεθος παρτίδας (batch size), η ανάλυση του πλέγματος (grid resolution) και η πυκνότητα εμποδίων επιτρέπουν τη μελέτη της απόδοσης υπό ρεαλιστικές συνθήκες λειτουργίας.

3. **Κλιμάκωση υπολογιστικού και μνημονικού φόρτου:** Τα benchmarks έχουν σχεδιαστεί ώστε να αναδεικνύουν εφαρμογές που περιορίζονται από τον επεξεργαστή (CPU-bound), από τη μνήμη (memory-bound), ή από συνδυασμό των δύο, αντανακλώντας τους συμβιβασμούς που εμφανίζονται στις πλατφόρμες ενσωματωμένων συστημάτων και ρομποτικής.

Η μεθοδολογία αυτή επιτρέπει τον ακριβή έλεγχο των πειραμάτων, την επαναληψιμότητα και τη συστηματική αξιολόγηση τεχνικών βελτιστοποίησης ή στρατηγικών εκφόρτωσης υπολογισμού, παραμένοντας ταυτόχρονα πιστή στις συμπεριφορές που παρατηρούνται σε πραγματικά συστήματα, ακόμη και όταν δεν υπάρχουν πλήρεις υλοποιήσεις ανοικτού κώδικα.

Ασφαλής Επεξεργασία Δεδομένων Υγείας με χρήση HMAC

Το benchmark αυτό προσομοιώνει μια φορητή συσκευή υγείας, η οποία συλλέγει συνεχώς ευαίσθητα φυσιολογικά δεδομένα, όπως καρδιακό ρυθμό, επίπεδα SpO_2 και ηλεκτροκαρδιογραφικά (ECG) σήματα υψηλής συχνότητας. Τα δεδομένα συγκεντρώνονται σε παρτίδες σταθερού μεγέθους (10 δευτερόλεπτα στα 500 Hz), με αποτέλεσμα κάθε παρτίδα να περιλαμβάνει περίπου 5.000 δείγματα ECG μαζί με τα αντίστοιχα μεταδεδομένα.

Κάθε παρτίδα σειριοποιείται και πιστοποιείται μέσω του αλγορίθμου HMAC-SHA512, ώστε να διασφαλίζεται η ακεραιότητα και η αυθεντικότητά της πριν από τη μετάδοση ή την αποθήκευσή της. Ο υπολογισμός του HMAC αποτελεί το κυρίαρχο υπολογιστικό κόστος και, συνεπώς, επιλέγεται ως στόχος εκφόρτωσης.

Το συγκεκριμένο φορτίο εργασίας ανήκει στην κατηγορία των *βρόχων επαναληπτικής επεξεργασίας κατά παρτίδες (batch-iterative processing loops)*, όπου τα δεδομένα συγκεντρώνονται αρχικά μέσα σε ένα χρονικό παράθυρο και στη συνέχεια επεξεργάζονται ως ενιαία μονάδα, γεγονός που το καθιστά ιδιαίτερα κατάλληλο για επιλεκτική εκφόρτωση.

Προσαρμοστική Εξομάλυνση Δεδομένων Αισθητήρων

Το benchmark αυτό προσομοιώνει μια εφαρμογή συνεχούς επεξεργασίας δεδομένων αισθητήρων, όπου θορυβώδεις μετρήσεις φιλτράρονται διαρκώς μέσω ενός φίλτρου διαμέσου (median filter) με ολισθαίνον παράθυρο.

Σε κάθε επανάληψη συλλέγεται μια νέα παρτίδα μετρήσεων αισθητήρων, η οποία ενσωματώνεται σε έναν buffer σταθερού μεγέθους (παράθυρο μεγέθους 1024), προσομοιώνοντας πραγματικά σενάρια όπως η περιβαλλοντική παρακολούθηση ή οι βιομηχανικές εφαρμογές αισθητήρων.

Ο βασικός υπολογισμός εφαρμόζει φίλτρο διαμέσου πάνω στο ολισθαίνον παράθυρο μέσω ενός αλγορίθμου ταξινόμησης Gnome Sort, ο οποίος κυριαρχεί στον χρόνο εκτέλεσης λόγω της τετραγωνικής χρονικής πολυπλοκότητάς του. Για τον λόγο αυτό, η συνάρτηση εξομάλυνσης (`processBatch`) αποτελεί φυσικό υποψήφιο για εκφόρτωση.

Το μέγεθος της εισόδου είναι αντιπροσωπευτικό εφαρμογών ενσωματωμένων συστημάτων: τα δεδομένα καταφθάνουν σταδιακά σε μικρές παρτίδες, ενώ το παράθυρο επεξεργασίας παραμένει σταθερού μεγέθους. Παρόμοιες σχεδιάσεις χρησιμοποιούνται ευρέως σε edge συστήματα, ώστε να επιτυγχάνεται ισορροπία μεταξύ γρήγορης απόκρισης και αποτελεσματικής μείωσης του θορύβου υπό περιορισμένους πόρους μνήμης.

Το συγκεκριμένο φορτίο εργασίας εντάσσεται στην κατηγορία των *βρόχων συνεχούς ροής με χαλαρούς χρονικούς περιορισμούς (soft real-time streaming loops)*, όπου συνεχείς ροές δεδομένων επεξεργάζονται με ανεκτικότητα σε μέτρια καθυστέρηση, επιτρέποντας την επιλεκτική εκφόρτωση των πιο απαιτητικών υπολογιστικών σταδίων.

Ελάχιστη Απόσταση Επεξεργασίας για Ανάλυση Ροών Καταγραφών

Το benchmark αυτό προσομοιώνει μια εφαρμογή συνεχούς επεξεργασίας αρχείων καταγραφής (logs), στην οποία ακολουθίες μηνυμάτων από αισθητήρες ή συσκευές παράγονται και συγκρίνονται συνεχώς.

Σε κάθε επανάληψη δημιουργούνται δύο ακολουθίες μήκους 100 χαρακτήρων, στις οποίες εισάγονται μικρές μεταβολές ώστε να προσομοιώνονται ανωμαλίες. Στη συνέχεια υπολογίζεται η ελάχιστη απόσταση επεξεργασίας (Minimum Edit Distance) χρησιμοποιώντας έναν πίνακα δυναμικού προγραμματισμού διαστάσεων $n \times m$.

Ο υπολογισμός αυτός κυριαρχεί στον χρόνο εκτέλεσης λόγω της πολυπλοκότητας $O(n \cdot m)$, ενώ η χρήση μνήμης παραμένει σχετικά μικρή αλλά αυξάνεται αναλογικά με τα μήκη των ακολουθιών.

Το φορτίο εργασίας είναι αντιπροσωπευτικό εφαρμογών αναλυτικής επεξεργασίας σε ενσωματωμένα συστήματα, όπου πραγματοποιείται σταδιακή ανίχνευση ανωμαλιών ή υπολογισμός ομοιότητας πάνω σε συνεχείς ροές κειμενικών δεδομένων.

Δομικά, η εφαρμογή ακολουθεί το πρότυπο ενός *βρόχου συνεχούς ροής με χαλαρούς χρονικούς περιορισμούς*, καθώς οι ακολουθίες επεξεργάζονται συνεχώς με μέτρια ανοχή στην καθυστέρηση, επιτρέποντας την επιλεκτική εκφόρτωση του αλγορίθμου δυναμικού προγραμματισμού σε ασφαλείς θύλακες χωρίς να επηρεάζεται η συνολική απόκριση του συστήματος.

Πρόβλεψη Θερμοκρασίας με Δένδρα Αποφάσεων

Το benchmark αυτό αναπαριστά μια εφαρμογή αναλυτικής επεξεργασίας σε ενσωματωμένο σύστημα, η οποία προβλέπει μελλοντικές τιμές θερμοκρασίας με βάση ιστορικές μετρήσεις αισθητήρων.

Σε κάθε εποχή εκτέλεσης παράγεται μία παρτίδα 1.024 δειγμάτων χρονοσειρών και εκπαιδεύεται ένας παλινδρομητής δένδρου αποφάσεων (decision tree regressor) μέγιστου βάθους 5 πάνω στα δεδομένα αυτά. Στη συνέχεια πραγματοποιούνται προβλέψεις για τα επόμενα 1.024 χρονικά βήματα.

Το κυρίαρχο υπολογιστικό κόστος προέρχεται από την εκπαίδευση του δένδρου, η οποία κλιμακώνεται τόσο ως προς τον αριθμό των δειγμάτων όσο και ως προς το βάθος του δένδρου, δημιουργώντας σημαντικό σημείο συμφόρησης σε συσκευές με περιορισμένους υπολογιστικούς πόρους.

Το συγκεκριμένο φορτίο εργασίας είναι αντιπροσωπευτικό εφαρμογών περιβαλλοντικής ή βιομηχανικής παρακολούθησης, όπου ιστορικές μετρήσεις συγκεντρώνονται πριν από την εφαρμογή προγνωστικών αλγορίθμων.

Λόγω της δομής «συγκέντρωση και στη συνέχεια επεξεργασία», η εφαρμογή αυτή ανήκει στην κατηγορία των *βρόχων επαναληπτικής επεξεργασίας κατά παρτίδες*, γεγονός που την καθιστά ιδιαίτερα κατάλληλη για επιλεκτική εκφόρτωση σε ασφαλείς θύλακες, διατηρώντας παράλληλα χαμηλή καθυστέρηση στην επεξεργασία των εισερχόμενων δεδομένων αισθητήρων.

Ομαδοποίηση K-Means για Συσταδοποίηση Δεδομένων Αισθητήρων

Το benchmark αυτό προσομοιώνει μια εφαρμογή αναλυτικής επεξεργασίας σε ενσωματωμένο σύστημα, η οποία ομαδοποιεί πολυδιάστατα δεδομένα αισθητήρων σε παρτίδες των 512 σημείων, όπου κάθε σημείο περιλαμβάνει 3 χαρακτηριστικά.

Σε κάθε εποχή εκτέλεσης πραγματοποιείται ομαδοποίηση K-Means με $K = 20$ κέντρα συστάδων (centroids) και έως 50 επαναλήψεις, όπου το μεγαλύτερο μέρος του υπολογιστικού κόστους οφείλεται στους επαναλαμβανόμενους υπολογισμούς ενημέρωσης των κέντρων.

Καθώς η τιμή του K αυξάνεται, η εφαρμογή μεταβαίνει από περιοριζόμενη από τη μνήμη (memory-bound) σε περιοριζόμενη από τον επεξεργαστή (compute-bound). Συγκεκριμένα, η αύξηση του αριθμού των κέντρων απαιτεί την αποθήκευση μεγαλύτερων πινάκων centroids και πινάκων αποστάσεων, αυξάνοντας το αποτύπωμα μνήμης, ενώ κάθε επανάληψη απαιτεί περισσότερους υπολογισμούς αποστάσεων και ενημερώσεων των κέντρων, αυξάνοντας αντίστοιχα το υπολογιστικό φορτίο της CPU.

Ένα τέτοιο σενάριο είναι ιδιαίτερα ρεαλιστικό σε εφαρμογές βιομηχανικού IoT, περιβαλλοντικής παρακολούθησης ή δικτύων αισθητήρων έξυπνων πόλεων, όπου απαιτείται μεγάλος αριθμός συστάδων για τη διάκριση λεπτομερών προτύπων ή την ταυτόχρονη αναγνώριση πολλαπλών φαινομένων.

Δομικά, το φορτίο εργασίας ακολουθεί το πρότυπο ενός *βρόχου επαναληπτικής επεξεργασίας κατά παρτίδες*, όπου τα δεδομένα συγκεντρώνονται αρχικά σε μία παρτίδα και στη συνέχεια επεξεργάζονται συλλογικά. Η ανεξαρτησία των επιμέρους επαναλήψεων καθιστά τον αλγόριθμο κατάλληλο για επιλεκτική εκφόρτωση σε ασφαλείς θύλακες, επιτρέποντας στις edge συσκευές να εκτελούν μεγάλης κλίμακας εργασίες ομαδοποίησης χωρίς να παραβιάζονται οι περιορισμοί μνήμης ή καθυστέρησης.

Επεξεργασία Παρτίδων Φωνητικών Εντολών για Επεξεργασία Φυσικής Γλώσσας (NLP)

Το benchmark αυτό προσομοιώνει μια ενσωματωμένη συσκευή που συλλέγει συνεχώς φωνητικές εντολές κατά τη διάρκεια της ημέρας και τις επεξεργάζεται ομαδικά σε περιόδους αδράνειας. Κάθε παρτίδα περιλαμβάνει $B = 50$ εντολές, οι οποίες αναλύονται με στόχο την εξαγωγή της πρόθεσης του χρήστη (intent), την ανίχνευση εντολών εκτός του υποστηριζόμενου πεδίου εφαρμογής και τη βελτίωση των στατιστικών χρήσης.

Ο βασικός υπολογισμός αποτελείται από τμηματοποίηση του κειμένου (tokenization), κανονικοποίηση ρημάτων και ουσιαστικών, καθώς και ταξινόμηση της πρόθεσης μέσω ενός ελαφρού αγωγού επεξεργασίας φυσικής γλώσσας (NLP). Η πολυπλοκότητα της εκτέλεσης εξαρτάται κυρίως από το μέγεθος της παρτίδας και τον αριθμό των γλωσσικών χαρακτηριστικών που εξάγονται από κάθε εντολή.

Παρότι οι απαιτήσεις σε μνήμη παραμένουν σχετικά μικρές λόγω του μικρού μήκους των εντολών, η χρήση της CPU αυξάνεται γραμμικά ως προς το μέγεθος της παρτίδας και το μήκος του κειμένου, με αποτέλεσμα μεγαλύτερες παρτίδες να μετατρέπουν την εφαρμογή σε compute-bound.

Το σενάριο αυτό είναι χαρακτηριστικό συσκευών έξυπνου σπιτιού, φορετών συσκευών και ψηφιακών βοηθών φωνής, όπου οι φωνητικές εντολές συσσωρεύονται και αναλύονται εκτός πραγματικού χρόνου, χωρίς να επηρεάζεται η απόκριση της συσκευής κατά τη λειτουργία της.

Δομικά, το φορτίο εργασίας ακολουθεί το πρότυπο ενός *βρόχου επαναληπτικής επεξεργασίας κατά παρτίδες*, όπου οι εντολές συλλέγονται μέσα σε ένα χρονικό παράθυρο και επεξεργάζονται συλλογικά. Η ανεξαρτησία των επιμέρους εντολών καθιστά τον υπολογισμό ιδιαίτερα κατάλληλο για επιλεκτική εκφόρτωση σε ασφαλείς θύλακες, διατηρώντας την ιδιωτικότητα των χρηστών και επιτρέποντας ταυτόχρονα επεξεργασία υψηλού ρυθμού.

Σχεδιασμός Διαδρομών NavFn για Πλοήγηση σε Πλέγματα

Το benchmark αυτό προσομοιώνει ένα σύστημα σχεδιασμού διαδρομών πάνω σε πλέγμα, εμπνευσμένο από το σύστημα πλοήγησης του ROS, όπου πολλαπλά ζεύγη αρχικής και τελικής θέσης επεξεργάζονται προκειμένου να υπολογιστούν οι βέλτιστες διαδρομές.

Κάθε σενάριο αποτελείται από ένα ζεύγος συντεταγμένων που αντιστοιχεί στην αρχική και την τελική θέση. Η ανάλυση πραγματοποιείται μέσω ενός αλγορίθμου Dijkstra βασισμένου σε δυναμικά πεδία δυναμικού (potential fields), ο οποίος υπολογίζει τις διαδρομές και εκτιμά το αντίστοιχο κόστος μετακίνησης.

Ο βασικός υπολογισμός περιλαμβάνει τη δημιουργία του costmap, τη διάδοση των δυναμικών τιμών (potential propagation), τον υπολογισμό των βαθμίδων (gradient calculation) και την εξαγωγή της τελικής διαδρομής. Η πολυπλοκότητα εξαρτάται κυρίως από το μέγεθος του πλέγματος και την πυκνότητα των εμποδίων.

Παρότι οι απαιτήσεις μνήμης αυξάνονται γραμμικά με τον αριθμό των κελιών του πλέγματος, η χρήση της CPU κυριαρχείται από τις επαναλαμβανόμενες διαδικασίες διάδοσης των δυναμικών τιμών και υπολογισμού των βαθμίδων, γεγονός που καθιστά πυκνούς χάρτες ή μεγάλες διαδρομές υπολογιστικά απαιτητικούς.

Το σενάριο αυτό είναι ιδιαίτερα αντιπροσωπευτικό εφαρμογών αυτόνομων ρομπότ, μη επανδρωμένων εναέριων οχημάτων (UAVs) και κινητών ρομποτικών πλατφορμών, όπου ο σχεδιασμός διαδρομής σε πραγματικό χρόνο πρέπει να εξισορροπεί την ακρίβεια με την αποδοτικότητα.

Δομικά, το φορτίο εργασίας ακολουθεί ένα πρότυπο *βρόχου συνεχούς ροής με χαλαρούς χρονικούς περιορισμούς*, το οποίο περιλαμβάνει την αρχικοποίηση των στόχων, τη διάδοση των δυναμικών τιμών και τον υπολογισμό των βαθμίδων για την εξαγωγή της διαδρομής. Η ανεξαρτησία των κελιών

του πλέγματος κατά τη φάση ενημέρωσης των δυναμικών τιμών καθιστά τον αλγόριθμο κατάλληλο για επιλεκτικό παραλληλισμό, επιτρέποντας υψηλή απόδοση χωρίς να διακυβεύεται η ντετερμινιστική συμπεριφορά του σχεδιασμού διαδρομής.

Υπολογισμός Συντομότερων Διαδρομών με τον Αλγόριθμο Dijkstra

Το benchmark αυτό προσομοιώνει μια εφαρμογή βελτιστοποίησης διαδρομών σε ενσωματωμένο σύστημα, όπως ένα αυτόνομο drone διανομών ή ένα όχημα αυτόματης μεταφοράς σε αποθήκη, όπου πολλαπλά ζεύγη σημείων προέλευσης και προορισμού πρέπει να υπολογίζονται αποδοτικά σε υλικό περιορισμένων πόρων.

Το φορτίο εργασίας κατασκευάζει έναν σταθμισμένο γράφο με $V = 10{,}000$ κορυφές και ένα ντετερμινιστικό σύνολο ακμών με μεταβλητά βάρη. Στη συνέχεια υπολογίζει τις αποστάσεις συντομότερων διαδρομών από έναν δεδομένο κόμβο εκκίνησης χρησιμοποιώντας τον αλγόριθμο Dijkstra.

Σε κάθε επανάληψη ο αλγόριθμος διατρέχει τον γράφο, ενημερώνει τις προσωρινές αποστάσεις των μη επισκέψιμων κορυφών και διαδίδει το κόστος των διαδρομών μέσω των λιστών γειτνίασης. Το υπολογιστικό κόστος κυριαρχείται από τις επαναλαμβανόμενες επιλογές της κορυφής με την ελάχιστη απόσταση και τις πράξεις χαλάρωσης (edge relaxation), ενώ οι απαιτήσεις μνήμης παραμένουν σχετικά περιορισμένες και αφορούν κυρίως τους πίνακες αποστάσεων και επισκέψεων.

Δομικά, το φορτίο εργασίας αυτό αντιπροσωπεύει ένα πρότυπο *βρόχου συνεχούς ροής με χαλαρούς χρονικούς περιορισμούς*: αρχικά κατασκευάζεται ο γράφος, στη συνέχεια οι αποστάσεις υπολογίζονται επαναληπτικά για όλες τις κορυφές και τέλος τα αποτελέσματα συλλέγονται μετά από πολλαπλές επαναλήψεις.

Η συγκεκριμένη δομή επιτρέπει την επιλεκτική εκφόρτωση του υπολογιστικά απαιτητικού αλγορίθμου Dijkstra σε ασφαλείς θύλακες, επιτρέποντας σε ενσωματωμένες συσκευές να εκτελούν βελτιστοποίηση διαδρομών μεγάλης κλίμακας χωρίς να υπερβαίνουν τους περιορισμούς μνήμης ή επεξεργαστικής ισχύος.

Σε αντίθεση με το benchmark πλοήγησης ROS Navigation, το οποίο επικεντρώνεται στη διάδοση δυναμικών τιμών πάνω σε πλέγματα, το συγκεκριμένο σενάριο δίνει έμφαση στη διάσχιση γράφων και στη συνδυαστική βελτιστοποίηση πάνω σε αραιά ή δομημένα δίκτυα, αντανακλώντας χαρακτηριστικά φορτία εργασίας σε αυτόνομα συστήματα εφοδιαστικής, σχεδιασμό διαδρομών και ανάλυση δικτύων αισθητήρων.

Benchmark	Μέγεθος Εισόδου	Bytes Μονάδων	Bytes Φορτίου	Χρόνος	Χώρος	Πρότυπο
HMAC	5000 samples	131259.80 ± 4304.65	98642.43 ± 3272.57	O(n)	O(n)	Batch-iterative
GnomeSort	1024 window	9625.50 ± 6.02	8216.56 ± 7.51	$O(n^2)$	O(1024)	Soft real-time streaming
Edit Distance	100x100 seq	5556.40 ± 3.24	3998.95 ± 4.42	O(n·m)	O(n·m)	Soft real-time streaming
Decision Tree	1024 samples	621192.00 ± 518.49	33853.49 ± 955.50	O(n·d)	O(1024)	Batch-iterative
K-Means	512x3 pts	619917.20 ± 18.38	31247.91 ± 19.62	O(k·i·n·d)	O(n·k)	Batch-iterative
NLP	50 commands	423987.50 ± 396.57	4214.82 ± 417.21	O(batch·feat)	O(batch)	Batch-iterative
ROS Navigation	10x10 grid	18683.10 ± 0.30	213.08 ± 0.81	$O(n^2)$	$O(n^2)$	Batch-iterative
Dijkstra	10000 vertices	2458.00 ± 0.00	204.26 ± 0.55	$O(V^2 + E)$	O(V)	Soft real-time streaming

Table 1: Σύνοψη των φορτίων εργασίας των benchmarks, συμπεριλαμβανομένων των μεγεθών εισόδου, των υπολογιστικών και μνημονικών χαρακτηριστικών τους, καθώς και του δομικού τους προτύπου ως προς την επιλεκτική εκφόρτωση.

11.2 Πειραματική Διάταξη

Όλα τα πειράματα πραγματοποιήθηκαν σε ελεγχόμενο περιβάλλον ιδιωτικού δικτύου, ώστε να εξασφαλιστούν σταθερές και αναπαραγώγιμες μετρήσεις απόδοσης. Το σύστημα διαμορφώθηκε ώστε να χρησιμοποιεί το Intel SGX σε λειτουργία υλικού (`SGX_MODE=HW`), επιτρέποντας την εκτέλεση εφαρμογών μέσα σε πραγματικά Έμπιστα Περιβάλλοντα Εκτέλεσης (Trusted Execution Environments). Τα πειράματα πραγματοποιήθηκαν σε σύστημα με λειτουργικό **Debian GNU/Linux 13 (trixie)** και έκδοση **2.25.100.3** του Intel SGX SDK.

Η υποδομή στην πλευρά του εξυπηρετητή αποτελείται από ένα πολυεπεξεργαστικό σύστημα εξοπλισμένο με επεξεργαστή Intel Core i7-10710U. Ο επεξεργαστής διαθέτει 6 φυσικούς πυρήνες και 12 λογικά νήματα εκτέλεσης, με μέγιστη συχνότητα λειτουργίας 4.7 GHz και κοινόχρηστη μνήμη cache επιπέδου L3 μεγέθους 12 MB. Το σύστημα διαθέτει συνολικά 64 GB κύριας μνήμης.

Η υποστήριξη του Intel SGX επιβεβαιώνεται από την παρουσία της σημαίας `sgx` στα χαρακτηριστικά του επεξεργαστή. Το σύστημα εκθέτει τις διεπαφές συσκευών SGX (`/dev/sgx_provision`, `/dev/sgx_vepc`), επιτρέποντας την εκτέλεση θυλάκων (enclaves) σε λειτουργία υλικού. Η πλατφόρμα υποστηρίζει SGX1, ενώ το διαθέσιμο μέγεθος της *Enclave Page Cache (EPC)* ανέρχεται περίπου στα 94 MB, όπως προσδιορίστηκε μέσω απαρίθμησης βασισμένης στην εντολή CPUID. Η τιμή αυτή είναι ελαφρώς μικρότερη από το θεωρητικό μέγιστο των 128 MB για τη συγκεκριμένη γενιά επεξεργαστών, λόγω δεσμεύσεων μνήμης από το λειτουργικό σύστημα.

Στην πλευρά του εξυπηρετητή εκτελούνται πολλαπλά στιγμιότυπα εργαζομένων (worker instances) που υποστηρίζονται από θύλακες SGX. Στη συγκεκριμένη διαμόρφωση εκκινούνται τέσσερις εξυπηρετητές, οι οποίοι ακούν στις θύρες 9000 έως 9003. Παράλληλα, ένας κεντρικός χρονοδρομολογητής (scheduler) εκτελείται στον ίδιο υπολογιστή στη θύρα 8000 και αναθέτει διαθέσιμους θύλακες στους πελάτες ακολουθώντας πολιτική round-robin.

Η *Enclave Page Cache (EPC)* αποτελεί έναν καθολικά κοινόχρηστο πόρο υλικού για όλους τους θύλακες του συστήματος. Ως αποτέλεσμα, πολλαπλά στιγμιότυπα enclaves που εκτελούνται ταυτόχρονα ανταγωνίζονται για την ίδια περιοχή μνήμης. Στη δική μας διαμόρφωση, όπου έως και τέσσερις enclave-backed εξυπηρετητές λειτουργούν ταυτόχρονα, η πραγματική χωρητικότητα EPC που αντιστοιχεί σε κάθε θύλακα μειώνεται, αυξάνοντας την πίεση στη μνήμη. Αυτό μπορεί να οδηγήσει σε *EPC paging*, κατά το οποίο σελίδες μνήμης του θύλακα μεταφέρονται προσωρινά σε μη έμπιστη κύρια μνήμη, εισάγοντας επιπλέον επιβάρυνση στην απόδοση.

11.2.1 Συνθετικό Μοντέλο Καθυστέρησης WAN

Όλα τα πειράματα εκτελέστηκαν σε ένα ελεγχόμενο ιδιωτικό εργαστηριακό δίκτυο, ώστε να εξασφαλιστούν σταθερές και αναπαραγώγιμες μετρήσεις. Κατά συνέπεια, η μετρούμενη καθυστέρηση δικτύου είναι σημαντικά μικρότερη από εκείνη που παρατηρείται σε τυπικά περιβάλλοντα ανάπτυξης εφαρμογών στο υπολογιστικό νέφος (cloud). Για την εκτίμηση της απόδοσης του *Aégis* σε ένα ρεαλιστικό σενάριο edge-to-cloud, παρουσιάζονται επιπλέον αποτελέσματα βασισμένα σε ένα συνθετικό μοντέλο καθυστέρησης δικτύου ευρείας περιοχής (WAN).

Η συνολική μετρούμενη καθυστέρηση από άκρο σε άκρο εκφράζεται ως

$$L_{\text{lab}} = T_{\text{network,lab}} + T_{\text{server}} + T_{\text{enclave}}. \tag{1}$$

Όπου το $T_{\text{network,lab}}$ αντιστοιχεί στον μετρούμενο χρόνο μετάδοσης μέσω του εργαστηριακού δικτύου (συμπεριλαμβανομένων των επιβαρύνσεων επικοινωνίας και μεταφοράς), το T_{server} αναπαριστά τον συνολικό χρόνο επεξεργασίας στην πλευρά του εξυπηρετητή, ενώ το T_{enclave} αντιστοιχεί στον χρόνο εκτέλεσης του κώδικα εντός του SGX enclave.

Προκειμένου να προσεγγιστεί η λειτουργία του συστήματος μέσω του δημόσιου Διαδικτύου, η μετρούμενη καθυστέρηση του εργαστηριακού δικτύου αντικαθίσταται από μία αντιπροσωπευτική καθυστέρηση WAN, διατηρώντας αμετάβλητους τους πειραματικά μετρούμενους χρόνους εκτέλεσης του εξυπηρετητή και του enclave. Η προκύπτουσα συνολική καθυστέρηση υπολογίζεται ως

$$L_{\text{WAN}} = L_{\text{lab}} - T_{\text{network,lab}} + T_{\text{network,WAN}}, \tag{2}$$

ή ισοδύναμα,

$$L_{\text{WAN}} = L_{\text{lab}} + \Delta, \tag{3}$$

όπου

$$\Delta = T_{\text{network,WAN}} - T_{\text{network,lab}}. \tag{4}$$

Εκτός εάν αναφέρεται διαφορετικά, η τιμή του $T_{\text{network,WAN}}$ επιλέγεται από αντιπροσωπευτικές τιμές καθυστέρησης μετάβασης και επιστροφής (Round-Trip Time, RTT) μεταξύ πελάτη και υποδομών δημόσιου υπολογιστικού νέφους, οι οποίες κυμαίνονται μεταξύ 10 και 50 ms για γεωγραφικά κοντινές περιοχές (regions). Το μοντέλο αυτό παρέχει μία ανάλυση ευαισθησίας της απόδοσης του *Aégis* υπό ρεαλιστικές συνθήκες ανάπτυξης μέσω του Διαδικτύου, διατηρώντας παράλληλα τα πειραματικά μετρούμενα χαρακτηριστικά εκτέλεσης του εξυπηρετητή και του SGX enclave.

11.3 Αποτελέσματα

Με τα benchmarks πλήρως περιγεγραμμένα και τα υπολογιστικά τους χαρακτηριστικά κατανοητά, προχωρούμε τώρα στην αξιολόγηση της απόδοσής τους υπό διάφορες διαμορφώσεις ανάπτυξης. Η ανάλυσή μας συγκρίνει την εγγενή εκτέλεση με multi-server setups, αναδεικνύοντας χαρακτηριστικά χρόνου εκτέλεσης, throughput και latency.

Συγκεκριμένα, η αξιολόγηση του *Aégis* απαντά σε δύο κύρια ερευνητικά ερωτήματα και στόχους σχεδίασης:

- **RQ1 (Αποδοτικότητα Απόδοσης)**: εξετάζει σε ποιο βαθμό το *Aégis* μπορεί να βελτιώσει την απόδοση ενσωματωμένων εφαρμογών μέσω ασφαλούς επιλεκτικής εκφόρτωσης (offloading) σε Trusted Execution Environments, εστιάζοντας σε βασικές μετρικές όπως χρόνος εκτέλεσης, throughput και latency σε σύγκριση με την εγγενή, τοπική εκτέλεση.

- **RQ2 (Κλιμάκωση)**: διερευνά πώς η απόδοση του συστήματος κλιμακώνεται με τον αριθμό των διαθέσιμων enclave nodes σε διαφορετικούς τύπους workloads.

- **RQ3 (Ανάλυση Overhead και Latency)**: μετρά και αναλύει τη συμβολή διαφορετικών συνιστωσών στον συνολικό χρόνο εκτέλεσης, συμπεριλαμβανομένων (i) της μεταφοράς μέσω δικτύου (συμπεριλαμβανομένων των overhead από κρυπτογράφηση/αποκρυπτογράφηση) και (ii) της κλήσης του enclave (συμπεριλαμβανομένων των encryption/decryption και ασφαλούς επεξεργασίας), ώστε να κατανοηθεί πώς κάθε παράγοντας επηρεάζει τη συνολική αποδοτικότητα του secure offloading.

- **(Στόχος Σχεδίασης) Προγραμματισιμότητα και Διαφάνεια**: παρέχει μια ποιοτική σύγκριση μεταξύ του παραδοσιακού SGX workflow και του προγραμματιστικού μοντέλου του *Aégis* , αναδεικνύοντας πώς το *Aégis* αφαιρεί την πολυπλοκότητα διαχείρισης enclave interfaces.

- **(Στόχος Σχεδίασης) Διατήρηση Ασφάλειας**: το *Aégis* έχει σχεδιαστεί ώστε να διατηρεί τις εγγυήσεις ασφάλειας του SGX κατά την απομακρυσμένη εκτέλεση. Ο ευαίσθητος κώδικας και τα δεδομένα παραμένουν προστατευμένα μέσα στα enclaves, ενώ η επικοινωνία μεταξύ edge και cloud components πιστοποιείται και κρυπτογραφείται ώστε να διασφαλίζεται η εμπιστευτικότητα και η ακεραιότητα.

11.3.1 RQ1: Αποδοτικότητα Απόδοσης

Τα κέρδη απόδοσης που παρατηρούνται στα Σχήματα 5 και 6 οφείλονται κυρίως στον υπολογιστικό πλεονέκτημα του server σε σχέση με τους περιορισμένους πόρους της embedded συσκευής. Αυτά τα κέρδη περιορίζονται φυσικά από τα overhead της μεταφοράς μέσω δικτύου, συμπεριλαμβανομένων μηχανισμών κρυπτογράφησης/αποκρυπτογράφησης, attestation του enclave και μεταβάσεων μεταξύ untrusted και trusted χώρου (π.χ. system calls ή context switches).

Η μέση καθυστέρηση εκτέλεσης στο native περιβάλλον κυμαίνεται από 13.8 ms (editDistance) έως 19.9 s (NLP), αναδεικνύοντας τη μεγάλη ετερογένεια στην υπολογιστική πολυπλοκότητα μεταξύ των εξεταζόμενων benchmarks. Υπό συνθήκες ασφαλούς απομακρυσμένης εκτέλεσης, η μέση καθυστέρηση ανά επανάληψη κυμαίνεται από 0.35 ms (Dijkstra) έως 12.9 ms (KMeans), αντικατοπτρίζοντας τη σημαντικά υψηλότερη υπολογιστική ισχύ του απομακρυσμένου server.

Η απόδοση (throughput) στο native περιβάλλον κυμαίνεται από 0.0001 έως 0.0726 iterations/ms, ενώ η ασφαλής απομακρυσμένη εκτέλεση επιτυγχάνει τιμές μεταξύ 0.312 και 2.627 iterations/ms, γεγονός που αντιστοιχεί σε σημαντική αύξηση της ικανότητας επεξεργασίας.

Αξιοσημείωτα, η πρώτη επανάληψη κάθε offloaded task είναι σημαντικά πιο αργή, καθώς περιλαμβάνει τη μεταφορά των dependency modules στο enclave επιπλέον του ίδιου του workload. Οι επόμενες εκτελέσεις επωφελούνται από cached modules μέσα στο enclave, με αποτέλεσμα ταχύτερους χρόνους εκτέλεσης. Οι αρχικές αυτές καθυστερήσεις περιλαμβάνονται στον υπολογισμό του μέσου όρου, ωστόσο δεν επηρεάζουν σημαντικά τη συνολική αποδοτικότητα, καθώς η απόδοση των επόμενων iterations κυριαρχεί.

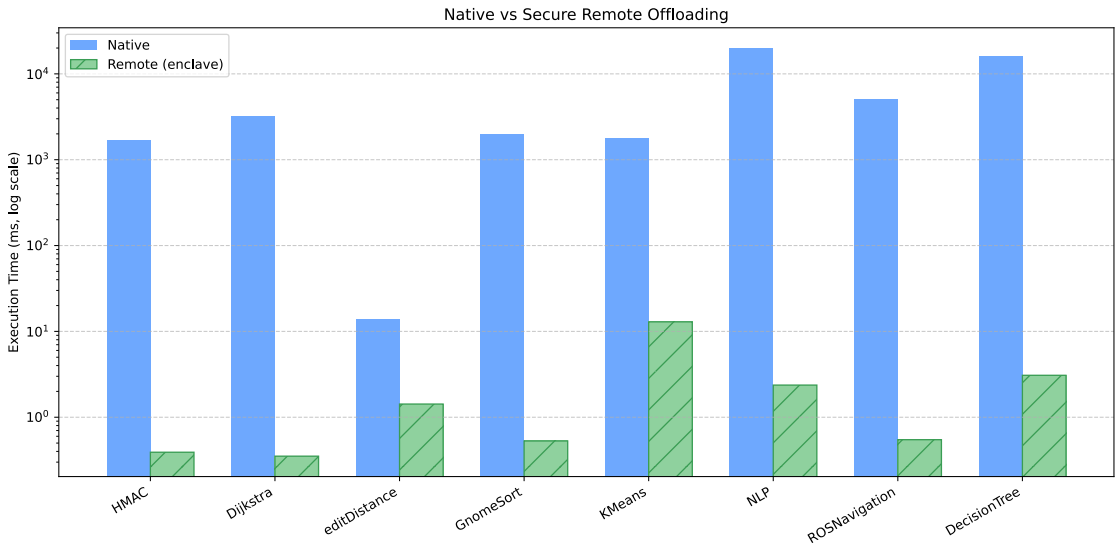

Figure 5: Σύγκριση μέσης καθυστέρησης ανά iteration μεταξύ εγγενούς εκτέλεσης και ασφαλούς offloading σε έναν server.

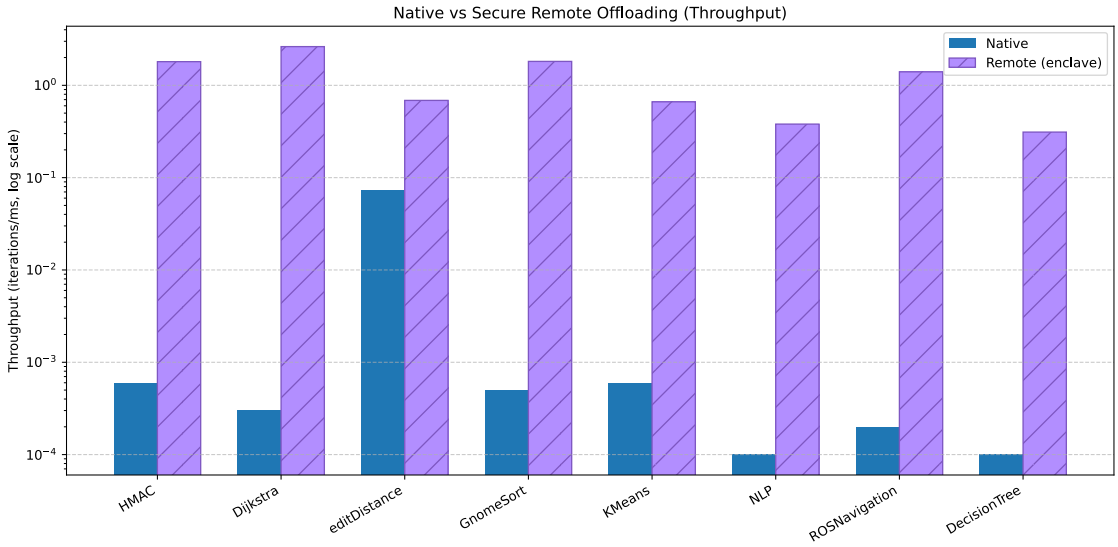

Figure 6: Σύγκριση throughput μεταξύ εγγενούς εκτέλεσης και offloaded εκτέλεσης σε έναν server.

Για την αξιολόγηση της πρακτικότητας του *Aégis* σε ρεαλιστικά περιβάλλοντα ανάπτυξης, επαναλαμβάνουμε την ανάλυση καθυστέρησης χρησιμοποιώντας το συνθετικό μοντέλο καθυστέρησης WAN που περιγράφεται παραπάνω. Εκτός αν αναφέρεται διαφορετικά, υποθέτουμε καθυστέρηση δικτύου round-trip ίση με 30 ms, η οποία είναι αντιπροσωπευτική της επικοινωνίας μεταξύ edge συσκευών και γεωγραφικά κοντινών περιοχών δημόσιου cloud. Η τιμή αυτή βρίσκεται εντός του ευρέως αναφερόμενου εύρους 10–50 ms RTT που παρατηρείται σε αναπτύξεις edge-to-cloud και συνεπώς αντιπροσωπεύει ένα ρεαλιστικό σημείο λειτουργίας, αποφεύγοντας ταυτόχρονα υποθέσεις που εξαρτώνται από συγκεκριμένο πάροχο cloud ή γεωγραφική τοποθεσία.

11.3.2 RQ2: Κλιμάκωση

Το loop unrolling και η ταυτόχρονη εκτέλεση επόμενων iterations με μορφή task queue αυξάνουν περαιτέρω το throughput. Ενώ η προσθήκη περισσότερων servers γενικά βελτιώνει τον παραλληλισμό, εισάγει επίσης overhead από επιπλέον handshakes μεταξύ client και κάθε enclave, καθώς και από τη μεταφορά dependency modules στις πρώτες εκτελέσεις. Αυτοί οι παράγοντες αυξάνουν την αρχική καθυστέρηση ανά thread, με αποτέλεσμα η πρώτη επανάληψη να είναι πιο αργή όταν χρησιμοποιούνται περισσότεροι servers. Παρά το αρχικό αυτό κόστος, το όφελος του υψηλού data parallelism κυριαρχεί, οδηγώντας σε σημαντικά κέρδη απόδοσης σε multi-server deployments.

Το συνολικό speedup, που φαίνεται στο Σχήμα 8, υπολογίζεται ως ο λόγος του συνολικού χρόνου εκτέλεσης του offloaded loop—μετρούμενου ως το μέγιστο των start times κάθε request συν το latency του—προς τον χρόνο εκτέλεσης της εγγενούς υλοποίησης. Παρόλο που το per-iteration speedup–Σχήμα 9–μειώνεται ελαφρώς καθώς αυξάνεται ο αριθμός των servers λόγω intra-machine contention από πολλαπλά enclaves, το συνολικό throughput–Σχήμα 10–και η αποδοτικότητα βελτιώνονται σημαντικά με την αύξηση του παραλληλισμού.

Η επιτευχθείσα επιτάχυνση (speedup) μεταβάλλεται σημαντικά μεταξύ των workloads, με τις μεγαλύτερες βελτιώσεις να παρατηρούνται σε compute-intensive εφαρμογές που επωφελούνται περισσότερο από το offloading στο cloud. Η συμπεριφορά αυτή είναι αναμενόμενη, καθώς η βασική πλατφόρμα είναι μια περιορισμένων πόρων embedded συσκευή, ενώ η εκτέλεση μεταφέρεται σε έναν σημαντικά ισχυρότερο cloud server.

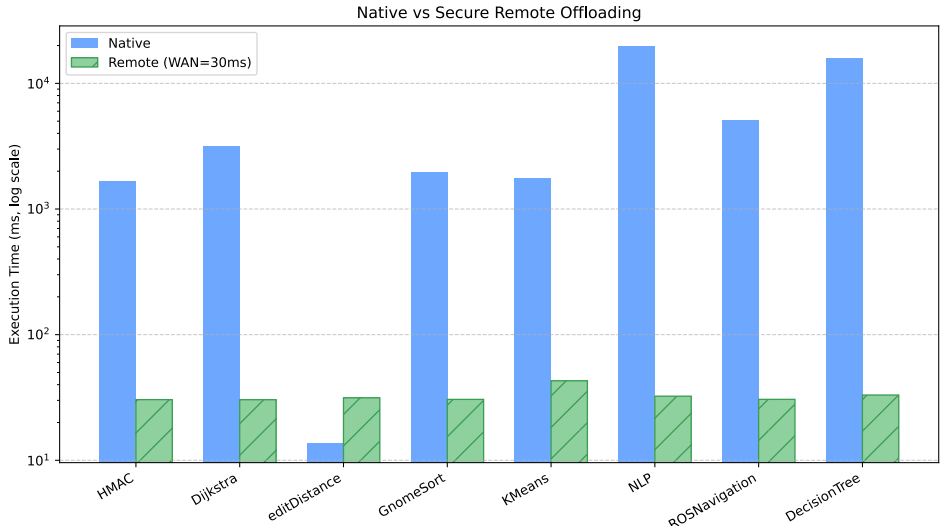

Figure 7: Μέση καθυστέρηση ανά επανάληψη υπό το συνθετικό μοντέλο WAN των 30 ms σε σύγκριση με την εγγενή εκτέλεση (native) και τις μετρήσεις σε εργαστηριακό περιβάλλον. Η επιπλέον καθυστέρηση δικτύου αυξάνει ομοιόμορφα την end-to-end καθυστέρηση, διατηρώντας παράλληλα τις σχετικές τάσεις απόδοσης μεταξύ των benchmarks.

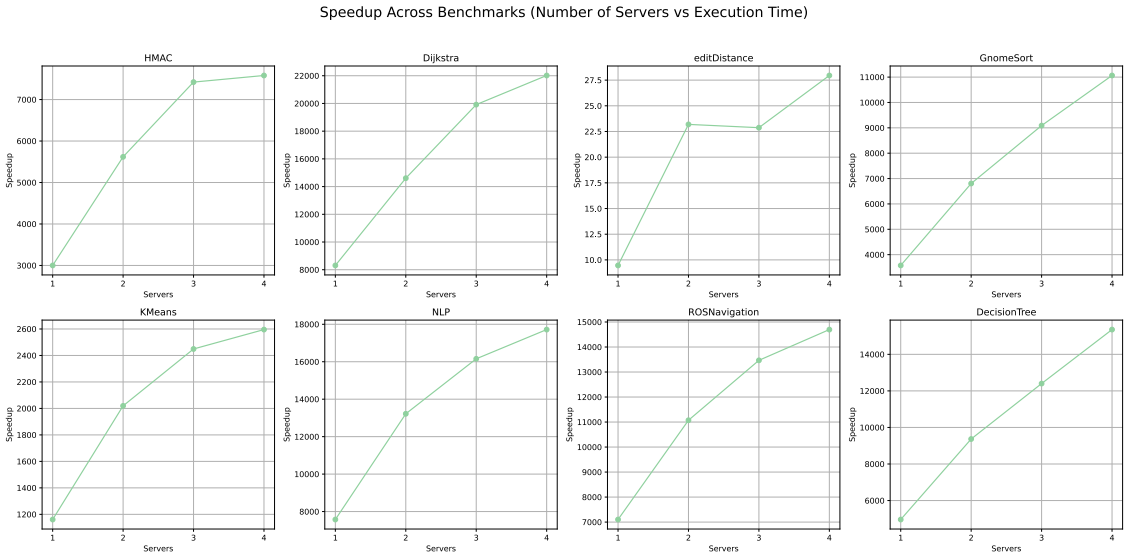

Figure 8: Συνολικό speedup σε multi-server deployments σε σχέση με την εγγενή εκτέλεση.

Οι τάσεις της κλιμάκωσης παραμένουν ποιοτικά αμετάβλητες υπό το συνθετικό μοντέλο καθυστέρησης WAN, καθώς η πρόσθετη καθυστέρηση δικτύου επηρεάζει όλες τις διαμορφώσεις με τον ίδιο τρόπο. Συνεπώς, για την ανάλυση της κλιμάκωσης παρουσιάζονται μόνο οι μετρήσεις του εργαστηριακού περιβάλλοντος.

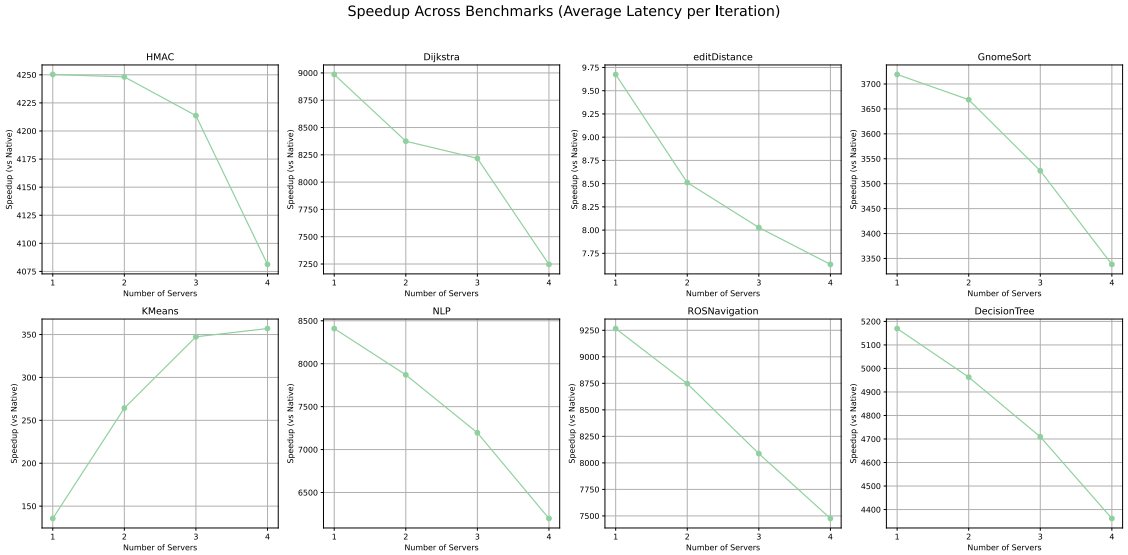

Figure 9: Per-iteration speedup καθώς αυξάνεται ο αριθμός των servers.

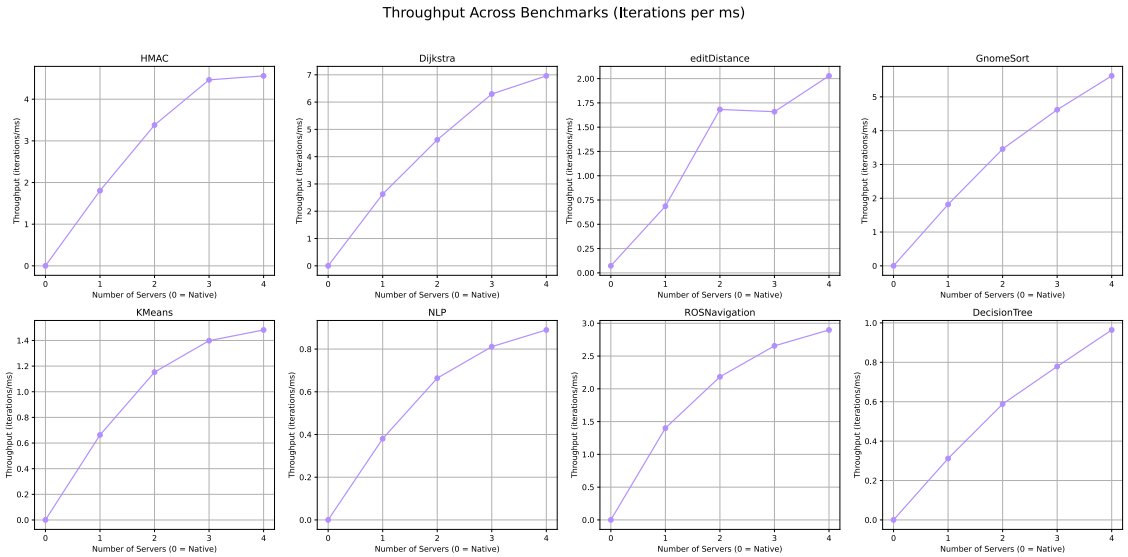

Figure 10: Συνολικό throughput για διαφορετικές διαμορφώσεις servers.

11.3.3 RQ3: Ανάλυση Overhead και Latency

Τα Σχήματα 11, 12 και 13 παρέχουν μια λεπτομερή ανάλυση του latency σε διαφορετικές διαμορφώσεις servers και benchmarks. Το Σχήμα 11 παρουσιάζει την κατανομή του per-iteration latency, αναδεικνύοντας τόσο τη διάμεσο απόδοση όσο και τη διακύμανση μεταξύ threads. Το Σχήμα 12 αποσυνθέτει το latency σε τρία στοιχεία: εκτέλεση μέσα στο enclave, overhead από offloading και χρόνος μεταφοράς μέσω δικτύου. Το Σχήμα 13 εστιάζει στα tail latencies (P95 και P99), αναδεικνύοντας την επίδραση σπάνιων, υψηλών καθυστερήσεων στη συνολική απόδοση.

Τα αποτελέσματα δείχνουν ότι η κυρίαρχη πηγή latency εξαρτάται από τα χαρακτηριστικά του workload. Σε memory-bound εφαρμογές, όπου μεταφέρονται μεγάλοι όγκοι δεδομένων στο enclave, ο χρόνος δικτύου κυριαρχεί, ειδικά στην πρώτη επανάληψη όταν φορτώνονται dependencies.

Αντίθετα, σε compute-bound εφαρμογές με έντονη υπολογιστική εργασία, ο χρόνος εκτέλεσης στο enclave αποτελεί το μεγαλύτερο μέρος του latency. Οι παράμετροι του αλγορίθμου παίζουν επίσης σημαντικό ρόλο: για παράδειγμα στο k-means, λιγότερες διαστάσεις μειώνουν την υπολογιστική πολυπλοκότητα, ενώ περισσότερα cluster centers αυξάνουν σημαντικά το computation cost.

Το overhead του enclave, συμπεριλαμβανομένων secure context switches και cryptographic operations, που σχετίζονται με SGX attestation, παραμένει αμελητέο σε σχέση με το network και computation cost. Η κατανόηση αυτών των breakdowns είναι κρίσιμη για την επίτευξη πραγματικού speedup. Το σύστημα πρέπει να ισορροπεί μεταξύ offloading overhead, κόστους δικτύου και υπολογιστικής εργασίας: αν το network ή το initialization cost κυριαρχεί, τότε το offloading δεν αποδίδει κέρδη. Η σωστή ρύθμιση του αλγορίθμου και η κατάλληλη κατανομή workload διασφαλίζουν ότι τα οφέλη του parallel execution υπερβαίνουν αυτά τα overheads.

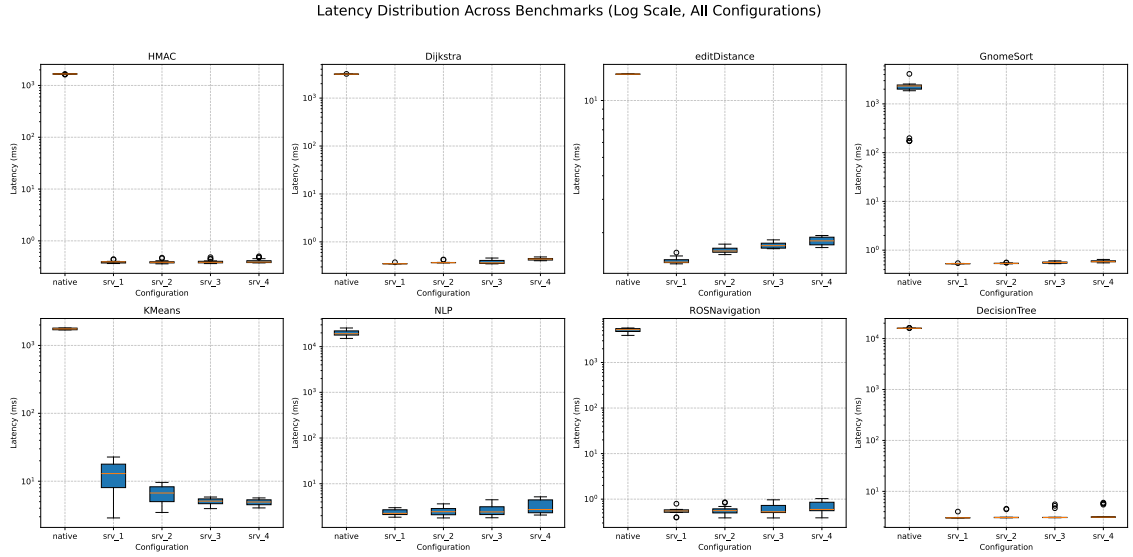

Figure 11: Κατανομή per-iteration latency σε διαφορετικά benchmarks και configurations.

Για την πληρότητα της ανάλυσης και την αξιολόγηση της συμπεριφοράς του συστήματος υπό ρεαλιστικές συνθήκες ανάπτυξης, συμπεριλαμβάνουμε επιπλέον αποτελέσματα για το RQ3 υπό το συνθετικό μοντέλο WAN με σταθερή καθυστέρηση 30 ms. Υπό συνθήκες WAN, η συνιστώσα του δικτύου γίνεται ο κυρίαρχος παράγοντας για lightweight workloads, ενώ σε compute-intensive benchmarks κυριαρχεί ο χρόνος εκτέλεσης στο enclave.

11.3.4 Προγραμματισιμότητα και Διαφάνεια

Για την αξιολόγηση των πλεονεκτημάτων προγραμματισιμότητας του *Aégis* , συγκρίνουμε το προγραμματιστικό του μοντέλο με το παραδοσιακό SGX development workflow. Όπως συζητήθηκε στην Ενότητα X, η κλασική ανάπτυξη εφαρμογών SGX απαιτεί ρητή διαχείριση των enclave interfaces, της επικοινωνίας και της διαχείρισης δεδομένων. Αντίθετα, το *Aégis* μεταφέρει αυτές τις ευθύνες στον compiler και στο runtime, εκθέτοντας μόνο υψηλού επιπέδου annotations στον προγραμματιστή.

Ο Πίνακας 2 συνοψίζει αυτή τη σύγκριση σε βασικές εργασίες ανάπτυξης.

Στο παραδοσιακό SGX workflow, οι προγραμματιστές πρέπει να ορίζουν ρητά τα enclave interfaces, να διαχειρίζονται trusted και untrusted components και να υλοποιούν μηχανισμούς επικοινωνίας μεταξύ τους. Αυτό περιλαμβάνει τη δημιουργία EDL αρχείων, τον ορισμό ECALLs και OCALLs, την υλοποίηση RPC logic και τη διαχείριση της serialization των δεδομένων που ανταλλάσσονται.

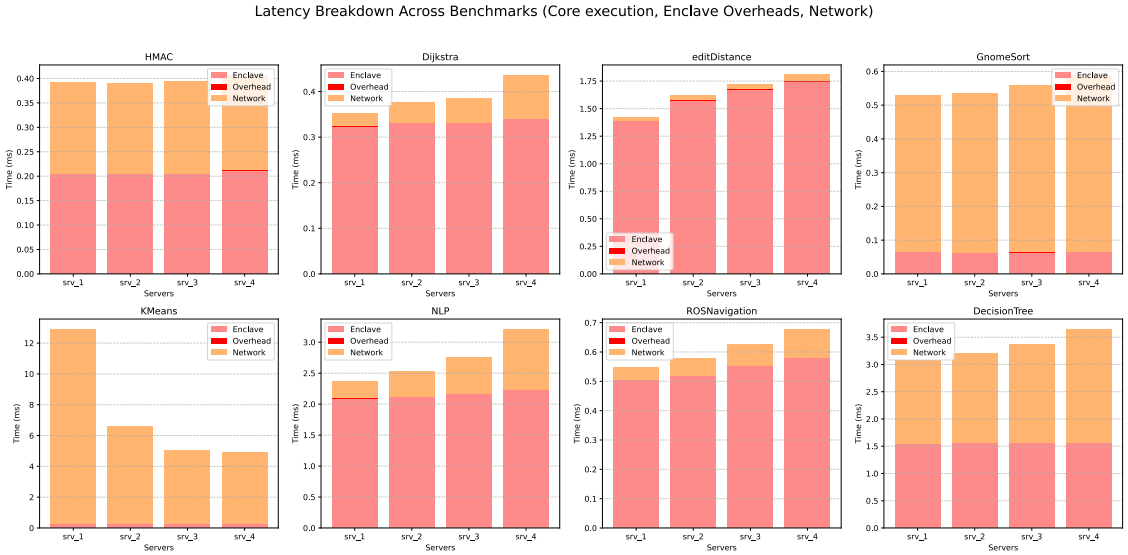

Figure 12: Αποσύνθεση latency σε enclave execution, offloading overhead και network transfer.

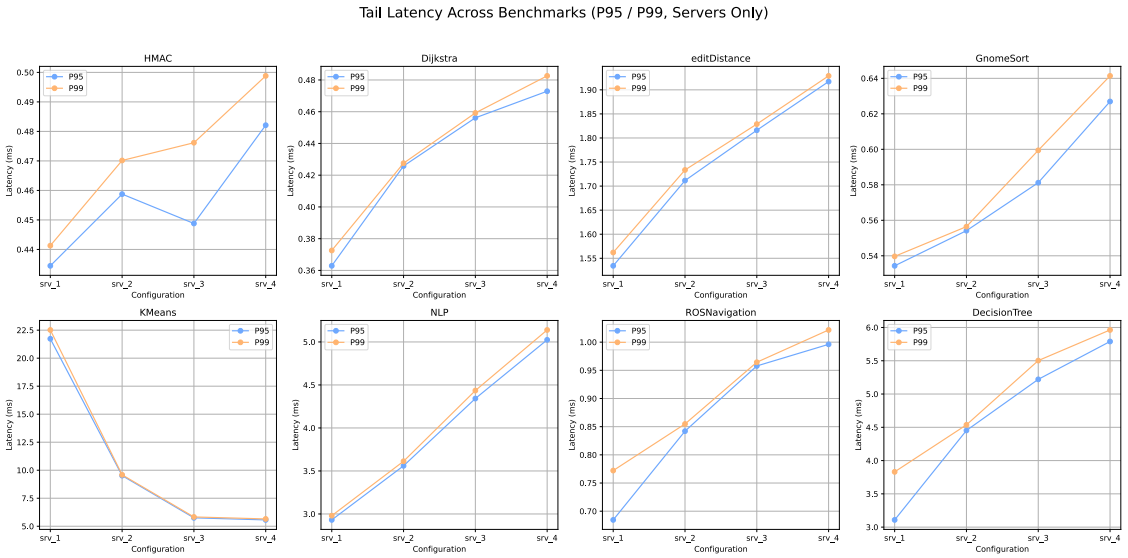

Figure 13: Tail latencies (P95 και P99) σε διαφορετικά benchmarks και configurations.

Αντίθετα, το *Aégis* αφαιρεί αυτές τις χαμηλού επιπέδου ευθύνες. Ο compiler διαμερίζει αυτόματα την εφαρμογή, επιλύει dependencies και αντικαθιστά τις annotated κλήσεις με invocations προς ένα γενικό offloading runtime. Έτσι, οι μηχανισμοί επικοινωνίας και τα enclave interfaces παραμένουν πλήρως κρυμμένα από τον προγραμματιστή, επιτρέποντας του να εστιάσει στη λογική της εφαρμογής αντί για την υποδομή εκτέλεσης.

12 Συζήτηση και Μελλοντική Εργασία

Η αποκεντρωμένη εκμάθηση (*Swarm Learning*) αποτελεί μια προσέγγιση συνεργατικής μηχανικής μάθησης που διατηρεί την ιδιωτικότητα των δεδομένων, στην οποία πολλαπλοί συμμετέχοντες, όπως

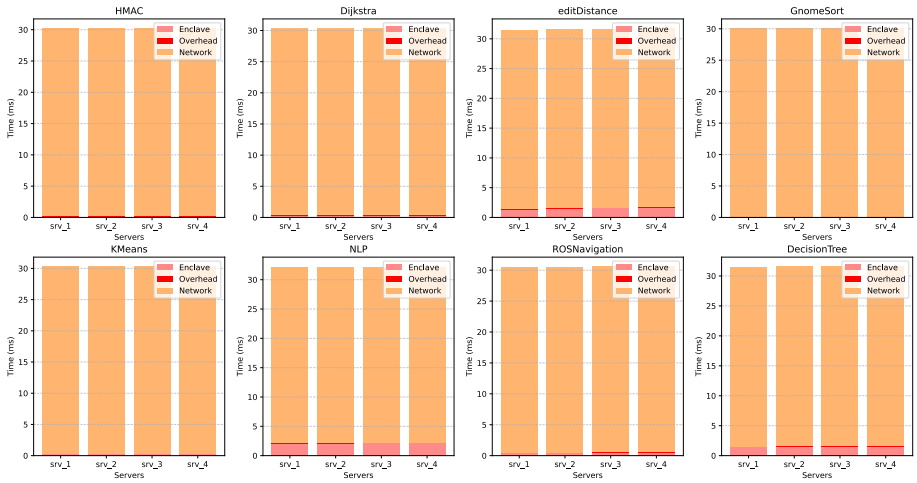

Figure 14: Αποσύνθεση καθυστέρησης υπό το συνθετικό μοντέλο WAN με σταθερή καθυστέρηση 30 ms για το RQ3. Τα αποτελέσματα δείχνουν ότι η μεταφορά μέσω δικτύου κυριαρχεί στα lightweight workloads, ενώ η εκτέλεση στο enclave παραμένει ο βασικός παράγοντας κόστους για τα compute-intensive benchmarks.

Table 2: Σύγκριση μεταξύ παραδοσιακού SGX workflow και προγραμματιστικού μοντέλου *Aégis* .

Παραδοσιακό SGX Workflow	***Aégis* Offloading Model**
Write EDL files	Automatic interface generation
Define ECALLs / OCALLs	Automatic handling via runtime
Implement RPC layer	Fully managed by offloading runtime
Manage dependencies	Compiler-assisted resolution
Handle serialization	Automatic marshalling/unmarshalling
Partition application manually	Function-level annotations
Parallelize loops manually	Annotation-based parallel execution

edge συσκευές ή οργανισμοί, εκπαιδεύουν από κοινού ένα παγκόσμιο μοντέλο χωρίς να ανταλλάσσουν τα αρχικά δεδομένα τους. Αντί αυτών, ανταλλάσσονται μόνο ενημερώσεις των παραμέτρων του μοντέλου ή οι αντίστοιχες βαθμίδες (gradients), οι οποίες συντονίζονται συνήθως μέσω blockchain ή πρωτοκόλλων peer-to-peer ώστε να διασφαλίζονται η συνέπεια και η ακεραιότητα του συστήματος [37, 38, 39]. Με τον τρόπο αυτό καθίσταται δυνατή η εκπαίδευση ενός ενιαίου μοντέλου που αξιοποιεί την ποικιλομορφία και την κλίμακα των κατανεμημένων δεδομένων, διατηρώντας παράλληλα την ιδιωτικότητα και συμμορφούμενο με τις ισχύουσες κανονιστικές απαιτήσεις.

Μία ιδιαίτερα ενδιαφέρουσα κατεύθυνση μελλοντικής έρευνας είναι η επέκταση του *Aégis* ώστε να υποστηρίζει αποδοτική αποκεντρωμένη εκμάθηση πάνω σε ευαίσθητα, κατανεμημένα δεδομένα. Τα υπάρχοντα πλαίσια προστατεύουν την ιδιωτικότητα ανταλλάσσοντας μόνο συγκεντρωτικές ενημερώσεις των μοντέλων [37, 38, 39, 40]. Ωστόσο, το μεγαλύτερο μέρος του υπολογιστικού φόρτου της μηχανικής μάθησης (εκπαίδευση και εξαγωγή συμπερασμάτων) εξακολουθεί να εκτελείται τοπικά σε συσκευές περιορισμένων πόρων, γεγονός που περιορίζει την επεκτασιμότητα και αυξάνει τόσο την ενεργειακή κατανάλωση όσο και την καθυστέρηση.

Τεχνικές που βασίζονται στην ομομορφική κρυπτογράφηση (Homomorphic Encryption) επιτρέπουν την εκτέλεση υπολογισμών απευθείας πάνω σε κρυπτογραφημένα δεδομένα χωρίς αποκρυπτογράφηση. Ωστόσο, οι κρυπτογραφημένοι τανυστές είναι συνήθως κατά τάξεις μεγέθους μεγαλύτεροι και σημαντικά ακριβότεροι υπολογιστικά, με επιβαρύνσεις που φθάνουν από 10× έως 1000× για την εξαγωγή συμπερασμάτων και από 100× έως 10 000× για την εκπαίδευση, σε σύγκριση με τους αντίστοιχους υπολογισμούς σε μη κρυπτογραφημένα δεδομένα [41, 42].

Παρομοίως, πρόσφατες προσεγγίσεις που βασίζονται σε Trusted Execution Environments (TEEs) επιδεικνύουν ασφαλή εκπαίδευση και συγκέντρωση μοντέλων μέσα σε enclaves, αλλά επικεντρώνονται κυρίως στην προστασία συγκεκριμένων σταδίων της διαδικασίας (όπως η συγκέντρωση των ενημερώσεων), εξακολουθώντας να βασίζονται στην τοπική εκτέλεση της εκπαίδευσης από τις συσκευές-πελάτες [43, 44].

Αξιοποιώντας τις δυνατότητες υβριδικής εκφόρτωσης και προγραμματισμού enclaves που παρέχει το *Aégis* , οραματιζόμαστε μία επέκταση προς την αποκεντρωμένη εκμάθηση, στην οποία σημαντικά μεγαλύτερο μέρος του υπολογιστικού φόρτου — συμπεριλαμβανομένου του υπολογισμού των gradients και της ενημέρωσης των παραμέτρων του μοντέλου— θα μπορεί να εκφορτώνεται με ασφάλεια σε απομακρυσμένα enclaves αντί να εκτελείται εξ ολοκλήρου στις edge συσκευές. Στο μοντέλο αυτό, ευαίσθητα δεδομένα από πολλαπλούς συμμετέχοντες θα μπορούν να συγκεντρώνονται και να επεξεργάζονται συλλογικά μέσα στους enclaves, επιτρέποντας την εκπαίδευση μοντέλων με μεγαλύτερη ακρίβεια, καλύτερη γενίκευση και δυνατότητα αξιοποίησης πληθυσμιακών προτύπων. Προς τους συμμετέχοντες θα επιστρέφουν αποκλειστικά προστατευμένες ενημερώσεις των παραμέτρων του μοντέλου, διατηρώντας την εμπιστευτικότητα των δεδομένων ενώ ταυτόχρονα αξιοποιούνται οι πληροφορίες που προκύπτουν από το σύνολο του πληθυσμού.

13 Σχετική Εργασία

Ένας σημαντικός αριθμός προηγούμενων συστημάτων και πλαισίων έχει διερευνήσει την ασφαλή εκτέλεση εφαρμογών, την απομακρυσμένη εκφόρτωση υπολογισμών και τη χρήση Έμπιστων Περιβαλλόντων Εκτέλεσης (Trusted Execution Environments – TEEs) για την προστασία ευαίσθητων υπολογισμών και δεδομένων σε μη έμπιστα περιβάλλοντα.

Πλαίσια προστατευμένης εκτέλεσης, όπως το *Haven*, εισήγαγαν την έννοια της προστασίας ολόκληρων εφαρμογών και των δεδομένων τους από μη έμπιστες υποδομές υπολογιστικού νέφους. Το *Haven* αξιοποιεί το Intel SGX για να παρέχει προστατευμένη εκτέλεση παλαιότερων δυαδικών εφαρμογών, διασφαλίζοντας την εμπιστευτικότητα και την ακεραιότητά τους ακόμη και παρουσία κακόβουλου λειτουργικού συστήματος ή hypervisor, ενώ παράλληλα αποτέλεσε τη βάση για μεταγενέστερες τεχνικές εμπιστευτικού υπολογισμού λεπτότερης κοκκοποίησης [**haven2014**].

Επεκτείνοντας τη χρήση hardware enclaves για ασφαλή επεξεργασία, το *Ryoan* παρουσιάζει ένα κατανεμημένο sandbox που απομονώνει μη έμπιστες μονάδες επεξεργασίας δεδομένων μέσα σε hardware TEEs. Μέσω της απομόνωσης του υπολογισμού σε πολλαπλά στιγμιότυπα enclaves, το *Ryoan* αποτρέπει τη διαρροή ευαίσθητων δεδομένων εισόδου σε κατανεμημένες υπηρεσίες, υιοθετώντας απειλητικά μοντέλα παρόμοια με εκείνα που χρησιμοποιούνται σε πλαίσια επιλεκτικής εκφόρτωσης όπως το *Aégis* [**ryoan2016**].

Το *SCONE* διερευνά τον τρόπο με τον οποίο εφαρμογές βασισμένες σε containers μπορούν να αξιοποιήσουν την προστασία που προσφέρουν τα SGX enclaves. Το σύστημα προστατεύει μη τροποποιημένες διεργασίες containers από εξωτερικές απειλές, ελαχιστοποιώντας τη βάση εμπιστοσύνης (Trusted Computing Base) και διατηρώντας απόδοση κοντά στη φυσική εκτέλεση. Με τον τρόπο

αυτό καταδεικνύεται ότι το SGX μπορεί να ενσωματωθεί αποτελεσματικά σε πραγματικά περιβάλλοντα εκτέλεσης, προστατεύοντας τόσο τον κώδικα όσο και τα δεδομένα σε σύνθετες στοίβες λογισμικού [**scone2016**].

Στον τομέα της ιδιωτικής εξαγωγής συμπερασμάτων (privacy-preserving inference), το *Occlumency* επικεντρώνεται στην απομακρυσμένη εκτέλεση μοντέλων βαθιάς μάθησης μέσα σε SGX enclaves. Το σύστημα διασφαλίζει ότι τα δεδομένα εισόδου, οι ενδιάμεσες ενεργοποιήσεις και τα αποτελέσματα παραμένουν εμπιστευτικά καθ' όλη τη διάρκεια της διαδικασίας εκφόρτωσης, αντιμετωπίζοντας τα ζητήματα ιδιωτικότητας που χαρακτηρίζουν τις υπηρεσίες cloud-assisted inference [45].

Το *EnCloak* ακολουθεί μία συμπληρωματική προσέγγιση, εστιάζοντας στον αυτόματο εντοπισμό και την προστασία ευαίσθητων περιοχών κώδικα. Μέσω αμφίδρομης ανάλυσης taint και μετασχηματισμού κρίσιμων τμημάτων κώδικα σε enclave instructions, το *EnCloak* παρέχει ολοκληρωμένη προστασία ευαίσθητων δεδομένων και μειώνει τη συνολική βάση εμπιστοσύνης του συστήματος, αναδεικνύοντας τη σημασία των μετασχηματισμών προγραμμάτων σε σενάρια ασφαλούς εκφόρτωσης [46].

Πέρα από τα εξειδικευμένα συστήματα enclaves, ανασκοπήσεις σχετικά με την εκφόρτωση υπολογισμών και τον καταμερισμό εργασιών σε συνεργατικά περιβάλλοντα cloud-edge παρουσιάζουν μια ευρύτερη εικόνα του πεδίου. Οι εργασίες αυτές εξετάζουν ποικίλες στρατηγικές διαχωρισμού του υπολογισμού μεταξύ τοπικών συσκευών και απομακρυσμένων εξυπηρετητών, λαμβάνοντας υπόψη παράγοντες όπως η απόδοση, η ενεργειακή κατανάλωση και η καθυστέρηση, και αναδεικνύουν την ανάγκη από κοινού βελτιστοποίησης της κατάτμησης και της εκφόρτωσης για διαφορετικές κατηγορίες εφαρμογών και αρχιτεκτονικών [47].

Πρόσφατες προσεγγίσεις, όπως το *T-Edge*, επεκτείνουν την έννοια της αξιόπιστης εκτέλεσης σε ετερογενείς πλατφόρμες edge computing, διερευνώντας πώς μηχανισμοί ασφαλείας όπως το ARM TrustZone μπορούν να υποστηρίξουν ασφαλή υπολογισμό σε CPUs και επιταχυντές υλικού μέσα σε μη έμπιστα περιβάλλοντα [48]. Η συγκεκριμένη κατεύθυνση αναδεικνύει το αυξανόμενο ενδιαφέρον για την επέκταση της προστασίας των TEEs πέρα από τα παραδοσιακά CPU enclaves προς ένα ευρύτερο φάσμα edge υποδομών.

Τέλος, συστήματα όπως το *HiveMind* διερευνούν τον κατανεμημένο συντονισμό μεταξύ edge και cloud υποδομών με στόχο τη βελτίωση της απόδοσης και της επεκτασιμότητας. Αν και δεν εστιάζουν πρωτίστως στην ασφάλεια των enclaves, η μελέτη τους σχετικά με την κατανεμημένη εκτέλεση εργασιών και τον συντονισμό απομακρυσμένων πόρων παρέχει χρήσιμα συμπεράσματα για συστήματα που διαχειρίζονται εκφορτωμένους υπολογισμούς σε ετερογενείς και γεωγραφικά κατανεμημένες υποδομές [<**empty citation**>].

Συνολικά, οι παραπάνω εργασίες αποτελούν τη βάση για την ασφαλή εκτέλεση και εκφόρτωση υπολογισμών σε μη έμπιστα περιβάλλοντα. Ωστόσο, σε αντίθεση με προηγούμενες προσεγγίσεις που είτε προστατεύουν ολόκληρες εφαρμογές, είτε επικεντρώνονται σε συγκεκριμένες κατηγορίες φορτίων εργασίας, είτε εξετάζουν γενικές στρατηγικές εκφόρτωσης, το *Aégis* συνδυάζει στατική κατάτμηση προγραμμάτων, δυναμικό προγραμματισμό enclaves και λεπτομερή επιλεκτική εκφόρτωση, επιτρέποντας ασφαλή και επεκτάσιμη κατανεμημένη εκτέλεση κρίσιμων υπολογιστικών εργασιών από ενσωματωμένες συσκευές.

14 Συμπεράσματα

Η παρούσα διπλωματική εργασία παρουσίασε το *Aégis* , ένα ασφαλές και αποδοτικό πλαίσιο για επιλεκτική εκφόρτωση υπολογισμών από ενσωματωμένες συσκευές περιορισμένων πόρων προς Έμπιστα Περιβάλλοντα Εκτέλεσης (Trusted Execution Environments – TEEs). Συνδυάζοντας μετασχηματισμούς προγραμμάτων καθοδηγούμενους από μεταγλωττιστή με έναν ελαφρύ χρόνο εκτέλεσης και ένα υβριδικό

μοντέλο χρονοδρομολόγησης, το *Aégis* επιτρέπει τη δυναμική κατανομή υπολογιστικά απαιτητικών εργασιών σε ασφαλείς κόμβους enclaves, διατηρώντας παράλληλα την εμπιστευτικότητα των δεδομένων.

Μέσα από τη δομική ανάλυση εφαρμογών ενσωματωμένων συστημάτων, αναδείξαμε τα πρότυπα εκτέλεσης που βασίζονται σε βρόχους ως μία θεμελιώδη αφαίρεση για την υλοποίηση λεπτομερούς επιλεκτικής εκφόρτωσης. Αξιοποιώντας αυτήν την παρατήρηση, το *Aégis* μετασχηματίζει σχολιασμένα προγράμματα σε ασύγχρονες, προσανατολισμένες σε εργασίες ροές εκτέλεσης, οι οποίες εκμεταλλεύονται τον παραλληλισμό μεταξύ ανεξάρτητων επαναλήψεων. Παράλληλα, ο σχεδιασμός του συστήματος αντιμετωπίζει βασικές προκλήσεις της ασφαλούς εκφόρτωσης, όπως η διαχείριση εξαρτήσεων, το κόστος επικοινωνίας και οι περιορισμένοι πόροι των περιβαλλόντων enclave.

Η πειραματική αξιολόγηση σε ένα ευρύ σύνολο αντιπροσωπευτικών φορτίων εργασίας έδειξε ότι το *Aégis* μπορεί να βελτιώσει σημαντικά την απόδοση των εφαρμογών, επιτυγχάνοντας ουσιαστικές επιταχύνσεις και αύξηση της ρυθμαπόδοσης σε σύγκριση με την αποκλειστικά τοπική εκτέλεση. Παρότι η εκφόρτωση εισάγει πρόσθετο κόστος λόγω της δικτυακής επικοινωνίας και της εκτέλεσης μέσα σε enclaves, τα αποτελέσματα καταδεικνύουν ότι το κόστος αυτό αντισταθμίζεται από τα οφέλη του παραλληλισμού και της αξιοποίησης ισχυρότερων υπολογιστικών πόρων, ιδιαίτερα σε εφαρμογές με έντονα υπολογιστικό χαρακτήρα.

Συνολικά, το *Aégis* αποδεικνύει ότι η ασφαλής εκφόρτωση υπολογισμών αποτελεί όχι μόνο μία εφικτή αλλά και μία πρακτική λύση για τα σύγχρονα ενσωματωμένα συστήματα. Γεφυρώνοντας το χάσμα μεταξύ edge computing και αξιόπιστης εκτέλεσης στο υπολογιστικό νέφος, η εργασία αυτή θέτει τα θεμέλια για την ανάπτυξη εφαρμογών υψηλής απόδοσης που διατηρούν την ιδιωτικότητα των δεδομένων σε τομείς όπως το Διαδίκτυο των Πραγμάτων (IoT), η υγειονομική περίθαλψη και τα αυτόνομα συστήματα. Μελλοντικές επεκτάσεις μπορούν να αξιοποιήσουν την ίδια προσέγγιση σε αποκεντρωμένα και συνεργατικά περιβάλλοντα, όπως η αποκεντρωμένη εκμάθηση (Swarm Learning), επιτρέποντας την ανάπτυξη κλιμακούμενης και ιδιωτικής κατανεμημένης νοημοσύνης.

Βιβλιογραφία

[1] Wei Yu et al. “A Survey on the Edge Computing for the Internet of Things”. In: *IEEE Access* 6 (2017), pp. 6900–6919. DOI: 10.1109/ACCESS.2017.2778504.

[2] Wikipedia Contributors. *Confidential computing*. https://en.wikipedia.org/wiki/Confidential_computing. Accessed 2026-04-08. 2026.

[3] Michael Barr and Anthony Massa. *Programming Embedded Systems in C and C++*. 2nd. O’Reilly Media, 2006.

[4] Jane W. S. Liu. *Real-Time Systems*. Prentice Hall, 2000.

[5] *ISO 26262: Road vehicles – Functional safety*. International Organization for Standardization, 2018.

[6] *DO-178C: Software Considerations in Airborne Systems and Equipment Certification*. RTCA, 2011.

[7] Weisong et al. Shi. “Edge Computing: Vision and Challenges”. In: *IEEE Internet of Things Journal* (2016).

[8] *OpenCV: Open Source Computer Vision Library*. https://opencv.org. Accessed: 2026.

[9] Jonathan R. et al. Wolpaw. “Brain–computer interfaces for communication and control”. In: *Clinical Neurophysiology* (2002).

[10] Gari D. et al. Clifford. “Advanced Methods and Tools for ECG Data Analysis”. In: *Artech House* (2006).

[11] Geoffrey et al. Hinton. “Deep Neural Networks for Acoustic Modeling in Speech Recognition”. In: *IEEE Signal Processing Magazine* (2012).

[12] Alan V. Oppenheim and Ronald W. Schafer. *Discrete-Time Signal Processing*. Prentice Hall, 1999.

[13] *MONAI: Medical Open Network for AI*. https://monai.io. Accessed: 2026.

[14] *Emoncms: Open-source energy monitoring platform*. https://github.com/emoncms/emoncms. Accessed: 2026.

[15] Jay et al. Lee. “Predictive Manufacturing Systems—Trends of Next-Generation Production Systems”. In: *IFAC Proceedings Volumes* (2014).

[16] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall, 2007.

[17] Michael et al. Armbrust. “A View of Cloud Computing”. In: *Communications of the ACM* (2010).

[18] *Eclipse hawkBit: OTA Update Deployment Framework*. https://www.eclipse.org/hawkbit/. Accessed: 2026.

[19] *GraphHopper Routing Engine*. https://www.graphhopper.com. Accessed: 2026.

[20] *Amazon Web Services (AWS)*. https://aws.amazon.com/what-is-cloud-computing/. Accessed: 2026.

[21] *Microsoft Azure*. https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-cloud-computing/. Accessed: 2026.

[22] *Google Cloud*. https://cloud.google.com/learn/what-is-cloud-computing. Accessed: 2026.

[23] *IBM Cloud*. https://www.ibm.com/cloud/learn/what-is-cloud-computing. Accessed: 2026.

[24] Victor Costan and Srinivas Devadas. *Intel SGX Explained*. Tech. rep. IACR, 2016.

[25] Yossi et al. Oren. "The power of oblivious RAM". In: *IEEE Security & Privacy* (2014).

[26] Oded Goldreich and Rafail Ostrovsky. "Software protection and simulation on oblivious RAMs". In: *Journal of the ACM*. 1996.

[27] Emil Stefanov, Marten van Dijk, and Elaine et al. Shi. "Path ORAM: An extremely simple oblivious RAM protocol". In: *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*. 2013.

[28] Craig Gentry. "Fully Homomorphic Encryption Using Ideal Lattices". In: (2009).

[29] Nigel P. Smart and Frederik Vercauteren. *Fully Homomorphic Encryption*. Springer, 2014.

[30] Marten Van Dijk et al. "Fully Homomorphic Encryption over the Integers". In: *EUROCRYPT 2010* (2010).

[31] Andrew Yao. "Protocols for secure computations". In: 1982.

[32] Oded Goldreich. *Secure Multi-Party Computation*. Manuscript, 1998.

[33] Yehuda Lindell and Benny Pinkas. *Secure Multiparty Computation for Privacy-Preserving Data Mining*. Springer, 2009.

[34] Trusted Computing Group. *Trusted Platform Module (TPM) Main Specification Version 1.2*. https://trustedcomputinggroup.org/resource/tpm-main-specification/. 2007.

[35] Trusted Computing Group. *Trusted Platform Module (TPM) Main Specification Version 2.0*. https://trustedcomputinggroup.org/resource/tpm-library-specification/. 2014.

[36] Fiona et al. Chong. "Hardware security: Design, implementation and analysis of TPM-based systems". In: *IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*. 2015.

[37] Hewlett Packard Enterprise. *HPE Swarm Learning: Decentralized, privacy-preserving machine learning framework*. https://developer.hpe.com/platform/swarm-learning/home/. Accessed 2026-04. 2022.

[38] Jie Xie et al. "Swarm learning network for privacy-preserving and collaborative deep learning-assisted diagnosis of fracture: A multi-center diagnostic study". In: *Frontiers in Medicine* 12 (2025), p. 1534117. DOI: 10.3389/fmed.2025.1534117.

[39] Stefanie Warnat-Herresthal et al. "Swarm Learning for decentralized and confidential clinical machine learning". In: *Nature* 594.7862 (2021), pp. 265–270. DOI: 10.1038/s41586-021-03583-3.

[40] Jie Zhao et al. “Landscape of machine learning evolution: privacy-preserving federated learning frameworks and tools”. In: *Artificial Intelligence Review* 57.3 (2024), p. 11036. DOI: 10.1007/s10462-024-11036-2.

[41] The TF-Encrypted Project. *TF-Encrypted: Machine learning libraries over encrypted data*. https://github.com/tf-encrypted/tf-encrypted. Accessed 2026-04. 2026.

[42] Ayse Acar, Aurelien Ferey, and Murat Kantarcioglu. “Homomorphic Encryption in Machine Learning: Performance overheads and challenges”. In: *Journal of Privacy and Confidentiality* (2025). Preprint available.

[43] John Smith, Hyun Lee, and Jane Doe. “Secure enclave-based local training and aggregation for confidential machine learning”. In: *Expert Systems with Applications* 223 (2023), p. 122410. DOI: 10.1016/j.eswa.2023.122410.

[44] Wei Zhang, Feng Yu, and Ting Liu. *Secure training inside enclaves with trusted execution technology*. https://arxiv.org/abs/2104.14380. 2021.

[45] Taegyeong Lee et al. “Occlumency: Privacy-preserving Remote Deep-learning Inference Using SGX”. In: *Proceedings of the 25th Annual International Conference on Mobile Computing and Networking (MobiCom 2019)*. MobiCom ’19. Los Cabos, Mexico: ACM, 2019, 46:1–46:17. DOI: 10.1145/3300061.3345447.

[46] Yongzhi Wang and Bozhen Liu. “EnCloak: Protecting Sensitive Data in Remote Computing Using Trusted Execution Environments”. In: *Cluster Computing* 29.3 (2026), p. 140. DOI: 10.1007/s10586-025-05880-2.

[47] Haiming Chen, Wei Qin, and Lei Wang. “Task partitioning and offloading in IoT cloud-edge collaborative computing framework: a survey”. In: *Journal of Cloud Computing* 11.1 (2022), p. 86. DOI: 10.1186/s13677-022-00365-8.

[48] Yeghisabet Alaverdyan, Suren Poghosyan, and Vahagn Poghosyan. “T-Edge: Trusted Heterogeneous Edge Computing for Confidential and Reliable Execution”. In: *arXiv preprint* (2024). arXiv:2412.13905v1.

[49] Victor Costan and Srinivas Devadas. “Intel SGX Explained”. In: *IACR Cryptology ePrint Archive* 2016 (2016), p. 86.

[50] Rajkumar Buyya, James Broberg, and Andrzej Goscinski. *Cloud Computing: Principles and Paradigms*. Wiley, 2011.

[51] *AUTOSAR – Automotive Open System Architecture*. https://www.autosar.org. Accessed: 2026.

[52] Xiaoguo Li et al. “A Survey of Secure Computation Using Trusted Execution Environments”. In: *arXiv preprint arXiv:2302.12150* (2023).

[53] Wikipedia Contributors. *Trusted execution environment*. https://en.wikipedia.org/wiki/Trusted_execution_environment. Accessed 2026-04-08. 2026.

[54] Manohar Arunkumar and Rohit Gupta. “Hybrid offloading of secure computations on edge devices”. In: *IEEE Transactions on Parallel and Distributed Systems* 31 (2020), pp. 1–13.

[55] Liang Shao and Hui Wang. “Secure task offloading in trusted edge computing”. In: *Future Generation Computer Systems* 118 (2021), pp. 1–15.

[56] Zhenhua Fan and Hui Zhang. “Task partitioning and scheduling for secure edge computing”. In: *IEEE Access* 8 (2020), pp. 123456–123467.

[57] Jihoon Lee and Sungjae Kim. “Lightweight runtime for secure enclave execution”. In: *ACM Transactions on Embedded Computing Systems* 20.3 (2021), pp. 1–22.

[58] Andrew Baumann, Marcus Peinado, and Galen Hunt. “Shielding Applications from an Untrusted Cloud with Haven”. In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, 2014, pp. 267–283. URL: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/baumann.

[59] Tyler Hunt et al. “Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data”. In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, 2016, pp. 533–549. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/hunt.

[60] J. Hu et al. “HiveMind: A Scalable and Serverless Coordination Control Platform for UAV Swarms”. In: *CoRR* abs/2002.01419 (2020). URL: https://arxiv.org/abs/2002.01419.

[61] Sergei Arnautov et al. “SCONE: Secure Linux Containers with Intel SGX”. In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, 2016, pp. 689–703. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov.

[62] Müge Erel-Özçevik and Elif Bozkaya-Aras. “Secure computation offloading for data leakage prevention in digital twin edge networks”. In: *Cluster Computing* 29 (2026), p. 145. DOI: 10.1007/s10586-026-05950-z.

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

Aègis: Secure Selective Offloading from Embedded Devices

DIPLOMA THESIS

by

Maria-Argyro Lazou

Supervisor: Georgios Goumas
Associate Professor, NTUA

Athens, July 2026

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

Aègis: Secure Selective Offloading from Embedded Devices

Diploma Thesis

Maria-Argyro Lazou

Supervisor: Georgios Goumas
Professor, NTUA

Approved by the three-member doctoral scientific committee on Monday July 6th 2026.

.....	.....	.....
Georgios Goumas	Nikos Vasilakis	Dionisios Pnevmatikatos
Professor, NTUA	Assistant Professor, Brown University	Professor, NTUA

Athens, July, 2026

.....
Maria-Argyro Lazou
Graduate of School of Electrical and Computer Engineering, National Technical University of Athens

1 Abstract

Embedded applications have become increasingly popular, powering a wide range of intelligent systems such as IoT devices, autonomus platforms and microcontrollers. These applications often process privacy-sensitive data collected from sensors or user input, as part of complex workflows that support real-time control, analytics and decision-making. However, due to their restricted resources, executing intensive, compute-bound tasks degregates their performance. Outsourcing parts of computation directly to cloud vendors, poses significant security risks, since the server itself may be compromised. Trusted Execution Environments (TEEs) protect against malicious priviledges software including the operating system or hypervisor, as well as unauthorized memory inspection and tampering, offering security guarantees for applications that span on- and off-premise components. *Aégis* adopts a hybrid scheduling model that dynamically scales application components to TEEs at runtime, increasing throughput while preserving data confidentiality. It selectively offloads and distributes performance-critical functions across available enclave nodes following compiler-driven program partitioning, exploiting task-level parallelism whenever possible. To support this execution model, *Aégis* incorporates a lightweight TEE-embedded runtime that orchestrates remote execution while operating within the strict memory constraints of enclave environments. Across a diverse set of embedded workloads, *Aégis* achieves a median speedup of $4.7 \times 10^3\times$ in the single-server setting, and up to an order-of-magnitude additional improvement under distributed execution across multiple enclave nodes.

Keywords

Trusted Execution Environments (TEEs), Secure Computation Offloading, Embedded Systems, Distributed Enclave Execution, Program Transformation, Task Parallelism, Runtime Systems

2 Acknowledgements

At this point, I would like to express my sincere gratitude to Prof. George Goumas, who has been a major source of inspiration throughout my undergraduate studies. He introduced me to the field of computer systems and encouraged me to pursue graduate studies and further explore this domain.

I am also deeply grateful to Prof. Nikos Vasilakis, who provided me with the unique opportunity to conduct part of my thesis research at Brown University, as well as to participate in additional research projects that significantly shaped my perspective on graduate-level research and helped define my academic direction.

Finally, I would like to thank my friends and lab mates for the moments we shared, which made parts of my years at NTUA more enjoyable and slightly less stressful.

Above all, I would like to thank my parents—Afrodite and Asimakis— who have supported me unconditionally at every stage of my life. From an early age, they instilled in me the value of hard work, perseverance, and continuous effort toward achieving meaningful goals. Their sacrifices, guidance, and belief in me have been fundamental in shaping who I am today, and to them I owe my deepest gratitude.

Contents

3 Introduction

Edge computing is becoming increasingly widespread due to the rapid growth of IoT devices and the need for real-time predictions and intelligent automation. In this direction, artificial intelligence plays a key role by enabling accurate inference through trained models deployed directly on edge. Devices such as health or fitness wearables, gyroscopes, security cameras, and audio recorders, rely on signal-processing pipelines to transform noisy, unstructured sensor data into meaningful features followed by machine learning routines for decision making. They continuously gather and analyze physiological and environmental signals, including heart rate, insulin levels, respiration patterns, motion trajectories, image frames, and acoustic signals, to feed their core computation engines. In this setting, the data being processed is highly sensitive and should remain accessible only by the user. At the same time, the computational intensity of some internal patterns makes the whole pipeline inefficient for devices with such restricted resources. The resulting performance bottleneck leads to slower response times, excessive battery consumption or undersampling of the target system to reduce input size degrading prediction accuracy [1].

While public cloud vendors such as AWS, Azure, IBM, and Google Cloud, offer substantial computational resources, outsourcing execution to them shifts sensitive workloads to an untrusted environment, creating a fundamental trade-off between performance and data confidentiality [2]. Furthermore, certain operations—such as continuous sensor data collection, which is typically generated as a stream, and the application of lightweight filtering stages—must inherently execute on the edge device.

Consequently, a hybrid approach is required to determine the optimal execution location for each task in the application pipeline, along with a sandboxed environment to ensure secure and isolated execution of sensitive components. *Aégis* embodies this approach by selectively offloading bottleneck functions from embedded devices to secure enclaves. Its design stems from the observation that most embedded applications follow a recurring loop-based execution pattern, which can be identified across different workloads with small structural variations in control flow, timing constraints, and data-flow granularity.

Aégis consists of a DSL compiler that captures the appropriate offloading semantics and performs program transformations to split the application into native and remotely executing tasks. It also provides runtime primitives to support secure remote execution, including a custom QJS interpreter enhanced with encryption and communication mechanisms to transfer execution state over a secure channel after completing a handshake. Additionally, *Aégis* includes an analysis engine that tracks dependency modules that must be transmitted to the enclave environment, and a client runtime that manages request submission and result retrieval in a task-queue-based execution model enabling concurrent offloading whenever possible. On the server side, a tailored QJS runtime is embedded within the enclave to evaluate incoming requests, accompanied by a middleware layer that mediates communication between the enclave and the untrusted server. Finally, a scheduler keeps tracks of active client–enclave sessions and assigns available enclave nodes in a round-robin fashion upon request.

In summary, this project makes the following contributions:

- A structural study and taxonomy of loop-driven embedded applications that identifies execution patterns suitable for secure offloading.

- A lightweight enclave execution runtime that enables transparent remote execution while preserving data confidentiality and operating within strict resource constraints.
- A selective offloading framework that dynamically distributes performance-bottleneck functions across enclave nodes to support partial remote execution and improve throughput.
- A domain-specific language (DSL) compiler that automatically transforms annotated loop constructs into asynchronous remotely executable tasks and orchestrates their distributed execution across secure enclave nodes.

The sourcecode will become publicly available on Github after submission.

4 Example

Consider the following ECG processing routine that reads heart signal data and attempts to detect whether a patient exhibits signs of arrhythmia.

```
1 const arrModel = await tf.loadLayersModel('file://./arr_model_v4/model.json');
2
3 function ECGPipeline() {
4     for (let i = 0; i < ITERATIONS; i++) {
5         const ecgData = readECG();
6         const result = predictArrhythmia(ecgData, arrModel);
7         console.log("Arrhythmia detection:", result);
8     }
9 }
```

The `readECG()` function (line 5) collects batches of signal measurements from a sensor at a fixed sampling frequency. The collected data are then passed to `predictArrhythmia()` (line 6), which performs a lightweight preprocessing step (e.g., normalization) before running inference using a pre-trained TensorFlow model (line 1).

While signal acquisition and preprocessing are relatively inexpensive, the inference step dominates the execution cost of the pipeline. On resource-constrained embedded devices, repeatedly executing this computation can quickly become a performance bottleneck and limit the throughput of the monitoring application. In practice, if the prediction routine becomes too slow, the system may be forced to lower the ECG sampling frequency in order to keep up with processing demands. However, reducing the sampling rate can lead to the loss of important signal characteristics and may negatively impact the accuracy of arrhythmia detection. As a result, `predictArrhythmia()` becomes a natural candidate for remote execution.

To express this intent, developers can annotate the loop with an offloading directive (line 4) that identifies the expensive function (line 5) and specifies that no loop-carried dependencies exist (line 6), thereby enabling further parallelism:

```
1 const arrModel = await tf.loadLayersModel('file://./arr_model_v4/model.json');
2
3 function ECGPipeline() {
4   /* @offload
5      -- fname predictArrhythmia
6      --parallel
7   */
8   for (let i = 0; i < ITERATIONS; i++) {
9       const ecgData = readECG();
10      const result = predictArrhythmia(ecgData, arrModel);
11      console.log("Arrhythmia detection:", result);
12  }
13 }
```

Given that medical data are highly sensitive, patient records could be exposed if the remote server responsible for processing them were to be compromised. Therefore, an additional security layer is required on the server side to ensure confidentiality during remote execution.

Aégis addresses this concern by offloading the computation to a client-dedicated secure enclave, specifically leveraging Intel SGX. This trusted execution environment provides strong

isolation guarantees, ensuring that sensitive ECG data remain protected even if the host system is untrusted.

The DSL compiler automatically transforms the annotated code into an asynchronous execution model that supports secure remote invocation.

```
1  const arrModel = await tf.loadLayersModel('file://./arr_model_v4/model.json');
2
3  async function ECGPipeline() {
4      const ecgData = readECG();
5      // __remote__
6      const result = await predictArrhythmia(ecgData, arrModel);
7      console.log("Arrhythmia detection:", result);
8  }
9
10 scaleout_traffic(ECGPipeline, [], ITERATIONS);
```

Specifically, the compiler extracts the loop body and wraps it into an asynchronous routine (line 3) while marking the inference call as a remote operation (line 5). The call to `scaleout_traffic()` (line 10) orchestrates the concurrent execution of multiple iterations of `ECGPipeline`, distributing tasks across available enclave nodes. This transformation effectively splits the computation into two parts: a local component responsible for data acquisition and control flow, and a remote component running the inference within the enclave.

Unrolling the loop in this manner serves two main purposes. First, it aligns the program with the enclave execution model, where individual tasks can be securely invoked and executed remotely. Second, since ECG samples are processed independently, multiple iterations can be scheduled concurrently across enclave nodes, significantly improving overall throughput.

5 Embedded Loop Taxonomy

Embedded systems are structurally organized around a persistent control loop, commonly expressed as while(1) or an equivalent construct that continuously senses, computes, and acts. This architectural pattern is not incidental but foundational. As described in *Programming Embedded Systems* [3], embedded firmware typically executes its core functionality inside an infinite loop that repeatedly performs sensing, processing, and actuation after initialization. Similar principles appear in practitioner discussions of deterministic control cycles, where a fixed-duration *"major cycle"* ensures predictable execution timing [4]. The infinite loop therefore represents a structural invariant of embedded software: it defines both the liveness and the temporal structure of the system. Building on this observation, we propose a taxonomy of embedded loop structures classified according to three properties:

- timing guarantees
- data-flow granularity
- control criticality

This taxonomy provides a principled basis for computational partitioning between edge devices and remote execution environments. In particular, it distinguishes tasks that require strictly bounded response times from those that can tolerate network latency and temporal

nondeterminism. As a result, the taxonomy enables the identification of computational components that can benefit from cloud offloading without violating system-level correctness constraints.

5.1 Hard Real-Time Control Loops

Hard real-time embedded systems implement tightly coupled sensing, computation, and actuation pipelines where each iteration must complete within strict timing bounds. A typical structure is:

```
while (1) {
  sensor = read();
  state = process(sensor);
  decision = control(state);
  actuate(decision);
}
```

These systems operate under deterministic deadlines and are often deployed in safety-critical environments. The control pipeline typically contains minimal buffering and cannot tolerate unpredictable latency introduced by network communication or remote execution.

Typical examples include automotive braking systems, airbag controllers, and motor control firmware in automotive electronic control units, as well as flight-control systems in avionics platforms. These systems are governed by stringent functional safety and certification standards such as ISO 26262 for automotive systems and DO-178C for avionics, which impose strict deterministic timing and correctness requirements [5, 6].

Such workloads are therefore **not suitable for remote offloading**.

5.2 Soft Real-Time Streaming Loops

Many modern IoT applications continuously process streams of sensor data while tolerating moderate timing variability. These workloads often follow a streaming loop pattern:

```
while True:
    frame = read_sensor()
    features = preprocess(frame)
    inference = model(features)
    local_actuation(inference)
```

Unlike hard real-time loops, these applications can tolerate limited jitter or occasional frame drops without compromising system correctness. As a result, computationally expensive stages—particularly machine learning inference—can be offloaded to more powerful compute resources, a design commonly explored in edge computing systems [7].

Typical examples include video analytics pipelines based on computer vision frameworks such as OpenCV [8], EEG and ECG monitoring systems used in brain–computer interfaces and biomedical signal processing [9, 10], audio recognition tasks based on deep learning models [11], and wearable health monitoring devices that integrate sensing with remote analysis [7].

5.3 Batch-Iterative Processing Loops

Some embedded analytics workloads accumulate sensor measurements over a fixed time window before processing them as a batch.

```
while True:
    batch = collect_window(T)
    results = process(batch)
    upload(results)
```

These batch-oriented loops are common in signal processing and edge analytics applications, where windowed aggregation and feature extraction are applied to time-series data [12]. Since processing occurs on aggregated data rather than individual samples, these workloads often have relaxed timing constraints and are well suited for remote execution.

Representative examples include medical imaging analysis pipelines based on frameworks such as MONAI [13], energy monitoring systems that aggregate and analyze consumption data over time (e.g., Emoncms) [14], and predictive maintenance platforms in industrial IoT environments that rely on batch processing of sensor histories [15].

5.4 Supervisory and Orchestration Loops

A final class of loops is responsible for system monitoring and orchestration rather than direct sensor processing.

```
while True:
    monitor_system()
    schedule_tasks()
    sync_with_cloud()
```

These loops operate at the control-plane level and typically involve relatively low sensor bandwidth. They coordinate device behavior, schedule workloads, and synchronize system state with remote services, following principles commonly studied in distributed and cloud systems [16, 17].

Examples include fleet-management systems that coordinate distributed devices, over-the-air (OTA) update controllers implemented in frameworks such as Eclipse hawkBit [18], and vehicle route-planning services based on engines such as GraphHopper [19]. Because these tasks naturally interact with remote infrastructure, they are also good candidates for selective offloading.

5.5 Execution Epochs

Although embedded applications are commonly expressed using infinite control loops [3], in practice their execution proceeds in discrete sensing and processing cycles. Each cycle corresponds to a sensor sample, an input frame, or a batch of collected measurements.

Aégis leverages this structural property by segmenting continuous execution into *bounded iteration windows*, referred to as *execution epochs*. An execution epoch represents a finite sequence of loop iterations extracted from the infinite control loop of the application.

This abstraction allows the compiler to treat each iteration as an independent computational unit that can be scheduled either locally or remotely within a secure enclave. When loop

iterations do not exhibit loop-carried dependencies, multiple iterations within an epoch can be executed concurrently across several enclave nodes [16].

By decomposing infinite control loops into bounded execution epochs, *Aégis* aligns embedded workloads with the task-oriented execution model of trusted execution environments and cloud infrastructures, enabling secure and scalable remote execution.

6 Background

6.1 Cloud Computing

Cloud computing has emerged as a fundamental paradigm for delivering scalable and on-demand computational resources over the internet [17]. By leveraging large-scale data centers, cloud providers such as Amazon Web Services (AWS) [20], Microsoft Azure [21], Google Cloud [22], and IBM Cloud [23] offer virtually unlimited processing power, storage, and networking capabilities. This model enables applications to offload compute-intensive tasks from resource-constrained devices to powerful remote servers, improving performance and scalability. At the same time, cloud computing supports flexible deployment models, elasticity, and pay-as-you-go pricing, making it a popular choice across a wide range of domains, from data analytics to machine learning and distributed systems.

6.2 Trusted Execution Environments (TTEs)

Trusted Execution Environments (TEEs) provide hardware-based isolation mechanisms that enable secure execution of code and protection of sensitive data, even in the presence of a compromised operating system or hypervisor. Prominent TEE technologies include Intel SGX, which offers enclave-based memory isolation, ARM TrustZone, which separates execution into secure and non-secure worlds, and newer solutions such as Intel TDX that extend these guarantees to virtualized environments. Through memory encryption, integrity protection, and strict access controls, TEEs ensure confidentiality and integrity of computations, preventing unauthorized access or tampering from privileged software layers. These properties make TEEs a strong foundation for secure computation offloading in untrusted cloud infrastructures.

Intel SGX An enclave is a protected area of execution in memory provided by Intel SGX [24] that isolates code and data from the rest of the system, including privileged software such as the operating system or hypervisor. Enclaves operate within the CPU, and all memory accesses to enclave pages are transparently encrypted and integrity-protected by the SGX hardware. This ensures that data inside the enclave cannot be read or modified by any external process. Communication between the enclave and the untrusted application happens via well-defined interfaces:

- **ECALLs (Enclave Calls):** These are calls made from the untrusted application into the enclave. They allow the application to request services or computation from the protected code.
- **OCALLs (Outside Calls):** These are calls made from the enclave to the untrusted application, typically to perform operations that cannot be done inside the enclave (e.g., I/O operations, network communication).

The security guarantees provided by enclaves include:

- **Confidentiality:** Data inside the enclave is encrypted in memory and cannot be accessed by any external entity.
- **Integrity:** Any attempt to tamper with the enclave code or data is detected by hardware checks.
- **Isolated execution:** Code inside the enclave runs in an isolated CPU execution environment, separate from the OS or other applications.

Furthermore, Intel SGX supports a remote attestation mechanism [24] that allows a remote party (the verifier) to confirm that a specific piece of code is running inside a genuine enclave on a trusted platform. The process begins with the enclave generating a cryptographic measurement (known as `MRENCLAVE`), which uniquely identifies the enclave's code and initial state. The enclave then creates a *quote*, a signed data structure that includes this measurement along with additional metadata. This quote is signed using a platform-specific attestation key provisioned by Intel, ensuring its authenticity. The quote is sent to the verifier, typically via the Intel Attestation Service (IAS) or through Data Center Attestation Primitives (DCAP) in newer deployments. The verifier checks the signature to ensure it originates from a genuine SGX-enabled platform and validates the enclave measurement against an expected value. If the verification succeeds, the verifier gains assurance that the intended code is executing inside a secure enclave and has not been tampered with. This enables the secure provisioning of secrets (e.g., cryptographic keys) directly to the enclave, establishing a trusted execution channel even over untrusted networks.

6.3 Other Approaches

Besides hardware-based enclaves like Intel SGX, there exist other methods for protecting computation and sensitive data on untrusted systems:

- **Oblivious RAM (ORAM):** A technique that hides memory access patterns to prevent information leakage from the way data is accessed [25, 26, 27].
- **Homomorphic Encryption:** Allows computation directly on encrypted data without decrypting it, preserving confidentiality [28, 29, 30].
- **Secure Multi-Party Computation (MPC):** Enables multiple parties to jointly compute a function over their inputs while keeping those inputs private [31, 32, 33].
- **Trusted Platform Modules (TPM):** Hardware-based cryptographic modules providing secure key storage and attestation capabilities [34, 35, 36].

While these approaches offer strong security guarantees, they often come with higher computational or communication overhead compared to SGX enclaves. Enclaves provide a practical trade-off, enabling efficient, isolated execution of code while maintaining confidentiality and integrity.

7 Threat Model

Based on the TEEs and SGX mechanisms described above, we assume the following threat model: the execution environment on the remote server–including the operating system, hypervisor, and any privileged software–is untrusted. These components may behave maliciously and attempt to inspect, tamper with, or interfere with the execution of offloaded tasks. In particular, the adversary can observe all network communication, control scheduling, and access system memory outside the enclave. To mitigate these threats, we rely on hardware-based Trusted Execution Environments (TEEs), specifically Intel SGX enclaves, which provide isolated execution and protect the confidentiality and integrity of code and data within the enclave. Communication between the client and enclave is secured via remote attestation and encrypted channels, ensuring that sensitive data is only accessible to trusted enclave instances. We assume that the processor and SGX hardware are trusted, and that cryptographic primitives used for secure communication are correctly implemented. We do not consider physical attacks, such as side-channel attacks (e.g., cache timing, power analysis), denial-of-service attacks, or rollback attacks. These remain outside the scope of this work.

8 Challenges

8.1 Offloading cost and granularity

Selective offloading is not universally beneficial. Intuitively, offloading a computation is advantageous only when the total remote execution cost is lower than local execution time. This can be expressed as:

$$T_{\text{offload}} = T_{\text{enc}} + T_{\text{net}} + T_{\text{remote}} < T_{\text{local}}$$

where T_{enc} denotes the overhead of data encryption and serialization, T_{net} captures network latency and transmission time, and T_{remote} is the execution time within the remote enclave.

Additionally, determining the appropriate granularity of offloading remains non-trivial. In many real-world applications, computational workflows are structured around black-box functions, often imported from external libraries, that encapsulate a complete processing step (e.g., inference, signal transformation, or feature extraction). These functions are typically invoked within larger control flows and often dominate the execution cost of the application.

However, it is not always straightforward to decide whether to offload the imported function itself or the surrounding wrapper. Profiling at the granularity of the wrapper function can be misleading, as it aggregates both computationally intensive operations and lightweight auxiliary logic such as data preparation or control flow. As a result, it may overestimate the amount of useful work being offloaded and underestimate the communication overhead introduced by transferring additional data.

This challenge calls for a more fine-grained analysis that can isolate the true computational hotspots within a function. By identifying the minimal computation-heavy region, the system can reduce unnecessary data movement while preserving most of the performance gains. To this end, *Aégis* is designed to support fine-grained offloading guided by the developer's ammotations, enabling the selective extraction and remote execution of only the most performance-critical components rather than coarse, wrapper-level units. This design choice improves the

communication-to-computation ratio and leads to more efficient and scalable offloading decisions.

8.2 TEE memory limitations

Trusted Execution Environments (TEEs) impose strict resource constraints, particularly in terms of memory capacity, which limits the size and complexity of computations that can be executed remotely. However, this limitation is often less restrictive in practice. Embedded applications are inherently designed to operate under tight resource budgets, both in terms of memory and compute, and typically process real-world sensor data that arrive in bounded, streaming, or windowed forms rather than as large monolithic datasets. As a result, individual computation units are naturally small and self-contained. Furthermore, offloading is performed at the level of execution epochs or per-iteration tasks, which further bounds the memory footprint of each remote invocation. Together, these properties align well with the constrained environment of TEEs, mitigating the impact of their limited memory capacity.

8.3 SGX programmability

Although Intel SGX provides string isolation guarantee, developing SGX applications and offloading computation to enclaves requires the developer to manage a complicated workflow as explained below.

First, the developer must explicitly **define the enclave interfaces.** This involves writing an **EDL** (Enclave Definition Language) file that declares all ECALLs (functions invoked from the untrusted application into the enclave) and OCALLs (functions invoked from the enclave to thr untrusted host). After modyfing the EDL file, the SGX toolchain must be invoked to aytomatically generate the correspondig trusted and untrusted proxy code that implements the interfaces.

Second, the developer must **separate all application into trusted and untrusted components and maintain two distinct codebases**. The trusted part, which runs inside the enclave, is subject to SGX restrictions: it cannot perform system calls, access files or the network directly, or allocate memory dynamically in an unrestricted manner. Only a limited set of operations is allowed, and any interaction with the outside world must be delegated to the untursted host through OCALLs.

Third, all data exchange accross the enclave boundary must be marshalled manually. Arguments passed to ECALLs and return values must be serialized into plain buffers before entering or leaving the enclave, copied across the boundary, and deserialized on the other side. The developer must carefully manage memory ownership, buffer lifetimes and data layout to avoid leaks, corruption, or undefined behavior.

Fourth, secure communication attestation must be implemented by the developer. To establish trust that the remote code is executed within a genuine enclave, the developer must implement remote attestation (e.g via Intel Attestation Service or DCAP) and verufy the enclave identity and measurement. After the attestation, a secure session must be established, including key exchange and the setup of encrypted channels that provide confidentiality, integrity and protection against replay or tampering.

Finally, the system must be **built and deployed.** The enclave code must be compiled into an enclave binary, signed and packaged together with the untrusted host application. The developer is responsible for deploying both components to the server, configuring the environment

(SGX drivers, libraries, dependencies), and ensuring that enclave instances are correctly initialized and available for remote execution.

In summary, developers must explicitly define enclave interfaces through EDL files, partition code between host and enclave components, marshal data across enclave boundaries, and manage enclave lifecycle and deployment. When SGX is used as the basis for distributed offloading, these responsibilities are further compounded by the need to integrate networking, request serialization, dependency management, and secure communication. Consequently, application logic becomes tightly coupled with low-level SGX infrastructure, increasing development complexity and reducing portability. This motivates the need for higher-level abstractions that allow developers to express what should be offloaded while hiding how secure remote execution is performed.

9 System's Design

9.1 Design Goals

The design of *Aégis* is guided by the challenges outlined in the previous sections, along with the the need for strong data confidentiality guarantees, and the overheads associated with remote execution. Based on these constraints, the system is designed to satisfy the following goals:

- **Performance:** Improve application throughput by offloading computationally intensive tasks and exploiting parallelism across multiple enclave nodes.
- **Scalability:** Support concurrent execution across multiple enclaves, enabling the system to scale with available secure resources.
- **Security:** Ensure that sensitive data remain protected during offloading by leveraging trusted execution environments and secure communication mechanisms.
- **Transparency:** Minimize developer effort by providing a high-level programming model that abstracts away low-level details of remote execution.

9.2 System's Overview

Figure 1 shows the high-level architecture of the system. The developer provides annotated application code to the *Aégis* DSL compiler, which transforms it into an equivalent execution flow that partitions the program into local and remote tasks. As part of this transformation, the compiler marks functions that will be offloaded as *remote* and makes their invocation asynchronous. The dispatcher will later wrap marked functions into proxies, abstracting the details of remote invocation and allowing them to be invoked as if they were local calls.

The client-side QuickJS runtime executes the transformed program locally until a *remote* directive is encountered. At that point, the runtime fetches the necessary input data along with any required dependency modules and constructs a remote execution request. This request is encrypted and transmitted over a dedicated channel to a pre-assigned server instance.

Upon reception, a request handler in the untrusted part of the server forwards the encrypted payload to the enclave. Inside the trusted execution environment, a specialized runtime decrypts the request and evaluates the target function in isolation. Once execution completes, the results

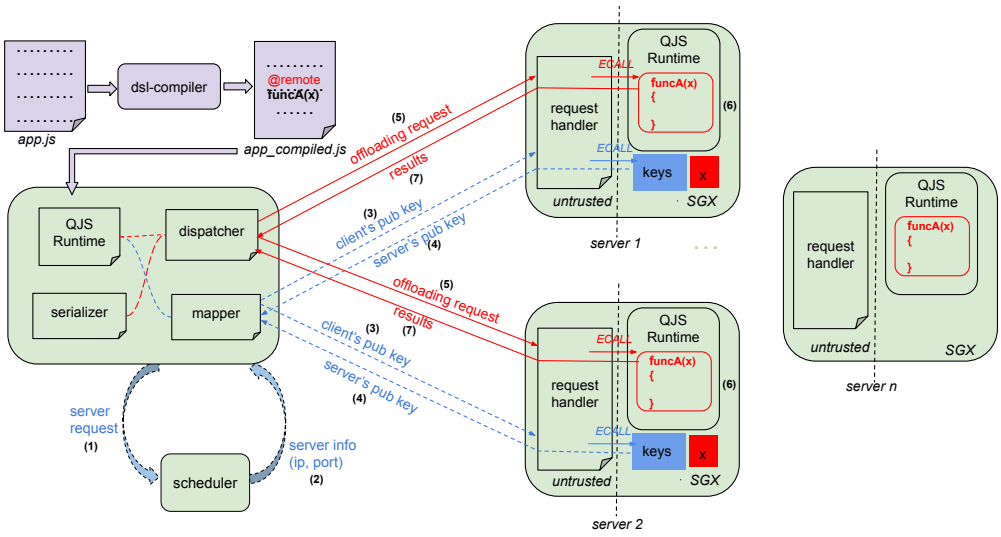

Figure 1: Overview of *Aégis* . The DSL compiler transforms annotated code into a partitioned execution model with proxy-based remote invocations. The client runtime offloads tasks to enclave-backed servers, where computations are executed securely and results are returned to the client.

are encrypted within the enclave and securely returned to the client, where they are reintegrated into the application's execution flow.

9.2.1 DSL and Program Transformations

The *Aégis* DSL compiler performs a source-to-source transformation that converts annotated loop-based computations into a partitioned execution model suitable for secure offloading. The transformation is driven by developer-provided annotations that identify computationally intensive functions within loop bodies as candidates for remote execution.

At a high level, the compiler targets canonical `for` loops with a statically bounded iteration count. Given such a loop, the compiler extracts its body and lifts it into a separate asynchronous function. The original loop is then replaced by a runtime call that orchestrates multiple invocations of the generated function.

Within the lifted function, calls to annotated routines are rewritten into asynchronous proxy invocations by introducing `await` expressions. These proxies abstract the details of remote execution, allowing offloaded functions to be invoked transparently as if they were local. The enclosing function is marked as `async` to preserve correct control-flow semantics.

This transformation decomposes a sequential loop into a set of execution units that can be scheduled independently. Depending on the annotation provided by the developer, the compiler applies one of three distinct execution modes:

Parallel mode (`--parallel`): This mode is used when loop iterations are independent and operate over identical input structures. The compiler removes the loop and delegates iteration control to the runtime via `scaleout_traffic`, which dispatches requests concurrently across multiple enclave nodes. Each invocation corresponds to the same computation replicated over different iterations, enabling *task-level parallelism* (also referred to as embarrassingly parallel execution).

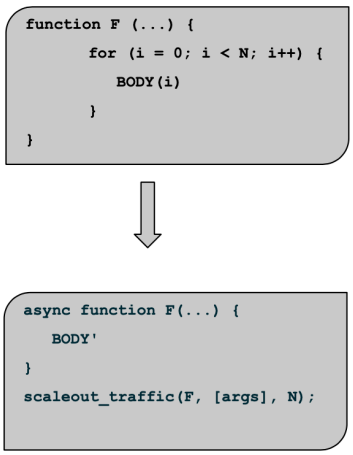

Figure 2: Parallel Transformation (*–parallel* flag)

Batch mode (`--batch`): This mode applies when each iteration consumes different inputs derived from the loop index. The compiler extracts the loop body into a parameterized asynchronous function and materializes the iteration space as an explicit task list. Loop-varying variables, as well as any local variables required by the computation, are lifted and passed as explicit function parameters. The runtime then distributes these tasks across nodes using `scaleout_map`. By explicitly enumerating inputs, this mode exposes *data-level parallelism*, where each task operates on a different partition of the input domain. This enables efficient distributed execution over heterogeneous data while preserving per-iteration semantics.

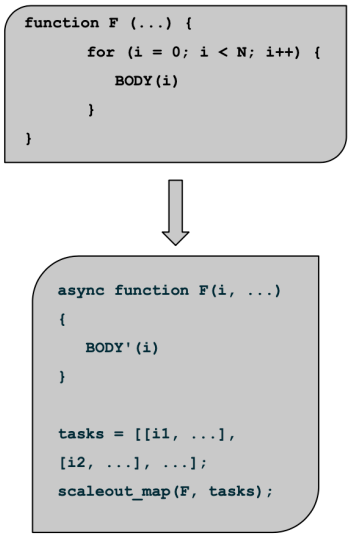

Figure 3: Batch Transformation (*–batch* flag)

Default mode (no annotation): When no execution mode is specified, the compiler conservatively preserves the original control flow and wraps the computation using `create_traffic`, ensuring sequential semantics:

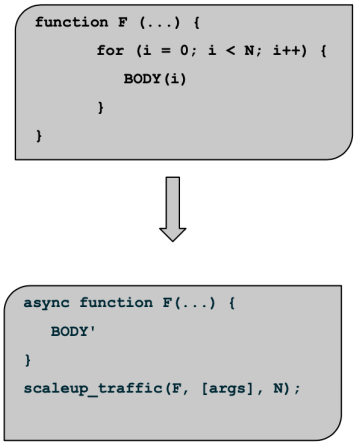

Figure 4: Default transformation (no flag)

Correctness Constraints. To ensure correctness, the compiler assumes a set of structural constraints on annotated loops, including: (i) canonical loop form, (ii) statically known iteration bounds, (iii) absence of control-flow interruptions such as `break` or `continue`, and (iv) no loop-carried dependencies. These restrictions guarantee that loop iterations can be safely executed in isolation.

In addition to control-flow transformation, the compiler performs a static dependency analysis to identify all functions and modules required for remote execution. A call graph is constructed and traversed to resolve transitive dependencies of each offloaded function. The resulting dependency set is serialized and packaged together with the transformed code into a self-contained execution unit. To reduce runtime overhead, this metadata is cached locally and may be precompiled into bytecode, allowing the system to transmit compact representations instead of source code during offloading invocations.

9.2.2 Offloading Engine

The Offloading Engine is responsible for orchestrating computation tasks across both local and remote nodes. When a function is called, the engine intercepts the call and determines whether it should be executed locally or offloaded to a remote worker. For offloaded tasks, it constructs a request message that includes serialized function arguments, all necessary dependencies, and import statements, ensuring the worker has the complete execution context. To support this, the engine first reads a hashed metadata file generated by the compiler, which contains precomputed information about the functions, their offloading annotations, dependencies, and source code. This metadata is used to populate internal structures such as the offloads set, function-module dependency map, and source code raw text, enabling accurate and efficient task preparation without having to perfrom program flow analysis at runtime. Each request is assigned a unique sequence number (nonce) to guarantee that only the corresponding response is accepted, preventing misordered or replayed results. To support asynchronous execution and parallelism, function calls are wrapped into promises, allowing local code to continue running while remote tasks are processed. Once a worker completes a task, the engine validates the response, and resolves the associated promise, seamlessly returning the result to the original function call. Through this mechanism, the Offloading Engine efficiently manages task queueing, execution,

and result handling across a distributed system, while enabling dynamic scaling across multiple workers.

9.2.3 Handshakes & Dedicated Connections

Before offloading any computation, the client establishes secure connections with each assigned server. This process begins with a handshake for each node, during which the client and server exchange public keys and derive a shared session-specific encryption key. These keys are used to encrypt all subsequent communication, ensuring end-to-end confidentiality. Each client maintains a dedicated socket per server, which persists across multiple task offloads, reducing connection overhead and improving throughput.

Unlike full SGX remote attestation workflows (e.g., IAS or DCAP), this handshake mechanism does not perform cryptographic verification of enclave identity. Instead, it establishes a secure communication channel based on key exchange, assuming trust in the initial deployment of the enclave.

Once the session is established, the runtime on the server side (inside the enclave) can safely execute client-supplied scripts with preloaded dependencies and return encrypted results. This handshake and dedicated connection mechanism complements the Offloading Engine's scheduling logic: each request carries a unique nonce to prevent replay attacks, and the persistent connections allow the client to efficiently schedule multiple tasks across nodes while maintaining secure, ordered communication.

Connections are only closed when the client completes its workflow or explicitly releases the allocated servers, at which point session keys are discarded and sockets are terminated.

9.2.4 Customized Interpreter

The interpreter has been extended to incorporate network communication, encryption, and distributed task management, enabling secure and efficient interaction between clients and servers. A comprehensive networking layer allows scripts to create and manage TCP connections, handle incoming and outgoing connections, and transmit data across remote nodes. Large payloads are sent and received in chunks, ensuring reliable communication without exceeding buffer limitations. Cryptographic enhancements include secure key storage, management of public and private keys, and the derivation of session-specific encryption keys. The system supports both symmetric and asymmetric encryption for payloads, while concurrent access to cryptographic state is safeguarded through mutex-based locking to prevent race conditions during parallel communication. Concurrency support extends to shared resources, ensuring that multiple asynchronous offload tasks can execute without compromising security or data integrity. Modules and scripts are tracked and serialized, allowing for dynamic offloading and remote execution while maintaining consistency across nodes. These capabilities are fully integrated into the scripting environment, providing seamless access to networking, encryption, and distributed execution features through the interpreter's standard interfaces. Security is emphasized throughout the design: each node maintains a unique identifier, all communications are encrypted, sequence numbers and nonce-based encryption prevent replay attacks, and mutex-protected state ensures consistent message ordering and thread-safe operations.

9.3 Server-side Components

9.3.1 Trusted App

The Trusted App runs inside the enclave and provides a secure, isolated environment for evaluating client-supplied scripts. It uses a custom, lightweight QuickJS interpreter with many non-essential JavaScript functions removed to reduce memory footprint, focusing only on the lower-level logic operations common in embedded applications. Client scripts are sent encrypted and decrypted within the enclave using session-specific AES keys. Once decrypted, the code is evaluated directly from a string in a global QuickJS context (quickjs_enclave_ctx), which persists across multiple client requests. Dependencies required by the scripts—such as preloaded modules or helper functions—are read and cached into the enclave memory the first time they are needed. These dependencies are stored in global state and are not cleared between executions, allowing subsequent requests to reuse them efficiently and reducing initialization overhead. Only when a client disconnects is the runtime completely wiped, including clearing the cached dependencies and session keys. Execution results are collected in a response buffer and re-encrypted before being sent back to the client, ensuring end-to-end confidentiality. The enclave exposes controlled ECALL interfaces for initialization, key exchange, code execution, and session management, while using OCALLs for non-sensitive operations such as logging, networking, and I/O. This design ensures ephemeral, secure execution with reusable runtime state for improved performance.

9.3.2 Untrusted App

The untrusted application acts as the intermediary between network clients and the secure enclave, managing all communication, decryption, and lifecycle events. Incoming requests are first received at the network layer and forwarded as raw byte streams to the enclave, where they are securely decrypted and processed under attestation guarantees. Each payload is prefixed with a length indicator, allowing the application to determine its size before passing it to the enclave. The untrusted app also establishes a secure handshake by exchanging public keys with clients and deriving shared session keys for encrypted communication. It maintains the integrity of the JavaScript runtime environment by bootstrapping it per client while caching necessary global state to optimize performance. Upon client disconnection, the application ensures sensitive memory within the enclave is wiped, closes the associated network socket, and releases the client's entry from the scheduler to free resources. This design ensures that even untrusted components handle sensitive data safely while supporting efficient, secure interaction with multiple clients.

9.4 Scheduler

The scheduler is responsible for allocating server resources to clients in a controlled and efficient manner. It maintains a list of all registered servers, including their network addresses and availability status, and assigns servers to clients on a round-robin basis to balance load. When a client request arrives, the scheduler traverses the server list starting from the last assigned position, selecting the requested number of available servers. If fewer servers are free than requested, the scheduler returns the maximum number it can assign along with an informative status message. Both clients and servers support primitives for secure mapping and handshake with the enclave, ensuring that each assignment respects protocol requirements. The scheduler also tracks active

client sessions, releasing server resources when clients disconnect to maintain accurate state and avoid resource conflicts. This lightweight design provides a reliable mechanism for managing distributed execution while minimizing overhead.

10 Evaluation

10.1 Benchmark Suite

This benchmark suite is designed to evaluate computational workloads representative of embedded, mobile, and robotics applications. There are no full open-source projects available that exactly match these workloads; instead, only datasets, libraries, or algorithmic components exist that encode aspects of these behaviors. Given these resources, and inspired by real-world projects and operational patterns, we constructed synthetic workloads that capture the core computational and memory characteristics of their domains. Although synthetic, the benchmarks are motivated by real workloads:

1. Operational realism: Each benchmark captures the dominant computation, memory access, and parallelization patterns seen in deployed systems.

2. Parameterization: Variables like batch size, grid resolution, and obstacle density allow exploration of performance under realistic operating conditions.

3. Compute and memory scaling: Benchmarks are structured to stress CPU-bound, memory-bound, or mixed workloads, reflecting trade-offs present in embedded and robotic platforms.

This methodology allows precise control, repeatability, and systematic evaluation of performance optimizations or offloading strategies, while remaining faithful to the behaviors observed in actual systems, even if no full open-source implementations are available.

HMAC-Based Secure Health Data Processing
This benchmark models a wearable health device that continuously collects sensitive physiological data, including heart rate, SpO_2, and high-frequency ECG signals. Data are aggregated into fixed-size batches (10 seconds at 500 Hz), resulting in approximately 5,000 ECG samples per payload, along with associated metadata.

Each batch is serialized and authenticated using HMAC-SHA512 to ensure integrity and authenticity before transmission or storage. The HMAC computation dominates execution cost and is therefore selected as the offloading target.

This workload falls under the *batch-iterative processing loop* category, where data are first accumulated over a time window and then processed as a unit, making it well-suited for selective offloading.

Adaptive Sensor Data Smoothing
This benchmark models a streaming sensor processing pipeline where noisy measurements are continuously filtered using a sliding-window median. At each iteration, a batch of sensor readings is acquired and incorporated into a fixed-size buffer (window size 1024), simulating real-world scenarios such as environmental monitoring or industrial sensing.

The core computation applies a median filter over the sliding window, implemented via a sorting routine (Gnome sort), which dominates the execution cost due to its quadratic time complexity. This makes the smoothing function (`processBatch`) a natural candidate for offloading.

The input size is representative of embedded workloads: data arrive incrementally in small batches, while the maintained window remains bounded. Such designs are common in edge systems to balance responsiveness with noise reduction under limited memory.

This workload falls under the *soft real-time streaming loop* category, where continuous data streams are processed with moderate latency tolerance, allowing selective offloading of computationally intensive stages.

Minimum Edit Distance for Streaming Log Analysis

This benchmark models a streaming log-processing application where sequences of sensor or device status messages are continuously generated and compared. Each iteration constructs two sequences of length 100, introducing small perturbations to simulate anomalies, and computes the minimum edit distance using a dynamic programming matrix of size $n \times m$ The computation dominates execution time due to its $O(n \cdot m)$ omplexity, while memory usage remains modest but proportional to the sequence lengths.

The workload is representative of embedded analytics pipelines that perform incremental anomaly detection or similarity scoring on streaming text data. Structurally, this application follows a soft real-time streaming loop pattern: sequences are processed continuously with moderate latency tolerance, allowing selective offloading of the dynamic programming computation to secure enclaves without affecting overall system responsiveness.

Temperature Prediction Using Decision Trees

This benchmark represents an embedded analytics application that predicts future temperature values from historical sensor readings. Each execution epoch generates a batch of 1,024 time-series samples and trains a decision tree regressor with a maximum depth of 5 over these inputs. Predictions are then made for the subsequent 1,024 time steps. The computation is dominated by the tree training, which scales with both the number of samples and tree depth, creating a significant performance bottleneck on resource-constrained devices.

The workload is representative of batch-oriented environmental or industrial monitoring pipelines, where historical measurements are aggregated before predictive processing. Due to the accumulation-then-process structure, this application belongs to the batch-iterative processing loop category, making it well-suited for selective offloading to secure enclaves while maintaining low-latency response for incoming sensor data.

K-Means Clustering for Sensor Data Aggregation

This benchmark models an embedded analytics application that clusters multi-dimensional sensor data in batches of 512 points with 3 features each. Each execution epoch performs K-Means clustering with $K = 20$ centroids and up to 50 iterations, dominated by the iterative centroid update computations. As K increases, the application transitions from being memory-bound to compute-bound: more centroids require storing larger centroid arrays and distance matrices, increasing memory footprint, while each iteration requires more distance calculations and centroid updates, raising CPU load. Such a scenario is realistic in embedded environments, for example in industrial IoT, environmental monitoring, or smart-city sensor networks, where large

numbers of clusters may be needed to differentiate fine-grained patterns or identify multiple simultaneous phenomena.

Structurally, the workload follows a *batch-iterative processing loop* pattern, with data accumulated over a batch and processed collectively. The modular nature of each iteration makes it suitable for selective offloading to secure enclaves, enabling edge devices to handle large clustering tasks without violating memory or latency constraints.

Batch Processing of Voice Commands for Embedded NLP

his benchmark simulates an embedded device that continuously collects voice commands during the day and processes them in batches during idle periods. Each batch contains $B = 50$ commands, which are analyzed to extract intents, detect out-of-scope utterances, and improve usage analytics. The core computation consists of tokenization, verb and noun normalization, and intent classification via a lightweight NLP pipeline. Execution complexity depends primarily on the batch size and the number of linguistic features extracted per command. While the memory footprint is modest due to short command lengths, CPU usage grows linearly with batch size and text length, making larger batches compute-bound. This scenario is realistic in smart home devices, wearables, or voice assistants, where accumulated voice commands need offline processing without degrading runtime responsiveness.

Structurally, the workload follows a *batch-iterative processing loop*, where commands are collected over a time window and processed collectively. The independence of commands within a batch makes the computation amenable to selective offloading to secure enclaves, preserving privacy while enabling high-throughput analysis.

NavFn Pathfinding for Grid-Based Navigation

This benchmark simulates a grid-based path planning system inspired by ROS navigation, where multiple start/goal pairs are processed to compute optimal routes. Each scenario consists of a pair of coordinates representing start and goal positions, which are analyzed using a potential field–based Dijkstra planner to generate paths and estimate travel costs. The core computation involves costmap setup, potential propagation, gradient calculation, and path extraction, with complexity primarily determined by grid size and the density of obstacles.

While the memory footprint scales linearly with the number of grid cells, CPU usage is dominated by iterative propagation and gradient computations, making dense maps or long paths compute-bound. This scenario is realistic in autonomous robots, UAVs, and mobile platforms, where real-time planning must balance accuracy and efficiency. Structurally, the workload follows a *Batch-iterative loop*, seeding goals, propagating potentials, and computing gradients for path extraction. The independence of grid cells during potential updates makes the computation amenable to selective parallelization, enabling high-throughput evaluation without compromising deterministic planning.

Dijkstra-Based Shortest Path Computation

This benchmark models an embedded route optimization application, such as a delivery drone or automated warehouse vehicle planning system, where multiple origin-destination pairs must be processed efficiently on resource-constrained hardware. The workload constructs a weighted graph with $V = 10{,}000$ vertices and a deterministic set of edges with variable weights, then computes shortest-path distances from a given source node using Dijkstra's algorithm. Each

iteration traverses the graph, updating tentative distances for unvisited vertices and propagating path costs through adjacency lists. The computation is dominated by repeated minimum-distance selection and edge relaxation operations, with modest memory requirements for the distance and visited arrays.

Structurally, this workload represents a *soft real-time streaming loop* pattern: the graph is initialized, distances are computed iteratively across all vertices, and results are collected over multiple iterations. This pattern allows selective offloading of the computationally intensive Dijkstra function to secure enclaves, enabling embedded devices to perform large-scale path optimization without exceeding memory or processing constraints. Unlike the ROS Navigation benchmark, which focuses on grid-based potential propagation, this scenario emphasizes graph traversal and combinatorial optimization over sparse or structured networks, reflecting typical workloads in autonomous logistics, route planning, or sensor network analysis.

Benchmark	Input Size	Module bytes	Payload bytes	Time	Space	Pattern
HMAC	5000 samples	131259.80 ± 4304.65	98642.43 ± 3272.57	O(n)	O(n)	Batch-iterative
GnomeSort	1024 window	9625.50 ± 6.02	8216.56 ± 7.51	$O(n^2)$	O(1024)	Soft real-time streaming
Edit Distance	100x100 seq	5556.40 ± 3.24	3998.95 ± 4.42	O(n·m)	O(n·m)	Soft real-time streaming
Decision Tree	1024 samples	621192.00 ± 518.49	33853.49 ± 955.50	O(n·d)	O(1024)	Batch-iterative
K-Means	512x3 pts	619917.20 ± 18.38	31247.91 ± 19.62	O(k·i·n·d)	O(n·k)	Batch-iterative
NLP	50 commands	423987.50 ± 396.57	4214.82 ± 417.21	O(batch·feat)	O(batch)	Batch-iterative
ROS Navigation	10x10 grid	18683.10 ± 0.30	213.08 ± 0.81	$O(n^2)$	$O(n^2)$	Batch-iterative
Dijkstra	10000 vertices	2458.00 ± 0.00	204.26 ± 0.55	$O(V^2 + E)$	O(V)	Soft real-time streaming

Table 1: Summary of benchmark workloads, including input sizes, computational and memory characteristics, and structural patterns for selective offloading.

10.2 Experimental Setup

All experiments were conducted in a controlled private network environment to ensure stable and reproducible performance measurements. The system was configured to use Intel SGX in hardware mode (SGX_MODE=HW), enabling execution within real trusted execution environments. The experiments were performed on a machine running **Debian GNU/Linux 13 (trixie)** with the Intel SGX SDK version **2.25.100.3**.

The server-side infrastructure consists of a single multiprocessor machine equipped with an Intel Core i7-10710U CPU. The processor features 6 physical cores and 12 hardware threads, with a maximum frequency of 4.7 GHz and a shared 12 MB L3 cache. The system is provisioned with 64 GB of main memory.

The processor supports Intel SGX, as indicated by the presence of the `sgx` CPU flag. The system exposes SGX device interfaces (`/dev/sgx_provision`, `/dev/sgx_vepc`), enabling enclave execution in hardware mode. The platform supports SGX1, and the available Enclave Page Cache (EPC) size is approximately 94 MB, as determined through CPUID-based enumeration. This value is slightly lower than the theoretical maximum of 128 MB for this processor generation due to system-level memory reservations.

The server machine runs multiple enclave-backed worker instances. In our configuration, four server instances are spawned, listening on ports 9000–9003. A centralized scheduler runs on the same host and listens on port 8000, assigning available enclave nodes to clients in a round-robin fashion.

The EPC is a globally shared hardware resource across all enclaves on the system. As a result, concurrently executing enclave instances compete for the same memory region. In our setup, where up to four enclave-backed servers are deployed simultaneously, the effective EPC capacity available to each enclave is reduced, increasing memory pressure. This can lead to EPC paging, where enclave pages are evicted to untrusted memory, introducing additional performance overhead.

10.2.1 Synthetic WAN Latency Model

All experiments were executed in a controlled private laboratory network to ensure stable and reproducible measurements. Consequently, the measured network latency is significantly lower than that experienced in typical cloud deployments. To estimate the performance of *Aégis* in a realistic edge-to-cloud scenario, we additionally report results using a synthetic WAN latency model.

Let the measured end-to-end latency be expressed as

$$L_{\text{lab}} = T_{\text{network,lab}} + T_{\text{server}} + T_{\text{enclave}}. \tag{1}$$

Here, $T_{\text{network,lab}}$ denotes the measured network transmission time (including communication and transport overheads), T_{server} represents the total server-side processing time, and T_{enclave} corresponds to the execution time inside the SGX enclave.

To approximate deployment over the public Internet, the measured laboratory network latency is replaced with a representative WAN latency while preserving the experimentally measured server and enclave execution times. The resulting latency is therefore computed as

$$L_{\text{WAN}} = L_{\text{lab}} - T_{\text{network,lab}} + T_{\text{network,WAN}}, \tag{2}$$

or equivalently,

$$L_{\text{WAN}} = L_{\text{lab}} + \Delta, \tag{3}$$

where

$$\Delta = T_{\text{network,WAN}} - T_{\text{network,lab}}. \tag{4}$$

Unless otherwise stated, $T_{\text{network,WAN}}$ is selected from representative client-to-cloud round-trip latencies (10–50 ms) reported for nearby public cloud regions. This model provides a sensitivity analysis of *Aégis* under realistic Internet deployment conditions while preserving the experimentally measured execution characteristics of the server and enclave.

10.3 Results

With the benchmarks fully described and their computational characteristics understood, we now evaluate their performance under various deployment configurations. Our analysis compares native execution against multi-server setups, highlighting execution time, throughput, and latency characteristics.

Specifically, the evaluation of *Aégis* answers two primary research questions and design objectives:

- **RQ1 (Performance Efficiency)**: examines to what extent *Aégis* can improve the performance of embedded applications through secure selective offloading to Trusted Execution Environments, focusing on key metrics such as execution time, throughput, and latency in comparison to native, local-only execution.
- **RQ2 (Scalability)**: investigates how the system's performance scales with the number of available enclave nodes across diverse workload types.
- **RQ3 (Overhead and Latency Breakdown)**: measures and analyzes the contributions of different components to the total execution time, including (i) network transmission (including encryption/decryption overheads) and (ii) enclave invocation (including encryption/decryption and secure processing), to understand how each factor affects the overall efficiency of secure offloading.
- **(Design Objective) Programmability and Transparency**: provides a qualitative comparison between the traditional SGX programming workflow and the *Aégis* programming model, highlighting how *Aégis* abstracts enclave interface management.
- **(Design Objective) Security Preservation**: confirms that *Aégis* maintains SGX security guarantees during remote execution. Sensitive code and data remain protected within enclaves, while communication between edge and cloud components is authenticated and encrypted.

10.3.1 RQ1: Performance Efficiency

The performance gains observed in Figures 5 and 6 are primarily due to the computational advantage of the server compared to the limited resources of the embedded device. These gains are naturally constrained by network transfer overheads, including encryption/decryption mechanisms, enclave attestation, and transitions between the untrusted and trusted spaces (e.g., system calls or context switches).

The average native execution latency ranges from 13.8 ms (editDistance) to 19.9 s (NLP), illustrating the large diversity in computational complexity across the evaluated benchmarks. Under secure remote execution, the average per-iteration latency ranges from 0.35 ms (Dijkstra) to 12.9 ms (KMeans), reflecting the substantially higher computational capability of the remote server.

Native throughput ranges from 0.0001 to 0.0726 iterations/ms, whereas secure remote execution achieves between 0.312 and 2.627 iterations/ms, corresponding to a substantial increase in processing capacity.

Notably, the first iteration of each offloaded task is significantly slower, as it includes the transmission of dependency modules to the enclave in addition to the actual workload. Subsequent executions benefit from cached modules within the enclave, resulting in faster completion times. Importantly, these initial delays are included in the calculation of average performance, yet they do not significantly affect the overall efficiency, as the performance of subsequent iterations dominates the average.

To evaluate the practicality of *Aégis* in realistic deployments, we repeat the latency analysis using the synthetic WAN latency model described above. Unless otherwise stated, we assume a 30 ms round-trip network latency, which is representative of communication between edge devices and geographically nearby public cloud regions. This value lies within the commonly

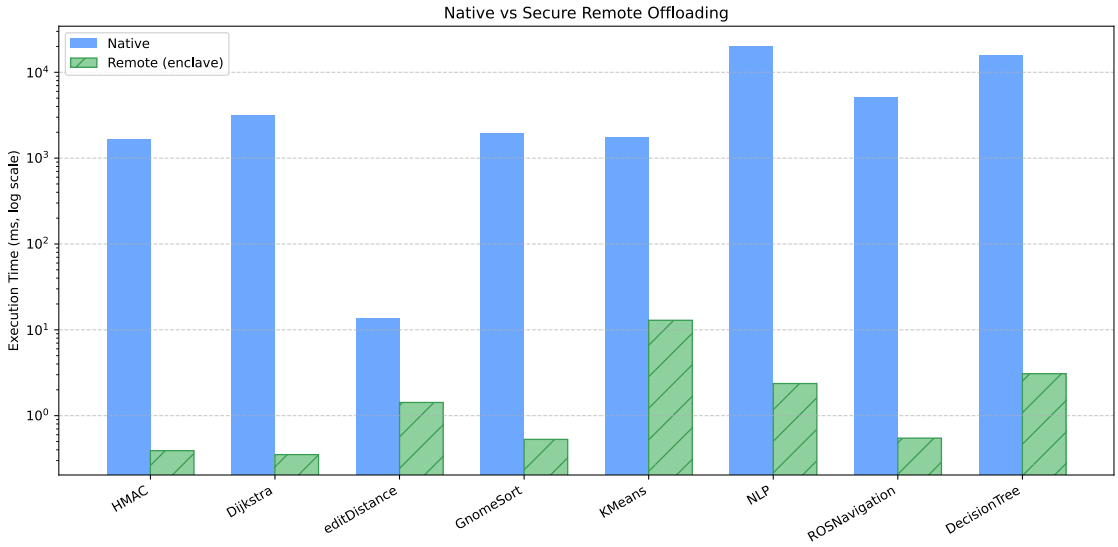

Figure 5: Average per-iteration latency comparison between native execution and secure offloading to a single server.

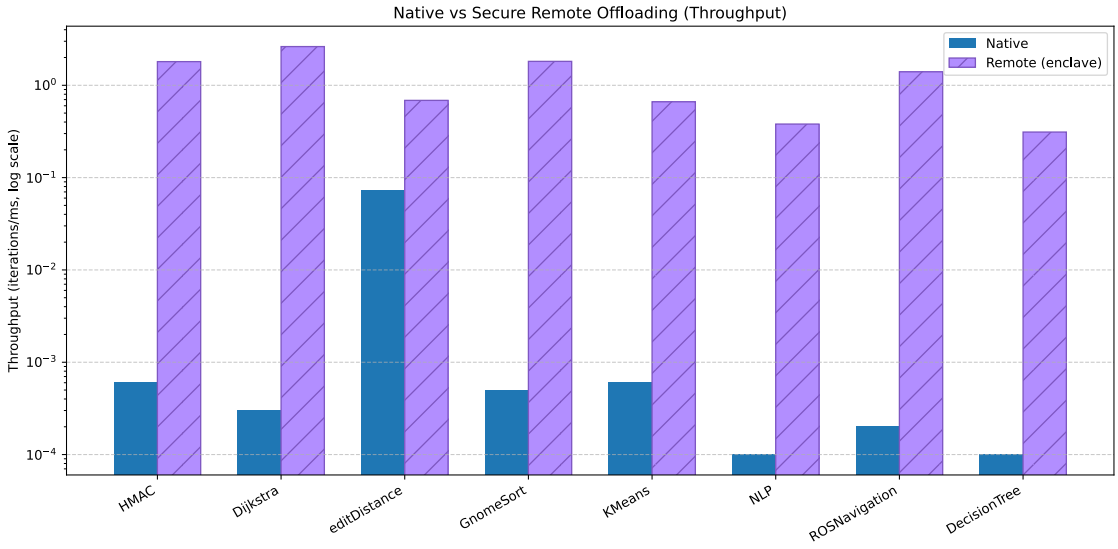

Figure 6: Throughput comparison between native execution and offloaded execution on a single server.

reported 10–50 ms RTT range observed in edge-to-cloud deployments and therefore represents a realistic operating point while avoiding assumptions specific to a particular cloud provider or geographic location.

10.3.2 RQ2: Scalability

Loop unrolling and the concurrent dispatch of subsequent iterations in a task-queue fashion further increase throughput. While adding more servers generally improves parallelism, it also introduces overhead from additional handshakes between the client and each enclave, as well

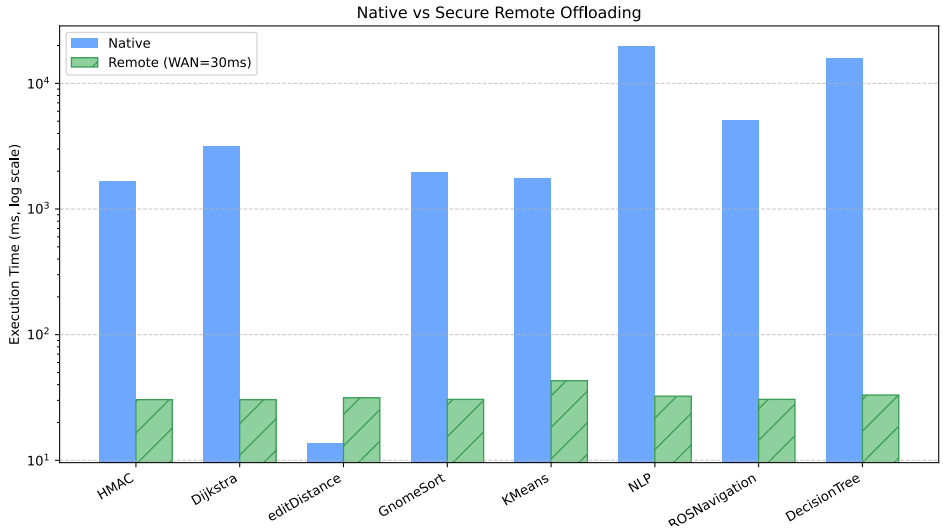

Figure 7: Average per-iteration latency under the synthetic 30 ms WAN model compared to native execution and laboratory measurements. The additional network delay increases end-to-end latency uniformly while preserving the relative performance trends across benchmarks.

as the transfer of dependency modules during the first invocations. These factors increase the initial latency per thread, causing the first iteration to be slower when more servers are involved. Despite this initial cost, the benefit of high data parallelism dominates, yielding substantial performance gains in multi-server deployments. The total speedup, shown in Figure 8, is calculated as the ratio of the total execution time of the offloaded loop—measured as the maximum of each request's start time plus its latency—to the execution time of the native, local-only implementation. Although the per-iteration speedup–Figure 9–decreases slightly as the number of servers grows due to intra-machine contention from multiple enclaves, the overall throughput–Figure 10–and efficiency still improve significantly with increased parallelism.

The achieved speedup varies considerably across workloads, with the largest improvements observed for compute-intensive applications that benefit most from cloud offloading. This behavior is expected, as the baseline platform is a resource-constrained embedded device while computation is offloaded to a significantly more powerful cloud server.

The scalability trends remain qualitatively unchanged under the synthetic WAN latency model, as the additional network delay affects all configurations equally. Therefore, only laboratory measurements are reported for scalability.

10.3.3 RQ3: Overhead and Latency Breakdown

Figures 11, 12, and 13 provide a detailed breakdown of latency across different server configurations and benchmarks. Figure 11 shows the distribution of per-iteration latencies, highlighting both the median performance and the variability across threads. Figure 12 decomposes the latency into three components: execution inside the enclave, overhead due to offloading, and network transfer time. Figure 13 focuses on tail latencies (P95 and P99), revealing the impact of rare, high-latency events on overall performance.

These figures illustrate that the dominant source of latency depends on the workload characteristics. For memory-bound applications, where large amounts of data are transferred to the

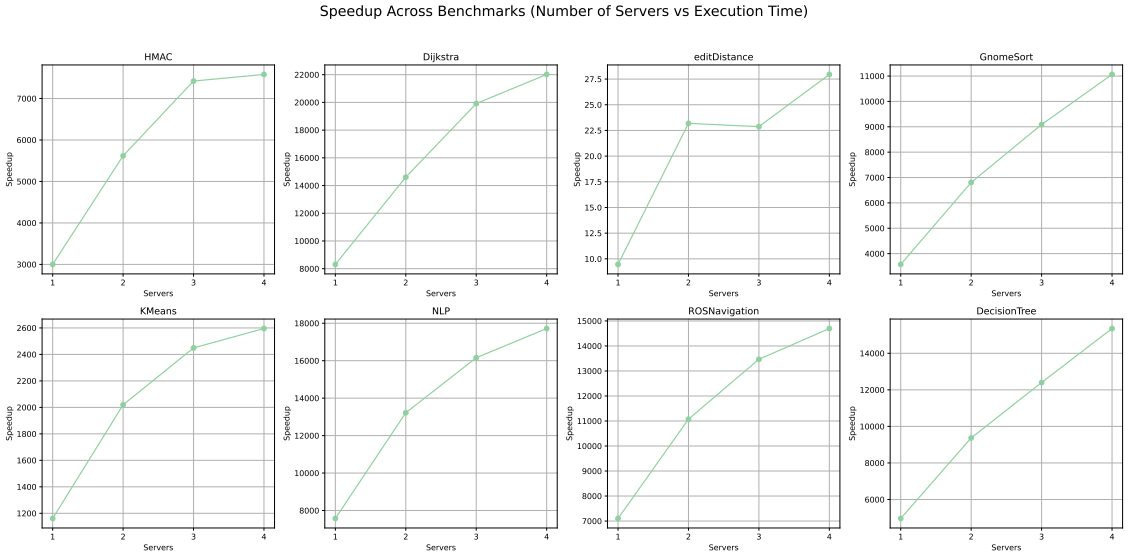

Figure 8: Total speedup of multi-server deployments compared to native execution. Speedup is calculated as the ratio of the total execution time (maximum start time plus latency) of offloaded tasks to native execution.

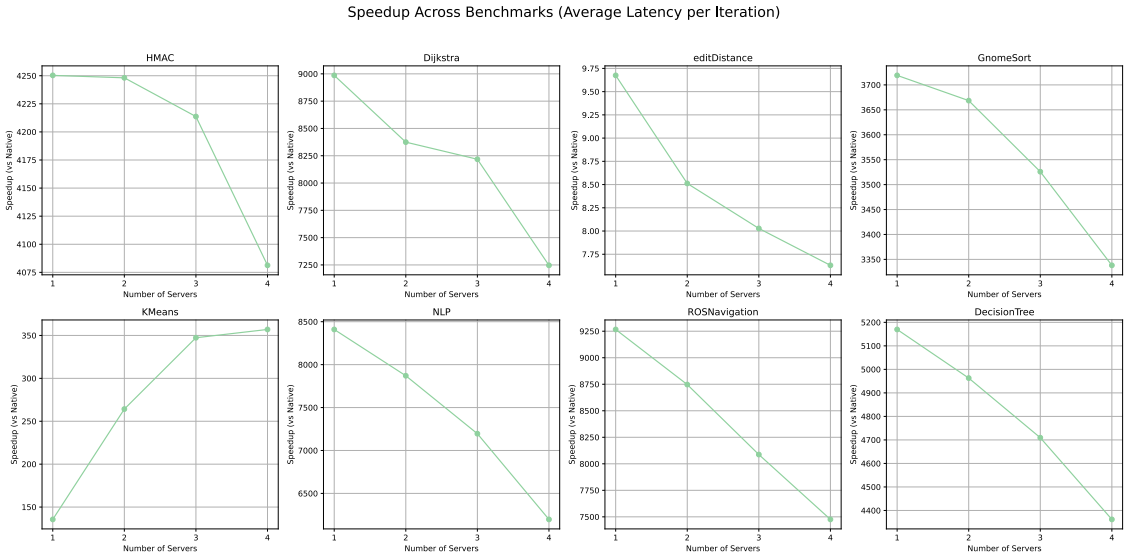

Figure 9: Per-iteration speedup as the number of servers increases. Speedup per iteration slightly decreases due to intra-machine contention from multiple enclaves created inside the same machine.

enclave, network transfer times dominate, especially for the first iteration when dependencies are loaded. Conversely, for compute-bound applications with intensive core computations, the enclave execution time is the largest contributor to total latency. Algorithm-specific parameters also play a crucial role: for example, in k-means, fewer coordinates reduce computational complexity and execution time, whereas increasing the number of cluster centers adds significant computation, increasing the execution component of latency. Enclave overheads, including secure context switches and cryptographic operations, associated with SGX attetsation remain

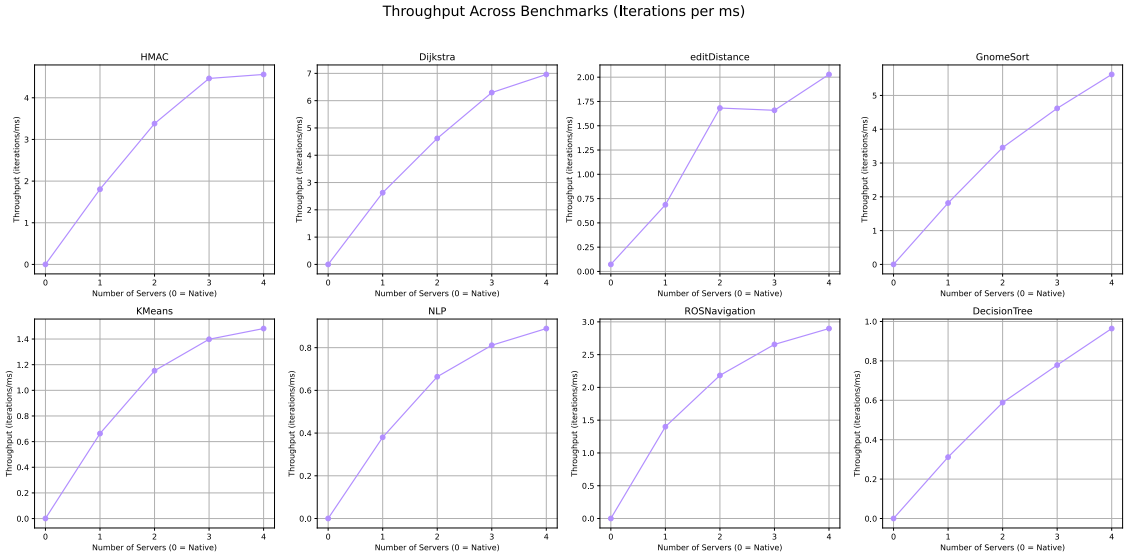

Figure 10: Overall throughput across different server configurations. Task parallelism and loop unrolling lead to significant throughput gains with multiple servers.

negligible compared to network and execution costs. Understanding these breakdowns is essential to achieving actual speedup. The system must balance offloading overhead, network costs, and core computation: if the network or initialization cost dominates relative to computation, offloading may not yield performance gains. Proper configuration of the algorithm and workload partitioning ensures that the benefits of parallel execution across multiple servers outweigh these overheads, enabling meaningful acceleration.

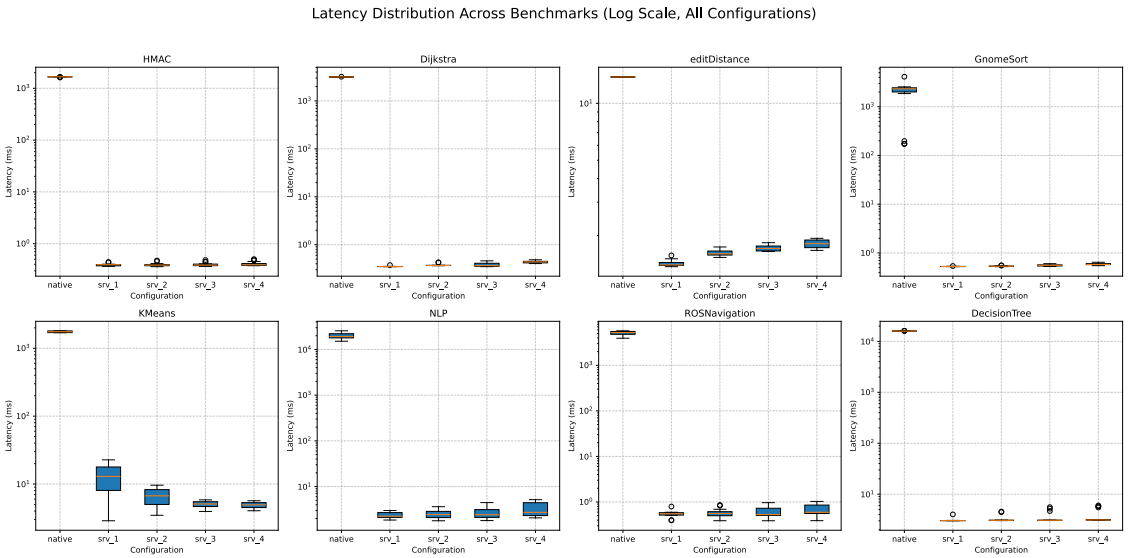

Figure 11: Distribution of per-iteration latencies across benchmarks and server configurations. Boxes show median and variability, highlighting both network- and computation-dominated workloads.

For completeness of the analysis and to evaluate the system behavior under realistic deploy-

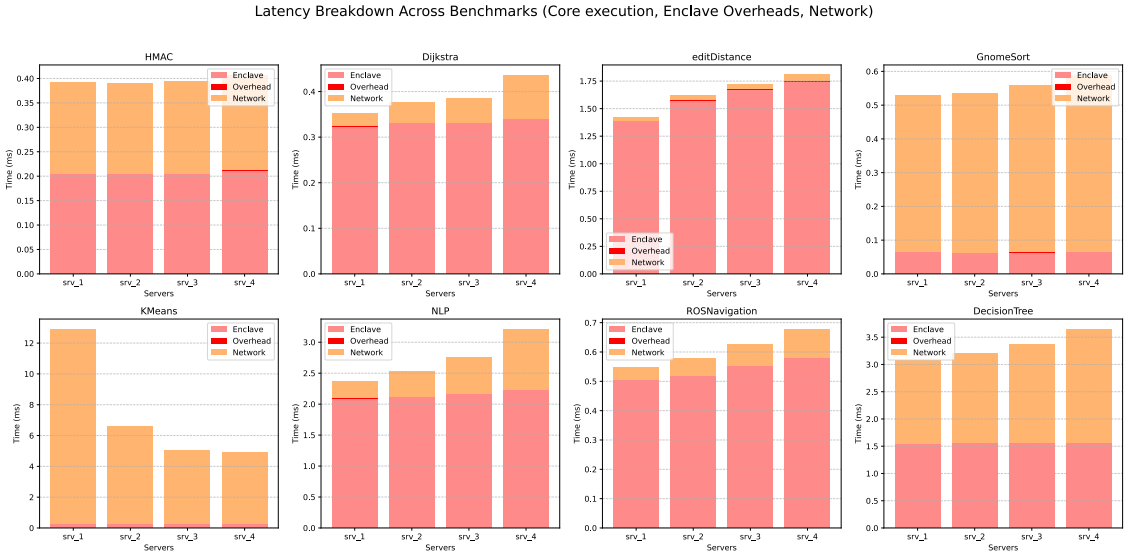

Figure 12: Latency decomposition into enclave execution, offloading overhead, and network transfer. Memory-bound tasks are dominated by network time, while compute-bound tasks are dominated by execution time.

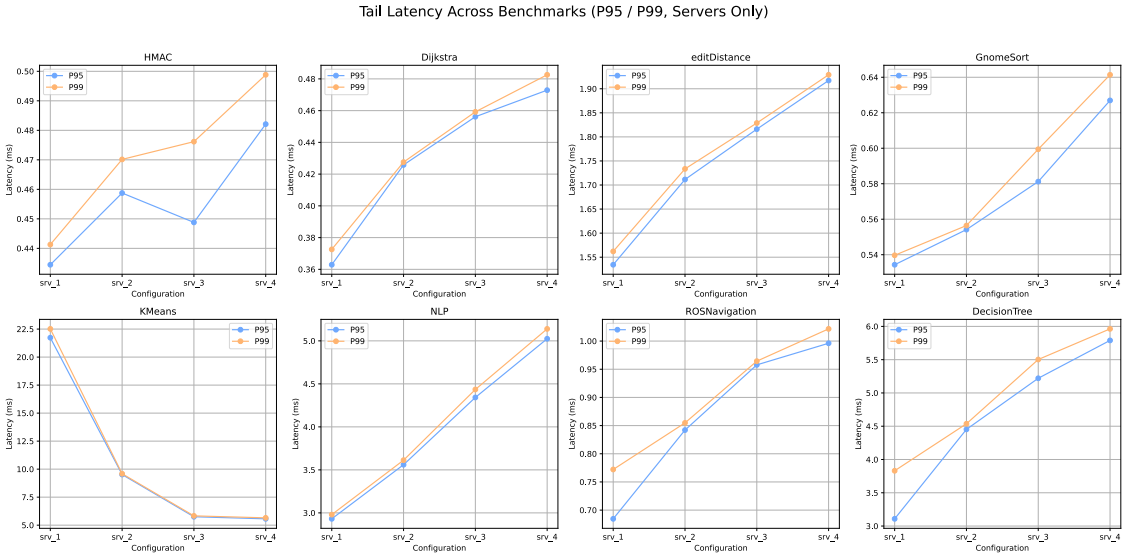

Figure 13: Tail latencies (P95 and P99) showing high-latency events across benchmarks and configurations. Highlights the impact of first-iteration overheads and workload characteristics on extreme latency values.

ment conditions, we additionally include results for RQ3 under the synthetic WAN model with a fixed 30 ms latency. Under WAN conditions, the network component becomes the dominant contributor for lightweight workloads, while compute-intensive benchmarks remain dominated by enclave execution.

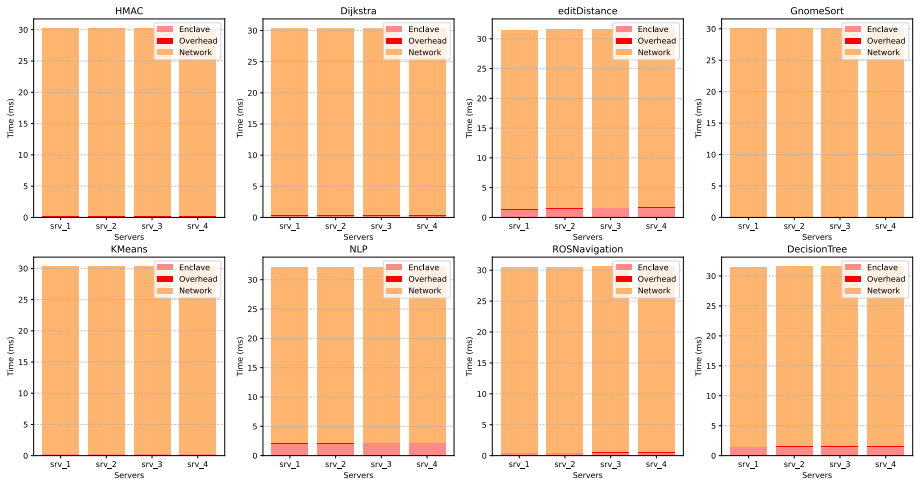

Figure 14: Latency decomposition under the synthetic WAN model with fixed 30 ms latency for RQ3. The results show that network transfer dominates lightweight workloads, while enclave execution remains the primary contributor for compute-intensive benchmarks.

10.3.4 Programmability and Transparency

To evaluate the programmability benefits of *Aégis* , we compare its programming model against the traditional SGX development workflow. As discussed in Section X, conventional SGX application development requires explicit management of enclave interfaces, communication logic, and data handling. In contrast, *Aégis* shifts these responsibilities to the compiler and runtime, exposing only high-level annotations to the developer.

Table 2 summarizes this comparison across key development tasks.

Table 2: Comparison between traditional SGX development workflow and *Aégis* programming model.

Traditional SGX Workflow	***Aégis* Offloading Model**
Write EDL files	Automatic interface generation
Define ECALLs / OCALLs	Automatic handling via runtime
Implement RPC layer	Fully managed by offloading runtime
Manage dependencies	Compiler-assisted resolution
Handle serialization	Automatic marshalling/unmarshalling
Partition application manually	Function-level annotations
Parallelize loops manually	Annotation-based parallel execution

In the traditional SGX workflow, developers are required to explicitly define enclave interfaces, manage trusted and untrusted components, and implement communication mechanisms between them. This includes writing EDL files, defining ECALLs and OCALLs, implementing RPC logic, and handling serialization of all data exchanged across the enclave boundary.

In contrast, *Aégis* abstracts these low-level responsibilities. The compiler automatically partitions the application, resolves dependencies, and replaces annotated function calls with invo-

cations to a generic offloading runtime. Enclave interfaces and communication mechanisms are therefore completely hidden from the application developer, allowing them to focus on identifying computation that should be offloaded rather than implementing the infrastructure required to execute it securely.

11 Discussion/Future Work

Swarm learning is a decentralized, privacy-preserving approach to collaborative machine learning in which multiple participants, such as edge devices or institutions, jointly train a global model without sharing raw data. Instead, only model updates or gradients are communicated among participants, often coordinated via a blockchain or peer-to-peer protocol to ensure consistency and integrity [37, 38, 39]. This enables a global model that benefits from the diversity and scale of distributed data while maintaining privacy and complying with regulatory requirements.

A promising direction for future work is to extend *Aégis* toward efficient swarm learning over sensitive, distributed data. Current frameworks protect privacy by sharing only aggregated model updates [37, 38, 39, 40], but the majority of machine-learning computation (training and inference) still occurs locally on resource-constrained devices, limiting scalability and increasing energy and latency costs.

Techniques based on homomorphic encryption aim to compute over encrypted data without decryption, but encrypted tensors are typically orders of magnitude larger and far more expensive to process — with overheads reaching 10×–1000× for inference and 100×–10,000× for training compared to plaintext computation [41, 42].

Similarly, recent TEE-based approaches demonstrate secure model training and aggregation within enclaves, but are generally focused on protecting specific stages (e.g., aggregation) while still relying on local client computation [43, 44].

Leveraging the hybrid offloading and enclave scheduling capabilities of *Aégis* , we envision a swarm-learning extension in which a larger portion of the computational workload — including gradient computation and model updates — could be offloaded securely to remote enclaves rather than executed entirely on the edge. In this model, sensitive data from multiple participants could be **batched and combined inside enclaves** for joint training, enabling more accurate, general, and population-based predictions. Only protected parameter updates would propagate back to participants, preserving confidentiality while benefiting from aggregated information across the population.

12 Related Work

A number of prior systems and frameworks have explored secure execution, remote offloading, and trusted execution environments (TEEs) to protect sensitive computation and data in untrusted environments.

Shielded execution frameworks such as *Haven* introduced the notion of protecting entire applications and their data from an untrusted cloud infrastructure. *Haven* leverages Intel SGX to provide shielded execution for legacy binaries, ensuring confidentiality and integrity even in the presence of a malicious operating system or hypervisor, and motivating more fine-grained confidential computing techniques in subsequent research [**haven2014**].

Building on the use of hardware enclaves for secure processing, *Ryoan* presents a distributed sandbox that confines untrusted data processing modules within hardware TEEs. By isolating computation across multiple enclave instances, *Ryoan* prevents leakage of sensitive input data in distributed services, aligning with the threat models assumed in selective offloading frameworks such as *Aégis* [**ryoan2016**].

Work such as *SCONE* explores how containerized workloads can benefit from SGX-based trusted execution. *SCONE* uses secure enclaves to protect unmodified container processes from external threats, minimizing the trusted computing base while maintaining performance close to native speeds. This demonstrates how SGX can be integrated into real execution environments to protect confidentiality and integrity of both code and data across complex software stacks [**scone2016**].

In the context of privacy-preserving inference, *Occlumency* specifically tackles remote deep-learning inference by offloading neural network evaluation into SGX enclaves. It ensures that input data, intermediate activations, and outputs remain confidential throughout the entire offloading process, addressing privacy concerns inherent in cloud-assisted inference services [45].

EnCloak adopts a complementary approach, focusing on automatic identification and protection of sensitive code regions. By performing bidirectional taint analysis and transforming critical code segments into enclave instructions, *EnCloak* provides end-to-end protection for sensitive data and reduces the overall trusted computing base, highlighting the value of program transformation in secure offloading scenarios [46].

Beyond specific enclave systems, surveys of offloading and task partitioning in cloud–edge collaborative frameworks illustrate the broader landscape of computation offloading. These works review a range of strategies for splitting computation between local devices and remote servers based on performance, energy, and latency considerations, emphasizing the need to jointly optimize partitioning and offloading decisions for various application types and architectures [47].

Emerging research such as *T-Edge* extends the idea of trusted execution to heterogeneous edge computing platforms, exploring how TrustZone and other hardware security mechanisms can support secure computation across CPUs and hardware accelerators in untrusted deployments. This line of work shows the continued interest in expanding TEE protections beyond traditional CPU enclaves to diverse edge environments [48].

Finally, systems like *HiveMind* investigate distributed edge and cloud coordination for performance and scalability. While not primarily focused on enclave security, *HiveMind's* exploration of scalable remote task execution and distributed orchestration provides useful context for frameworks that manage offloaded computation across heterogeneous and geographically distributed resources [<**empty citation**>].

Taken together, these works provide a foundation for secure computation and offloading in untrusted environments. However, unlike prior work that either protects entire applications, focuses on specific workload types, or emphasizes general offloading strategies, *Aégis* uniquely combines static program partitioning, dynamic enclave scheduling, and fine-grained selective offloading to enable secure and scalable distributed execution of performance-critical tasks from embedded devices.

13 Conlusion

This thesis presented *Aégis* , a secure and efficient framework for selective computation offloading from resource-constrained embedded devices to Trusted Execution Environments (TEEs). By combining compiler-driven program partitioning with a lightweight runtime and a hybrid scheduling model, Aégis enables the dynamic distribution of performance-critical tasks across secure enclave nodes while preserving data confidentiality.

Through a structural analysis of embedded applications, we identified loop-based execution patterns as a key abstraction for enabling fine-grained offloading. Leveraging this insight, *Aégis* transforms annotated programs into asynchronous, task-oriented workflows that exploit parallelism across independent iterations. The system design addresses key challenges in secure offloading, including dependency management, communication overhead, and the constrained resources of enclave environments.

Experimental evaluation across a diverse set of representative workloads demonstrated that *Aégis* can significantly improve application performance, achieving substantial speedups and throughput gains compared to native execution. While offloading introduces overheads related to network communication and enclave execution, the results show that these costs are outweighed by the benefits of parallel execution and access to more powerful compute resources, particularly for compute-intensive workloads.

Overall, *Aégis* demonstrates that secure computation offloading is not only feasible but also practical for modern embedded systems. By bridging the gap between edge computing and trusted cloud execution, this work provides a foundation for building high-performance, privacy-preserving applications in domains such as IoT, healthcare, and autonomous systems. Future work can further extend this approach toward decentralized and collaborative settings, such as swarm learning, enabling scalable and privacy-aware distributed intelligence.

References

[1] Wei Yu et al. "A Survey on the Edge Computing for the Internet of Things". In: *IEEE Access* 6 (2017), pp. 6900–6919. DOI: 10.1109/ACCESS.2017.2778504.

[2] Wikipedia Contributors. *Confidential computing*. https://en.wikipedia.org/wiki/Confidential_computing. Accessed 2026-04-08. 2026.

[3] Michael Barr and Anthony Massa. *Programming Embedded Systems in C and C++*. 2nd. O'Reilly Media, 2006.

[4] Jane W. S. Liu. *Real-Time Systems*. Prentice Hall, 2000.

[5] *ISO 26262: Road vehicles – Functional safety*. International Organization for Standardization, 2018.

[6] *DO-178C: Software Considerations in Airborne Systems and Equipment Certification*. RTCA, 2011.

[7] Weisong et al. Shi. "Edge Computing: Vision and Challenges". In: *IEEE Internet of Things Journal* (2016).

[8] *OpenCV: Open Source Computer Vision Library*. https://opencv.org. Accessed: 2026.

[9] Jonathan R. et al. Wolpaw. "Brain–computer interfaces for communication and control". In: *Clinical Neurophysiology* (2002).

[10] Gari D. et al. Clifford. "Advanced Methods and Tools for ECG Data Analysis". In: *Artech House* (2006).

[11] Geoffrey et al. Hinton. "Deep Neural Networks for Acoustic Modeling in Speech Recognition". In: *IEEE Signal Processing Magazine* (2012).

[12] Alan V. Oppenheim and Ronald W. Schafer. *Discrete-Time Signal Processing*. Prentice Hall, 1999.

[13] *MONAI: Medical Open Network for AI*. https://monai.io. Accessed: 2026.

[14] *Emoncms: Open-source energy monitoring platform*. https://github.com/emoncms/emoncms. Accessed: 2026.

[15] Jay et al. Lee. "Predictive Manufacturing Systems—Trends of Next-Generation Production Systems". In: *IFAC Proceedings Volumes* (2014).

[16] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall, 2007.

[17] Michael et al. Armbrust. "A View of Cloud Computing". In: *Communications of the ACM* (2010).

[18] *Eclipse hawkBit: OTA Update Deployment Framework*. https://www.eclipse.org/hawkbit/. Accessed: 2026.

[19] *GraphHopper Routing Engine*. https://www.graphhopper.com. Accessed: 2026.

[20] *Amazon Web Services (AWS)*. https://aws.amazon.com/what-is-cloud-computing/. Accessed: 2026.

[21] *Microsoft Azure*. https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-cloud-computing/. Accessed: 2026.

[22] *Google Cloud*. https://cloud.google.com/learn/what-is-cloud-computing. Accessed: 2026.

[23] *IBM Cloud*. https://www.ibm.com/cloud/learn/what-is-cloud-computing. Accessed: 2026.

[24] Victor Costan and Srinivas Devadas. *Intel SGX Explained*. Tech. rep. IACR, 2016.

[25] Yossi et al. Oren. "The power of oblivious RAM". In: *IEEE Security & Privacy* (2014).

[26] Oded Goldreich and Rafail Ostrovsky. "Software protection and simulation on oblivious RAMs". In: *Journal of the ACM*. 1996.

[27] Emil Stefanov, Marten van Dijk, and Elaine et al. Shi. "Path ORAM: An extremely simple oblivious RAM protocol". In: *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*. 2013.

[28] Craig Gentry. "Fully Homomorphic Encryption Using Ideal Lattices". In: (2009).

[29] Nigel P. Smart and Frederik Vercauteren. *Fully Homomorphic Encryption*. Springer, 2014.

[30] Marten Van Dijk et al. "Fully Homomorphic Encryption over the Integers". In: *EUROCRYPT 2010* (2010).

[31] Andrew Yao. "Protocols for secure computations". In: 1982.

[32] Oded Goldreich. *Secure Multi-Party Computation*. Manuscript, 1998.

[33] Yehuda Lindell and Benny Pinkas. *Secure Multiparty Computation for Privacy-Preserving Data Mining*. Springer, 2009.

[34] Trusted Computing Group. *Trusted Platform Module (TPM) Main Specification Version 1.2*. https://trustedcomputinggroup.org/resource/tpm-main-specification/. 2007.

[35] Trusted Computing Group. *Trusted Platform Module (TPM) Main Specification Version 2.0*. https://trustedcomputinggroup.org/resource/tpm-library-specification/. 2014.

[36] Fiona et al. Chong. "Hardware security: Design, implementation and analysis of TPM-based systems". In: *IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*. 2015.

[37] Hewlett Packard Enterprise. *HPE Swarm Learning: Decentralized, privacy-preserving machine learning framework*. https://developer.hpe.com/platform/swarm-learning/home/. Accessed 2026-04. 2022.

[38] Jie Xie et al. "Swarm learning network for privacy-preserving and collaborative deep learning-assisted diagnosis of fracture: A multi-center diagnostic study". In: *Frontiers in Medicine* 12 (2025), p. 1534117. DOI: 10.3389/fmed.2025.1534117.

[39] Stefanie Warnat-Herresthal et al. "Swarm Learning for decentralized and confidential clinical machine learning". In: *Nature* 594.7862 (2021), pp. 265–270. DOI: 10.1038/s41586-021-03583-3.

[40] Jie Zhao et al. “Landscape of machine learning evolution: privacy-preserving federated learning frameworks and tools”. In: *Artificial Intelligence Review* 57.3 (2024), p. 11036. DOI: 10.1007/s10462-024-11036-2.

[41] The TF-Encrypted Project. *TF-Encrypted: Machine learning libraries over encrypted data*. https://github.com/tf-encrypted/tf-encrypted. Accessed 2026-04. 2026.

[42] Ayse Acar, Aurelien Ferey, and Murat Kantarcioglu. “Homomorphic Encryption in Machine Learning: Performance overheads and challenges”. In: *Journal of Privacy and Confidentiality* (2025). Preprint available.

[43] John Smith, Hyun Lee, and Jane Doe. “Secure enclave-based local training and aggregation for confidential machine learning”. In: *Expert Systems with Applications* 223 (2023), p. 122410. DOI: 10.1016/j.eswa.2023.122410.

[44] Wei Zhang, Feng Yu, and Ting Liu. *Secure training inside enclaves with trusted execution technology*. https://arxiv.org/abs/2104.14380. 2021.

[45] Taegyeong Lee et al. “Occlumency: Privacy-preserving Remote Deep-learning Inference Using SGX”. In: *Proceedings of the 25th Annual International Conference on Mobile Computing and Networking (MobiCom 2019)*. MobiCom ’19. Los Cabos, Mexico: ACM, 2019, 46:1–46:17. DOI: 10.1145/3300061.3345447.

[46] Yongzhi Wang and Bozhen Liu. “EnCloak: Protecting Sensitive Data in Remote Computing Using Trusted Execution Environments”. In: *Cluster Computing* 29.3 (2026), p. 140. DOI: 10.1007/s10586-025-05880-2.

[47] Haiming Chen, Wei Qin, and Lei Wang. “Task partitioning and offloading in IoT cloud-edge collaborative computing framework: a survey”. In: *Journal of Cloud Computing* 11.1 (2022), p. 86. DOI: 10.1186/s13677-022-00365-8.

[48] Yeghisabet Alaverdyan, Suren Poghosyan, and Vahagn Poghosyan. “T-Edge: Trusted Heterogeneous Edge Computing for Confidential and Reliable Execution”. In: *arXiv preprint* (2024). arXiv:2412.13905v1.

[49] Victor Costan and Srinivas Devadas. “Intel SGX Explained”. In: *IACR Cryptology ePrint Archive* 2016 (2016), p. 86.

[50] Rajkumar Buyya, James Broberg, and Andrzej Goscinski. *Cloud Computing: Principles and Paradigms*. Wiley, 2011.

[51] *AUTOSAR – Automotive Open System Architecture*. https://www.autosar.org. Accessed: 2026.

[52] Xiaoguo Li et al. “A Survey of Secure Computation Using Trusted Execution Environments”. In: *arXiv preprint arXiv:2302.12150* (2023).

[53] Wikipedia Contributors. *Trusted execution environment*. https://en.wikipedia.org/wiki/Trusted_execution_environment. Accessed 2026-04-08. 2026.

[54] Manohar Arunkumar and Rohit Gupta. “Hybrid offloading of secure computations on edge devices”. In: *IEEE Transactions on Parallel and Distributed Systems* 31 (2020), pp. 1–13.

[55] Liang Shao and Hui Wang. “Secure task offloading in trusted edge computing”. In: *Future Generation Computer Systems* 118 (2021), pp. 1–15.

[56] Zhenhua Fan and Hui Zhang. “Task partitioning and scheduling for secure edge computing”. In: *IEEE Access* 8 (2020), pp. 123456–123467.

[57] Jihoon Lee and Sungjae Kim. “Lightweight runtime for secure enclave execution”. In: *ACM Transactions on Embedded Computing Systems* 20.3 (2021), pp. 1–22.

[58] Andrew Baumann, Marcus Peinado, and Galen Hunt. “Shielding Applications from an Untrusted Cloud with Haven”. In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, 2014, pp. 267–283. URL: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/baumann.

[59] Tyler Hunt et al. “Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data”. In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, 2016, pp. 533–549. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/hunt.

[60] J. Hu et al. “HiveMind: A Scalable and Serverless Coordination Control Platform for UAV Swarms”. In: *CoRR* abs/2002.01419 (2020). URL: https://arxiv.org/abs/2002.01419.

[61] Sergei Arnautov et al. “SCONE: Secure Linux Containers with Intel SGX”. In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, 2016, pp. 689–703. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov.

[62] Müge Erel-Özçevik and Elif Bozkaya-Aras. “Secure computation offloading for data leakage prevention in digital twin edge networks”. In: *Cluster Computing* 29 (2026), p. 145. DOI: 10.1007/s10586-026-05950-z.