Ατομική Διπλωματική Εργασία

Μελέτη Σιωπηλών Αλλοιώσεων Δεδομένων σε Εφαρμογές Βασισμένες σε Μικροϋπηρεσίες

Κενδέας Δημητρίου

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Μάιος 2025

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Μελέτη Σιωπηλών Αλλοιώσεων Δεδομένων σε Εφαρμογές Βασισμένες σε Μικροϋπηρεσίες

Κενδέας Δημητρίου

Επιβλέπων Καθηγητής
Χάρης Βώλος

Η Ατομική Διπλωματική Εργασία υποβλήθηκε προς μερική εκπλήρωση των απαιτήσεων απόκτησης του πτυχίου Πληροφορικής του Τμήματος Πληροφορικής του Πανεπιστημίου Κύπρου

Μάιος 2025

Ευχαριστίες

Θα ήθελα να εκφράσω την ειλικρινή μου ευγνωμοσύνη προς τον επιβλέποντα καθηγητή μου, Δρ. Χάρη Βώλο, Καθηγητή στο Τμήμα Πληροφορικής του Πανεπιστημίου Κύπρου, για την εμπιστοσύνη που έδειξε στην επιλογή του θέματος και για τη σταθερή καθοδήγηση και υποστήριξή του καθ' όλη τη διάρκεια της διπλωματικής μου εργασίας.

Θα ήθελα επίσης να ευχαριστήσω θερμά τον Στυλιανό Βασιλείου για την ουσιαστική συμβολή του και τις πολύτιμες συμβουλές που μου προσέφερε σε όλα τα στάδια της ερευνητικής και συγγραφικής διαδικασίας.

Περίληψη

Στην παρούσα διπλωματική εργασία μελετάται πειραματικά το φαινόμενο των σιωπηλών αλλοιώσεων δεδομένων SDCs σε εφαρμογές αρχιτεκτονικής microservices, με έμφαση στον τρόπο με τον οποίο επηρεάζεται η ροή ελέγχου σε περιπτώσεις τέτοιων σφάλματων. Οι SDCs, σε αντίθεση με τα εμφανή σφάλματα του συστήματος, δεν συνοδεύονται από άμεσα ανιχνεύσιμες ενδείξεις όπως σφάλματα κατάρρευσης, αλλά δύνανται να μεταβάλλουν σιωπηλά τα αποτελέσματα, υπονομεύοντας την αξιοπιστία κρίσιμων εφαρμογών.

Αναπτύχθηκε ένα αυτοματοποιημένο εργαλείο έγχυσης σφαλμάτων, βασισμένο στο Intel Pin, για microservices που ακολουθούν το μοντέλο query-response. Το εργαλείο επιτρέπει την προσομοίωση σιωπηλών αλλοιώσεων δεδομένων (Silent Data Corruptions - SDCs) σε εντολές που εκτελούνται από υπολογιστικές μονάδες. Ως πειραματικό περιβάλλον χρησιμοποιήθηκε η εφαρμογή HDSearch της μSuite, η οποία βασίζεται σε αρχιτεκτονική microservices και εκτελεί αριθμητικά εντατικές λειτουργίες αναζήτησης ομοιότητας εικόνων.

Κατά τη διάρκεια των πειραμάτων, η αξιολόγηση επικεντρώθηκε στη σύγκριση της ροής ελέγχου και της απόκρισης του συστήματος μεταξύ εκτελέσεων με και χωρίς έγχυση σφαλμάτων, με στόχο την κατανόηση ενδεχόμενων αποκλίσεων στη συμπεριφορά των microservices υπό την παρουσία σιωπηλών αλλοιώσεων δεδομένων.

Περιεχόμενα

Κεφάλαιο 1

Εισαγωγή

1.1 Πλαίσιο

Στη σύγχρονη εποχή, οι διαδικτυακές υπηρεσίες μεγάλης κλίμακας λειτουργούν πάνω σε τεράστιες υπολογιστικές υποδομές που αποτελούνται από εκατομμύρια διανεμημένους εξυπηρετητές. Η ορθότητα, η αξιοπιστία και η ανθεκτικότητα των υπολογισμών είναι απαραίτητες για την απρόσκοπτη λειτουργία αυτών των υποδομών. Ωστόσο, ένα αναδυόμενο και ιδιαίτερα ανησυχητικό φαινόμενο είναι οι σιωπηλές αλλοιώσεις δεδομένων, οι οποίες ενδέχεται να οφείλονται σε εσωτερικά ελαττώματα υλικού, φθορά τρανζίστορ λόγω γήρανσης, θερμικές διακυμάνσεις ή πολύπλοκες μικροαρχιτεκτονικές αλληλεπιδράσεις.

Οι SDCs αποτελούν μια ιδιαίτερα ύπουλη μορφή σφάλματος, καθώς δεν συνοδεύονται από άμεσες ενδείξεις δυσλειτουργίας, όπως σφάλματα κατάρρευσης ή εξαιρέσεις, αλλά οδηγούν σε λανθασμένα αποτελέσματα που συνεχίζουν να διαδίδονται εντός του συστήματος. Οι συνέπειες αυτών των αλλοιώσεων μπορεί να είναι καταστροφικές, καθώς υπονομεύουν την ακεραιότητα των δεδομένων, την αξιοπιστία του λογισμικού και τη συνοχή του συστήματος.

Παρότι το πρόβλημα των SDCs έχει μελετηθεί εκτενώς στο πλαίσιο της μνήμης και των αποθηκευτικών μέσων, η εμφάνισή τους σε υπολογιστικές μονάδες αποτελεί μια νέα και ιδιαίτερα απαιτητική πρόκληση. Καθώς οι διαστάσεις των τρανζίστορ συνεχίζουν

να μειώνονται, οι υπολογιστικοί πυρήνες καθίστανται ολοένα και πιο ευάλωτοι σε σφάλματα που δεν εντοπίζονται άμεσα, γεγονός που μπορεί να επηρεάσει την ακρίβεια και την αξιοπιστία των αριθμητικών διαδικασιών**.** Επομένως, η κατανόηση και η αντιμετώπιση των SDCs σε επίπεδο υπολογιστικών μονάδων είναι κρίσιμη για τη διατήρηση της συνολικής αξιοπιστίας των σύγχρονων συστημάτων.

Ορισμένες δομές του επεξεργαστή παρουσιάζουν ιδιαίτερη ευπάθεια σε SDCs, όπως:

- Οι πράξεις κινητής υποδιαστολής (floating-point arithmetic),
- Οι εντολές διανυσματικής επεξεργασίας SIMD,[1]

Είναι αξιοσημείωτο ότι τα σφάλματα σε πράξεις κινητής υποδιαστολής τείνουν να προκαλούν λεπτές, ανεπαίσθητες αποκλίσεις στα αποτελέσματα, που συχνά διαφεύγουν των κλασικών μηχανισμών εντοπισμού σφαλμάτων. Το φαινόμενο αυτό οφείλεται σε μεγάλο βαθμό στον τρόπο με τον οποίο αναπαρίστανται οι αριθμοί κινητής υποδιαστολής (σύμφωνα με το πρότυπο IEEE-754), όπου ακόμη και μια ελάχιστη αλλοίωση στο fractional part μπορεί να επιφέρει μικρή αριθμητική απόκλιση, χωρίς να προκαλέσει άμεσο σφάλμα ή κατάρρευση του συστήματος. Τέτοιες αλλοιώσεις τείνουν να διαφεύγουν των κλασικών μηχανισμών ανίχνευσης, ειδικά όταν η μεταβολή εντοπίζεται σε χαμηλής σημασίας bits, με αποτέλεσμα τη σιωπηλή διάδοση του σφάλματος σε επόμενα στάδια υπολογισμού[1]. Το γεγονός αυτό είναι ιδιαίτερα κρίσιμο για εφαρμογές που απαιτούν υψηλή αριθμητική ακρίβεια, όπως η επεξεργασία σήματος, η μηχανική μάθηση και οι επιστημονικοί υπολογισμοί.

Καθώς η ανθεκτικότητα των σύγχρονων υπολογιστικών εφαρμογών, και ιδίως εκείνων που βασίζονται σε αρχιτεκτονική microservices, εξαρτάται σε μεγάλο βαθμό από την ικανότητά τους να ανταποκρίνονται ορθά ακόμη και όταν προκύπτουν σφάλματα σε χαμηλό επίπεδο, όπως οι σιωπηλές αλλοιώσεις δεδομένων, η παρούσα εργασία εστιάζει στη μελέτη των συνεπειών των SDCs στο εσωτερικό μιας συγκεκριμένης microservice-based εφαρμογής. Συγκεκριμένα, αναλύεται πώς αυτά τα σφάλματα διαδίδονται και επηρεάζουν τη συμπεριφορά της εφαρμογής κατά τη διάρκεια της εκτέλεσης.

1.2 Στόχοι

Σκοπός της παρούσας διπλωματικής εργασίας είναι η πειραματική διερεύνηση της επίδρασης των Silent Data Corruptions στη λειτουργικότητα εφαρμογών που βασίζονται σε αρχιτεκτονική microservices. Ειδικότερα, εξετάζεται κατά πόσο οι αποκλίσεις στη ροή ελέγχου μιας εκτέλεσης με εισαγόμενα σφάλματα, σε σύγκριση με την κανονική εκτέλεση, μπορούν να αξιοποιηθούν ως ενδείξεις παρουσίας SDCs.

Η εργασία επικεντρώνεται στη διερεύνηση του τρόπου με τον οποίο τα σφάλματα σε υπολογιστικές μονάδες χαμηλού επιπέδου — όπως η αριθμητική-λογική μονάδα (ALU) και η μονάδα κινητής υποδιαστολής (FPU) — επηρεάζουν τη ροή ελέγχου και την ακρίβεια των αποκρίσεων σε εφαρμογές με υψηλό υπολογιστικό φόρτο. Ιδιαίτερη έμφαση δίνεται σε κρίσιμα σημεία της εφαρμογής όπου παρατηρείται εντατική αριθμητική επεξεργασία, καθώς σε αυτά τα σημεία εμφανίζεται αυξημένη πιθανότητα εκδήλωσης SDCs, ιδιαίτερα υπό συνθήκες θερμικής καταπόνησης ή παρουσίας ελαττωμάτων στο υλικό.

Για την υλοποίηση των πειραμάτων, αναπτύχθηκε ένα παραμετροποιήσιμο και επαναχρησιμοποιήσιμο εργαλείο έγχυσης σφαλμάτων βασισμένο στο εργαλείο Intel Pin[7], με στόχο την προσομοίωση SDCs σε συγκεκριμένες κατηγορίες εντολών. Στόχος του εργαλείου είναι η προσομοίωση σιωπηλών αλλοιώσεων δεδομένων σε συγκεκριμένες κατηγορίες εντολών καθώς και η εξαγωγή του instruction sequence κατά την εκτέλεση, με σκοπό τη σύγκριση golden και faulty run. Το εργαλείο έχει σχεδιαστεί ώστε να ενσωματώνεται με ευκολία σε εφαρμογές βασισμένες σε αρχιτεκτονική μικροϋπηρεσιών τύπου query – response.

1.3 Μεθοδολογία

Η μεθοδολογία της παρούσας μελέτης επικεντρώθηκε στην ανάπτυξη Pintools με χρήση του Intel Pin API[7], με σκοπό τη δημιουργία ενός πλήρως αυτοματοποιημένου και επαναχρησιμοποιήσιμου συστήματος προσομοίωσης SDCs σε εφαρμογές αρχιτεκτονικής microservices.

Αρχικά, πραγματοποιήθηκε βιβλιογραφική ανασκόπηση της σχετικής ερευνητικής βιβλιογραφίας με στόχο την κατανόηση της φύσης, των χαρακτηριστικών και των επιπτώσεων των Silent Data Corruptions σε σύγχρονα υπολογιστικά συστήματα. Παράλληλα, μελετήθηκε εις βάθος η αρχιτεκτονική των microservices, με σκοπό την επιλογή ενός κατάλληλου πειραματικού περιβάλλοντος. Για τις ανάγκες της παρούσας εργασίας, επιλέχθηκε η χρήση του μSuite[9] — μιας ερευνητικής σουίτας εφαρμογών μικροϋπηρεσιών, σχεδιασμένης για την αξιολόγηση επιδόσεων και την ανάλυση σφαλμάτων σε κατανεμημένα περιβάλλοντα. Το μSuite περιλαμβάνει διαφορετικές εφαρμογές με ποικίλα χαρακτηριστικά υπολογιστικού φόρτου και επικοινωνίας, οι οποίες προσομοιώνουν συνθήκες πραγματικών υπηρεσιών cloud.

Η παρούσα μελέτη επικεντρώνεται στην εφαρμογή HDSearch της μSuite[9], η οποία αποτελεί μια πλατφόρμα αναζήτησης ομοιότητας εικόνων, βασισμένη σε χαρακτηριστικά υψηλής διαστασιμότητας (high-dimensional features). Η εφαρμογή HDSearch ακολουθεί αρχιτεκτονική microservices τύπου query-response, όπου το Bucket Server service εκτελεί το μεγαλύτερο μέρος των υπολογισμών κατά τη φάση της σύγκρισης διανυσμάτων. Η φύση και η λειτουργία της HDSearch την καθιστούν ιδανική περίπτωση μελέτης για την αξιολόγηση της επίδρασης των SDCs σε εφαρμογές με έντονη αριθμητική δραστηριότητα.

Στο πλαίσιο της υλοποίησης για περιβάλλοντα microservices, χρησιμοποιήθηκε το Intel Pin API[7] για την ανάπτυξη εργαλείων έγχυσης σφαλμάτων, με στόχο την προσομοίωση SDCs σε επίπεδο bit. Δεδομένης της αυξημένης πιθανότητας εμφάνισης SDCs σε μονάδες υπολογισμού, και ιδίως σε λειτουργίες κινητής υποδιαστολής ή εντολές SIMD, η μελέτη επικεντρώθηκε στην αλλοίωση αποτελεσμάτων που παράγονται από την ALU, την FPU και τις μονάδες επέκτασης SSE.

Στη συνέχεια, η εφαρμογή HDSearch μελετήθηκε εις βάθος και προσαρμόστηκε κατάλληλα, ώστε να είναι εφικτή η πειραματική παρακολούθησή της μέσω του αναπτυγμένου Pintool. Κατά τη διάρκεια της υλοποίησης, διαπιστώθηκε ότι η ακολουθία εντολών εντός των εσωτερικών βιβλιοθηκών του λειτουργικού συστήματος δεν είναι πλήρως ντετερμινιστική, κάτι που δυσχεραίνει τη σύγκριση μεταξύ εκτελέσεων με και χωρίς fault injection. Για την αντιμετώπιση του προβλήματος,

υιοθετήθηκε παράλληλη εκτέλεση, κατά την οποία δημιουργείται μια θυγατρική διεργασία (child process) πριν την εκτέλεση των αριθμητικών υπολογισμών κινητής υποδιαστολής. Με αυτόν τον τρόπο, κατέστη δυνατή η σύγκριση της "καθαρής" εκτέλεσης (golden run) από τη θυγατρική διεργασία με την "επηρεασμένη" εκτέλεση στην αρχική διεργασία, υπό τις ίδιες ακριβώς συνθήκες εισόδου.

Ειδικότερα, διερευνάται κατά πόσο οι αποκλίσεις στη ροή ελέγχου μιας εκτέλεσης με αλλοιώσεις από τη συνήθη εκτέλεση μπορούν να αξιοποιηθούν ως ενδείξεις για την παρουσία Silent Data Corruptions (SDCs). Μέσω της εκτέλεσης πολλαπλών επαναλήψεων του HDSearch, με στοχαστική επιλογή εντολών και διαφορετικά bit masks σε επιλεγμένα ευπαθή σημεία της εφαρμογής, εξετάζονται σενάρια που σχετίζονται με διαφορετικές εκφάνσεις αντίδρασης του συστήματος σε SDCs: από πλήρη κατάρρευση της εφαρμογής (crash), μέχρι μερική αλλοίωση της απόκρισης ή και περιπτώσεις πλήρους σιωπής του σφάλματος, στις οποίες το microservice φαίνεται να λειτουργεί κανονικά, παρότι το αποτέλεσμα είναι ήδη αλλοιωμένο.

1.4 Επισκόπηση Κεφαλαίων

Η παρούσα διπλωματική εργασία οργανώνεται σε επτά κεφάλαια, τα οποία αναπτύσσονται με τρόπο που οδηγεί προοδευτικά από το θεωρητικό υπόβαθρο στην πειραματική αξιολόγηση και ανάλυση των αποτελεσμάτων.

- Κεφάλαιο 2:
  Παρουσιάζεται μια εις βάθος εισαγωγή στις σιωπηλές αλλοιώσεις δεδομένων, με έμφαση στον ορισμό τους, τα βασικά χαρακτηριστικά τους και τη σημασία τους για τη λειτουργική αξιοπιστία των σύγχρονων υπολογιστικών συστημάτων.

- Κεφάλαιο 3:
  Περιγράφεται η αρχιτεκτονική των microservices και αναλύεται η εφαρμογή HDSearch της μSuite, η οποία χρησιμοποιείται ως βασικό πειραματικό περιβάλλον στην παρούσα μελέτη.

- Κεφάλαιο 4:
  Παρουσιάζεται η σχεδίαση και υλοποίηση ενός αυτοματοποιημένου εργαλείου έγχυσης σφαλμάτων, βασισμένου στο Intel Pin[7], το οποίο στοχεύει σε εντολές που εκτελούνται από κρίσιμες υπολογιστικές μονάδες όπως η ALU μέσω καταχωρητών γενικού σκοπού και η FPU/SSE μέσω καταχωρητών XMM.

- Κεφάλαιο 5:
  Αναλύονται οι τροποποιήσεις που πραγματοποιήθηκαν στην εφαρμογή HDSearch, καθώς και τα προβλήματα που προέκυψαν κατά την υλοποίηση και εκτέλεση των πειραμάτων.

- Κεφάλαιο 6:
  Περιγράφεται αναλυτικά η πειραματική μεθοδολογία, συμπεριλαμβανομένης της προετοιμασίας του περιβάλλοντος εκτέλεσης, της διαδικασίας επανάληψης των πειραμάτων και της επιλογής σεναρίων έγχυσης.

- Κεφάλαιο 7:
  Παρουσιάζονται τα αποτελέσματα για κάθε σενάριο έγχυσης, με ανάλυση των αποκρίσεων του συστήματος μέσω σύγκρισης μεταξύ golden και faulty εκτελέσεων. Ιδιαίτερη έμφαση δίνεται στην παρατήρηση διαφορών στη ροή ελέγχου, στην εκδήλωση σφαλμάτων (crashes), στην παραγωγή εσφαλμένων αποκρίσεων ή και στην απουσία ορατών αποκλίσεων, παρά την ύπαρξη σφάλματος.

Η εργασία ολοκληρώνεται με τα συμπεράσματα και προτάσεις για μελλοντική έρευνα, εστιάζοντας σε τεχνικές ενίσχυσης της ανθεκτικότητας εφαρμογών microservices απέναντι σε σιωπηλά σφάλματα.

Κεφάλαιο 2

Επισκόπηση των SDCs

2.1 Τι είναι SDCs

Οι σιωπηλές αλλοιώσεις δεδομένων – silent data corruptions είναι σφάλματα που συμβαίνουν εντός υπολογιστικών συστημάτων χωρίς να ενεργοποιούν άμεσα μηχανισμούς ανίχνευσης σφαλμάτων. Σε αντίθεση με τα παραδοσιακά σφάλματα, τα οποία εντοπίζονται και διορθώνονται μέσω διαδικασιών ελέγχου, οι SDCs παραμένουν λανθάνουσες και δύσκολα ανιχνεύσιμες, αποτελώντας σημαντική απειλή για την αξιοπιστία και την ακεραιότητα εφαρμογών που βασίζονται σε δεδομένα.

Όταν δημιουργηθούν σφάλματα SDCs το σύστημα συνεχίζει να λειτουργεί φαινομενικά κανονικά, παρόλο που τα δεδομένα ή τα αποτελέσματα που παράγονται είναι λανθασμένα. Αυτό τις καθιστά ιδιαίτερα επικίνδυνες, καθώς μπορούν να διαδοθούν αθόρυβα στο σύστημα και να οδηγήσουν σε λανθασμένα αποτελέσματα, αστάθεια λειτουργίας ή ακόμα και σε παραβιάσεις ασφάλειας. Οι SDCs μπορούν να εμφανιστούν σε διάφορα επίπεδα του υπολογιστικού συστήματος, όπως στη μνήμη, στα αποθηκευτικά μέσα, στα κανάλια επικοινωνίας, και πιο πρόσφατα σε υπολογιστικές μονάδες CPU, που αποτελούν και το επίκεντρο της παρούσας μελέτης.

2.2 Η σύγχρονη πρόκληση των Silent Data Corruptions

Αρχικά, τα SDCs θεωρούνταν εξαιρετικά σπάνια φαινόμενα, που προκαλούνταν από κοσμική ακτινοβολία, και εκδηλώνονταν με παροδικά, μη επαναλαμβανόμενα bit flips σε SRAM ή καταχωρητές. Ωστόσο, σύγχρονες μελέτες από τη Meta[2], την Google και την Alibaba αποκάλυψαν ότι πολλές SDCs είναι δομικής φύσεως, προερχόμενες από:

- Ελαττωματικούς επεξεργαστές
- Γήρανση του υλικού
- Θερμικές ασταθείς καταστάσεις
- Προβλήματα cache coherence και συντονισμού cores.

Αυτό οδήγησε στη διαπίστωση ότι το πρόβλημα είναι πολύ πιο συχνό απ' ό,τι πιστευόταν αρχικά, με ποσοστά εμφάνισης που φτάνουν την 1 στις 1000 CPUs σε πραγματικές παραγωγικές συνθήκες[2].

Τα νέα computational SDCs μπορούν να πλήξουν διάφορες λειτουργικές εντολές, ιδίως:

- Floating-point υπολογισμούς
- Vectorized εντολές

Αυτό σημαίνει ότι ειδικοί τύποι workloads είναι πιο ευάλωτοι, ιδίως όταν εμπλέκονται υψηλού ρυθμού υπολογισμοί (π.χ. machine learning, video processing), παράλληλη εκτέλεση σε multithreaded περιβάλλον, ή χρήση κρυπτογράφησης και βάσεων δεδομένων.

Η σοβαρότητα αυτών των προβλημάτων επιβάλλει την ανάγκη για στρατηγικές ανίχνευσης σε επίπεδο εφαρμογών, καθώς τα χαμηλότερα επίπεδα του συστήματος (hardware/kernel) αδυνατούν να τα αναγνωρίσουν. Όπως αναφέρεται στη μελέτη της Meta[2], απαιτείται συνεργασία ανάμεσα σε hardware και software μηχανισμούς ώστε να προστατευθούν κρίσιμες εφαρμογές από σιωπηλές βλάβες, που διαφορετικά θα εκλαμβάνονταν ως τυπικά bugs.

Συνολικά, η αντιμετώπιση των SDCs δεν είναι πλέον τεχνική επιλογή, αλλά απαίτηση αξιοπιστίας για σύγχρονα, μεγάλης κλίμακας υπολογιστικά περιβάλλοντα.

2.3 Στρατηγικές Ανίχνευσης SDCs

Η αντιμετώπιση των SDCs βασίζεται σήμερα σε δύο στάδια[2]:

Out-of-production testing: είναι μια διαδικασία αξιολόγησης μηχανών που βρίσκονται εκτός παραγωγής, μέσω ελεγχόμενων μοτίβων εισόδου και σύγκρισης εξόδων με αναμενόμενες τιμές. Εφαρμόζεται σε φάσεις συντήρησης, όπως firmware ή kernel αναβαθμίσεις, χωρίς να επηρεάζεται το παραγωγικό περιβάλλον. Η Meta υλοποίησε το εργαλείο Fleetscanner, το οποίο εντοπίζει τέτοιες ευκαιρίες και εκτελεί ελέγχους για silent data corruption (SDC), επιτρέποντας την έγκαιρη ανίχνευση σφαλμάτων με ελάχιστο κόστος ανά μηχανή.[2]

Επιπρόσθετα, η Alibaba[1] παρουσιάζει ένα ολοκληρωμένο εργαλείο Toolchain το οποίο προσομοιώνει cloud workload για να εντοπιστεί κατά πόσο ένας επεξεργαστής είναι ελαττωματικός . Για την εφαρμογή του toolchain διακόπτεται προσωρινά η κανονική λειτουργία του server, φορτώνονται δυναμικά τα επιλεγμένα testcases και εκτελείται αυτοματοποιημένη ανίχνευση σιωπηλών σφαλμάτων δεδομένων, όπου τα μηχανήματα υποβάλλονται σε περιοδικούς ελέγχους ανά ομάδες, διάρκειας περίπου δύο εβδομάδων για κάθε ομάδα. Το toolchain περιλαμβάνει 633 testcases που προσομοιώνουν σύνθετα φορτία εργασίας στο cloud — από αριθμητικούς υπολογισμούς κινητής υποδιαστολής και branch prediction μέχρι προσπέλαση cache και πολυνηματικά σενάρια — και ένα ευέλικτο πλαίσιο εργασίας το οποίο, βάσει προδιαγραφών χρήστη, επιλέγει δυναμικά τα κατάλληλα testcases, καθορίζει τη σειρά εκτέλεσης, διαχειρίζεται πόρους όπως ο χρόνος CPU και η ταυτόχρονη εκτέλεση και παρακολουθεί την εμφάνιση SDCs συγκρίνοντας τα αποτελέσματα με τις αναμενόμενες τιμές.

Ακόμη, ένα ενδιαφέρον εργαλείο που θα μπορούσε να χρησημοποιηθεί ως Out of production testing είναι το SAGA[6]. Η προσέγγιση του SAGA[6] (Surrogate-Assisted Genetic Algorithm) εστιάζει στην ταχεία δημιουργία power viruses, δηλαδή προγραμμάτων που προκαλούν μέγιστη κατανάλωση ισχύος στην CPU. Αν και το σύστημα δεν στοχεύει άμεσα στην ανίχνευση silent data corruptions (SDCs), μπορεί να αξιοποιηθεί έμμεσα για τον εντοπισμό ελαττωματικών επεξεργαστών. Συγκεκριμένα, τα

προγράμματα που παράγει το SAGA[6] οδηγούν τον επεξεργαστή σε καταστάσεις θερμικού και ενεργειακού stress, αυξάνοντας την πιθανότητα να εκδηλωθούν intermittent faults ή latent σφάλματα που δεν είναι ανιχνεύσιμα υπό κανονική χρήση. Εάν κατά την εκτέλεση αυτών των power viruses καταγραφούν ασυνήθιστες αποκλίσεις στην κατανάλωση ισχύος, στην επίδοση, ή εμφανιστούν σφάλματα στα δεδομένα εξόδου, τότε αυτές οι συμπεριφορές μπορούν να υποδεικνύουν την παρουσία ελαττωματικού πυρήνα.

In-production testing: η Meta ανέπτυξε το Ripple, το οποίο ελέγχει κατά τη διάρκεια της κανονικής λειτουργίας των εφαρμογών, καθώς και το Hardware Sentinel.

Το Hardware Sentinel[3] αποτελεί ένα καινοτόμο, vendor-agnostic πλαίσιο ανίχνευσης σιωπηλών σφαλμάτων δεδομένων σε επίπεδο εφαρμογών, αξιοποιώντας τυπικούς δείκτες αποτυχίας λογισμικού — όπως segmentation faults, core dumps, application crashes και καταγραφές log — για να εντοπίσει αποκλίσεις που προκύπτουν από ελαττωματικούς επεξεργαστές. Μελετώντας έξι χρόνια δεδομένων αποτυχιών εφαρμογών και συστήματος σε ένα hyperscale fleet, το Hardware Sentinel[3] συσχετίζει ανωμαλίες kernel exceptions με patterns στο runtime των εφαρμογών. Η κορυφαία επιτυχία του αποδεικνύεται από την ανίχνευση εκατοντάδων ελαττωματικών CPUs, την υπεροχή του σε επτά γενιές επεξεργαστών και 13 τύπους workloads, καθώς και την ανεξάρτητη επιβεβαίωση των σφαλμάτων μέσω αναλύσεων failure analysis.

Τέλος, ένας εναλλακτικός τρόπος αντιμετώπισης των ελαττωματικών CPU είναι η μεθοδολογία που εισάγει το εργαλείο Harpocrates[5]. Η διαφοροποίηση του Harpocrates[5] σε σχέση με τις προαναφερθείσες προσεγγίσεις έγκειται στο γεγονός ότι δεν στοχεύει στον εντοπισμό σφαλμάτων υπό συνθήκες αυξημένου θερμικού ή ενεργειακού φορτίου, αλλά στη συστηματική παραγωγή σύντομων λειτουργικών προγραμμάτων ελέγχου που εκτελούνται σε συνθήκες κανονικής λειτουργίας. Το εργαλείο αξιοποιεί τεχνικές hardware-in-the-loop, δηλαδή την επανατροφοδότηση από λεπτομερή προσομοίωση της μικροαρχιτεκτονικής, για τη βελτιστοποίηση των παραγόμενων προγραμμάτων. Με τον τρόπο αυτό επιτυγχάνεται υψηλή κάλυψη συγκεκριμένων δομών του επεξεργαστή και εξαιρετικά αποτελεσματική ανίχνευση σιωπηλών σφαλμάτων (SDCs), χωρίς να απαιτούνται ακραίες συνθήκες λειτουργίας.

Κατά συνέπεια, το Harpocrates[5] αποτελεί μια πιο στοχευμένη προσέγγιση, η οποία επιτρέπει την έγκαιρη διάγνωση δομικών αδυναμιών ή latent defects στον επεξεργαστή, που διαφορετικά ενδέχεται να παρέμεναν αδιάγνωστα από συμβατικές ή stress-testing μεθόδους.

2.4 Περιγραφή Πραγματικού Περιστατικού SDC

Στο πλαίσιο μιας εκτεταμένης μελέτης αξιοπιστίας που διεξήχθη από την Alibaba Cloud[1], καταγράφηκε ένα από τα πλέον χαρακτηριστικά παραδείγματα Silent Data Corruption (SDC) σε παραγωγικό περιβάλλον. Το περιστατικό αφορά έναν επεξεργαστή που παρήγαγε λανθασμένα αποτελέσματα σε υπολογισμούς ελέγχου ακεραιότητας δεδομένων (checksums). Συγκεκριμένα, σε μία εφαρμογή αποθήκευσης, η οποία χρησιμοποιούσε υπολογισμό checksum για να επαληθεύει την ορθότητα των δεδομένων που ανταλλάσσονταν μεταξύ client και daemon threads, παρατηρήθηκαν επανειλημμένα σφάλματα.

Το σφάλμα εκδηλωνόταν ως ασυμφωνία στο αποτέλεσμα του checksum, οδηγώντας την εφαρμογή να θεωρεί λανθασμένα ότι τα δεδομένα είχαν υποστεί αλλοίωση. Το αξιοσημείωτο στην περίπτωση αυτή ήταν ότι το σφάλμα δεν οφειλόταν σε λογική αποτυχία του λογισμικού, αλλά εμφανιζόταν αποκλειστικά όταν ο κώδικας εκτελούνταν σε έναν συγκεκριμένο επεξεργαστή της υποδομής. Μετά από εβδομάδες εντατικής ανάλυσης, εντοπίστηκε ότι η αιτία του προβλήματος ήταν ένα ελάττωμα σε συγκεκριμένο core του επεξεργαστή, το οποίο παρήγαγε λάθος αποτέλεσμα μόνο υπό ορισμένες θερμικές συνθήκες.

Οι μηχανικοί της Alibaba διαπίστωσαν ότι το SDC ήταν αναπαραγώγιμο, δηλαδή μπορούσε να επαναληφθεί με την εκτέλεση της ίδιας εντολής στον ίδιο core και σε αντίστοιχο θερμικό περιβάλλον. Αυτό απέκλεισε την πιθανότητα τυχαίου soft error και επιβεβαίωσε την υλική προέλευση του προβλήματος. Το περιστατικό αυτό υπογράμμισε την αναγκαιότητα χρήσης προηγμένων εργαλείων testing, ακόμα και σε φάσεις μετά την παραγωγή (in-production), και οδήγησε την Alibaba στην ανάπτυξη μηχανισμών όπως το Farron, που συνδυάζει προτεραιοποίηση testcases με θερμική παρακολούθηση για την έγκαιρη ανίχνευση τέτοιων φαινομένων.

Αυτό το περιστατικό αποτελεί χαρακτηριστικό παράδειγμα της πολυπλοκότητας και της δυσκολίας ανίχνευσης των SDCs, καθώς και του τρόπου με τον οποίο μικρά, τοπικά ελαττώματα μπορούν να έχουν ευρύ αντίκτυπο σε κρίσιμες υπηρεσίες cloud.

Κεφάλαιο 3

Αρχιτεκτονική και Υποδομή Μικροϋπηρεσιών

3.1 Αρχιτεκτονική Microservices

Η αρχιτεκτονική microservices (μικροϋπηρεσιών) είναι ένα μοντέλο ανάπτυξης λογισμικού όπου η εφαρμογή χωρίζεται σε ανεξάρτητα υποσυστήματα (services), καθένα από τα οποία εκτελεί μια συγκεκριμένη λειτουργία. Σε αντίθεση με τις μονολιθικές εφαρμογές, όπου όλη η λογική βρίσκεται σε έναν ενιαίο server, τα microservices επιτρέπουν μεγαλύτερη ευελιξία, επεκτασιμότητα και ανεξαρτησία στην ανάπτυξη.

Κάθε microservice τρέχει σε δικό του container – μια απομονωμένη και ελαφριά μονάδα που περιέχει το απαραίτητο περιβάλλον εκτέλεσης. Τα containers αποτελούν μια τεχνολογία πακεταρίσματος λογισμικού, όπου κάθε εφαρμογή «τυλίγεται» με όλα τα απαραίτητα αρχεία για την εκτέλεσή της – όπως ο κώδικας, οι βιβλιοθήκες, τα εργαλεία συστήματος και τα αρχεία ρυθμίσεων. Επιτρέπουν την απομόνωση των εφαρμογών μεταξύ τους, ενώ μοιράζονται τον ίδιο πυρήνα του λειτουργικού συστήματος (figure 3.1). Ένα container μπορεί να εκτελεστεί πανομοιότυπα σε οποιοδήποτε υπολογιστικό περιβάλλον, ανεξαρτήτως έκδοσης λειτουργικού συστήματος ή kernel version, αρκεί να ταιριάζει η αρχιτεκτονική (π.χ. x86_64).

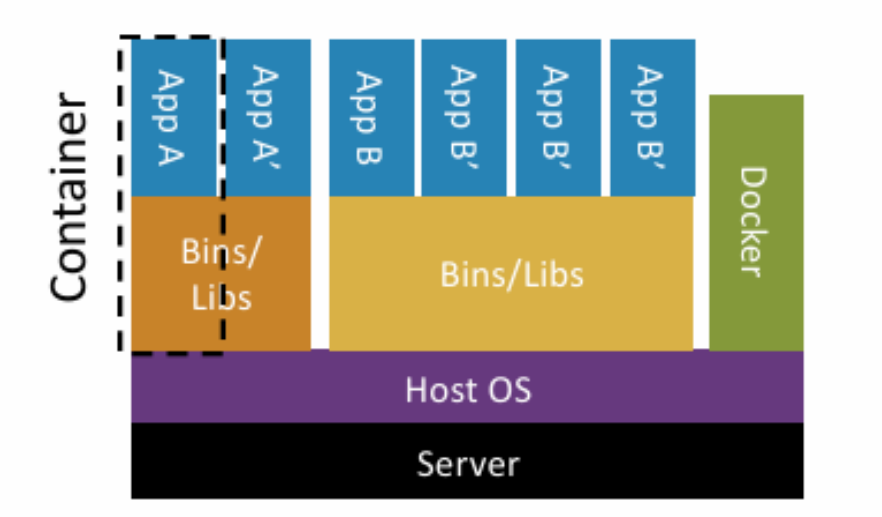

Figure 3.1 – Container-based application isolation.

Η αρχιτεκτονική microservices συχνά οργανώνεται σε τρία βασικά επίπεδα (figure 3.2):

- Frontend: Είναι υπεύθυνο για την αλληλεπίδραση με τον χρήστη (π.χ. εφαρμογές web ή mobile).
- Logic Layer (Application Tier): Εκεί εκτελείται η επιχειρησιακή λογική, όπως έλεγχοι, επεξεργασία δεδομένων και routing. Εδώ συχνά διατηρείται προσωρινή κατάσταση (state), όπως π.χ. ένα καλάθι αγορών.
- Database Layer: Περιέχει την κεντρική βάση δεδομένων.

Για καλύτερη απόδοση, πολλές υπηρεσίες χρησιμοποιούν επίπεδα caching (π.χ. Memcached, Redis), ώστε να μειωθεί ο φόρτος προς τη βάση δεδομένων.

Η επικοινωνία μεταξύ των microservices γίνεται μέσω RPC (Remote Procedure Calls) ή REST APIs. Τα RPCs επιτρέπουν σε ένα πρόγραμμα να καλέσει μια συνάρτηση που εκτελείται σε άλλο server σαν να ήταν τοπική, ενώ τα REST APIs βασίζονται στο πρωτόκολλο HTTP και χρησιμοποιούν μεθόδους όπως GET, POST, PUT, DELETE για την ανταλλαγή δεδομένων.

Ο φόρτος εργασίας μετακινείται μέσω αιτημάτων (requests) που φτάνουν από το frontend στο backend μέσω load balancers οι οποίοι κατανέμουν τα requests στα κατάλληλα μηχανήματα ή containers.

Η χρήση microservices διευκολύνει την παράλληλη ανάπτυξη από διαφορετικές ομάδες, την ανεξάρτητη συντήρηση και την ταχύτερη απόκριση σε αλλαγές.

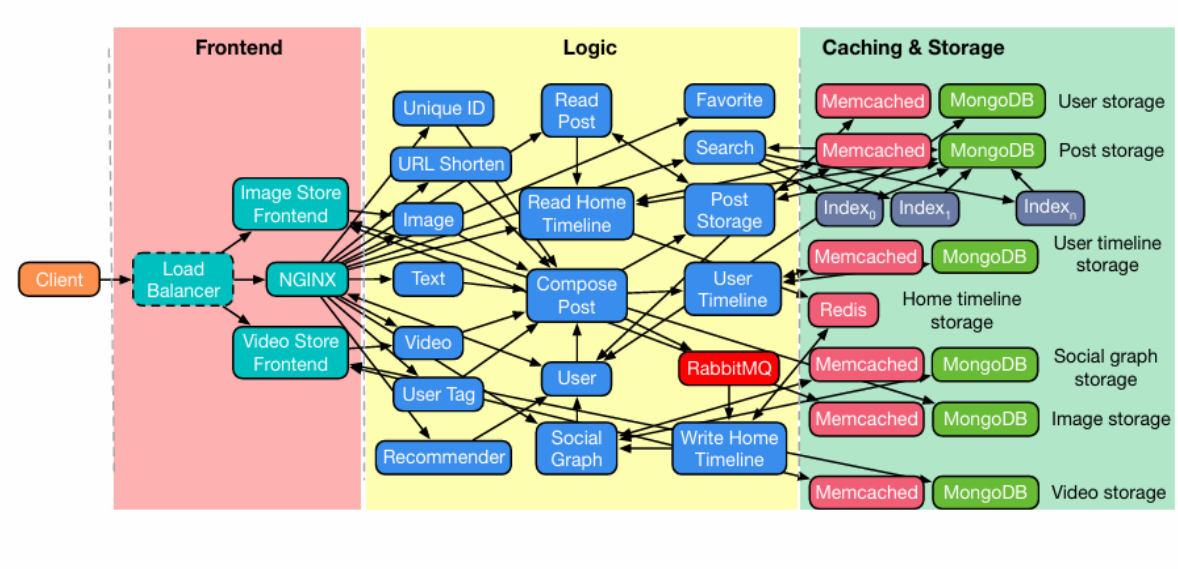

Figure 3.2 – DeathStarBench microservice multitier system overview.

3.2 Application που μελετήθηκε - μSuite HDSearch

Καθώς η αρχιτεκτονική microservices γίνεται όλο και πιο διαδεδομένη σε εφαρμογές μεγάλης κλίμακας, είναι απαραίτητο να υπάρχουν αξιόπιστα εργαλεία που να επιτρέπουν τη μελέτη της συμπεριφοράς και της απόδοσής τους. Σε αυτό το πλαίσιο, εντάσσεται το μSuite[9] – ένα σύνολο από microservice εφαρμογές που έχουν σχεδιαστεί ώστε να προσομοιώνουν On-Line Data Intensive (OLDI) περιβάλλοντα, όπως αναζητήσεις, προτάσεις ή επεξεργασία δεδομένων σε πραγματικό χρόνο.

Μια από τις βασικές εφαρμογές του μSuite είναι το HDSearch, το οποίο υλοποιεί ένα σύστημα αναζήτησης παρόμοιων εικόνων. Η λειτουργία του βασίζεται στην εξαγωγή και σύγκριση διανυσμάτων χαρακτηριστικών εικόνων. Το HDSearch ακολουθεί πλήρως την αρχιτεκτονική τριών επιπέδων που περιγράψαμε προηγουμένως (figure 3.3):

- Το Front-end microservice (load generator) προσομοιώνει workload και εξάγει το διάνυσμα χαρακτηριστικών της κάθε εικόνας.

- Το Mid-tier microservice εφαρμόζει τεχνικές Locality-Sensitive Hashing (LSH) και δρομολογεί το αίτημα στα κατάλληλα leaf services.

- Τα Leaf microservices (bucket service) εκτελούν την αναζήτηση σε επιμέρους τμήματα της βάσης δεδομένων και επιστρέφουν τις πιο παρόμοιες εικόνες στο mid-tier για τελική συγχώνευση των αποτελεσμάτων.

Για τη δημιουργία φορτίου, χρησιμοποιήθηκε ο Open-loop Load Generator, ο οποίος στέλνει αιτήματα με ρυθμό που καθορίζεται από κατανομή Poisson. Στον open-loop τα αιτήματα αποστέλλονται ανεξάρτητα, επιτρέποντας τη μελέτη του συστήματος σε συνθήκες συμφόρησης και ιδιαίτερης tail latency.

Το HDSearch χρησιμοποιήθηκε στο πλαίσιο αυτής της εργασίας σε περιβάλλον Docker Compose[13] που εγκαταστάθηκε σε κατανεμημένες μηχανές στο CloudLab[12], επιτρέποντας την εύκολη παρακολούθηση, απομόνωση και επανεκκίνηση των επιμέρους υπηρεσιών. Η συγκεκριμένη εγκατάσταση αποτέλεσε τη βάση για τη μελέτη του συστήματος σε συνθήκες Silent Data Corruption (SDC), με τη χρήση δυναμικού εργαλείου παρακολούθησης εκτέλεσης Intel Pin Tool για την εισαγωγή τεχνητών σφαλμάτων σε κρίσιμες εντολές του συστήματος.

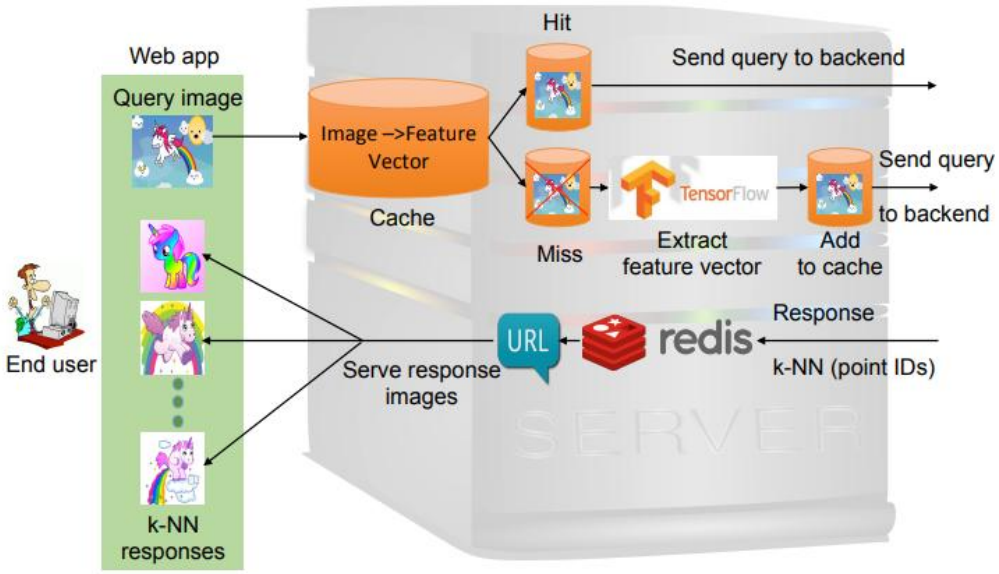

Figure 3.3 – HDSearch: frontend presentation microservice

Κεφάλαιο 4

Fault injection Pintool

4.1 Dynamic Binary Instrumentation και Intel Pin tool

Η πειραματική μελέτη που πραγματοποιείται στην παρούσα εργασία βασίζεται στη δυναμική ανάλυση εκτελέσιμου κώδικα (Dynamic Binary Instrumentation – DBI). Dynamic Binary Instrumentation είναι η διαδικασία εισαγωγής νέου κώδικα σε ένα πρόγραμμα κατά τη διάρκεια της εκτέλεσής του. Ο νέος αυτός κώδικας μπορεί να παρακολουθεί (monitor) ή ακόμη και να αλλάζει τη συμπεριφορά του προγράμματος.

Ο μηχανισμός ενός DBI εργαλείου, περιλαμβάνει μια σειρά από εσωτερικά υποσυστήματα (figure 4.1). Ο Code Fetcher αναλαμβάνει να διαβάσει τον εκτελέσιμο κώδικα, ενώ το Instrumentation Engine προσθέτει τον επιπλέον κώδικα (instrumentation) στα κατάλληλα σημεία, όπως σε blocks ή εντολές. Στη συνέχεια, ο JIT Compiler μεταγλωττίζει αυτόν τον νέο, τροποποιημένο κώδικα σε αποδοτική μορφή. Ο κώδικας αποθηκεύεται προσωρινά στον Code Cache, ώστε να μπορεί να επαναχρησιμοποιηθεί γρήγορα. Ο Dispatcher δρομολογεί τη σωστή ροή εκτέλεσης μέσα στο σύστημα. Όλα αυτά εκτελούνται πάνω στο λειτουργικό περιβάλλον (OS – Machine) του υπολογιστή.

Από την πλευρά του χρήστη, το DBI εργαλείο αποτελείται από συναρτήσεις που δηλώνουν τι πρέπει να παρακολουθηθεί και τι να εκτελεστεί. Για παράδειγμα, η instrument_bb() καθορίζει ποια blocks θα παρακολουθούνται, προσθέτοντας εντολές που καλούν τις συναρτήσεις analysis callbacks κατά τη διάρκεια της εκτέλεσης. Η bb_callback() περιέχει τον κώδικα που εκτελείται κάθε φορά που φτάνει το πρόγραμμα σε αυτά τα blocks (παράδειγμα στο figure 4.2). Η συνάρτηση init() αρχικοποιεί το εργαλείο και ενεργοποιεί τη λειτουργία παρακολούθησης.

Η επικοινωνία μεταξύ του PinTool και του DBI engine γίνεται μέσω μιας ειδικής βιβλιοθήκης (API), που επιτρέπει στο εργαλείο να ελέγχει και να καταγράφει τη συμπεριφορά του εκτελέσιμου προγράμματος με μεγάλη ακρίβεια.

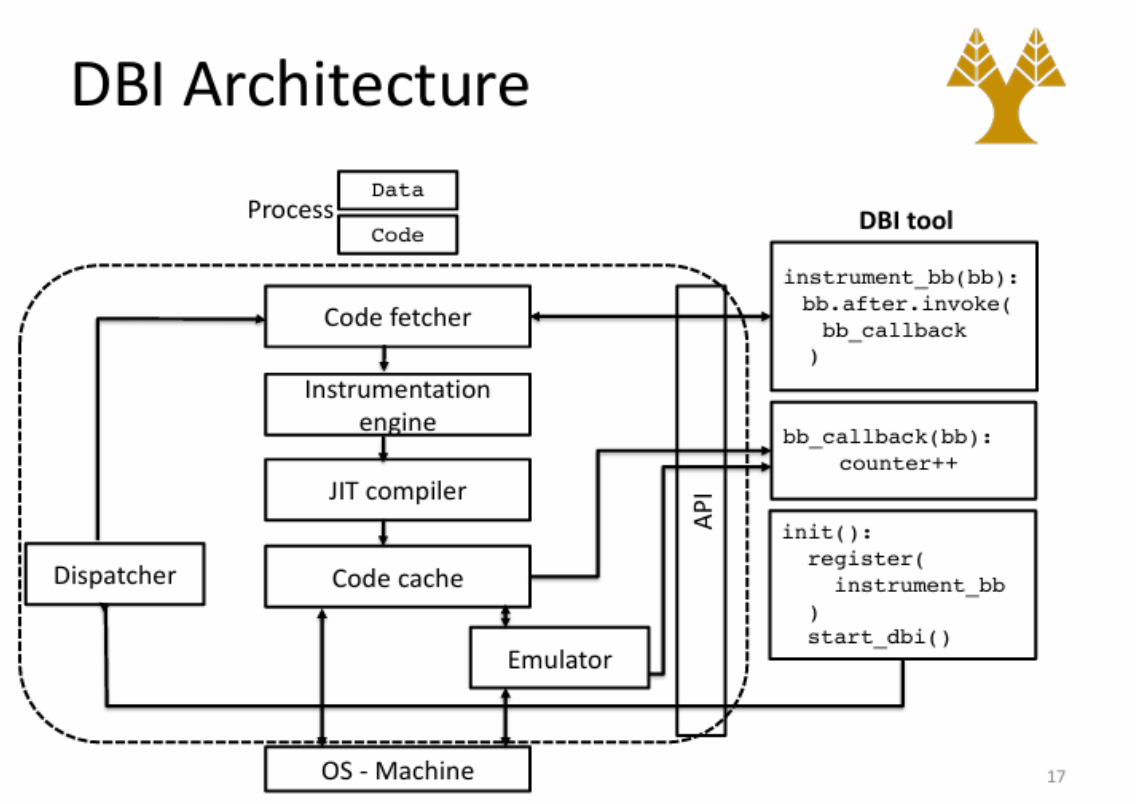

Figure 4.1 – DBI system and tool interface.

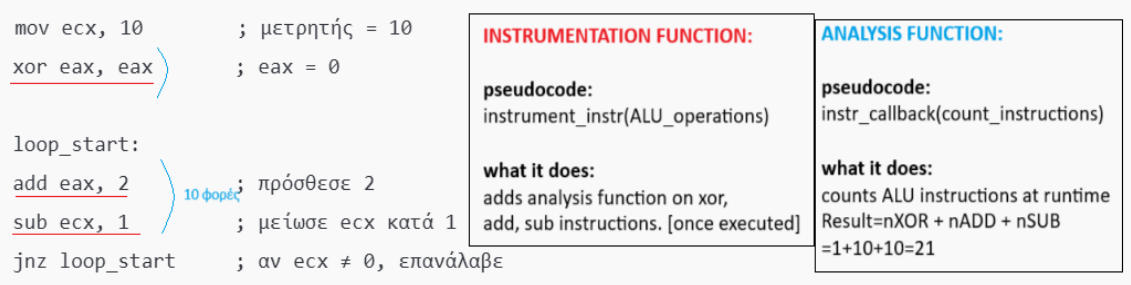

Figure 4.2 – Basic example of code instrumentation and runtime analysis

Η ανάγκη για την εισαγωγή ελεγχόμενων σφαλμάτων σε επιλεγμένα σημεία του εκτελέσιμου κώδικα, καθώς και για την παρακολούθηση της συμπεριφοράς των microservices με υψηλό βαθμό ακρίβειας και λεπτομέρειας, ανέδειξε την απαίτηση για ένα εξειδικευμένο εργαλείο δυναμικής ανάλυσης.

Ένα από τα πιο γνωστά και ισχυρά εργαλεία DBI είναι το Intel Pin[7], το οποίο επιτρέπει τη δυναμική παρεμβολή κώδικα χωρίς να απαιτείται αλλαγή στο εκτελέσιμο αρχείο. Το Pin λειτουργεί σαν ένας just-in-time μεταγλωττιστής: παίρνει σαν είσοδο ένα block κώδικα και το μεταφράζει σε νέο κώδικα με τα analysis functions. Ο χρήστης γράφει τα δικά του εργαλεία (Pintools), τα οποία περιλαμβάνουν instrumentation συναρτήσεις που ορίζουν πού θα προστεθεί κώδικας, και analysis συναρτήσεις που υλοποιούν τι θα εκτελεστεί σε κάθε σημείο (όπως μέτρηση εντολών, παρακολούθηση τιμών καταχωρητών ή εισαγωγή σφαλμάτων).

Το Intel Pin[7] υποστηρίζει πολλαπλά επίπεδα παρακολούθησης (instrumentation granularity), επιτρέποντας την παρέμβαση τόσο σε επίπεδο μεμονωμένων εντολών και συναρτήσεων, όσο και σε επίπεδο ολόκληρων εκτελέσιμων αρχείων (image-level). Στο πλαίσιο της παρούσας εργασίας αξιοποιούνται κυρίως οι τεχνικές Instruction Instrumentation και Routine Instrumentation, με στόχο τον εντοπισμό και τη στοχευμένη παρέμβαση σε κρίσιμες εντολές ή σε κάποιες συναρτήσεις της εφαρμογής HDSearch. Η επιλογή του Intel Pin[7] καθιστά εφικτή την ακριβή και ελεγχόμενη

εισαγωγή σφαλμάτων, υποστηρίζοντας τη μελέτη της επίδρασης φαινομένων σιωπηρής αλλοίωσης δεδομένων.

4.2 Μηχανισμός Έγχυσης Σφαλμάτων

Καθώς δεν υπάρχει κάποιο υπάρχον εργαλείο fault injection που να μπορεί να χρησιμοποιηθεί απευθείας σε περιβάλλοντα microservice εφαρμογών σε συνδυασμό με την ανάγκη για μελέτη της αξιοπιστίας τέτοιων συστημάτων οδήγησε στην ανάπτυξη αυτοματοποιημένων εργαλείων ειδικά σχεδιασμένων για αυτά τα περιβάλλοντα.

Η πειραματική αξιολόγηση των SDCs απαιτεί τον σχεδιασμό κατάλληλων εργαλείων που να προσομοιώνουν με ακρίβεια τον τρόπο εμφάνισης σφαλμάτων σε πραγματικά, ελαττωματικά συστήματα. Στο πλαίσιο αυτό, αναπτύχθηκε ένα custom Pintool, το οποίο επιτρέπει την εισαγωγή bit-level σφαλμάτων κατά την εκτέλεση σε κρίσιμα σημεία του application, βασισμένο στη δυναμική ανάλυση κώδικα του Intel Pin Tool.

Η εμφάνιση σιωπηλών αλλοιώσεων δεδομένων σε σύγχρονες υπολογιστικές πλατφόρμες έχει συνδεθεί, σύμφωνα με πρόσφατες μελέτες – και κυρίως τη μελέτη της Alibaba[1] –με μονάδες υπολογισμού του επεξεργαστή, όπως η Arithmetic Logic Unit (ALU), η Floating Point Unit (FPU) και οι SIMD επεκτάσεις τύπου SSE/AVX. Οι μονάδες αυτές, λόγω της αυξανόμενης πολυπλοκότητας και της υψηλής αξιοποίησής τους από εφαρμογές υψηλών επιδόσεων, εμφανίζουν σφάλματα που μπορούν να περάσουν απαρατήρητα, προκαλώντας ωστόσο σημαντικές αλλοιώσεις στην ακρίβεια ή την ορθότητα των αποτελεσμάτων.

Λαμβάνοντας υπόψη τα παραπάνω ευρήματα, κατά τον σχεδιασμό των εργαλείων fault injection της παρούσας εργασίας δόθηκε έμφαση στην παρακολούθηση και παρέμβαση σε εντολές που ενεργοποιούν τις παραπάνω μονάδες, δηλαδή:

- ALU operations σε General Purpose Registers (GPRs)
- Floating Point και SSE εντολές μέσω XMM καταχωρητών

Η προσομοίωση του σφάλματος γίνεται μέσω της εγγραφής ενός τροποποιημένου αποτελέσματος στον καταχωρητή προορισμού της εντολής, πριν αυτό δεσμευθεί (write-

back) στο αρχικό σημείο προορισμού. Πιο συγκεκριμένα, αφού ολοκληρωθεί η αριθμητική πράξη από τη λειτουργική μονάδα, παρεμβάλλεται ένα τροποποιημένο αποτέλεσμα, μοντελοποιώντας έτσι ρεαλιστικά τη χρονική στιγμή και τον τρόπο με τον οποίο ένα bit flip θα μπορούσε να προκύψει στο υλικό.

Η βασική στρατηγική του πειραματικού σχεδιασμού βασίστηκε στη διάκριση των SDCs σε δύο βασικές κατηγορίες, οι οποίες αντιστοιχούν στους πιο συχνούς τρόπους εμφάνισης τέτοιων σφαλμάτων σε ελαττωματικούς επεξεργαστές.

Περίπτωση 1: Σφάλματα σε ALU εντολές

Η πρώτη κατηγορία αφορά σφάλματα που προκύπτουν σε λογικές και αριθμητικές πράξεις που εκτελούνται μέσω της Arithmetic Logic Unit (ALU), όπως and, or, add, mul,κ.ά σε general purpose registers (rax,rdi κλπ.).

Το injection tool αποφεύγει οποιαδήποτε επέμβαση σε κρίσιμους καταχωρητές, των οποίων η τροποποίηση μπορεί να οδηγήσει σε άμεση αποτυχία του προγράμματος και να αναιρέσει τον στόχο της μελέτης, που είναι η προσομοίωση σιωπηλών αλλοιώσεων χωρίς εμφανές σφάλμα ή κατάρρευση.

Συγκεκριμένα, εξαιρούνται από κάθε είδους injection οι καταχωρητές RSP (stack pointer) και RBP (base pointer). Ο RSP διατηρεί τη διεύθυνση της κορυφής της στοίβας (stack), και κάθε αλλοίωση της τιμής του ενδέχεται να οδηγήσει σε πρόσβαση σε μη έγκυρη μνήμη, προκαλώντας segmentation fault ή crash. Αντίστοιχα, ο RBP χρησιμοποιείται ως σταθερό σημείο αναφοράς (frame pointer) για την πρόσβαση σε παραμέτρους και τοπικές μεταβλητές εντός κάθε stack frame. Ένα bit flip στον RBP μπορεί να οδηγήσει σε λανθασμένη αποκωδικοποίηση δεδομένων ή, χειρότερα, σε ακούσια εκτέλεση άλλων περιοχών μνήμης.

Επιπλέον, εξαιρούνται ρητά από το fault injection και οι καταχωρητές κατάστασης (flag registers), οι οποίοι φέρουν πληροφορίες για το αποτέλεσμα λογικών ή αριθμητικών πράξεων.

Καθώς ο στόχος είναι η προσομοίωση ρεαλιστικών SDCs που περνούν απαρατήρητα, και όχι η πρόκληση εμφανών σφαλμάτων, η αλλοίωση αυτών των καταχωρητών θα οδηγούσε σε μη αντιπροσωπευτικά ή καταστροφικά αποτελέσματα. Για τον λόγο αυτό, το Pintool φιλτράρει εξ αρχής αυτές τις περιπτώσεις, διατηρώντας το injection εντός των λειτουργικά "ανεκτών" καταχωρητών.

Στην περίπτωση των πράξεων αριθμητικής/λογικής μονάδας, το σφάλμα προσομοιώνεται ως αντιστροφή ενός τυχαίου bit στην τιμή του output register. Το εργαλείο εντοπίζει ALU εντολές και σε επιλεγμένα injection spots, εκτελεί ένα bitwise XOR με μάσκα εντός των πρώτων 32 bits του καταχωρητή.

Αυτό επιτυγχάνεται με την ανάγνωση της τρέχουσας τιμής του καταχωρητή (REG_GPR), την αντιστροφή ενός LSB (π.χ. bit 7) και την εγγραφή της νέας τιμής στο execution context. Η επέμβαση πραγματοποιείται αμέσως μετά την εκτέλεση της αντίστοιχης εντολής και πριν συνεχιστεί η κανονική ροή εκτέλεσης του προγράμματος, διασφαλίζοντας ότι το σφάλμα θα επηρεάσει μόνο τις επόμενες πράξεις.

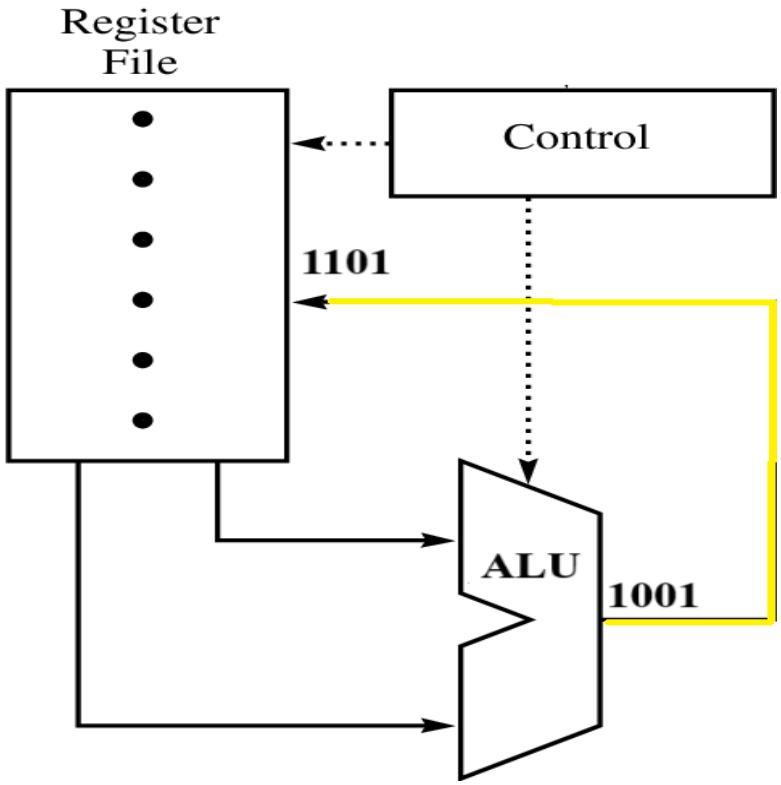

Figure 4.3 – Example of SDC in ALU operation.

Περίπτωση 2: Σφάλματα σε Floating Point και SIMD εντολές

Η δεύτερη κατηγορία σφαλμάτων επικεντρώνεται σε εκείνα που εντοπίζονται κατά την εκτέλεση εντολών κινητής υποδιαστολής (floating point) και SIMD εντολών, με έμφαση στις SSE (Streaming SIMD Extensions), οι οποίες εφαρμόζονται ταυτόχρονα σε πολλαπλά δεδομένα, όπως υπογραμμίζεται και στη μελέτη της Alibaba. Η μελέτη αυτή τεκμηριώνει ότι οι εντολές που αφορούν αριθμητικούς υπολογισμούς σε Floating Point Units (FPUs) είναι ιδιαίτερα ευάλωτες στην εμφάνιση Silent Data Corruptions (SDCs), με τα σφάλματα να εντοπίζονται συχνότερα στο κλασματικό μέρος της αριθμητικής αναπαράστασης, σύμφωνα με το πρότυπο IEEE-754. Αυτά τα σφάλματα προκαλούν μικρές αποκλίσεις στην ακρίβεια των αποτελεσμάτων, οι οποίες ενδέχεται να διαφεύγουν της ανίχνευσης από συμβατικούς μηχανισμούς ελέγχου σφαλμάτων. Αντίστοιχες ευπάθειες έχουν παρατηρηθεί και σε SIMD εντολές, όπου το ίδιο υπολογιστικό πρότυπο εφαρμόζεται ταυτόχρονα σε πολλαπλά δεδομένα, αυξάνοντας την πιθανότητα πολλαπλής διάδοσης του σφάλματος.

Για την ανάλυση αυτής της κατηγορίας, ο σχεδιασμός του εργαλείου Pintool επεκτάθηκε ώστε να εντοπίζει και να κατηγοριοποιεί τις floating point εντολές σε xmm registers με βάση δύο βασικά χαρακτηριστικά: το επίπεδο ακρίβειας (single ή double precision) και τον τύπο του υπολογισμού (scalar ή vectorized/SSE). Η κατηγοριοποίηση αυτή επέτρεψε την εφαρμογή στοχευμένων παρεμβάσεων, μέσω bit flip στο fraction part της αναπαράστασης IEEE-754, με βάση το αντίστοιχο μήκος: 23 bits για single precision και 52 bits για double precision.

Ο πίνακας που ακολουθεί συνοψίζει τις παραμέτρους του fault injection για κάθε κατηγορία:

Κατηγορία	Target bits	Bit flip περιορισμός
Single Precision - Scalar	32 bits	XOR με mask % 23 (LSBs)
Double Precision - Scalar	64 bits	XOR με mask % 52 (LSBs)
Single Precision - Vector	4 × 32 bits (XMM)	Επιλογή 1 από 4, XOR % 23
Double Precision - Vector	2 × 64 bits (XMM)	Επιλογή 1 από 2, XOR % 52

Table 4.1 – Floating point target categories

Η υλοποίηση περιλάμβανε έλεγχο των operands κάθε floating point εντολής, ώστε να κατηγοριοποιηθούν με βάση precision και είδος υπολογισμού αλλα και να εξαιρεθούν μη αριθμητικές πράξεις (π.χ. ucomiss, movaps) ή εντολές αναδιάταξης (vextracti128).

Συμπερασματικά στα figures 4.4 – 4.6 αποσαφηνίζουν τη δομή και λειτουργία των XMM καταχωρητών, οι οποίοι είναι οι βασικοί καταχωρητές FPU, SSE στην αρχιτεκτονική x86-64 και χρησιμοποιούνται τόσο για scalar όσο και για vector floating point υπολογισμούς. Όπως φαίνεται στο Σχήμα 2, εντολές τύπου mulss επηρεάζουν μόνο το κατώτερο τμήμα του XMM καταχωρητή, ενώ οι mulps εφαρμόζονται ταυτόχρονα σε τέσσερα στοιχεία 32-bit, γεγονός που απαιτεί διαφορετική στρατηγική επιλογής στόχου κατά το fault injection. Η προσέγγιση αυτή επιτρέπει τη ρεαλιστική προσομοίωση σφαλμάτων με πιθανό αντίκτυπο στην ακρίβεια των αποτελεσμάτων, χωρίς να διαταράσσεται η εκτελεσιμότητα του συστήματος.

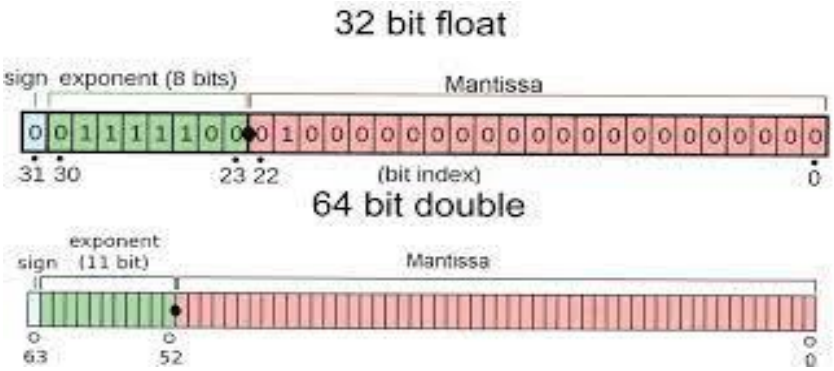

Figure 4.4 – Bit structure of single and double precision floating-point numbers.

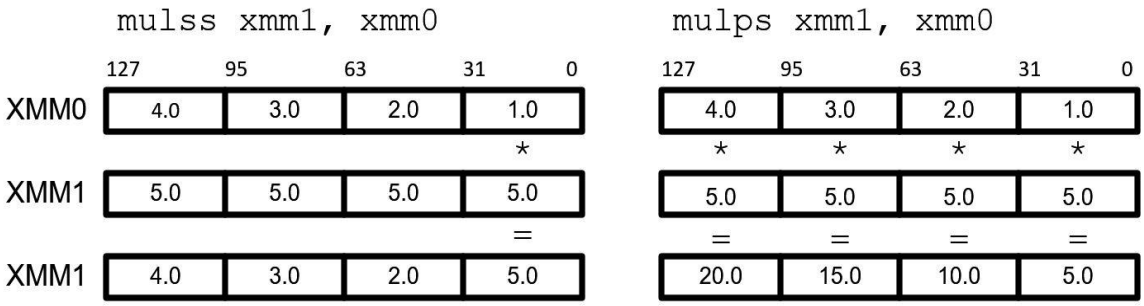

Figure 4.5 – Detecting scalar or vector operations of precision through instruction operand

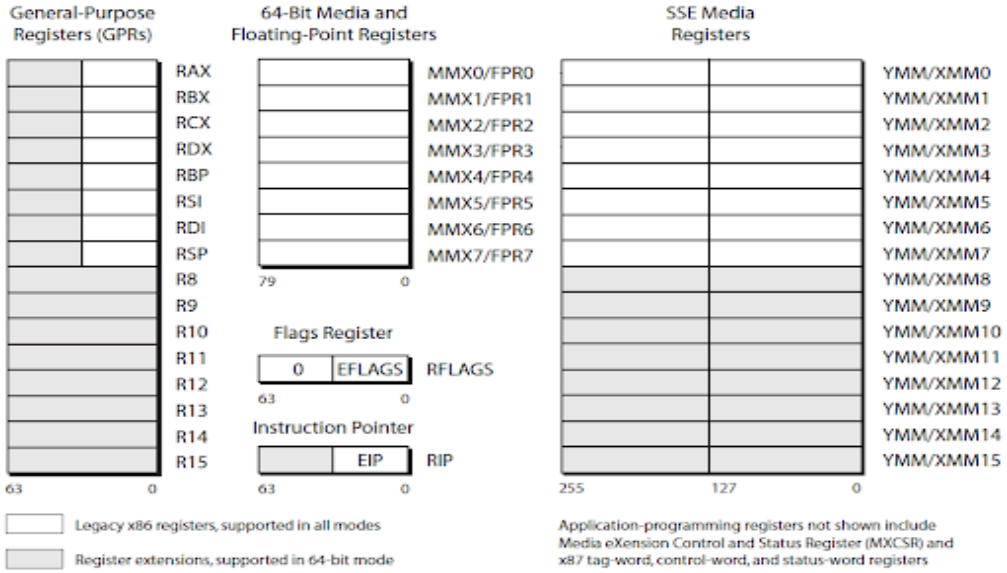

Figure 4.6 – x86-64 register organization

4.3 Fault injection window και Query Mapping

Η στοχευμένη μελέτη της ευπάθειας της εφαρμογής σε σιωπηλές αλλοιώσεις δεδομένων (SDCs) απαιτεί τον σαφή ορισμό ενός ενεργού injection window, εντός του οποίου ενεργοποιούνται οι μηχανισμοί παρακολούθησης και σφάλματος. Για να καταστεί δυνατή η δυναμική ενεργοποίηση αυτών των λειτουργιών, το Pin Tool βασίζεται στη σήμανση της εκτέλεσης μέσω προκαθορισμένων κλήσεων που εισάγονται από τον χρήστη στον κώδικα της εφαρμογής.

Συγκεκριμένα, το εργαλείο παρακολουθεί μέσω routine instrumentation την εκτέλεση των συναρτήσεων FaultInjectionBegin() και FaultInjectionEnd(), οι οποίες προσδιορίζουν την έναρξη και λήξη του injection window αντίστοιχα. Κατά την είσοδο στη FaultInjectionBegin(), το εργαλείο ενημερώνει εσωτερικές σημαίες (π.χ. inject = true, trace = true) ώστε να ξεκινήσει η καταγραφή των εντολών και η ενεργοποίηση του fault injection μηχανισμού. Αντίστροφα, με την είσοδο στη FaultInjectionEnd(), οι σημαίες αυτές μηδενίζονται, τερματίζοντας προσωρινά τη λειτουργία του εργαλείου. Η χρήση αυτής της τεχνικής προσφέρει προσαρμοστικότητα και επιτρέπει την εύκολη μεταφορά του εργαλείου σε άλλες εφαρμογές, απαιτώντας ελάχιστες αλλαγές στον κώδικα της εκάστοτε στόχευσης.

Επιπλέον, λαμβάνοντας υπόψη ότι τα microservices λειτουργούν με βάση το μοντέλο αίτημα-απόκριση (query-response), όπου κάθε εκτέλεση αντιστοιχεί σε ένα ερώτημα χρήστη, η αντιστοίχιση κάθε εκτέλεσης με συγκεκριμένο query επιτυγχάνεται μέσω δύο επιπλέον συναρτήσεων:

- Η QueryBegins(int id) καλείται κατά την έναρξη επεξεργασίας ενός νέου query και καταχωρεί το αναγνωριστικό id στο thread-local storage του αντίστοιχου νήματος.

- Η QueryResults(int id_result) καλείται κατά την ολοκλήρωση επεξεργασίας του query και παρέχει το τελικό αναγνωριστικό αποτέλεσμα (id_result), το οποίο καταγράφεται από το εργαλείο ώστε να συσχετιστεί με το αντίστοιχο query id και instruction trace.

Με την παρεμβολή (interposition) των κλήσεων αυτών, το Pintool έχει τη δυνατότητα να συνδέσει το κάθε instruction του low level εκτελέσιμου κώδικα με το αντίστοιχο query id και response id. Αυτός ο διαχωρισμός καθιστά δυνατή την ακριβή και ανεξάρτητη ανάλυση των επιπτώσεων κάθε fault injection, τόσο στη λειτουργική συμπεριφορά (μέσω του αποτελέσματος), όσο και στον έλεγχο ροής, διευκολύνοντας τον εντοπισμό αποκλίσεων σε σχέση με τις golden εκτελέσεις.

4.4 Παράλληλη Παρακολούθηση Golden και Faulty Εκτελέσεων

Ένας ευρέως διαδεδομένος τρόπος ανίχνευσης και αντιμετώπισης SDCs είναι η τριπλή εκτέλεση (Triple Modular Redundancy – TMR)[4], όπου το ίδιο πρόγραμμα εκτελείται σε τρεις διαφορετικούς επεξεργαστές και η τελική έξοδος καθορίζεται μέσω ψηφοφορίας. Ωστόσο, η προσέγγιση αυτή είναι ιδιαίτερα κοστοβόρα και μη πρακτική για περιβάλλοντα microservices, όπου κυρίαρχες απαιτήσεις είναι η χαμηλή καθυστέρηση και η αποδοτική αξιοποίηση πόρων.

Σαν βασική ιδέα την πιο πάνω μέθοδο ένα αλλο σημαντικό σημείο του tool ήταν η δημιουργία ταυτόχρονης παρακολούθησης των αποκρίσεων (replies) και της ροής εκτέλεσης (instruction-level control flow) τόσο για τις εκτελέσεις που φέρουν σφάλμα όσο και για τις "καθαρές" εκτελέσεις αναφοράς (golden runs). Η πρόκληση έγκειται στο να παραχθούν κρίσιμα τμήματα κώδικα δύο φορές σε ένα run, χωρίς ωστόσο να επηρεάζεται η κανονική λειτουργία του συστήματος. Για τον σκοπό αυτό, μελετήθηκαν δύο διαφορετικές προσεγγίσεις, βασισμένες σε multithreading και multiprocess τεχνικές.

4.4.1 Παρακολούθηση Εκτέλεσης μέσω shadow thread

Η πρώτη προσέγγιση αφορά την υποστήριξη παράλληλων νημάτων (threads). Αναπτύχθηκε ειδικό Pintool το οποίο εντοπίζει και παρακολουθεί τη δημιουργία shadow threads, τα οποία εκτελούν την ίδια συνάρτηση με τα πραγματικά νήματα, αλλά σε ξεχωριστό χώρο μνήμης και χωρίς να αλληλεπιδρούν με το υπόλοιπο σύστημα. Κατά την έναρξη ενός νέου ερωτήματος, η εφαρμογή δημιουργεί ένα shadow thread (σε

κατάσταση detached), το οποίο "αντιγράφει" τη συμπεριφορά των πραγματικών νημάτων, εκτελώντας ανεξάρτητα την process_request με παραμέτρους που δεσμεύονται εκ των προτέρων στη heap. (Figure 4.7)

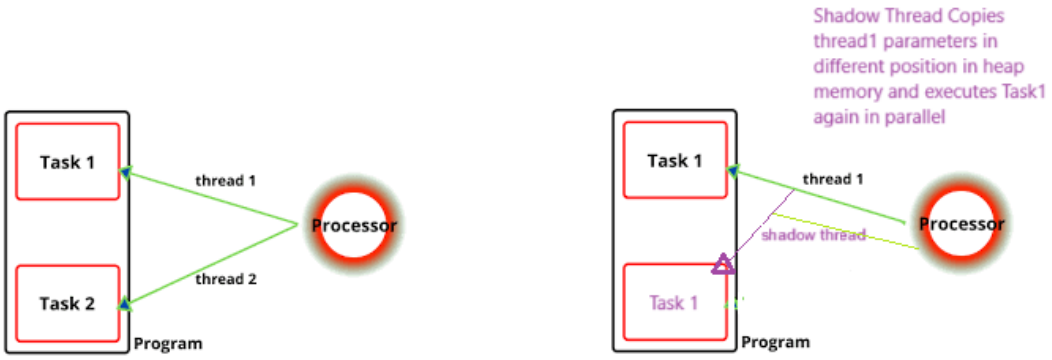

Figure 4.7 – Shadow thread execution model

Το εργαλείο χρησιμοποιεί Thread-Local Storage (TLS) για να αποθηκεύει ανά νήμα σημαντικά μεταδεδομένα, όπως query_id, response_id, instruction counter κρίσιμων εντολών καθώς και flags που δηλώνουν αν το νήμα βρίσκεται εντός injection window ή αν έχει ενεργοποιημένη την καταγραφή trace.

Το Pintool αναγνωρίζει ποια νήματα είναι shadow μέσω application-level tagging και ποιά όχι και τα θέτει είτε ως golden execution είτε ως faulty execution ανάλογα με τις ανάγκες του πειράματος. Επίσης κατα την δειάρκεια εκτέλεσεις ελέγχει τα μεταδεδομένα για να δεί κατα πόσο θα χρησημοποιήσει ορισμένα callbacks κατά την εκτέλεση των εντολών, π.χ "αν δεν είναι shadow thread ή είναι εκτός injection window μην κάνεις έγχυση σφάλματος" ή "αν το trace είναι off μην τυπώσεις αυτην την εντολή"

Για κάθε query, το εργαλείο αποθηκεύει ξεχωριστά: Το instruction sequence ,το αποτέλεσμα της συνάρτησης , τα οποία τυπώνει το thread που τα εξυπηρετά

Κατά το πέρας του injection window, η TLS δομή επαναφέρεται σε default τιμές (π.χ. inject=false, trace=false, instruction_count=0, is_shadow=false) ώστε να τερματιστεί η παρακολούθηση και να προετοιμαστεί η υποδομή για νέο query. Το shadow thread τερματίζεται αμέσως μετά την ολοκλήρωση της λειτουργίας του, χωρίς να επηρεάζει την ροή του προγράμματος.

4.4.2 Παρακολούθηση Εκτέλεσης μέσω fork()

Παρ' όλα αυτά, αφού η μελέτη εστιάζει στην ανάλυση του control flow μετά από την εμφάνιση SDCs και όχι στην αποτίμηση επιδόσεων, το κύριο εργαλείο που χρησιμοποιήθηκε ήταν η υλοποίηση μέσω instrumentation σε επίπεδο πολλαπλών διεργασιών, και συγκεκριμένα με χρήση της fork(), προκειμένου να επιτευχθεί πλήρης απομόνωση των εκτελέσεων. Η επιλογή αυτή υπαγορεύτηκε τόσο από λόγους απλότητας όσο και από την ανάγκη για ντετερμινιστική συμπεριφορά (βλ. Κεφάλαιο 5).

Η χρήση της fork() απαιτεί ιδιαίτερη προσοχή στο πλαίσιο microservice εφαρμογών, καθώς θα πρέπει να εφαρμόζεται αποκλειστικά σε σημεία όπου δεν υπάρχει thread concurrency, με threads όπου εκτελούν κοινή εργασία.

Καθώς ο parent περιμένει με waitpid το child process να τερματίσει, το ίδιο εκτελεί το window με το ίδιο request που παρέλαβε, και αποθηκεύει το reply, χωρίς να το αποστείλει στο mid-tier. Το child τερματίζεται πριν την ολοκλήρωση τις συνάρτησης υπολογισμών, ώστε να μην επηρεάσει την εξωτερική συμπεριφορά του συστήματος. Όταν τερματίσει το child, το parent process συνεχίζει την εκτέλεση κανονικά και παράγει το τελικό αποτέλεσμα.

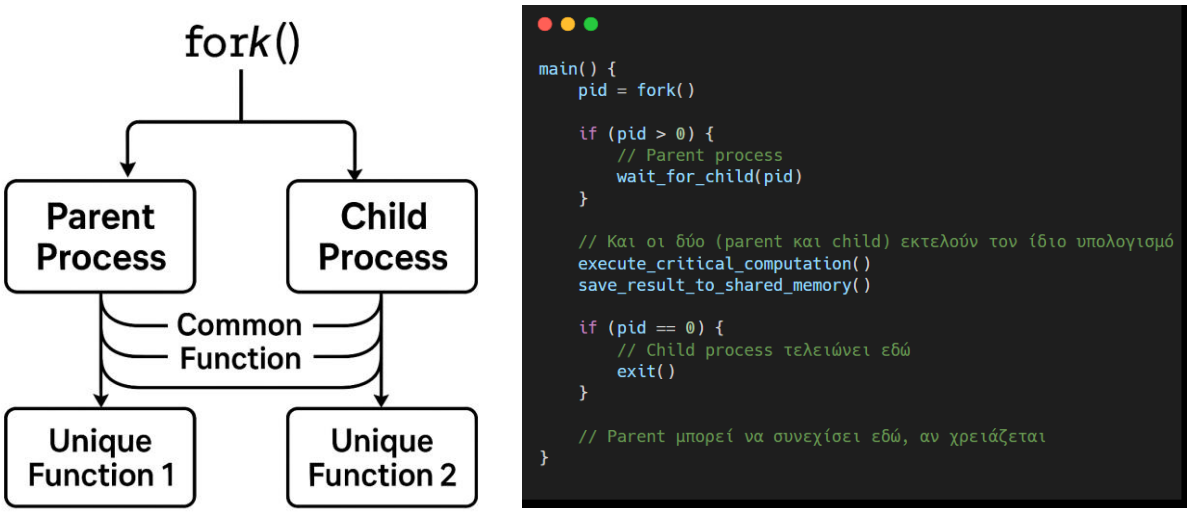

Figure 4.8 – Parent and child process behavior after fork()

Μιας και το child process αντιγράφει την κατάσταση μόνο του κυριου thread που κάνει τις υλοποιήσεις[11] (figure 4.9). Στο pin tool είναι απαραίτητο τα νήματα που σχετίζονται με τη διαχείριση GRPC να αγνοούνται από την υλοποίηση , προκειμένου να διασφαλιστεί ντετερμινιστική ροή εντολών (instruction sequence) μεταξύ parent και child διεργασιών.

Ενώ είμαστε σίγουροι ότι το child δημιουργήθηκε από το κύριο νήμα που θέλουμε να εγχείσουμε σφάλμα, αν ο parent process έχει ταυτόχρονα με την κύρια λειτουργία του, νήματα που κάνουν διαφορετικές εργασίες π.χ. grpc threads, το instruction sequence του parent ενδέχεται να έχει έξτρα εντολές που προέρχονται από αυτά. Αυτό είναι ένα από τα κύρια προβλήματα που εμφανίζονται με την μέθοδο του fork() που όμως με την χρήση δομής TLS και το injection window είναι εύκολα επιλίσιμο πρόβλημα μιας και το thread που εκτελεί άσχετες λειτουργίες εκτός του μελετημένου σημείου δεν θα μπεί ποτέ στο window, και στη TLS δομή του στο tool δεν θα τεθούν ποτέ τα flags λόγω του ότι είναι εκτός του window αυτου. Επομένως κατά τη χρηση των callbacks θα αγνοηθούν οι εντολές αυτές και δεν θα τυπωθούν στο instruction sequence ούτε θα εγχειθούν σε αυτές σφάλματα.

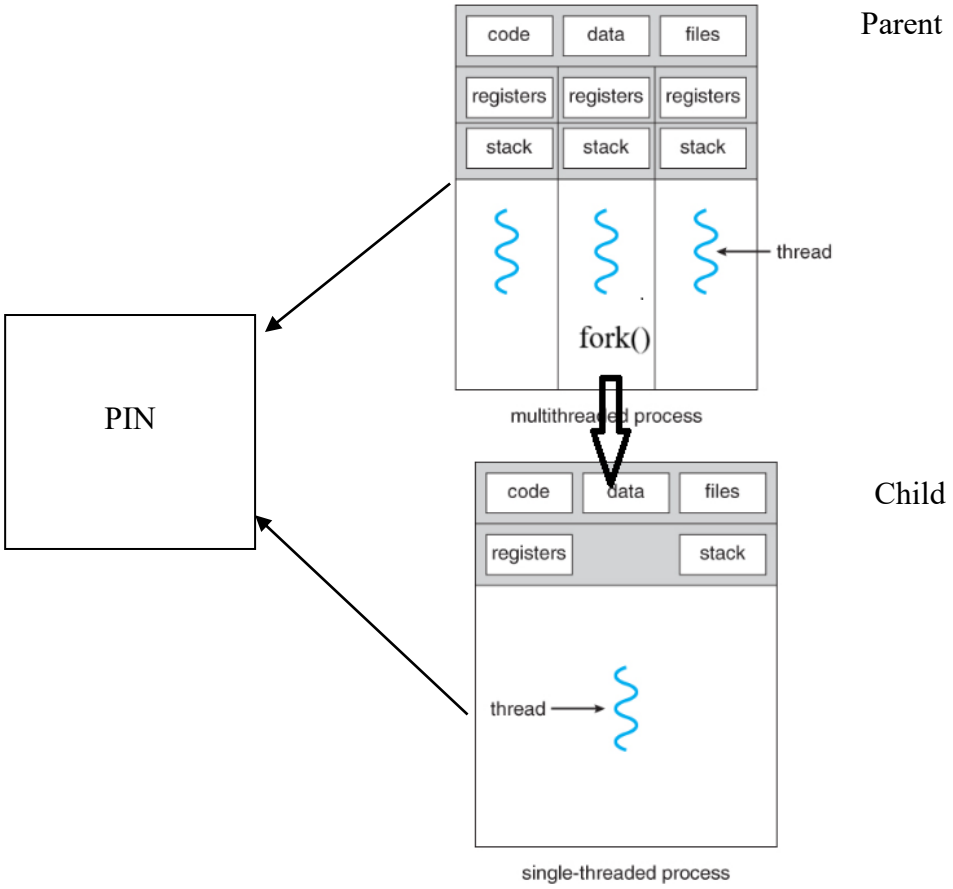

Figure 4.9 – Parent and child process behavior after fork()

Επιπρόσθετα πριν την κλήση fork(), το Pintool δεσμεύει θέση στη shared memory μέσω mmap() ώστε να επιτραπεί η επικοινωνία μεταξύ parent και child διεργασιών. Μετά την εκτέλεση της fork(), το Pin tool διαχειρίζεται κάθε διεργασία, child και parent ξεχωριστά (figure 4.9)και διαχειρίζεται injection window,tracing flags, και query_id, response_id, instruction counter μέσω τοπικών μεταβλητών στη δομή TLS, το Pintool αναγνωρίζει πιο process είναι πιο (PIN_GetPid()) και τα θέτει είτε ως golden execution είτε ως faulty execution ανάλογα με τις ανάγκες του πειράματος. Η χρήση της TLS δομής είναι απαραίτητη λόγω ύπαρξης grpc threads στο parent που δεν κληρονομούνται στο child.

Η συμπεριφορά του εργαλείου παραμένει παρόμοια: για κάθε ερώτημα, το Pin[7] καταγράφει το instruction sequence και την απόκριση ξεχωριστά για κάθε διεργασία. Όπως και στην προηγούμενη προσέγγιση, με το πέρας του injection window, οι μεταβλητές στο TLS επανέρχονται σε default τιμές (inject=false, trace=false, query_id=0, κ.λπ.) ώστε να τερματιστεί η παρακολούθηση και να προετοιμαστεί νέος κύκλος μελέτης, δηλαδή όταν πρόκειτε να εξηπηρετηθεί ένα νέο query. Το child process τερματίζεται αμέσως μετά την ολοκλήρωση της λειτουργίας του, χωρίς να επηρεάζει την ροή του προγράμματος, σαν να μην υπήρξε ποτέ δηλαδή.

Η υλοποίηση αυτής της τεχνικής ενδείκνυεται για περιπτώσεις όπου απαιτείται υψηλή ακρίβεια στην ανάλυση της εκτέλεσης, ενώ ενισχύει τον έλεγχο αποκλίσεων στον instruction-level control flow, γεγονός ιδιαίτερα χρήσιμο στη μελέτη fault propagation λόγω SDCs.

4.5 **Επιλογή σημείων εισαγωγής σφαλμάτων**

Η διαδικασία εισαγωγής σφαλμάτων που περιγράφεται σε αυτή την ενότητα είναι ο βασικός τρόπος που επιλέχθηκε για τη εισαγωγή σφαλμάτων στο Microservice και στοχεύει τόσο στη μελέτη της ανθεκτικότητας της εφαρμογής απέναντι σε τυχαία σφάλματα τύπου SDC όσο και στην μελέτη του control flow.

Για την υλοποίηση αυτής της μελέτης αξιοποιήθηκε η τεχνική process-based redundancy execution, μέσω της δημιουργίας ενός child process με χρήση της κλήσης fork(). Καθώς το child process εκτελεί πρώτο το injection window ως golden run, το pin tool αποκτά πλήρη εικόνα του instruction sequence που περιλαμβάνεται εντός του injection window για κάθε query. Αυτό επιτρέπει την ακριβή καταμέτρηση των υποψήφιων εκτελούμενων εντολών (instruction count), που ενδέχεται να εισαχθούν στην συνέχεια σφάλμα απο το parent, στο κρίσιμο τμήμα του κώδικα που αναλύεται μέσω τοπικού μετρητή instruction_count σε TLS δομή στο child process.

Η πληροφορία αυτή κοινοποιείται στο parent process μέσω μιας κοινόχρηστης περιοχής μνήμης (shared memory region) που δημιουργείται με χρήση της κλήσης συστήματος

mmap(). Η συγκεκριμένη μέθοδος επιτρέπει στον γονικό process να έχει πρόσβαση στο ακριβές πλήθος των instructions που παράχθηκε από το child process.

Με βάση αυτή την πληροφορία, το Pintool εφαρμόζει ένα στοχαστικό μηχανισμό επιλογής σημείων injection, ο οποίος επιλέγει ένα ή περισσότερα τυχαία σημεία στο instruction stream αποθηκεύοντας τα σε λίστα στο TLS του parent process όπου θα εφαρμοστεί το fault. Για παράδειγμα:

π.χ.: Για εισαγωγή $3^{ων}$ faults

```
injection_points = {
  rand() % total_instruction_count,
  rand() % total_instruction_count,
  rand() % total_instruction_count
};
```

Κατά την εκτέλεση του parent process, διατηρείται επίσης ένας τοπικός μετρητής (instruction_counter_fault) στο TLS που αυξάνεται με κάθε υποψήφια για injection εντολή που εκτελείται στο injection window. Όταν ο μετρητής φτάσει σε ένα από τα επιλεγμένα injection points, εφαρμόζεται το fault injection στον καταχωρητή που σχετίζεται με αυτή. Οι μετρητές αυτοί των processes μηδενίζονται κάθε φορά που ένα query βγαίνει εκτός του injection window, έτσι ώστε να υπολογίζονται για κάθε query ξεχωριστά.

Reminder: το σφάλμα μπορεί να είναι, bit flip σε XMM ή GPR καταχωρητή, όπως περιγράφηκε στα προηγούμενα τμήματα.

Με τον τρόπο αυτό, επιτυγχάνεται:

- Στοχαστική κάλυψη του injection space, χωρίς να απαιτείται εξαντλητική ή δομημένη αναπαραγωγή όλων των paths.

- Ανάλυση ανθεκτικότητας της εφαρμογής με ποσοτική μέτρηση: δηλαδή, για κάθε πείραμα μπορούμε να καταγράψουμε:
    - αν το πρόγραμμα συνέχισε κανονικά με σωστή απάντηση,

- αν το αποτέλεσμα ήταν λανθασμένο (silent fault),
- ή αν προκλήθηκε crash (π.χ. segmentation fault).

Η μέθοδος αυτή καθιστά δυνατή τη στατιστική αξιολόγηση της ευαισθησίας της εφαρμογής στα SDC, μετρώντας π.χ. πόσα fault injections απαιτούνται για να οδηγηθούμε σε λανθασμένα αποτελέσματα ή σφάλματα συστήματος.

Τέλος, η ευελιξία του εργαλείου επιτρέπει την εκτέλεση πολλαπλών πειραμάτων με τυχαία εισαγωγή σφαλμάτων σε διαφορετικούς καταχωρητές. Με αυτόν τον τρόπο μπορούν να πραγματοποιηθούν πολλαπλά runs ανά query, με διαφορετικό πλήθος και θέσεις faults κάθε φορά, επιτρέποντας τη συστηματική αξιολόγηση του ποσοστού σφαλμάτων (π.χ. silent faults ή crashes) υπό διαφορετικά επίπεδα επιβάρυνσης, σύμφωνα με την προσέγγιση τύπου *N-fault injection campaign*

Κεφάλαιο 5

HDSearch Microservice

5.1 Εισαγωγή

Η ανθεκτικότητα των σύγχρονων υπολογιστικών εφαρμογών, ιδίως αυτών που βασίζονται σε microservices, εξαρτάται από την ικανότητά τους να ανταποκρίνονται ορθά ακόμη και όταν προκύπτουν σφάλματα σε χαμηλό επίπεδο, όπως οι σιωπηλές αλλοιώσεις δεδομένων. Στην παρούσα εργασία, στόχος είναι η πειραματική μελέτη της επίδρασης τέτοιων σφαλμάτων στην ορθότητα και την απόκριση μιας κατανεμημένης εφαρμογής βασισμένης σε microservices. Πιο συγκεκριμένα, επιδιώκεται η ανάλυση της αλλαγής της ροής ελέγχου (control flow) και της απόκλισης στα αποτελέσματα των αποκρίσεων (responses), συγκρίνοντας την εκτέλεση fault injection με ένα καθαρό golden run.

Η εφαρμογή HDSearch αποτελεί μία ελαφριά, αλλά πλήρως λειτουργική microservice εφαρμογή, η οποία υλοποιείται σε πολυνηματικό περιβάλλον με διακριτά επίπεδα (frontend, mid-tier και backend). Έχει ως input μια σειρά από query ids (id εικόνας) και σαν output το response id για το κάθε query (ομοιότερη εικόνα του query).

Κατά την αρχική φάση ανάλυσης του πειραματικού συστήματος, διαπιστώθηκε ότι το μεγαλύτερο μέρος των υπολογιστικά κρίσιμων εντολών της εφαρμογής HDSearch

εκτελείται στο bucket server, και συγκεκριμένα στη συνάρτηση process_request. Σε αυτό το σημείο, διαφορετικά threads αναλαμβάνουν την παράλληλη επεξεργασία των αιτημάτων (queries) που αποστέλλονται από την ενδιάμεση υπηρεσία (mid-tier service), με στόχο την επιτάχυνση της απόκρισης. Κάθε νήμα εκτελεί τη συνάρτηση KNN_Calculation, η οποία υπολογίζει αποστάσεις μεταξύ του εισερχόμενου ερωτήματος και ενός συνόλου εικόνων από τη βάση δεδομένων, με χρήση της Ευκλείδειας απόστασης. Οι αποστάσεις αυτές χρησιμοποιούνται για την εύρεση των *k*-πλησιέστερων γειτόνων (*k*-Nearest Neighbors), βάσει των οποίων εξάγεται το αποτέλεσμα.

Αναγνωρίζοντας τον κρίσιμο ρόλο του bucket server στη διαδικασία υπολογισμού των τελικών αποτελεσμάτων, η ανάλυση σφαλμάτων επικεντρώθηκε στη συνάρτηση process_request, η οποία θεωρήθηκε ως primary point of intervention για την εισαγωγή σφαλμάτων. Η επιλογή αυτή βασίστηκε στο γεγονός ότι η συγκεκριμένη συνάρτηση αποτελεί τον πυρήνα της λογικής εξυπηρέτησης των αιτημάτων, εμπλέκεται άμεσα στον υπολογισμό αποστάσεων μέσω KNN algorithm και στη δρομολόγηση των αποτελεσμάτων, και επομένως επηρεάζει άμεσα την ορθότητα της τελικής απόκρισης του συστήματος.

5.2 Αλγόριθμος K-Nearest Neighbours

Ο αλγόριθμος K-Nearest Neighbours (KNN) αποτελεί τον υπολογιστικό πυρήνα της υπηρεσίας ανάκτησης ομοιοτήτων της εφαρμογής HDSearch. Η λειτουργία του στοχεύει στον εντοπισμό της πιο όμοιας εγγραφής (response) σε σχέση με ένα εισερχόμενο ερώτημα (query), το οποίο αναπαρίσταται ως διάνυσμα χαρακτηριστικών. Η διαδικασία υλοποιείται με καταμερισμό μεταξύ των επιπέδων της αρχιτεκτονικής της εφαρμογής, με το υπολογιστικά κρίσιμο στάδιο να εκτελείται στον bucket server.

Επιλογή Υποψηφίων Γειτόνων μέσω LSH:
Κατά την παραλαβή ενός ερωτήματος, το mid-tier service εφαρμόζει τεχνικές *Locality-Sensitive Hashing* (LSH), προκειμένου να περιορίσει τον αριθμό των εγγραφών που θα εξεταστούν. Η LSH επιλέγει ένα υποσύνολο πιθανών γειτόνων από τη βάση δεδομένων και το προωθεί στον bucket server για υπολογισμό αποστάσεων.

Υπολογισμός Αποστάσεων και Τελική Επιλογή:

Ο bucket server υλοποιεί το βασικό βήμα του KNN αλγορίθμου, συγκρίνοντας το διάνυσμα του ερωτήματος με τα διανύσματα των υποψήφιων γειτόνων μέσω υπολογισμού Ευκλείδειας απόστασης. Ο αντίστοιχος ψευδοκώδικας παρουσιάζεται παρακάτω:

```
Είσοδος:
    query_vector ← διάνυσμα χαρακτηριστικών του query
    candidate_vectors ← λίστα διανυσμάτων υποψήφιων γειτόνων

Βήματα:
    distances ← κενή λίστα

    Για κάθε vector στη candidate_vectors:
        distance ← EuklideanDistance_MKL(query_vector, candidate_vectors)
        Προσθήκη distance στη λίστα distances

    min_index ← min_element(distances)
    response_id ← αναγνωριστικό του υποψηφίου στη θέση min_index

Έξοδος:
    Επιστροφή του response_id
```

Ο υπολογισμός των αποστάσεων πραγματοποιείται με χρήση της βιβλιοθήκης Intel Math Kernel Library (MKL), η οποία παρέχει βελτιστοποιημένες ρουτίνες για αριθμητικές πράξεις σε SIMD καταχωρητές XMM. Μετά την εξαγωγή όλων των αποστάσεων, η συνάρτηση min_element() εντοπίζει τη μικρότερη απόσταση και επιλέγει τον αντίστοιχο δείκτη ως πλησιέστερο γείτονα. Το response id που σχετίζεται με αυτόν επιστρέφεται ως η τελική απάντηση για το ερώτημα.

5.3 Προσαρμογή της HDSearch για Υποστήριξη Fault Injection

Για τη μελέτη της ευπάθειας της εφαρμογής HDSearch σε σιωπηλές αλλοιώσεις δεδομένων (SDCs), κρίθηκε απαραίτητη η προσαρμογή του bucket server, ο οποίος υλοποιεί τη βασική λογική υπολογισμού αποτελεσμάτων μέσω της συνάρτησης process_request. Προκειμένου να καταστεί δυνατή η στοχευμένη παρακολούθηση και η εισαγωγή σφαλμάτων, ενσωματώθηκαν οι συναρτήσεις δημιουργείας του "injection window", δηλαδή της περιοχής του κώδικα μέσα στην οποία το Pintool θα είναι ενεργό.

Για τη σήμανση του injection window, εντάχθηκαν στο λογισμικό οι δύο ειδικές συναρτήσεις: FaultInjectionBegin() και FaultInjectionEnd() (figure 5,1). Οι συναρτήσεις αυτές τοποθετήθηκαν σε διάφορα σημεία της συνάρτησης process request αλλά σε όλα τα πειράματα είχαν ανάμεσα τους την knnCalculations οπου γίνονται και οι περισσότεροι υπολογισμοί για την εύρεση απόκρισης. Το PinTool μπορεί έτσι να εντοπίζει τα όρια του injection window μέσω routine instrumentation και να ενεργοποιείται δυναμικά μόνο κατά την εκτέλεση των εντολών εντός αυτού του πλαισίου.

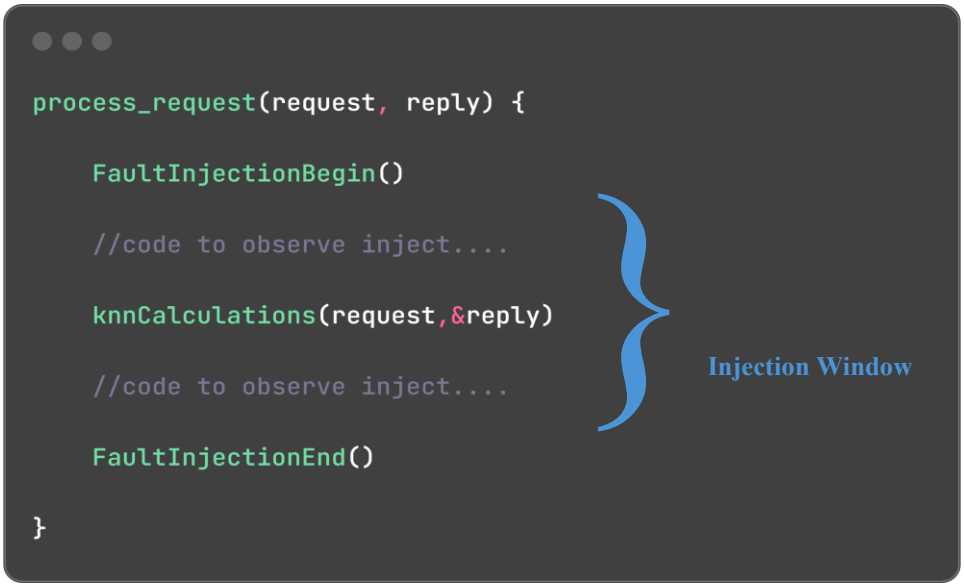

Figure 5.1 – Injection window in process request function

Επιπλέον, ενσωματώθηκε η συνάρτηση QueryBegins(int id), η οποία καλείται κάθε φορά που ένα thread ξεκινά την εξυπηρέτηση ενός νέου query και η QueryResult(int id), η οποία καλείτε στο τέλος τις εξυπηρέτησεις του query από την process request. Με την παρεμβολή (interposition) των κλήσεων αυτών, το Pintool έχει τη δυνατότητα να συνδέσει το κάθε basic block του low level εκτελούμενου κώδικα στο window με το αντίστοιχο query id – response id . Αυτό επιτρέπει την ανάλυση σε επίπεδο μεμονωμένων queries, τόσο ως προς τη ροή εντολών (instruction sequence), όσο και ως προς τα responses που παράγονται, διευκολύνοντας σημαντικά τη σύγκριση μεταξύ golden run και fault-injection run.

Συμπερασματικά, για τη σωστή λειτουργία αυτής της τεχνικής, συνιστάται η μεταγλώττιση της εφαρμογής με τις επιλογές -g -O0 στο Makefile. Η σημαία -g προσθέτει σύμβολα εντοπισμού σφαλμάτων (debug symbols), επιτρέποντας στο εργαλείο να αναγνωρίζει σωστά τις ονομασίες συναρτήσεων και να εντοπίζει τα σημεία εισαγωγής (routines) με ακρίβεια. Παράλληλα, η επιλογή -O0 απενεργοποιεί οποιαδήποτε βελτιστοποίηση κατά τη μεταγλώττιση, εξασφαλίζοντας ότι ο κώδικας παραμένει όσο το δυνατόν πιο προβλέψιμος και κοντά στην αρχική του δομή. Αυτή η προσέγγιση διευκολύνει σημαντικά τη διαδικασία instrumentation του Pin Tool, καθώς αποφεύγονται αλλαγές στη ροή εντολών ή συγχωνεύσεις συναρτήσεων που προκύπτουν από επιθετικές βελτιστοποιήσεις σε υψηλότερα επίπεδα.

5.4 Αξιολόγηση υπολογιστικών μονάδων που χρησιμοποιά το HDSearch

Όπως υποδεικνύει και η μελέτη της Alibaba[1], εντολές που περιλαμβάνουν υπολογισμούς κινητής υποδιαστολής, δηλαδή πράξεις στο FPU, είναι ιδιαίτερα επιρρεπείς σε SDCs, με τα σφάλματα να εμφανίζονται συχνότερα στο fraction part των αριθμών, προκαλώντας μικρές απώλειες ακρίβειας που συχνά περνούν απαρατήρητες από τα συνήθη συστήματα εντοπισμού σφαλμάτων. Παρόμοια ευπάθεια παρατηρήθηκε και στις SIMD μονάδες όπου μία εντολή εφαρμόζεται ταυτόχρονα σε πολλαπλά δεδομένα.

Η παραπάνω ταξινόμηση των SDC περιπτώσεων ανέδειξε την ανάγκη να διερευνηθεί κατά πόσο η εφαρμογή HDSearch του μSuite αξιοποιεί υπολογιστικές μονάδες που

είναι ευάλωτες σε τέτοια σφάλματα. Με τη βοήθεια του Intel Pintool, το οποίο διαθέτει δυνατότητες κατηγοριοποίησης εντολών βάσει αρχιτεκτονικής (μέσω INS_Category), καταγράφηκαν οι τύποι εντολών που χρησιμοποιούνται κατά την εκτέλεση της εφαρμογής.

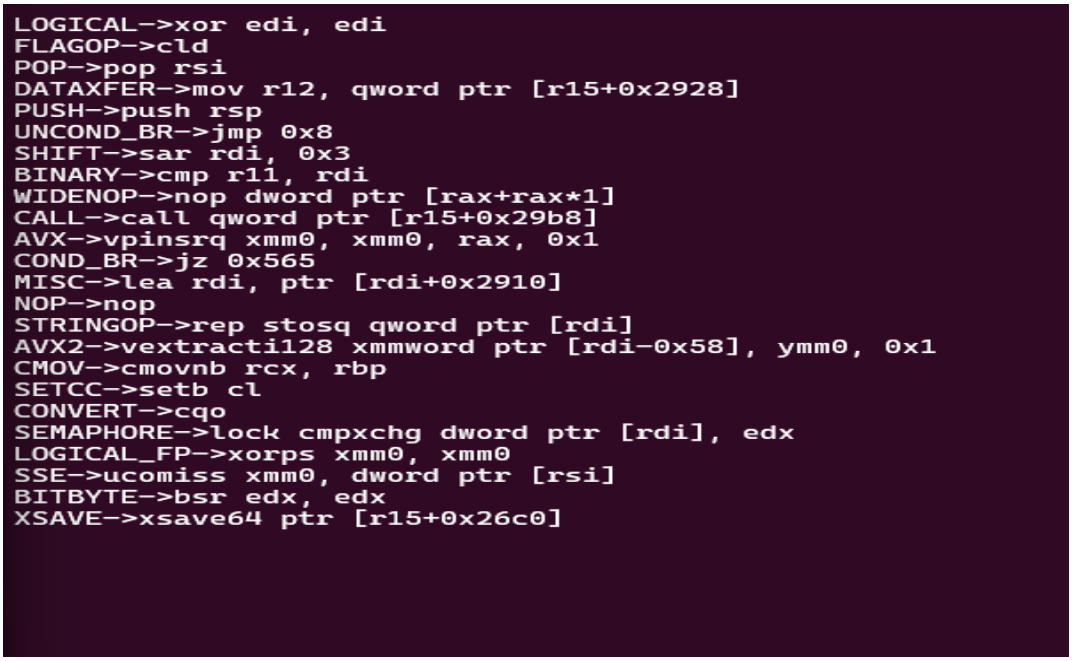

Figure 5.2 – Classification of instructions executed in the HDSearch bucket component

Η ανάλυση (fugure 5.2) αποκάλυψε ότι, εκτός από βασικές ALU εντολές, η εφαρμογή κάνει ευρεία χρήση και των εξής μονάδων:

1. **SSE Unit**
    - Κατηγορία εντολών Pin: SSE
    - Μονάδα CPU: SSE Floating Point Unit
    - Καταχωρητές: 128-bit XMM
    - Υποστηρίζει: Single και Double Precision Floating Point πράξεις
2. **AVX / AVX2 SIMD Floating Point Unit**
    - Κατηγορία εντολών Pin: AVX, AVX2

- Καταχωρητές: XMM, YMM, ZMM
- Υποστηρίζει: Παράλληλους υπολογισμούς σε πολλαπλά δεδομένα FP (SIMD)

3. **FPU Unit**
   - Κατηγορία εντολών Pin: LOGICAL_FP
   - Καταχωρητές: XMM
   - Υλοποιεί: Πράξεις κινητής υποδιαστολής, κυρίως σε Single Precision, scalar , μέσω της FPU

Η ποικιλία και η πολυπλοκότητα των κατηγοριών αυτών καθιστά απαραίτητο τον λεπτομερή έλεγχο των instruction operands, ώστε να διαχωρίζονται οι πραγματικά επικίνδυνες floating point πράξεις (όπως ADDSS, SUBSD, MULPS, VADDPS, VSQRTPD) από εκείνες που σχετίζονται με flags, συγκρίσεις ή αναδιάταξη δεδομένων (π.χ. ucomiss, vextracti128). Η διάκριση αυτή είναι κρίσιμη για τον εντοπισμό των σημείων εισαγωγής SDC που μπορούν να επηρεάσουν υπολογιστικά κρίσιμα δεδομένα, χωρίς να προκαλούν άμεση αποτυχία στην εκτέλεση του προγράμματος.

Καθώς η συνάρτηση KNN_calculation στον bucket server αποτελεί τον κύριο υπολογιστικό πυρήνα της εφαρμογής, παρατηρήθηκε ότι σε αυτό το σημείο λαμβάνει χώρα έντονη επεξεργασία δεδομένων, μέσω πλήθους αριθμητικών πράξεων κινητής υποδιαστολής σε μονάδες SIMD και FPU. Η υψηλή συγκέντρωση υπολογισμών καθιστά τη συγκεκριμένη περιοχή ιδιαίτερα επιρρεπή σε θερμικά φαινόμενα, καθώς ο επεξεργαστής μπορεί να οδηγηθεί σε τοπική υπερθέρμανση λόγω αυξημένου ενεργειακού φορτίου.

Η συσχέτιση υψηλής υπολογιστικής έντασης και αυξημένης θερμικής καταπόνησης ενισχύει την πιθανότητα εμφάνισης SDCs. Για τον λόγο αυτό, το injection σε αυτό το σημείο θεωρείται ιδιαίτερα ρεαλιστικό, καθώς προσομοιώνει καταστάσεις στις οποίες ένας επεξεργαστής μπορεί —λόγω stress ή θερμικής αστάθειας— να παράγει εσφαλμένα αποτελέσματα χωρίς να προκληθεί άμεσο crash.

Συμπερασματικά, διαπιστώθηκε ότι η συντριπτική πλειονότητα των πράξεων κινητής υποδιαστολής στην εφαρμογή HDSearch εκτελείται μέσω των XMM καταχωρητών, ανεξαρτήτως αν πρόκειται για scalar ή SSE εντολές. Η παρατήρηση αυτή οδήγησε στη στοχευμένη επιλογή των XMM registers ως κύριο στόχο fault injection, ώστε να επιτυγχάνεται ρεαλιστική προσομοίωση SDCs σε κρίσιμα υπολογιστικά μονοπάτια της εφαρμογής.

5.5 Μελέτη control flow bucket server

Για τη μελέτη της ροής εκτέλεσης κατά τα αρχικά στάδια της παρούσας εργασίας, χρησιμοποιήθηκε ένα cfg-tool, το οποίο παράγει Control Flow Graphs βασισμένους στη δυναμική ανάλυση των εντολών εκτέλεσης του προγράμματος.

Συγκεκριμένα, το εργαλείο καταγράφει το πλήρες instruction sequence κάθε εκτέλεσης και αναγνωρίζει πρότυπα αλλαγής ροής ελέγχου όπως jumps, calls, returns και syscalls, απομονώνοντας τα αντίστοιχα basic blocks κώδικα.

Για κάθε ανεξάρτητο ερώτημα (query) που επεξεργάζεται η συνάρτηση process_request, δημιουργείται ξεχωριστό αρχείο, το οποίο περιλαμβάνει τον αντίστοιχο CFG. Με αυτόν τον τρόπο, διευκολύνεται η αναλυτική χαρτογράφηση της ροής εκτέλεσης σε επίπεδο εντολών για κάθε περίπτωση, επιτρέποντας τον εντοπισμό κρίσιμων μοτίβων υπολογιστικής συμπεριφοράς.

5.6 Διαχείρηση μη ντετερμινιστικού instruction sequence

Αφού δημιουργήθηκε ντετερμινιστικό sequence των input στο Load generator με ψευδοτυχαίο τροπο επιλογής των query ids, συγκρίνοντας δύο ανεξάρτητα runs, χωρίς εισαγωγή σφαλμάτων, παρατηρήθηκαν αλλαγές στη σειρά εκτέλεσης των εντολών (instruction sequence), γεγονός που υποδηλώνει ότι η εφαρμογή, παρουσιάζει μη ντετερμινιστική εκτέλεση. Αυτό το φαινόμενο ενδέχεται να οδηγήσει σε ψευδώς θετικά αποτελέσματα κατά τη σύγκριση control flow μεταξύ golden και injected runs, αφού οι αποκλίσεις ενδέχεται να αποδίδονται σε εγγενή μη ντετερμινιστική συμπεριφορά του συστήματος και όχι σε σφάλμα.

Αυτό οδήγησε σε μια νέα κατεύθυνση της μελέτης: την ταυτοποίηση των εντολών που είναι μη ντετερμινιστικές και την προσπάθεια εξάλειψης ή ελέγχου της μεταβλητότητας στο instruction sequence.

Αρχικά, εξετάστηκε η δυνατότητα δημιουργίας ενός static binary για τον bucket server, με στόχο τον περιορισμό της μη ντετερμινιστικής συμπεριφοράς που προκαλείται από δυναμικά φορτωμένες βιβλιοθήκες συστήματος. Προς αυτή την κατεύθυνση, τροποποιήθηκε η διαδικασία μεταγλώττισης με την ενεργοποίηση της σημαίας -no-pie (Position Independent Executable), ώστε το εκτελέσιμο να έχει σταθερές διευθύνσεις μνήμης. Παράλληλα, απενεργοποιήθηκε η τεχνική ASLR (Address Space Layout Randomization) του λειτουργικού συστήματος, η οποία προκαλεί τυχαία κατανομή διευθύνσεων στη μνήμη.

Επιπρόσθετα, διερευνήθηκε το ενδεχόμενο η μη ντετερμινιστικότητα να οφείλεται στο concurrency και στο φαινόμενο του thread interleaving. Για τον λόγο αυτό, αρχικά πραγματοποιήθηκε δραστική μείωση του ρυθμού εισερχόμενων queries (queries per second), ώστε να περιοριστεί η πιθανότητα παράλληλης εκτέλεσης πολλαπλών threads που θα μπορούσαν να οδηγήσουν σε παράλληλες εγγραφές των ιχνών εκτέλεσης. Ωστόσο, η μη ντετερμινιστική συμπεριφορά συνέχισε να παρατηρείται, υποδεικνύοντας ότι η πηγή του προβλήματος δεν ήταν αποκλειστικά το concurrency. Στη συνέχεια, εφαρμόστηκε περαιτέρω μεθοδολογία, με την πλήρη απομόνωση των καταγραφών, όπου κάθε thread κατέγραφε το instruction trace σε ξεχωριστό αρχείο ανά query. Παρά και αυτήν την πλήρη απομόνωση των threads, το πρόβλημα της μεταβλητότητας στο instruction sequence παρέμεινε.

Παρά τις παραπάνω παρεμβάσεις, διαπιστώθηκε ότι ορισμένα τμήματα των βιβλιοθηκών του λειτουργικού συστήματος (π.χ. libc) συνέχιζαν να φορτώνονται δυναμικά και εμφάνιζαν μη προβλέψιμη συμπεριφορά σε κάθε εκτέλεση. Οι βιβλιοθήκες αυτές περιλαμβάνουν βασικές συναρτήσεις για τη διαχείριση μνήμης, εισόδου/εξόδου και χρονισμού, και συχνά περιέχουν εσωτερικές βελτιστοποιήσεις που βασίζονται σε cache ή multithreading. Ως αποτέλεσμα, η ακολουθία των εντολών που παρατηρείται κατά την εκτέλεση δεν είναι απόλυτα επαναλήψιμη, γεγονός που

δυσχεραίνει την ανάλυση των διαφορών μεταξύ των εκτελέσεων με και χωρίς fault injection. Έτσι, μετά από εκτεταμένη μελέτη, παρατηρήθηκε ότι η heap memory παρουσίαζε μη ντετερμινιστική συμπεριφορά, γεγονός που οδηγούσε σε διαφοροποιήσεις στη συμπεριφορά βασικών ρουτινών του συστήματος, όπως οι malloc, nss lookup και άλλες συστημικές συναρτήσεις. Για τον λόγο αυτό, υιοθετήθηκε η προσέγγιση double redundancy execution εντός της συνάρτησης process_request, ως ένα πιο αποτελεσματικό και επεκτάσιμο μέσο αντιμετώπισης της μη ντετερμινιστικότητας. Η τεχνική αυτή επιτρέπει τόσο τη δημιουργία σταθερής ροής εκτέλεσης ανά query, όσο και τη σύγκριση control flow και παραγόμενης απόκρισης (reply) εντός του ίδιου περάσματος του PinTool, καθιστώντας δυνατή την αξιόπιστη μελέτη των επιπτώσεων των SDCs.

5.7 Προσέγγιση χρήσης fork για εξασφάλιση Ντετερμινιστικότητας

Η ανάγκη για αξιόπιστη μελέτη του control flow και των πιθανών αποκλίσεων στην απόκριση των microservices, σε συνδυασμό με την επιδίωξη ενός deterministic περιβάλλοντος εκτέλεσης και την αξιολόγηση της ανθεκτικότητας του μSuite έναντι σιωπηλών αλλοιώσεων δεδομένων (SDCs), οδήγησε στην εφαρμογή της προσέγγισης double redundancy execution. Η προσέγγιση αυτή εφαρμόζεται στο πλαίσιο της συνάρτησης process_request του bucket server. Μέσω της σύγκρισης μεταξύ εκτελέσεων με και χωρίς fault injection, επιτυγχάνεται η συστηματική παρακολούθηση των αποκρίσεων (replies) και η ανάλυση διαφορών στον έλεγχο ροής, ενισχύοντας τη δυνατότητα εντοπισμού σφαλμάτων. Επιπλέον, η τεχνική αυτή μπορεί να αποτελέσει βάση για μελλοντική εφαρμογή σε πραγματικά περιβάλλοντα ως μία αμιγώς λογισμική στρατηγική ανίχνευσης ή απομόνωσης SDCs. Οι δύο τεχνικές που εφαρμόστηκαν ήταν:

Multithreading Redundancy Execution:

Η προσέγγιση αυτή δεν εξαλείφει τη μη ντετερμινιστική ροή εκτέλεσης, καθώς τα threads μοιράζονται κοινό χώρο διευθύνσεων και οι system routines (π.χ. malloc, I/O) εξακολουθούν να επηρεάζονται από τον προγραμματισμό του λειτουργικού. Η παρατήρηση αυτή οδήγησε στην ανάγκη υιοθέτησης μιας διαφορετικής στρατηγικής,

ικανής να διασφαλίσει ντετερμινιστικό instruction sequence ανεξαρτήτως εξωτερικών παραγόντων.

Process Redundancy Execution με fork():

Σε αντίθεση με τα threads, η χρήση της κλήσης fork() επιτρέπει τη δημιουργία child process με πλήρως ανεξάρτητο χώρο διευθύνσεων, καταχωρητές και περιβάλλον εκτέλεσης, καθιστώντας την εκτέλεσή του επαναλήψιμη και προβλέψιμη, χωρίς παρεμβολές από κοινόχρηστη κατάσταση του συστήματος. Επιπλέον, ο συνδυασμός αυτής της τεχνικής με την απενεργοποίηση του μηχανισμού ASLR (Address Space Layout Randomization) και τη μεταγλώττιση της εφαρμογής σε στατικά συνδεδεμένο binary συνέβαλε αποφασιστικά στην επίτευξη πλήρους ντετερμινιστικότητας, εξασφαλίζοντας ταυτόσημο instruction sequence μεταξύ επαναλαμβανόμενων εκτελέσεων του ίδιου ερωτήματος (query), όταν δεν υφίσταται fault injection.

Με αυτό τον τρόπο, επιτυγχάνεται πλήρης απομόνωση και είναι δυνατή η ακριβής σύγκριση τόσο του αποτελέσματος, όσο και του control flow (instruction sequence) μεταξύ των δύο εκτελέσεων. Η τεχνική αυτή εξαλείφει τις μη ντετερμινιστικές επιρροές από system routines και βιβλιοθήκες (libc, malloc, I/O), αλλά απαιτεί περιορισμό σε μονονηματική εκτέλεση, καθώς η fork() δεν είναι ασφαλής σε πολυνηματικό περιβάλλον, μιας και αντιγράφει σε τέτοιες περιπτώσεις το thread που την καλεί αγνοώντας τα υπόλοιπα threads του parent process.

Καθώς το κύριο μέλημα της παρούσας μελέτης είναι η διερεύνηση της συμπεριφοράς της εφαρμογής HDSearch υπό την παρουσία σιωπηλών σφαλμάτων (silent failures), κρίθηκε αποδεκτό να χρησιμοποιηθεί η τεχνική αυτή με την προυπόθεση ότι κάθε query εξυπηρετείται από ένα μόνο νήμα. Αν και διαπιστώθηκε η ταυτόχρονη ύπαρξη νημάτων, οπου υπήρχαν νήματα που διαχειρίζονται επικοινωνία μέσω GRPC, η μελέτη επικεντρώθηκε σε κρίσιμους υπολογιστικούς κόμβους, στους οποίους το ενεργό νήμα που καλεί τη fork() αντιστοιχεί στην κύρια λογική του query.

Συμπερασματικά σε πραγματικά συστήματα παραγωγής, η εκτέλεση των δύο διεργασιών πρέπει να πραγματοποιείται σε διαφορετικά CPUs, προκειμένου να

εξασφαλιστεί ότι πιθανό σφάλμα σε έναν επεξεργαστή δεν θα επηρεάσει και τις δύο εκτελέσεις. Μια τέτοια απαίτηση, αν και απαραίτητη για αξιοπιστία, εισάγει σημαντικό υπολογιστικό overhead, γεγονός που περιορίζει την πρακτική εφαρμογή της προσέγγισης σε μεγάλης κλίμακας, performance-sensitive περιβάλλοντα.

Κεφάλαιο 6

Πειραματική Μεθοδολογία

6.1 Υποδομή Πειραματικής Μελέτης – Το CloudLab

Για τη μελέτη του αντίκτυπου σφαλμάτων Silent Data Corruption (SDC) σε συστήματα που βασίζονται σε αρχιτεκτονική microservices, απαιτείται ένα υπολογιστικό περιβάλλον μεγάλων κέντρων δεδομένων που να προσφέρει πλήρη έλεγχο στον ερευνητή, πραγματικούς φυσικούς πόρους και ευελιξία στην παραμετροποίηση του συστήματος. Για τον λόγο αυτό, στην παρούσα εργασία χρησιμοποιήθηκε το CloudLab[12].

Το CloudLab[12] είναι μια ακαδημαϊκή πλατφόρμα πειραματισμού για συστήματα υποδομής και κατανεμημένες εφαρμογές. Προσφέρει στους ερευνητές πρόσβαση σε πλήρως ελεγχόμενα nodes με φυσική απομόνωση, δίνοντας τη δυνατότητα ανάπτυξης, ρύθμισης και μέτρησης σε πραγματικό υλικό, σε αντίθεση με τους περιορισμούς ενός εικονικού περιβάλλοντος.

6.2 Βήματα Προετοιμασίας Πειραματικού Περιβάλλοντος

Για την υλοποίηση του πειραματικού περιβάλλοντος και την εκτέλεση της εφαρμογής HDSearch του μSuite, πραγματοποιήθηκε μια σειρά από βήματα σε υπολογιστικούς κόμβους του CloudLab[12]. Η διαδικασία περιλαμβάνει την προετοιμασία του λειτουργικού συστήματος, την εγκατάσταση εργαλείων, τη ρύθμιση του Docker και τη λειτουργία του συστήματος μέσω Docker Compose[13]. Ακολουθεί αναλυτικά η διαδικασία:

Βήμα 1: Επιλογή και εκκίνηση κόμβου στο CloudLab

- Επιλέχθηκε ένα φυσικό μηχάνημα (bare-metal) αρχιτεκτονικής x86_64 AMD(c6525-100g, c6525-25g), με default image.
- Δημιουργήθηκε experiment με 1 node.
- Παραχωρήθηκαν 200 GB προσωρινού αποθηκευτικού χώρου στο mount point /mydata για το docker container.

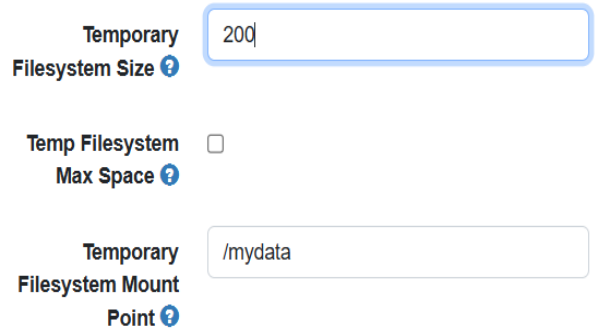

Figure 6.1 – Allocation of temporary storage for a Docker container in CloudLab.

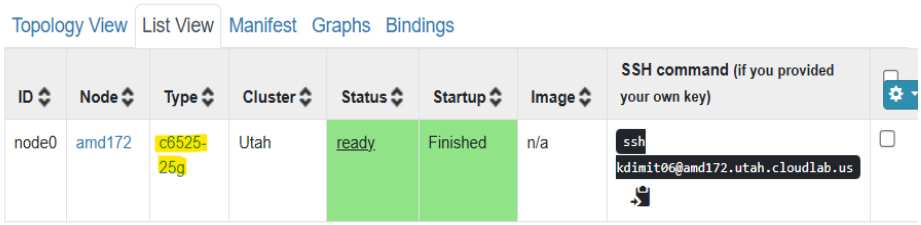

Figure 6.2 – Node type selection interface

Η πρόσβαση σε αυτό το περιβάλλον ήταν απαραίτητη για να πραγματοποιηθεί το set-up της αρχιτεκτονικής των microservices σε επίπεδο Docker Compose[13], και στη συνέχεια να μελετηθεί το πώς τα injected faults επηρεάζουν τα microservices υπό ρεαλιστικό φόρτο εργασίας.

Βήμα 2: Εγκατάσταση Intel Pin Tool [8]

Εγκαταστάθηκε το Intel Pin Tool, ένα δυναμικό εργαλείο ανάλυσης δυαδικού κώδικα (dynamic binary instrumentation tool), το οποίο χρησημοποιείθηκε στη συνέχεια για την εισαγωγή τεχνητών σφαλμάτων (fault injection) σε κρίσιμες εντολές της εφαρμογής.

Βήμα 3: Ανάπτυξη του HDSearch με Docker Compose

Η εφαρμογή HDSearch εγκαταστάθηκε στον φυσικό node του CloudLab. Όλα τα επιμέρους services εκτελούνται και επικοινωνούν μεταξύ τους μέσα από Docker instances. Για τους σκοπούς της μελέτης χρησιμοποιήθηκε ένα ενημερωμένο repository του πανεπηστημιίου[10]. Πραγματοποιήθηκαν επιπλέον τροποποιήσεις στο αρχικό setup ώστε το σύστημα να λειτουργεί σύμφωνα με τις απαιτήσεις της παρούσας μελέτης.

Βήμα 4: Δημιουργία custom PinTool

Στη συνέχεια δημιουργήθηκε custom PinTool με στόχο την προσομοίωση Silent Data Corruption σε κρίσιμα σημεία εκτέλεσης (π.χ. floating point εντολές) βασισμένο στην λειτουργια των microservices (multithreading περιβάλλον, με υποστήριξη fork()).

Όλα τα παραπάνω βήματα έχουν υλοποιηθεί και ενσωματώθηκαν (εκτος Βημα 1) σε public repository στο GitHub, το οποίο συνοδεύεται από τα απαραίτητα αρχεία και κώδικες για την αναπαραγωγή του πειράματος. Στο repository περιλαμβάνονται:

- Φάκελος με το εγκατεστημένο Pin Tool (MicroPinfi)
- Ο πηγαίος κώδικας (.cpp) και binaries (.so) των custom PinTools
- Οδηγίες εγκατάστασης,ρύθμισης και εκτέλεσης της εφαρμογής HDSearch και pintool στο docker compose instance.

Το repository είναι διαθέσιμο στο κεφάλαιο με τις βιβλιογραφίες[14]

6.3 **Injection Scenarios**

Στα παρακάτω σενάρια μελετάται η επίδραση των fault injections στον υπολογιστικά βαρυφορτωμένο αλγόριθμο KNN του bucket server.

Κατά τις αρχικές φάσεις των πειραμάτων διαπιστώθηκε ότι πολλές από τις βιβλιοθήκες χρήστη του λειτουργικού συστήματος, έχουν ισχυρή τάση να αξιοποιούν καταχωρητές γενικής χρήσης (GPRs) ως δείκτες στη μνήμη (memory pointers). Η παρατήρηση αυτή αποκτά ιδιαίτερη σημασία, καθώς ένας bit flip σε τέτοιου είδους καταχωρητή ενδέχεται να μεταβάλει τη διεύθυνση που δείχνει ο pointer, οδηγώντας το πρόγραμμα να προσπελάσει περιοχή μνήμης εκτός του επιτρεπόμενου address space. Το αποτέλεσμα ενός τέτοιου σφάλματος έχει μεγάλη πιθανότητα να είναι crash του προγράμματος καθώς το λειτουργικό σύστημα ανιχνεύει παράνομη πρόσβαση στη μνήμη.

Για τις ανάγκες της πειραματικής αξιολόγησης, οι εντολές που αξιοποιούν καταχωρητές γενικής χρήσης (GPRs) χωρίστηκαν σε δύο διακριτές κατηγορίες ανάλογα με το εκτελεστικό τους πλαίσιο: (α) GPR-based instructions εντός των system routines. και (β) GPR-based instructions εντός του application code. Ο διαχωρισμός αυτός κρίθηκε απαραίτητος, καθώς τα σφάλματα που εισάγονται σε καταχωρητές κατά την εκτέλεση system routines παρουσιάζουν αυξημένη πιθανότητα να οδηγήσουν σε μη έγκυρη πρόσβαση στη μνήμη (invalid memory access) και κατά συνέπεια σε segmentation fault, λόγω της συχνής χρήσης των GPRs ως pointers.

Επομένως, στο πλαίσιο της παρούσας πειραματικής αξιολόγησης, διαμορφώθηκαν τρεις κύριες κατηγορίες fault injection scenarios.

. Για κάθε σενάριο, εξετάζονται συστηματικά τα εξής χαρακτηριστικά:

- Ποσοστό εμφάνισης segmentation fault,
- Αξιοπιστία της απόκρισης (faulty reply),
- Αποκλίσεις στο control flow (instruction sequence).

Scenario 1: Bit Flip σε GPR καταχωρητές εντός system library routines

Πρόκειται για σφάλματα που εισάγονται σε καταχωρητές γενικής χρήσης (rax, rdi, κ.λπ.) κατά την εκτέλεση system library routines π.χ malloc . Οι καταχωρητές σε αυτό το context συχνά χρησιμοποιούνται ως δείκτες (pointers) προς περιοχές μνήμης. Ένα bit flip σε τέτοιο pointer μπορεί να μετατοπίσει τη διεύθυνση σε μη έγκυρο memory region, προκαλώντας segmentation fault (crash). Σε αυτό το σενάριο αναμένεται υψηλή πιθανότητα αποτυχίας.

Scenario 2: Bit Flip σε GPR καταχωρητές εντός Application Code

Το συγκεκριμένο σενάριο χρησιμοποιείται για να μελετηθεί πώς επηρεάζουν τα bit flips σε καταχωρητές γενικής χρήσης (GPRs) όταν αυτά εμφανίζονται σε εντολές του application-level κώδικα. Οι GPRs σε αυτό το πλαίσιο αξιοποιούνται κυρίως για την εκτέλεση αριθμητικών πράξεων, τον χειρισμό τοπικών μεταβλητών, αλλά και τη μεταφορά παραμέτρων μέσω του stack προς άλλες συναρτήσεις. Μέσω αυτής της προσέγγισης, επιδιώκεται να εξεταστεί κατά πόσο σφάλματα που ενσωματώνονται στο application code μπορούν να επηρεάσουν τη συνολική συμπεριφορά της εφαρμογής, να μεταβάλουν την ορθότητα των υπολογισμών ή να οδηγήσουν σε ανεπιθύμητες παρενέργειες ή ακομα και segmentation faults κατά την εκτέλεση.

Scenario 3: Bit Flip σε XMM καταχωρητές (Floating Point / SIMD)

Το τρίτο σενάριο στοχεύει αποκλειστικά σε αριθμητικές πράξεις κινητής υποδιαστολής (floating point operations), οι οποίες υλοποιούνται μέσω των XMM καταχωρητών σε FPU και SIMD μονάδες.

Το σενάριο αυτό έχει σχεδιαστεί για να μελετήσει την αλγοριθμική ανθεκτικότητα του KNN σε SDC σφάλματα: συγκεκριμένα, κατά πόσο μια μικρή αριθμητική παραμόρφωση στην τιμή απόστασης μπορεί να επηρεάσει την επιλογή των γειτονικών αποτελεσμάτων, να προκαλέσει αλλοίωση στην απόκριση (response error) ή, σε ακραίες περιπτώσεις, να συμβάλει σε αποτυχία εκτέλεσης (π.χ. segmentation fault)

Κεφάλαιο 7

Αποτελέσματα

7.1 Εισαγωγή

Αρχικά, πριν προχωρήσουμε στην ανάλυση των αποκλίσεων που παρατηρήθηκαν, κρίνεται σκόπιμο να παρουσιαστεί συνοπτικά η βασική λειτουργία του αλγορίθμου K-Nearest Neighbors (KNN), όπως εφαρμόζεται στο σύστημα. Στην παρούσα υλοποίηση, το mid-tier service επιλέγει ένα σύνολο υποψηφίων γειτόνων και το στέλνει στο bucket server. Στη συνέχεια, για κάθε query, οι αποστάσεις μεταξύ του σημείου-στόχου και των υποψηφίων υπολογίζονται μέσω της βιβλιοθήκης Intel MKL. Τέλος, η συνάρτηση min_element χρησιμοποιείται για την ανάκτηση του κοντινότερου γείτονα από τη λίστα των αποστάσεων των υποψηφίων, ολοκληρώνοντας τη διαδικασία απόφασης.

Στο παρόν κεφάλαιο παρουσιάζονται τα αποτελέσματα της πειραματικής αξιολόγησης των σεναρίων fault injection που εφαρμόστηκαν στην εφαρμογή HDSearch. Συγκεκριμένα, εξετάζονται τα τρία σενάρια που αναφέραμε πιο πρίν

1) Σφάλματα που εισάγονται σε καταχωρητές γενικής χρήσης (rax, rdi, κ.λπ.) κατά την εκτέλεση system library routines π.χ malloc . Οι καταχωρητές σε αυτό το context συχνά χρησιμοποιούνται ως δείκτες (pointers) προς περιοχές μνήμης. Ένα bit flip σε τέτοιο pointer μπορεί να μετατοπίσει τη διεύθυνση σε μη έγκυρο memory region, προκαλώντας segmentation fault (crash). Σε αυτό το σενάριο αναμένεται υψηλή πιθανότητα αποτυχίας..

2) Σφάλματα που εισάγονται σε καταχωρητές γενικής χρήσης(GPRs) όταν αυτά εμφανίζονται σε εντολές του application-level κώδικα. Μέσω αυτής της

προσέγγισης, επιδιώκεται να εξεταστεί κατά πόσο σφάλματα που ενσωματώνονται στο application code μπορούν να επηρεάσουν τη συνολική συμπεριφορά της εφαρμογής, να μεταβάλουν την ορθότητα των υπολογισμών ή να οδηγήσουν σε ανεπιθύμητες παρενέργειες ή ακομα και segmentation faults κατά την εκτέλεση.

3) Σφάλματα σε αριθμητικές πράξεις κινητής υποδιαστολής (floating point operations), οι οποίες υλοποιούνται μέσω των XMM καταχωρητών σε FPU και SIMD μονάδες. Το σενάριο αυτό έχει σχεδιαστεί για να μελετήσει την αλγοριθμική ανθεκτικότητα του KNN σε SDC σφάλματα: συγκεκριμένα, κατά πόσο μια μικρή αριθμητική παραμόρφωση στην τιμή απόστασης μπορεί να επηρεάσει την επιλογή των γειτονικών αποτελεσμάτων, να προκαλέσει αλλοίωση στην απόκριση (response error) ή, σε ακραίες περιπτώσεις, να συμβάλει σε αποτυχία εκτέλεσης (π.χ. segmentation fault)

Για κάθε σενάριο, εφαρμόστηκε σειρά πειραμάτων τύπου N-fault injection campaign, όπου ο αριθμός των εισαγόμενων σφαλμάτων (1 έως N) αυξάνεται σταδιακά. Κάθε πείραμα επαναλαμβάνεται 10–30 φορές με τυχαία επιλογή register και μάσκας (randomized injection), ώστε να επιτευχθεί στατιστικά αξιόπιστη εκτίμηση της ευπάθειας του συστήματος.

Μέσα από τη συστηματική ανάλυση των παραπάνω μετρικών, επιδιώκονται οι εξής στόχοι:

1. Αξιολόγηση της ανθεκτικότητας του bucket server σε σφάλματα, βάσει της ικανότητάς του να εντοπίζει επικίνδυνες καταστάσεις (π.χ. μέσω segmentation fault).

2. Ανάλυση της αξιοπιστίας των αποτελεσμάτων, σε περιπτώσεις όπου η εκτέλεση συνεχίζεται χωρίς crash, αλλά πιθανόν με αλλοιωμένη απόκριση.

3. Κατανόηση της επίδρασης των σφαλμάτων στη ροή ελέγχου (control flow), με στόχο την μελέτη τις αντίδρασης του application σε περίπτωση injection είτε

αυτό οδηγήσει σε αλλοίωση αποτελέσματος, απρόβλεπτη συμπεριφορά ή ακομα και όταν αυτό δεν δημιουργεί κάποια εμφανής αντίδραση του συστήματος.

7.2 Αποτελέσματα

Οι πειραματικοί πίνακες που παρουσιάζονται στο παρόν κεφάλαιο βασίζονται σε καμπάνιες τύπου N-fault injection, όπου για κάθε πείραμα εκτελούνται πολλαπλά queries με σταθερή, ντετερμινιστική σειρά, υπό διαφορετικό πλήθος σφαλμάτων (fault injections) ανά query. Κάθε στήλη στους πίνακες αντιστοιχεί στο πλήθος των injections που εφαρμόζονται σε κάθε query, ενώ κάθε γραμμή καταγράφει διαφορετικές μετρικές για τον αριθμό των injections κατά την εκτέλεση των πειραμάτων.

1. Queries: Ο συνολικός αριθμός των queries που εκτελέστηκαν πριν την εμφάνιση ενός σφάλματος που προκαλεί τερματισμό (π.χ. segmentation fault). Ο αριθμός αυτός καθορίζεται από την εφαρμογή και μπορεί να διαφέρει ανά run, καθώς η καμπάνια διακόπτεται και επανεκκινείται σε κάθε κρίσιμο σφάλμα.

2. Seg faults: Ο αριθμός των runs που οδήγησαν σε segmentation fault. Σε κάθε πείραμα εκτελείται πλήθος queries, όμως σε περιπτώσεις αυξημένων injections, είναι πιθανό κάποια σφάλματα να προκαλέσουν τερματισμό του προγράμματος προτού ολοκληρωθούν όλα τα queries.

3. False reply: Ο αριθμός των queries που επέζησαν από το injection αλλά παρήγαγαν λανθασμένη απόκριση σε σχέση με την κανονική (golden) εκτέλεση.

4. Overall error: Το ποσοστό queries (σε σχέση με το σύνολο) που παρουσίασαν σφάλμα είτε μέσω λανθασμένης απόκρισης είτε μέσω τερματισμού. Η τιμή αυτή αντανακλά την πιθανότητα ένα query να παράξει μη αναμενόμενη συμπεριφορά όταν εκτελείται με το δοθέν πλήθος injections.

Scenario 1: Αποτελέσματα Σεναρίου 1 – Bit Flip σε GPR Registers εντός System Routines

Για κάθε πλήθος injected σφαλμάτων (1 έως 5), πραγματοποιήθηκαν 20 runs με τυχαία επιλογή καταχωρητή και μάσκας, ώστε να αξιολογηθεί η πιθανότητα εμφάνισης segmentation fault, αλλοίωσης της απόκρισης, καθώς και αποκλίσεων στη ροή ελέγχου.

	1 Injection	2 Injections	3 Injections	4 Injections	5 Injections	6 Injections
Queries	59	32	25	24	22	20
Seg faults	20	20	20	20	20	20
False reply	1	0	0	0	0	0
Overall Error	40.68%	68.75%	80%	87.50%	90.91%	100%

Table 7.1 – Error rates observed under different numbers of fault injections per query targeting ALU operations in system routines

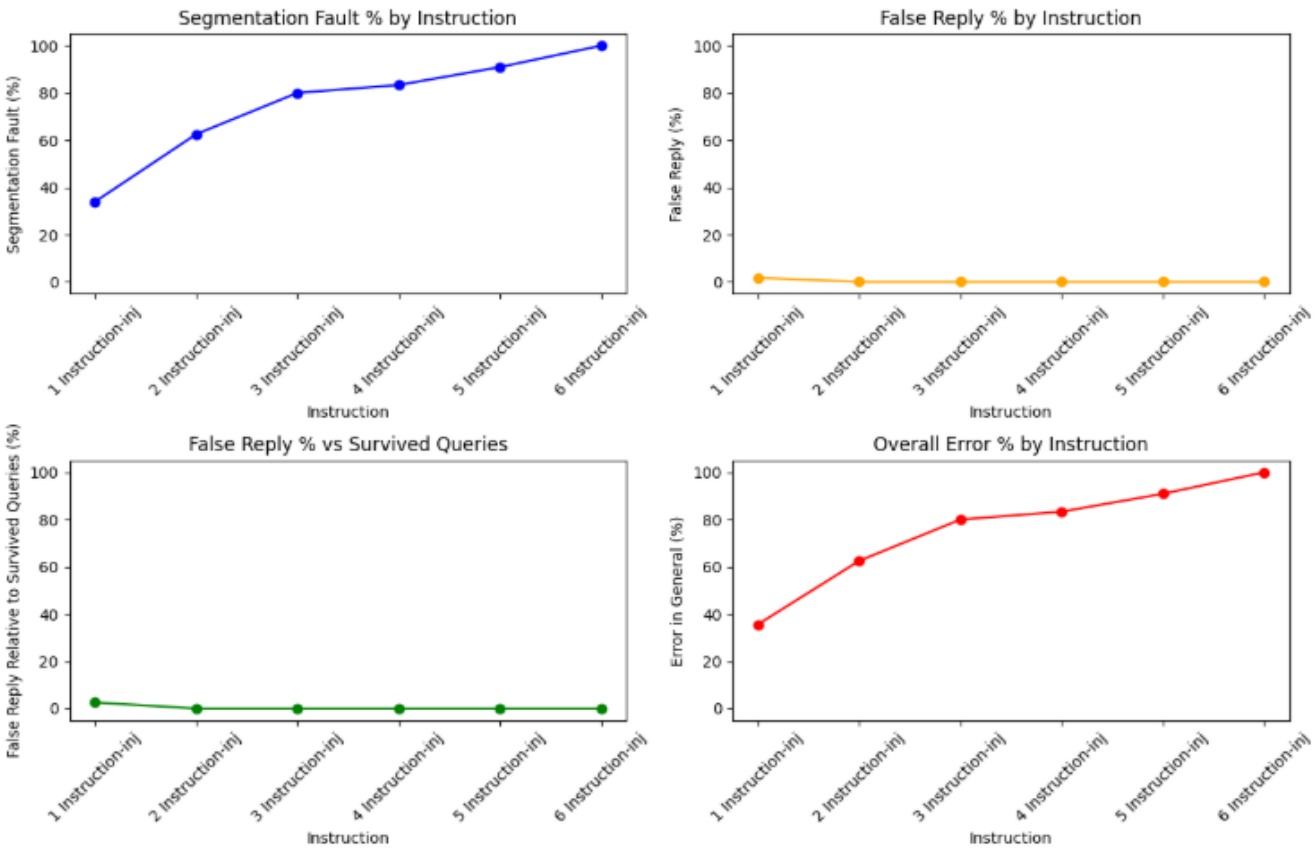

Figure 7.1 – Error behavior under increasing fault injections per query targeting ALU operations in system routines

Οι γραφικές παραστάσεις απεικονίζουν τα αποτελέσματα της fault injection καμπάνιας σε system routines, όπου εισήχθησαν bit-flips σε GPR καταχωρητές κατά την εκτέλεση OS-related εντολών. Στο μπλε διάγραμμα φαίνεται ότι το ποσοστό των segmentation faults αυξάνεται σχεδόν γραμμικά με τον αριθμό των injected instructions, φτάνοντας το 100% για 6 injections, γεγονός που καταδεικνύει την ευπάθεια των system routines στη αλλοίωση καταχωρητών. Αντίθετα, το κίτρινο γράφημα δείχνει ότι η συχνότητα λανθασμένων αποκρίσεων (false replies) παραμένει πολύ χαμηλή, σχεδόν μηδενική, με αυξανόμενα injections – τα περισσότερα faults οδηγούν σε crash πριν την παραγωγή αποτελέσματος. Στο πράσινο διάγραμμα παρουσιάζεται η αναλογία false replies σε σχέση με τα queries που επιβίωσαν (δεν τερματίστηκαν με crash), και πάλι το ποσοστό μειώνεται καθώς αυξάνεται η βαρύτητα του injection. Τέλος, το κόκκινο γράφημα συνοψίζει τη συνολική πιθανότητα σφάλματος (είτε segmentation fault είτε λάθος απάντηση), η οποία αυξάνεται σταθερά, επιβεβαιώνοντας ότι η αλλοίωση GPRs σε system-level code επηρεάζει σημαντικά το σύστημα.

Ανάλυση Segmentation Fault:

Παρατηρείται σχεδόν γραμμική αύξηση του ποσοστού εμφάνισης segmentation fault καθώς αυξάνεται ο αριθμός των fault injections. Για περισσότερα από έξι σφάλματα ανά query, η εκτέλεση κατέληγε σταθερά σε segmentation fault, υποδηλώνοντας την ιδιαίτερη ευαισθησία των system routines σε αλλοιώσεις καταχωρητών.

Ανάλυση Αξιοπιστίας Απόκρισης (False Reply):

Το ποσοστό λανθασμένων αποτελεσμάτων χωρίς συνοδευτικό crash ήταν εξαιρετικά χαμηλό. Συγκεκριμένα, μόνο μια περίπτωση false reply παρατηρήθηκε. Το γεγονός αυτό υποδηλώνει ότι τα περισσότερα κρίσιμα faults που προκύπτουν εντός του λειτουργικού συστήματος ανιχνεύονται και οδηγούν σε άμεσο τερματισμό της εκτέλεσης, περιορίζοντας τον αριθμό των "σιωπηλών" λανθασμένων αποκρίσεων. Τα faults που μπορούν να προκύψουν από System routines είναι μέσω αλλαγής register που δείχνει σε μνήμη η οποία χρησημοποιείτε για την μεταφορά της απάντησης

Αποκλίσεις στον Έλεγχο Ροής (Control Flow):

Σε περιπτώσεις όπου το fault injection οδηγεί σε crash, το instruction sequence του αντίστοιχου query αποκλίνει σημαντικά από τα υπόλοιπα, καθώς ενεργοποιούνται system calls όπως raise() και abort() που διαχειρίζονται την εξαίρεση ή και segmentation faults λόγω παράνομης πρόσβασης σε μνήμη. Η απόκλιση αυτή καθιστά ευδιάκριτο το path ενός crash, επιβεβαιώνοντας την αποτελεσματικότητα του λειτουργικού στην ανίχνευση κρίσιμων σφαλμάτων.

Μια ενδιαφέρον περίπτωση crash:

Κατά τη διάρκεια των πειραμάτων, παρατηρήθηκε μία επαναλαμβανόμενη περίπτωση σφάλματος που εκδηλώνεται περίπου μία φορά ανά 30 queries. Συγκεκριμένα, το εκάστοτε query φαίνεται να εκτελείται κανονικά, με την απόκριση να αποστέλλεται επιτυχώς στον load generator, χωρίς να εντοπίζεται άμεσο σφάλμα εντός της process_request του συγκεκριμένου query. Το σφάλμα αυτό συμβαίνει όταν η έγχυση

σφάλματος, αλλοιώνει με συγκερκιμένο τρόπο μετρητή σε ρουτίνα χαμηλού επιπέδου (nss_hosts_lookup) η οποία εμπλέκεται στη μεταφορά δεδομένων μεταξύ καταχωρητών τύπου XMM και της κύριας μνήμης. Έτσι, το σφάλμα αυτό είναι ανιχνεύσιμο μέσω ελέγχου control flow. Το σφάλμα δεν οδηγεί σε άμεση αστοχία της επεξεργασίας του τρέχοντος query, αλλά επιφέρει memory corruption στο εσωτερικό του bucket server, το οποίο τελικά προκαλεί τον τερματισμό του πριν την επεξεργασία του επόμενου αιτήματος.

Σε περιπτώσεις όπου παρατηρήθηκε αλλαγή στην απόκριση του συστήματος:

Η αλλαγή οφειλόταν σε αλλοίωση δείκτη στη μνήμη χωρίς να μεταβάλλετε η ροή ελέγχου. Συγκεκριμένα, κατά τα πειραματικά στάδια εντοπίστηκε περίπτωση όπου κατά την αποθήκευση της απόκρισης, ένας καταχωρητής που περιείχε δείκτη προς τη διεύθυνση μνήμης του response id αλλοιώθηκε από τη διαδικασία έγχυσης σφάλματος, με αποτέλεσμα να δείχνει σε διαφορετική θέση μνήμης που περιείχε την τιμή 0. Η αλλοίωση αυτή δεν επηρέασε τη ροή ελέγχου του προγράμματος και, συνεπώς, δεν ήταν δυνατό να εντοπιστεί μέσω ελέγχου control flow. Το αποτέλεσμα ήταν μια σιωπηλή αλλοίωση της πληροφορίας που στάλθηκε στον client, χωρίς την παραμικρή ένδειξη αποτυχίας κατά την εκτέλεση

Παρατηρήσεις

Το λειτουργικό σύστημα αντέδρασε αποτελεσματικά στα περισσότερα σφάλματα μέσω μηχανισμών όπως raise() και abort(), υποδεικνύοντας υψηλό επίπεδο προστασίας στον των system library routines. Ωστόσο, ορισμένα faults διαφεύγουν άμεσης ανίχνευσης και εκδηλώνονται με καθυστέρηση, είτε μέσω silent failure είτε ως μεταφερόμενο fault που εντοπίζεται αργότερα από άλλες υπηρεσίες του συστήματος.

Κατά την εκτέλεση των πειραμάτων fault injection παρατηρήθηκαν διάφορες μορφές κρίσιμων σφαλμάτων εκτέλεσης, οι οποίες υποδηλώνουν σοβαρή αλλοίωση της εσωτερικής κατάστασης του προγράμματος. Συγκεκριμένα, εντοπίστηκαν αστοχίες όπως heap corruption (malloc(): memory corruption, double free or corruption), αποτυχίες κατανομής μνήμης (rcmd: Cannot allocate memory), αποτυχίες σε δυναμική

σύνδεση συμβόλων (undefined symbol: __cxa_throw) και εσωτερικά assertion failures στον allocator (tc_idx < TCACHE_MAX_BINS). Τα σφάλματα αυτά αποδίδονται κυρίως σε αλλοιώσεις καταχωρητών γενικής χρήσης (GPRs), που χρησιμοποιούνται ως δείκτες σε δομές μνήμης. Σε όλα τα παραπάνω σενάρια, καθίσταται εμφανές ότι bit-level σφάλματα μπορούν να προκαλέσουν σημαντική αποσταθεροποίηση, με την εκδήλωση των προβλημάτων να εξαρτάται τόσο από το σημείο εισαγωγής του fault όσο και από το μονοπάτι εκτέλεσης που το ενεργοποιεί. Τα ευρήματα αυτά ενισχύουν την ανάγκη για μηχανισμούς απομόνωσης και ελέγχου σε επίπεδο λογισμικού, ιδιαίτερα σε πολυνηματικά microservice περιβάλλοντα.

Scenario 2: Bit Flip σε GPR καταχωρητές εντός Application Code

Στο δεύτερο σενάριο, πραγματοποιήθηκαν 30 ανεξάρτητα runs για κάθε περίπτωση (από 1 εώς 10 faults) με τυχαία εισαγωγή bit flips σε GPR καταχωρητές εντολών του application-level κώδικα.

	1 Injection	2 Injections	3 Injections	4 Injections	5 Injections
Queries	148	78	63	62	49
Seg faults	30	30	30	30	30
False reply	11	10	9	12	6
Overall Error	27.7%	51.28%	61.9%	67.74%	73.47%
	6 Injections	**7 Injections**	**8 Injections**	**9 Injections**	**10 Injections**
Queries	46	35	34	60	30
Seg faluts	31	30	31	58	30
False reply	6	2	1	1	0
Overall Error	80.04%	91.43%	94.12%	98.33%	100%

Table 7.2 – Error rates observed under different numbers of fault injections per query targeting ALU operations in application code routines

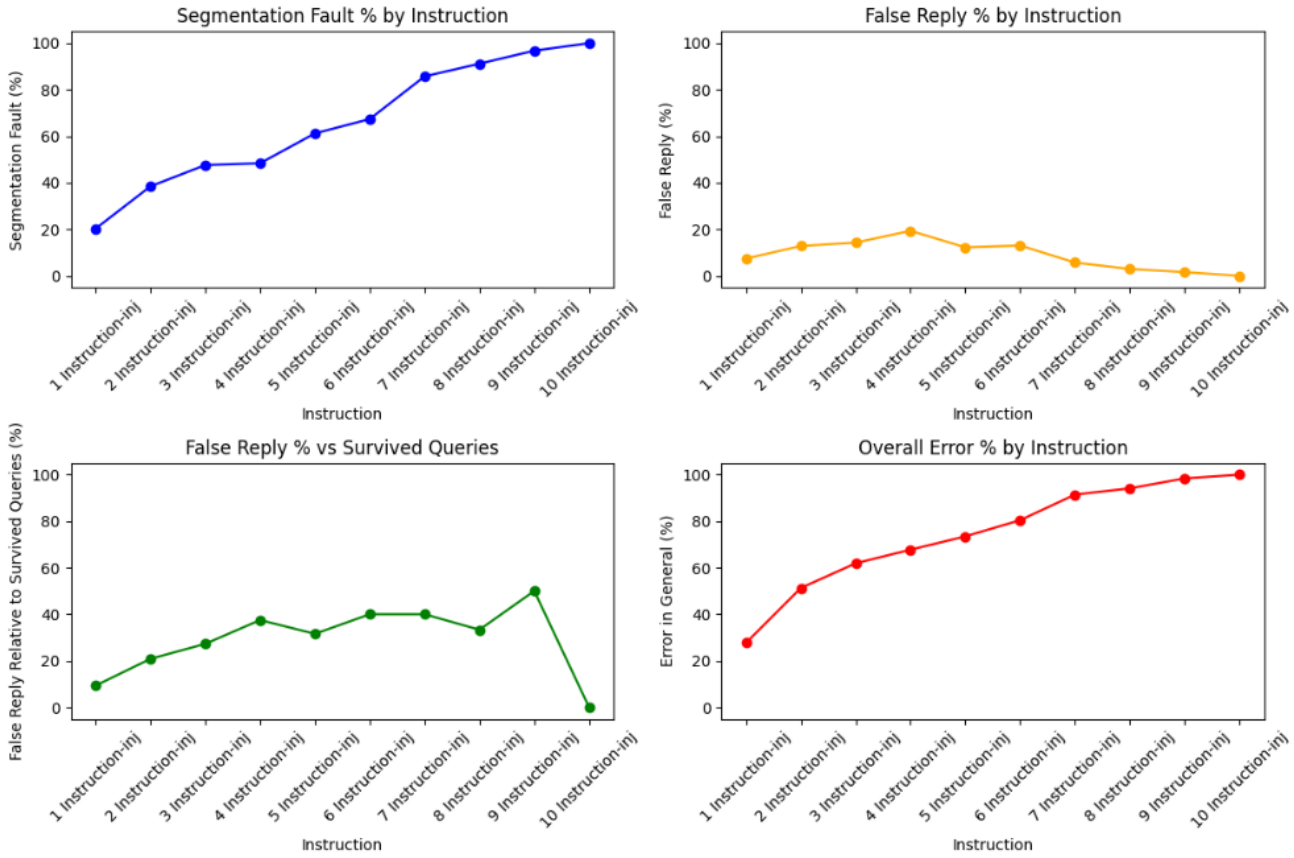

Figure 7.2 – Error behavior under increasing fault injections per query targeting ALU operations in application code routines

Οι γραφικές παραστάσεις απεικονίζουν τα αποτελέσματα της fault injection καμπάνιας στο application-level code με bit-flips σε GPR καταχωρητές. Η μπλε καμπύλη δείχνει ότι το ποσοστό segmentation faults αυξάνεται σταθερά όσο αυξάνεται ο αριθμός των injected εντολών, φτάνοντας σχεδόν το 100% στις 10 εντολές. Αυτό υποδηλώνει ότι το application γίνεται ιδιαίτερα ευάλωτο σε πολλαπλά injections. Η πορτοκαλί καμπύλη που αναπαριστά τα false replies δείχνει αυξομειώσεις στα πρώτα injections (1–4), ενώ από τα 5 και πάνω παραμένει χαμηλά – ενδεχομένως γιατί τα περισσότερα queries καταρρέουν πριν απαντήσουν. Η πράσινη καμπύλη, που δείχνει τα false replies σε σχέση με τα queries που επιβίωσαν, αποκαλύπτει ότι σημαντικό ποσοστό των επιβιωσάντων δίνει λανθασμένες απαντήσεις, ειδικότερα μετά απο τα 4 injections.Τέλος, η κόκκινη καμπύλη καταγράφει το συνολικό ποσοστό σφαλμάτων (είτε false replies είτε segmentation faults), δείχνοντας ξεκάθαρη αύξηση με τον αριθμό των injections και φτάνοντας σε επίπεδα πλήρους αποτυχίας. Αυτό καταδεικνύει ότι το injection σε GPRs στον application code επηρεάζει σε μεγάλο βαθμό το σύστημα.

Ανάλυση Segmentation Fault:

Καθώς αυξάνεται το πλήθος των injected faults εντός του application code, παρατηρείται σχεδόν γραμμική αύξηση του ποσοστού εμφάνισης segmentation faults. Για περισσότερα από 10 injected faults ανά query, το σύστημα οδηγείται σταθερά σε κατάρρευση.

Ανάλυση Αξιοπιστίας της απόκρισης (False Reply):

Η πιθανότητα παραγωγής λανθασμένης απόκρισης αυξάνεται σημαντικά έως και το τέταρτο injected fault. Πέραν αυτού του σημείου, η πιθανότητα σφάλματος σταθεροποιείται. Αυτό οφείλετε στον αυξανόμενο αριθμό των segmentation faults - crashes που παράγονται.

Αποκλίσεις στη ροή ελέγχου (Control Flow):

Μελέτη ροής σε περιπτώσεις Segmentation faults – Crashes:

Από την ανάλυση των injected runs προέκυψε ότι το μεγαλύτερο ποσοστό των crash οφείλονταν σε segmentation faults λόγω παρατεταμένης εκτέλεσης βρόχων του query, όπου εκτελούνταν επαναλαμβανόμενες πράξεις στην KNN . Η υπερβολική επανάληψη οδηγεί σε σταδιακή έξοδο από τα όρια των δεδομένων εισόδου, προκαλώντας παράνομες προσπελάσεις και τερματισμό από το λειτουργικό σύστημα.

Καταγράφηκαν δύο κύριοι τύποι τερματισμού της εφαρμογής:

1) Segmentation fault: Προσπέλαση εκτός ορίων μνήμης. Σφάλμα το οποίο είναι μεταφερόμενο.

Ένα παράδειγμα που καταγράφηκε ήταν ότι κατά την εκτέλεση πειράματος fault injection στη συνάρτηση mkl_blas_mc_xsaxpy της Intel MKL, η αλλοίωση της τιμής του καταχωρητή rdi (loop counter) προκάλεσε αύξηση της αρχικής του τιμής (από 0x260 *xor* mask => 0x4260), οδηγώντας σε σημαντικά περισσότερες επαναλήψεις του βρόχου που εκτελούνταν διάφορες FPU, SSE εντολές. Ενώ ο rdi μειωνόταν κανονικά

μέχρι να ικανοποιηθεί η συνθήκη εξόδου, ένας άλλος καταχωρητής —ο rsi, που χρησιμοποιείται ως pointer σε buffer— αυξανόταν σημαντικά σε κάθε επανάληψη. Η συνεχής αυτή αύξηση οδήγησε τελικά σε προσπέλαση εκτός των επιτρεπτών ορίων μνήμης από εντολή τύπου cvtps2pd xmm, [rsi+offset], προκαλώντας segmentation fault.

2) Abort : Σε αρκετές περιπτώσεις, το injected σφάλμα εντός της process_request δεν εκδηλωνόταν άμεσα, αλλά μεταφερόταν έμμεσα μέσω των παραμέτρων της συνάρτησης, δηλαδή μέσω του stack, σε επόμενες κλήσεις system-level συναρτήσεων. Οι αλλοιωμένοι καταχωρητές (GPRs) περνούσαν ως είσοδοι σε system routines του kernel, όπου χρησιμοποιούνταν για διαχείριση μνήμης ή I/O. Η παρουσία ενός λανθασμένου pointer, π.χ. σε malloc, free ή new, οδηγούσε σε καταστάσεις όπως memory corruption, double free, ή αποτυχία κατανομής μνήμης (bad_alloc), με αποτέλεσμα ο kernel να εντοπίζει το σφάλμα και να τερματίζει την εκτέλεση με abort(). Επίσης, σε ορισμένες περιπτώσεις, το σφάλμα οφειλόταν σε GPR που χρησιμοποιήθηκε σε εσφαλμένα σημεία ως pointer κατά την αποθήκευση floating point τιμών στη μνήμη, οδηγώντας σε καταστροφή δεδομένων ή παραβίαση προστατευμένων περιοχών. Αυτή η καθυστερημένη εκδήλωση του σφάλματος αναδεικνύει τη δυσκολία στον εντοπισμό silent faults που περνούν μέσω του runtime state της εφαρμογής.

Μελέτη ροής σε περιπτώσεις ανεπαίσθητων σφαλμάτων:

Κατά την ανάλυση της ροής εντολών διαπιστώθηκε ότι παρατηρήθηκαν αποκλίσεις στη ροή εκτέλεσης σε σημεία που σχετίζονταν με υπολογισμούς εντός της βιβλιοθήκης Intel MKL, η οποία χρησιμοποιείται για τον υπολογισμό αποστάσεων στη συνάρτηση KNN, καθώς και στη συνάρτηση min_element, η οποία εντοπίζει τον κοντινότερο γείτονα μέσα σε λίστα υποψηφίων. Ωστόσο, σε πολλές από αυτές τις περιπτώσεις, το τελικό αποτέλεσμα παρέμεινε ορθό, καθώς τα αποτελέσματα του *fault injection* συνέπιπταν με εκείνα του golden run, αποδεικνύοντας έτσι την ανθεκτηκότητα του KNN σε SDCs.

Η αιτία αυτών των αποκλίσεων εντοπίζεται σε αλλοιώσεις καταχωρητών που συμμετέχουν σε εντολές ως μετρητές βρόχων (for-loops) της mkl. Όταν τέτοιοι καταχωρητές τροποποιούνται λόγω fault injection, η λογική ροή της εκτέλεσης ενδέχεται να μεταβληθεί, οδηγώντας σε διαφορετική διαδρομή εκτέλεσης, κυρίως εντός

των εσωτερικών υπολογιστικών ρουτινών της βιβλιοθήκης Intel MKL. Αν και οι διαδρομές αυτές μπορεί να εκτελούν μαθηματικά ισοδύναμες εντολές ή να συγκλίνουν στο ίδιο τελικό αποτέλεσμα, η ίδια η ύπαρξη αποκλίσεων υποδεικνύει μια λανθάνουσα ευπάθεια στη ροή ελέγχου.

Μια επιπλέον κατηγορία καταγράφηκε σε περιπτώσεις αλλοίωσης καταχωρητών γενικού σκοπού (GPR), οι οποίοι χρησιμοποιούνται ως δείκτες προς διευθύνσεις μνήμης. Η μεταγενέστερη χρήση ενός τέτοιου καταχωρητή ως pointer κατά την αποθήκευση floating-point τιμής οδήγησε σε αλλοίωση καταχωρητή τύπου XMM, επηρεάζοντας τον υπολογισμό αποστάσεων. Σε περιπτώσεις όπου δεν επηρεάζοταν το αποτέλεσμα το control flow δεν είχε κάποια ένδειξη SDC αντίθετα σε περιπτώσεις αλλοίωσης αποτελέσματος, η συνάρτηση min_element ακολούθησε διαφορετική διαδρομή εκτέλεσης και επέλεξε λανθασμένο γείτονα ως πλησιέστερο, καταδεικνύοντας τη δυνατότητα έμμεσης επίδρασης σφαλμάτων διευθυνσιοδότησης σε κρίσιμους αριθμητικούς υπολογισμούς.

Παρατηρήσεις:

Το παραπάνω εύρημα αναδεικνύει ένα κρίσιμο σημείο στη μελέτη των SDCs: Για να αξιολογηθεί κατά πόσο ένα SDC, το οποίο μεταβάλλει το τελικό response, μπορεί να ανιχνευθεί έμμεσα μέσω αποκλίσεων στη ροή ελέγχου, απαιτούνται στοχευμένα injections σε συγκεκριμένους καταχωρητές που εμπλέκονται στη ροή της εκτέλεσης. Μόνο μέσω τέτοιων στοχευμένων πειραμάτων μπορεί να απομονωθεί ο ρόλος των μεταβλητών ελέγχου και να εκτιμηθεί με ακρίβεια η σχέση μεταξύ λογικής ροής και ακρίβειας αποτελέσματος. Αυτό υποδεικνύει ότι η ικανότητα ενός συστήματος να εντοπίζει SDCs μέσω αποκλίσεων ελέγχου εξαρτάται όχι μόνο από το μέγεθος ή τη σοβαρότητα του σφάλματος, αλλά και από το σημείο και το είδος του καταχωρητή που επηρεάζεται.

Scenario 3: Bit Flip σε Floating Point XMM **Καταχωρητές εντός της Συνάρτησης Intel mkl**

Στο τρίτο σενάριο πειραματισμού, πραγματοποιήθηκαν δέκα ανεξάρτητες εκτελέσεις για κάθε επίπεδο σφαλμάτων (από 50 έως 3050 injected faults), με τυχαία εισαγωγή bit flips σε floating point σε καταχωρητές τύπου XMM κατά την εκτέλεση εντολών εντός της συνάρτησης KNN, η οποία βασίζεται στη βιβλιοθήκη Intel mkl για τους υπολογισμούς αποστάσεων.

Η KNN αποτελεί τον υπολογιστικό πυρήνα της εφαρμογής HDSearch, καθώς εκτελεί πλήθος αριθμητικών πράξεων σε διανύσματα μεγάλης διαστασιμότητας, γεγονός που την καθιστά ιδιαίτερα επιρρεπή στην εμφάνιση SDCs λόγω της έντονης χρήσης των FPU,SEE μονάδων. Τα injection points επιλέχθηκαν εντός της mkl, ξεκινώντας από τα 50 faults, σημείο στο οποίο είχε παρατηρηθεί για πρώτη φορά αλλοίωση στην απόκριση χωρίς όμως στοχευμένα injections σε κρίσημες εντολές. Στόχος ήταν η αξιολόγηση της πιθανότητας εμφάνισης segmentation fault, αριθμητικής αλλοίωσης στην έξοδο, καθώς και αποκλίσεων στη ροή ελέγχου της εφαρμογής.

	50 Injections	**150 Injections**	**250 Injections**	**450 Injections**	**650 Injections**
Queries	380	380	190	190	190
Seg faults	0	0	0	0	0
False reply	1	4	6	5	12
Overall Error	0.26%	1.05%	3.16%	2.63%	6.32%

	850 Injections	1050 Injections	1250 Injections	1450 Injections	1650 Injections
Queries	190	190	190	190	190
Seg faults	0	0	0	0	0
False reply	6	7	13	14	10
Overall Error	3.16%	3.68%	6.84%	7.37%	5.26%
	1850 Injections	**2050 Injections**	**2250 Injections**	**2450 Injections**	**2650 Injections**
Queries	190	190	190	190	190
Seg faults	0	0	0	0	0
False reply	17	12	13	13	20
Overall Error	8.95%	6.32%	6.84%	6.84%	10.53%
	2850 Injections	**3050 Injections**			
Queries	190	380			
Seg faults	0	0			
False reply	22	44			
Overall Error	11.58%	11.58%			

Table 7.3 – Error rates observed under different numbers of fault injections per query targeting floating point operations

Figure 7.3 – Error behavior under increasing fault injections per query targeting floating point operations

Η μπλε γραφική παράσταση (Segmentation Fault %) δείχνει ότι, ανεξαρτήτως του πλήθους των injected faults, το ποσοστό εμφάνισης σφαλμάτων κατάρρευσης (segmentation faults) παραμένει στο 0%, γεγονός που επιβεβαιώνει ότι οι αλλοιώσεις σε XMM καταχωρητές δεν επηρεάζουν κρίσιμες περιοχές μνήμης, αλλά οδηγούν κυρίως σε σιωπηλές αριθμητικές αποκλίσεις. Στην πορτοκαλί καμπύλη (False Reply %), παρατηρείται σταδιακή αύξηση του ποσοστού των αποκρίσεων που αποκλίνουν αριθμητικά από τη σωστή τιμή, από ~0.3% στα 50 faults έως ~12% στα 3050 faults, υποδηλώνοντας ότι οι αλλοιώσεις στους καταχωρητές κινητής υποδιαστολής έχουν άμεσο αντίκτυπο στην ακρίβεια των αποτελεσμάτων. Παρόλα αυτά, παρατηρείται ότι απαιτούνται περισσότερες από 50 μη στοχευμένες εγχύσεις σφαλμάτων προκειμένου να παρουσιαστεί, σε σύνολο 190 queries, μία εσφαλμένη απόκριση. Το γεγονός αυτό αναδεικνύει την εγγενή ανθεκτικότητα του αλγορίθμου KNN έναντι σιωπηλών αλλοιώσεων δεδομένων (SDCs).

Ανάλυση Segmentation Fault Rate:

Η εισαγωγή σφαλμάτων σε floating point καταχωρητές τύπου XMM στο fractional part δεν οδηγεί ποτέ σε σφάλματα κατάρρευσης (segmentation faults), ακόμη και με αυξανόμενο πλήθος injected faults. Η παρατήρηση αυτή επιβεβαιώνει ότι τα bit flips που λαμβάνουν χώρα στους συγκεκριμένους καταχωρητές δεν επηρεάζουν κρίσιμες δομές μνήμης ή pointers, με αποτέλεσμα η εκτέλεση να συνεχίζεται κανονικά χωρίς εμφανείς δυσλειτουργίες. Το φαινόμενο αυτό είναι ενδεικτικό της σιωπηλής φύσης των SDCs όταν περιορίζονται σε αριθμητικές μονάδες υπολογισμού FPU, SSE.

Ανάλυση Αξιοπιστίας Απόκρισης (False Reply):

Η ακρίβεια της απόκρισης παρουσίασε σαφή εξάρτηση όχι μόνο από τον αριθμό των injected faults, αλλά κυρίως από την επιλεγμένη εντολή, τον στοχευμένο καταχωρητή και τη θέση (mask) του bit flip. Αυτό το συμπέρασμα εξάγεται από τα spikes στην γραφική παράσταση γεγονός που δείχνει μεγάλη σημασία στην τοποθεσία του σφάλματος.

Επιπρόσθετα, μιας και η επιλογή query – response γίνενετε από την ίδια βάση, (αλλά δεν γίνετε hashed πάντα η ίδια εικόνα στους υποψήφιους γείτονες). Μέσω της ανάλυσης των αποτελεσμάτων παρατηρήθηκε ότι queries των οποίων ο κοντινότερος γείτονας ήταν η ίδια η εικόνα ως response,

$$d(q,r) = 0,\ q = r,$$

στην κανονική εκτέλεση, εμφάνιζαν χαμηλότερη ευαισθησία σε σφάλματα σε fault εκτέλεση (σπάνιο αλλά όχι απίθανο), καθώς μικρές αλλοιώσεις στο fraction part των floating point υπολογισμών δεν επηρέαζαν σε τόσο μεγάλο βαθμό τις αποστάσεις ώστε να απομακρύνουν δύο ίδιες εικόνες και να επιλεγεί άλλη.

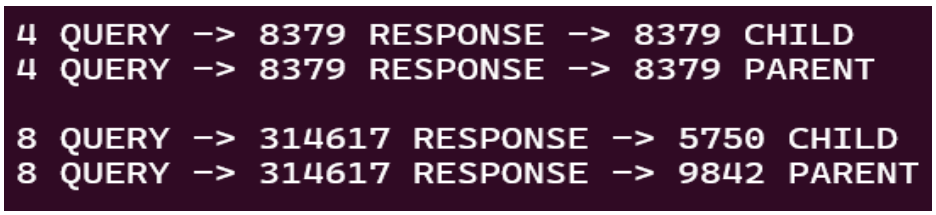

Figure 7.4 – Self-matching queries exhibit reduced sensitivity to floating-point faults.

Επίσης, αν εξερέσουμε τα ερωτήματα που είχαν ίδιο query – response id. Μέσω της ανάλυσης παρατηρήθηκε ότι συγκεκριμένα queries είχαν μεγαλύτερη πιθανότητα αλλαγής στην απόκριση σε σχέση με άλλα. Μάλιστα σημαντική παρατηρούμενη τάση αλλοίωσης παρατηρήθηκε ειδικά στα queries 6 και 8, αυτό έγκειται στο γεγονός ότι οι υποψήφιοι nearest neighbours(που στάλθηκαν από mid tier) είχαν αρκετά κοντινή απόσταση από το query,

δηλ. αν $d(q->r1) - d(q->r2) < \varepsilon$, *όπου r1,r2 υποψήφιοι γείτονες, και ε threshold που δείχνει μικρή διαφορά απόστασης*

συνεπώς με την αλλαγή της τελικής απόστασης query - neighbours από τα injections ένας άλλος γείτονας είχε πιο μικρή πραγματική απόσταση και επιλέχθηκε αυτός.

Τέλος, όταν η απόκριση άλλαζε για ένα query, το σφάλμα επαναλαμβανόταν σταθερά, ανεξαρτήτως μάσκας ή θέσης έγχυσης σφάλματος, δείχνωντας έτσι ότι το πολύ αν δεν επιλεγεί ο πρώτος κοντινότερος θα επιλεγεί ο δεύτερος. (Μπορούν να υπάρξουν ακρέες περιπτώσεις που θα επιλεγεί άλλος γείτονας αλλά δεν υφίσταται σε πραγματικές καταστάσεις SDC).

Αποκλίσεις στον Έλεγχο Ροής (Control Flow):

Κατά τη σύγκριση μεταξύ των εκτελέσεων αναφοράς και των αντίστοιχων fault-injected εκτελέσεων, παρατηρήθηκε ότι υπό σταθερό input και απουσία λειτουργικού

αντίκτυπου, δηλαδή αλλαγής στην απόκριση, στις περισσότερες περιπτώσεις σιωπηρής αλλοίωσης δεδομένων, δεν παρατηρείται μεταβολή στη ροή ελέγχου (control flow). Ωστόσο, η συνάρτηση min_element, η οποία είναι υπεύθυνη για την επιλογή της ελάχιστης τιμής μέσα σε ένα διάνυσμα αποστάσεων, παρουσιάζει ιδιαίτερο ενδιαφέρον ως σημείο παρατήρησης αποκλίσεων.

```
void DistCalc::GetNN(const MultiplePoints &dataset,

int min_index = distance(min_dists.begin(), min_element(min_dists.begin(),min_dists.end()));
```

Figure 7.5 – Determining the index of the closest neighbour using min_element

Η συγκεκριμένη ρουτίνα είναι υπεύθυνη για την επιλογή του στοιχείου με τη μικρότερη αριθμητική τιμή μέσα σε μια λίστα αποστάσεων που έχουν υπολογιστεί μεταξύ ενός query και των υποψήφιων γειτόνων (response candidates). Καθώς μια bit-level αλλοίωση επηρεάζει την αριθμητική ακρίβεια των υπολογισμών απόστασης, ενδέχεται να επιλεγεί διαφορετικός γείτονας ως πλησιέστερος, μεταβάλλοντας το τελικό αποτέλεσμα. Συγκεκριμένα, η γενικευμένη λογική της *min_element*:

```
for (int i = 0; i < values.size(); ++i) {
    if (values[i] < min_value) {
        min_value = values[i];
        min_index = i;
    }
}
```

Ο σχετικός στατικός ψευδοκώδικας σε assembly ακολουθεί τη μορφή:

```
;Αρχικοποίση τιμών στοίβας;
…
; Αρχικοποίηση min = i

0xAB: mov rax, [rbp-0x18]
0xAC: mov [rbp-0x8], rax              ; min = i

0xAD: lea rdi, [rbp-0x18]
0xAE: call operator++                 ; ++i

; Αρχή loop: while (i < values.size())

0xB0: mov rdi, result_of_++
0xB1: lea rsi, [rbp-0x20]             ; values.size()
0xB2: call operator<
0xB3: test al, al
0xB4: jz 0xC0                  ; αν i == values.size(), πήγαινε στην επιστροφή

; if (values[i] < min_value)

0xB5: mov rsi, [rbp-0x18]             ; current element
0xB6: mov rdi, [rbp-0x8]              ; current min
0xB7: call operator<

0xB8: test al, al
0xB9: jz 0xBD                         ; αν όχι, δεν αλλάζει το min
0xBA: mov rax, [rbp-0x18]             ; αλλιώς:
0xBB: mov [rbp-0x8], rax              ; min = i

; ++i και συνέχισε το loop

0xBD: lea rdi, [rbp-0x18]
0xBE: call operator++
0xBF: jmp 0xB0                        ; repeat

; Επιστροφή του min

0xC0: mov rax, [rbp-0x8]
0xC1: leave
0xC2: ret
```

Η εμφάνιση αλλαγής στη ροή ελέγχου (control flow) εντός της min_element εξαρτάται από την ενεργοποίση του if statement (κώδικας με κόκκινο χρώμα) και μπορεί να ερμηνευτεί με δύο τρόπους:

1. Όταν υπάρχει αλλαγή στο αποτέλεσμα, δηλαδή διαφορετικό min_index, τότε είναι βέβαιο ότι υπήρξε αλλαγή στη ροή ελέγχου μέσα στη min_element. Αυτό οφείλεται στο γεγονός ότι η αλλαγή του αποτελέσματος σημαίνει πως διαφορετικό στοιχείο πληρεί τη συνθήκη values[i] < min_value σε διαφορετικό βήμα της επανάληψης.

2. Μπορεί να παρατηρηθεί αλλαγή στον έλεγχο ροής ακόμη και όταν το τελικό αποτέλεσμα (min_index) παραμένει αμετάβλητο. Αυτό συμβαίνει όταν η αλλοίωση των δεδομένων οδηγεί σε διαφορετική πορεία εκτέλεσης (π.χ. ενεργοποίηση ή μη του if), αλλά τελικά επιλέγεται το ίδιο στοιχείο ως ελάχιστο.

Για παράδειγμα, έστω:

- **Golden min_list:** {**5**, 7, 6, **2**, 8} → επιστρέφει min_index = 3
- **Faulty min_list:** {**5**, **4**, 6, **2**, 8} (αλλοίωση στη δεύτερη θέση) → επιστρέφει επίσης min_index = 3

Σε αυτή την περίπτωση η αλλοίωση σε floating point επίπεδο (FPU, SSE) προκαλεί αλλαγή της ροής εκτέλεσης, αλλά όχι του τελικού αποτελέσματος. Δηλαδή, το if ενεργοποιείται διαφορετικά (golden: 5,2 faulty: 5,4,2), όμως η τελική τιμή παραμένει η ίδια.

Για την αξιόπιστη ανίχνευση τέτοιων φαινομένων, μπορεί να χρησιμοποιηθεί ένας βοηθητικός βρόχος, όπως:

for (int i = 0; i < min_index; i++) { / do nothing */ }*

Ο βρόχος αυτός, αν και λειτουργικά αδρανής, εξαρτάται άμεσα από την τιμή του min_index, και συνεπώς η καταγραφή του στο instruction trace επιτρέπει τη σίγουρη

ανίχνευση αλλαγής του αποτελέσματος, ακόμα και σε περιπτώσεις όπου η ροή εντός της min_element δεν αποκλίνει εμφανώς.

Συμπερασματικά, η min_element είναι ιδιαίτερα ευαίσθητη σε bit-level αλλοιώσεις, και οι αποκλίσεις στη ροή της συνδέονται στενά με τη σημασιολογική ορθότητα του αποτελέσματος. Η παρουσία ή απουσία αποκλίσεων αποτελεί ισχυρό δείκτη για την ανίχνευση ή τη διάψευση της ύπαρξης σφάλματος.

Παρατηρήσεις - Ανάλυση του Αλγορίθμου K-Nearest Neighbors (KNN):

Στην εφαρμογή HDSEARCH του μSuite, ο αλγόριθμος K-Nearest Neighbors (KNN) υλοποιείται εντός της συνάρτησης process_request του bucket server και αποτελεί το κρίσιμο υπολογιστικό στάδιο για την ανάκτηση της πιο παρόμοιας εικόνας σε σχέση με ένα ερώτημα (query). Η λειτουργία του βασίζεται στη σύγκριση ενός διανύσματος χαρακτηριστικών (feature vector) του query με ένα σύνολο διανυσμάτων από τη βάση δεδομένων που έχουν προεπιλεγεί από το mid-tier ως υποψήφιοι γείτονες.

Η μέτρηση ομοιότητας μεταξύ των διανυσμάτων γίνεται μέσω υπολογισμού αποστάσεων, με χρήση βελτιστοποιημένων συναρτήσεων της βιβλιοθήκης Intel MKL. Οι αποστάσεις αυτές αποθηκεύονται σε έναν πίνακα, και στη συνέχεια χρησιμοποιείται η ρουτίνα *min_element* για την εύρεση του index του γείτονα με τη μικρότερη απόσταση, δηλαδή του πιο παρόμοιου response.

Από τα πειραματικά δεδομένα προκύπτει ότι, οι αλλοιώσεις που εντοπίζονται στο fractional part των αποτελεσμάτων floating point πράξεων (π.χ. addss, mulps) σε XMM καταχωρητές, μπορούν να μεταβάλουν τις αποστάσεις σε τέτοιο βαθμό, ώστε να αλλάξει το αποτέλεσμα και κατ’ επέκταση η ροή στη min_element routine. Αυτό οδηγεί στην επιλογή διαφορετικού γείτονα — και συνεπώς διαφορετικού response ID.

Το φαινόμενο αυτό είναι εντονότερο σε queries όπου οι αποστάσεις μεταξύ του πρώτου και δεύτερου πλησιέστερου γείτονα είναι πολύ κοντινές (π.χ. $|d(q \rightarrow r1) - d(q \rightarrow r2)| < \varepsilon$). Σε αυτές τις περιπτώσεις, ακόμα και μία ελάχιστη αλλοίωση στο αποτέλεσμα απόστασης μπορεί να ανατρέψει τη σειρά ταξινόμησης και να αλλάξει την απόκριση.

Αντίθετα, queries όπου το response ID είναι το ίδιο με το query ID (δηλαδή d(q, r) = 0) αποδείχθηκαν σταθερά και ανθεκτικά, καθώς ακόμη και με injected faults, η απόσταση δεν αλλοιώνεται αρκετά ώστε να ανατραπεί η απόφαση.

Ολοκληρώνοντας την ανάλυση, προκύπτει ότι ο αλγόριθμος KNN και, γενικότερα, το microservice της HDSearch παρουσιάζουν αξιοσημείωτη ανθεκτικότητα έναντι σιωπηλών αλλοιώσεων δεδομένων (SDCs) που εντοπίζονται σε floating point αριθμητικές πράξεις. Το εύρημα αυτό είναι ιδιαιτέρως σημαντικό, καθώς αυτού του τύπου οι αλλοιώσεις είναι οι πιο πιθανές σε εφαρμογές υψηλής υπολογιστικής έντασης, όπως η HDSearch, όπου το μεγαλύτερο μέρος του υπολογιστικού φόρτου συγκεντρώνεται στη συνάρτηση knn.

Η ανθεκτικότητα αυτή επιβεβαιώθηκε πειραματικά, καθώς απαιτούνται περισσότερα από 50 τυχαία injected faults προκειμένου να παρατηρηθεί έστω και μία λανθασμένη απόκριση σε ένα σύνολο 380 queries. Το γεγονός αυτό αποδεικνύει ότι, υπό τυχαίες συνθήκες SDC, η πιθανότητα λειτουργικής αστοχίας είναι ιδιαίτερα χαμηλή.

Ωστόσο, για να επιτευχθεί λεπτομερής αποτίμηση της αξιοπιστίας του αλγορίθμου, απαιτείται στοχευμένη μελέτη συγκεκριμένων εγχύσεων σφαλμάτων σε καθορισμένους καταχωρητές και συγκεκριμένες εντολές που εμπλέκονται απευθείας στον υπολογισμό αποστάσεων. Παράλληλα, χρήζει περαιτέρω διερεύνησης η συμπεριφορά queries με πολύ κοντινές αποστάσεις μεταξύ υποψήφιων γειτόνων, όπου η ευαισθησία σε μικρές αριθμητικές αποκλίσεις είναι αυξημένη.

Η μελέτη τέτοιων περιπτώσεων, σε συνδυασμό με μειωμένο πλήθος injections και εντοπισμένο fault targeting, θα επιτρέψει την πιο ακριβή εκτίμηση της λειτουργικής αξιοπιστίας του συστήματος υπό ρεαλιστικά σενάρια SDC, και θα θέσει τα θεμέλια για μελλοντικές τεχνικές ανίχνευσης ή μετριασμού των επιπτώσεων τέτοιων σφαλμάτων σε περιβάλλοντα microservices.

Κεφάλαιο 8

Συμπεράσματα

8.1 Περίληψη

Στην παρούσα διπλωματική εργασία, πραγματοποιήθηκε μια ολοκληρωμένη διερεύνηση τεχνικών ένεσης σφαλμάτων, με στόχο την κατανόηση των σιωπηλών διαφθορών δεδομένων και των επιπτώσεών τους σε εφαρμογές microservice. Η μελέτη επικεντρώθηκε συγκεκριμένα στο μικροϋπηρεσιακό σύστημα HDSearch, το οποίο αποτελεί μέρος του μSuite[9]. Αρχικό κίνητρο αποτέλεσε η ανάγκη κατανόησης της συμπεριφοράς μικροϋπηρεσιών υπό συνθήκες μεταβαλλόμενης αξιοπιστίας, καθώς και η διερεύνηση του τρόπου με τον οποίο το σύστημα μπορεί να διαχειριστεί ένα πιθανό silent fault, με ιδιαίτερη έμφαση στην ανάλυση του control flow:
σε περιπτώσεις όπου δεν προκαλείτε κάποιο εμφανές σφάλμα, σε περιπτώσεις που υπάρχει αλλαγή στο response του application και σε περιπτώσεις που προκαλείται segmentation fault ή crash.

Για τη μελέτη της ροής ελέγχου και την εισαγωγή σφαλμάτων σε κρίσιμα σημεία του εκτελούμενου κώδικα, χρησιμοποιήθηκε το εργαλείο δυναμικής ανάλυσης Intel Pin[7], στο οποίο βασίστηκε η ανάπτυξη ενός αυτοματοποιημένου pin tool για fault injection. Ο στόχος ήταν η δημιουργία ενός εργαλείου που να μπορεί να προσομοιώσει με ρεαλιστικό τρόπο σφάλματα σε αριθμητικές μονάδες επεξεργασίας (ALU και FPU/SSE), εντός διαφορετικών μικροϋπηρεσιών. Το εργαλείο αυτό επέτρεψε την παρεμβολή σφαλμάτων πριν το write-back του αποτελέσματος στους καταχωρητές, ώστε να μοντελοποιηθεί η επίδραση ενός πραγματικού silent fault που προκύπτει σε επίπεδο υλικού.

Κατά τη διάρκεια της μελέτης, εντοπίστηκαν κρίσιμα ζητήματα που έπρεπε να αντιμετωπιστούν. Ένα από τα σημαντικότερα προβλήματα ήταν η εμφάνιση μη ντετερμινιστικού control flow μεταξύ διαφορετικών εκτελέσεων golden runs, με ίδιο input.

Για την αντιμετώπιση αυτού του φαινομένου, χρησιμοποιήθηκε η δημιουργία static binaries, απενεργοποιήθηκε η τεχνική ASLR του λειτουργικού συστήματος και επελέγει η χρήση της τεχνικής redundancy execution μέσω fork(). Η συγκεκριμένη προσέγγιση επισημάνθηκε και στη βιβλιογραφία ως ιδιαίτερα αποτελεσματική για την απομόνωση σφαλμάτων σε nondeterministic περιβάλλοντα.

Ένα ακόμα σημαντικό εύρημα αφορούσε τη μεγάλη πιθανότητα crash σε περιπτώσεις όπου γινόταν fault injection σε γενικής χρήσης καταχωρητές (GPRs) εντός συστημικών βιβλιοθηκών χρήστη, εξαιτίας του ρόλου των GPRs ως pointers.

Τέλος, η μελέτη οργανώθηκε σε τρία βασικά σενάρια injection: σε GPRs που επηρεάζουν λειτουργίες του συστημικών βιβλιοθηκών χρήστη, σε GPRs που επηρεάζουν απευθείας τον κώδικα του application, και σε floating-point καταχωρητές τύπου XMM.

Ένα βασικό συμπέρασμα που προέκυψε είναι ότι το σύστημα μSuite και ειδικά το HDSearch αποδείχθηκε ιδιαίτερα ανθεκτικό. Η πιθανότητα να προκύψει κάποιο εμφανές σφάλμα στην απόκριση σε περιπτώσεις εισαγωγής ενός σφάλματος σε ένα query ήταν εξαιρετικά μικρή, ακόμα και σε floating-point πράξεις, που και θεωρητικά παρουσιάζουν μεγαλύτερη πιθανότητα σφάλματος σε πραγματικά περιβάλλοντα, είχαν περιορισμένη επίδραση στο σύστημα. Ωστόσο, παρατηρήθηκε μία περίπτωση σφάλματος με infinite loop σε ρουτίνες αποδεύσμευσης χωρίς να εντοπιστεί απο το σύστημα,η οποία αν συνέβαινε σε παραγωγικό περιβάλλον, θα μπορούσε να έχει καταστροφικές συνέπειες.

8.2 Μελλοντική δουλειά

Η παρούσα εργασία έθεσε τα θεμέλια για την ανάλυση silent faults σε περιβάλλοντα microservice, εστιάζοντας στην ένεση σφαλμάτων σε κρίσιμα σημεία του κώδικα και την αντίδραση των microservices σε αυτά. Ωστόσο, προέκυψαν αρκετές κατευθύνσεις για μελλοντική έρευνα που μπορούν να επεκτείνουν και να εμβαθύνουν τα ευρήματα της μελέτης.

Μια σημαντική κατεύθυνση για μελλοντική εργασία αφορά τη διερεύνηση της απόκλισης ροής, που προκαλείται, από την ένεση σφαλμάτων στο FPU, στην std::min_element, συγκριτικά με τη φυσική απόκλιση που προκύπτει από την εκτέλεση διαφορετικών queries χωρίς παρουσία σφαλμάτων. Συγκεκριμένα, θα μπορούσε να εξεταστεί σε ποιο βαθμό μια αλλοίωση σε επίπεδο bit επηρεάζει την επιλογή του ελάχιστου στοιχείου, και πώς αυτή η αλλοίωση συγκρίνεται με την εγγενή μεταβλητότητα που υπάρχει στο σύστημα λόγω διαφορών στα δεδομένα εισόδου. Μια τέτοια ανάλυση θα επέτρεπε τον ποσοτικό προσδιορισμό του σφάλματος και θα συνέβαλλε στην κατανόηση του κατά πόσο τα σφάλματα αλλοιώνουν πραγματικά την έξοδο με τρόπο διακριτό από τη φυσική μεταβλητότητα του συστήματος, ανοίγοντας τον δρόμο για πιο στοχευμένες τεχνικές ανίχνευσης silent faults.

Επίσης, διαπιστώθηκε ότι η δημιουργία ενός πλήρως ντετερμινιστικού instruction sequence παραμένει τεχνικά και υπολογιστικά απαιτητική. Έτσι, είναι αναγκαία η ανάπτυξη πιο αποτελεσματικών μεθόδων σύγκρισης golden και fault run, ενδεχομένως μέσω ανάλυσης της κατάστασης των παραμέτρων και αποτελεσμάτων κάθε ρουτίνας και της παρακολούθησης των κρίσιμων τιμών σε καταχωρητές και μεταβλητές.

Επιπλέον, το fault injection σε καταχωρητές SSE/XMM αναδεικνύει την ανάγκη για βελτιστοποίηση αλγορίθμων, όπως ο KNN, ώστε να μπορούν να αντεπεξέλθουν σε σφάλματα αριθμητικής φύσης χωρίς να οδηγούνται σε λανθασμένα αποτελέσματα. Η μελλοντική εργασία θα μπορούσε να επικεντρωθεί στον σχεδιασμό ανθεκτικών εκδόσεων τέτοιων αλγορίθμων που διατηρούν την ακρίβεια υπό την παρουσία σφαλμάτων χαμηλού επιπέδου.

Μια ακόμη κατεύθυνση είναι η εφαρμογή της τεχνικής thread redundancy σε πραγματικά περιβάλλοντα παραγωγής, με στόχο τη μελέτη της αποδοτικότητας και του κόστους εντοπισμού silent faults μέσω συγκρίσεων μεταξύ threads. Η πρακτική αξιολόγηση αυτής της τεχνικής θα μπορούσε να οδηγήσει σε πραγματικές υλοποιήσεις μηχανισμών ανίχνευσης σφαλμάτων εντός μικροϋπηρεσιών.

Επιπρόσθετα, το εργαλείο fault injection που χρησιμοποιήθηκε στην παρούσα εργασία μπορεί να αποτελέσει τη βάση για την ενσωμάτωση επιπλέον τεχνικών εισαγωγής σφαλμάτων με στόχο την αύξηση της αυτοματοποίησης και της κάλυψης σε ετερογενή περιβάλλοντα. Για παράδειγμα, θα μπορούσε να υποστηριχθεί injection σε ευρύτερους καταχωρητές SIMD, όπως οι YMM και ZMM, ενισχύοντας τη δυνατότητα προσομοίωσης σφαλμάτων σε σύγχρονες αρχιτεκτονικές. Επίσης, η χρήση αναλύσεων που εντοπίζουν τα κρίσιμα execution paths του κώδικα, δηλαδή εκείνα που επηρεάζουν άμεσα την κατάσταση του συστήματος ή την έξοδο της υπηρεσίας, θα επέτρεπε την εστιασμένη ένεση σφαλμάτων σε σημεία με μεγαλύτερο αντίκτυπο. Με αυτόν τον τρόπο, το εργαλείο θα μπορούσε να εξελιχθεί σε ένα περισσότερο στοχευμένο και αποδοτικό σύστημα μελέτης της συμπεριφοράς μικροϋπηρεσιών υπό συνθήκες silent fault.

Τέλος, καθώς οι μικροϋπηρεσίες χρησιμοποιούνται όλο και περισσότερο σε κρίσιμες και σύνθετες εφαρμογές, καθίσταται απαραίτητη η διεπιστημονική συνεργασία μεταξύ ειδικών στην αξιοπιστία συστημάτων, την ανοχή σε σφάλματα και την ανάλυση λογισμικού. Μέσα από τη σύμπραξη αυτών των πεδίων, μπορούν να αναπτυχθούν ανθεκτικότερες τεχνικές εντοπισμού και αποκατάστασης σφαλμάτων, ενισχύοντας τη σταθερότητα και την αξιοπιστία των αρχιτεκτονικών microservice.

Βιβλιογραφία

[1] SOSP 2023: Shaobu Wang, Guangyan Zhang, Junyu Wei, Yang Wang, Jiesheng Wu, and Qingchao Luo. Understanding Silent Data Corruptions in a Large Production CPU Population. In Proceedings of the 29th Symposium on Operating Systems Principles. 2023
Link: https://dl.acm.org/doi/10.1145/3600006.3613149

[2] Harish Dattatraya Dixit, Laura Boyle, Gautham Vunnam, Sneha Pendharkar, Matt Beadon, and Sriram Sankar. 2022. Detecting silent data corruptions in the wild.
Link: https://arxiv.org/abs/2203.08989

[3] ASPLOS 2025: Rhea Dutta, Harish Dattatraya Dixit, Rik Van Riel, Gautham Vunnam, Sriram Sankar. Hardware Sentinel: Protecting Software Applications from Hardware Silent Data Corruptions. Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2
Link: https://dl.acm.org/doi/10.1145/3676641.3716258

[4] Riesen, Rolf E., Ferreira, Kurt Brian, Fiala, David, Mueller, Frank, & Engelmann, Christian (2011). Detection and Correction of Silent Data Corruption for Large-Scale High-Performance Computing.
Link: https://www.osti.gov/biblio/1110355

[5] N. Karystinos, O. Chatzopoulos, G. -M. Fragkoulis, G. Papadimitriou, D. Gizopoulos and S. Gurumurthi, "Harpocrates: Breaking the Silence of CPU Faults through Hardware-in-the-Loop Program Generation," *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, Buenos Aires, Argentina, 2024.
Link: https://ieeexplore.ieee.org/document/10609719

[6] Panteleimonas Chatzimiltis, Georgia Antoniou, Haris Volos, Yiannakis Sazeides, SAGA: A Surrogate Assisted Genetic Algorithm for Fast CPU Power Virus Generation

[7] Pin:
https://software.intel.com/sites/landingpage/pintool/docs/98484/Pin/html/index.html

[8] Download Pin:
https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-binary-instrumentation-tool-downloads.html

[9] μSuite: A Benchmark Suite for Microservices", Akshitha Sriraman and Thomas F. Wenisch, IEEE International Symposium on Workload Characterization, September 2018
Link:http://akshithasriraman.eecs.umich.edu/pubs/IISWC2018-%CE%BCSuite-preprint.pdf
https://github.com/wenischlab/MicroSuite

[10] μSuite ucy github repository: https://github.com/ucy-xilab/MicroSuite

[11] fork function :
https://pubs.opengroup.org/onlinepubs/000095399/functions/fork.html

[12] CloudLab: https://www.cloudlab.us/user-dashboard.php#profiles

[13] Docker compose: https://docs.docker.com/compose/

[14] To Github Repository μου: https://github.com/kendeasdimitriou/MicroSuite-hdsearch-pinjector/tree/master