ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΣΧΕΔΙΟ ΠΑΡΟΥΣΙΑΣΗΣ
ΑΤΟΜΙΚΗΣ ΔΙΠΛΩΜΑΤΙΚΗΣ ΕΡΓΑΣΙΑΣ

Μάιος 2014

Ατομική Διπλωματική Εργασία

SOFTWARE FAULT INJECTION

Ξάνθη Ζαχαρίου

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Μάιος 2014

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Software Fault Injection

Ξάνθη Ζαχαρίου

Επιβλέπων Καθηγητής
Γιάννος Σαζείδης

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

Μάιος 2014

Ευχαριστίες

Αρχικά, θα ήθελα να εκφράσω τις ευχαριστίες μου στον επιβλέπων καθηγητή μου κ. Γιαννάκη Σαζείδη για όλη τη βοήθεια και καθοδήγηση που μου παρείχε κατά τη διάρκεια της εκπόνησης της διπλωματικής μου εργασίας. Θα ήθελα ακόμη να τον ευχαριστήσω για το θέμα που μου ανέθεσε καθώς είναι ένα ενδιαφέρον θέμα που κατά την διεκπεραίωσης του με βοήθησε να αποκτήσω αρκετές γνώσεις για θέματα αξιοπιστίας ενός υπολογιστικού συστήματος και για θέματα μνήμης και απόδοσης , τα οποία είναι κρίσιμα για την τεχνολογία των υπολογιστών.

Επιπλέον, θα ήθελα να ευχαριστήσω την Παναγιώτα Νικολάου για τη βοήθεια και στήριξη που μου παρείχε σε όλη τη διάρκεια της διπλωματικής μου εργασίας , και ιδιαίτερα επεξηγώντας μου απορίες σχετικά με το θέμα της διπλωματικής μου.

Περίληψη

Από τη στιγμή που η τεχνολογία έχει εισβάλει στην καθημερινή μας ζωή , υπήρξε απαίτηση από την ανθρωπότητα όπως υπολογιστικά συστήματα είτε μεγάλα είναι αυτά είτε μικρά να λειτουργούν πιο γρήγορα και χωρίς την ύπαρξη λαθών την στιγμή που η τεχνολογία εστιαζόταν στο πόσο γρήγορα θα λειτουργήσει ένα τέτοιο σύστημα .Φυσικά σε κανένα υπαρκτό υπολογιστικό σύστημα δεν υπάρχει περίπτωση να μην υπάρξουνε σφάλματα. Πλέον η τεχνολογία των υπολογιστών εστιάζεται και σε άλλα επίσης σημαντικά σημεία όπως στο πόσο αξιόπιστο μπορεί να είναι ένα σύστημα έστω και με την ύπαρξη λαθών. Κάποια λάθη μπορεί να γίνονται αισθητά στον χρήστη και άλλα όχι .Όμως κάποτε υπάρχει απειροελάχιστη πιθανότητα να υπάρξουν σφάλματα σε ένα σύστημα με καταστροφικά αποτελέσματα . Ένας κύριος λόγος που υπάρχει η τεχνική του «software fault injection» , όπου γίνονται έρευνες για μείωση των σφαλμάτων είναι ή αντιμετώπισή τους . Σε αυτή την έρευνα θα εστιαστούμε στις υφιστάμενες τεχνικές και εργαλεία που υπάρχουν για το software fault injection και εγώ με την σειρά μου θα πειραματιστώ με την τεχνολογία «ένεση σφάλμα».

Περιεχόμενα

Κεφάλαιο 1

Εισαγωγή

1.1 Σκοπός Μελέτης

Ένα σημαντικό πρόβλημα στην ανάπτυξη των ανεκτικών σε σφάλματα συστημάτων είναι ο ακριβής προσδιορισμός των ιδιοτήτων της αξιοπιστίας του συστήματος . Σε αντίθεση με την απόδοση , η οποία μπορεί να αξιολογηθεί με τη χρήση των προγραμμάτων αναφοράς , ο βαθμός ανοχής βλάβης και η αξιοπιστία του συστήματος δεν μπορεί να αξιολογηθεί με ένα τέτοιο τρόπο , δεδομένου ότι δεν έχουμε συχνά την πολυτέλεια τα συστήματα να τρέχουν για πολλά χρόνια για να δούμε συμπεριφορά τους κάτω από επιδράσεις σφαλμάτων.Η γενικά αποδεκτή λύση στο πρόβλημα αυτό είναι να εισφέρει τις επιπτώσεις των σφαλμάτων σε ένα μοντέλο προσομοίωσης ή πρότυπη υλοποίηση , και να γίνει παρατήρηση της συμπεριφορά του συστήματος κάτω από τα σφάλματα που γίνονται με ένεση[1] . Βλάβη ένεση σε μια προσομοίωση είναι πολύ ευέλικτη , αλλά δυστυχώς πάρα πολύ χρονοβόρα . Από την άλλη πλευρά , είναι πολύ πιο δύσκολο για την ένεση (δηλ. ρεαλιστική) σφαλμάτων σε ένα πρωτότυπο να είναι ακριβής, όμως οι επιδράσεις των σφαλμάτων στους επιχειρησιακούς κώδικες μπορεί εύκολα να παρατηρηθούν . Έχουν αναπτυχθεί ισχυρές τεχνικές για να γίνει η μίμηση σφαλμάτων υλικού με τη χρήση του λογισμικού . Αυτές ενσωματώθηκαν σε ένα ευέλικτο σύστημα έγχυσης σφάλματος. Πολλές εμπορικές εφαρμογές , όμως , χρησιμοποιούν ιδιόκτητους μεταγλωττιστές και εντοπιστές σφαλμάτων , για τα οποία ο πηγαίος κώδικας δεν είναι διαθέσιμος . Πρόσφατα αναπτύχτηκε ένα νέο σύστημα ένεσης σφάλματος ,το FIESTA , όπου προσανατολίζεται σε ενσωματωμένα συστήματα COTS[2] . Αυτό το σύστημα ένεση υλοποιείται στο πλαίσιο του λειτουργικού συστήματος σε πραγματικό χρόνο. Κίνητρο της μελέτης αυτής είναι να καθοριστεί κατά πόσο αξιόπιστο είναι ένα σύστημα υπό την παρουσία σφαλμάτων και πως το λειτουργικό σύστημα με διάφορες τεχνικές μπορεί να αντεπεξέλθει σε αυτά. Επίσης για μεγάλες εταιρίες ή επιχειρήσεις που έχουν συστήματα στα οποία θα ήταν ιδανικό να μην υπάρχουν σφάλματα ,όμως αυτό είναι αδύνατο. Για αυτό γίνονται μελέτες όπως αυτά τα συστήματα να αντεπεξέρχονται γρήγορα από τα σφάλματα

διότι μπορεί να υπάρξουν καταστροφικά αποτελέσματα. Για παράδειγμα η αποτυχία του Ariane 501 (πύραυλος)[3], η οποία προκλήθηκε από την πλήρη απώλεια του προσανατολισμού και πληροφορίες για το υψόμετρο, 37 δευτερόλεπτα μετά την έναρξη της κύριας ακολουθίας, έγινε ανάφλεξη του κινητήρα (30 δευτερόλεπτα μετά την απογείωσή του). Αυτή η απώλεια πληροφοριών οφειλόταν σε περιγραφή και λάθη σχεδιασμού στο λογισμικό του αδρανειακού σύστημα αναφοράς. Η SRI (Système de Référence Inertielle or Inertial Reference System.) εξαίρεση του λογισμικού προκλήθηκε κατά τη διάρκεια της εκτέλεσης μιας μετατροπή των δεδομένων από 64-bit κινητής υποδιαστολής σε 16-bit signed ακέραια τιμή. Ο αριθμός κινητής υποδιαστολής που μετατράπηκε είχε αξία μεγαλύτερη από ότι θα μπορούσε να εκπροσωπείται από ένα 16-bit signed integer και είναι ο λόγος που ο πύραυλος Ariane 501 μετά από 40 περίπου δευτερόλεπτα μετά την εκτόξευση του εξερράγει. Αυτό το καταστροφικό αποτέλεσμα και εφόσον ήταν το πρώτο ταξίδι του πυραύλου η δεκαετής προσπάθεια ανάπτυξης κόστισε 7.000.000.000 δολάρια. Ο πύραυλος κατέστρεψε και το φορτίο του που αποτινόταν στα 500 εκατομμύρια δολάρια.. Ο αριθμός ήταν μεγαλύτερος από 32.767, ο μεγαλύτερος αποθηκευμένος ακέραιος σε ένα 16-bit signed integer, και έτσι απέτυχε η μετατροπή[4]. Αυτό κόστισε οικονομικά στην εταιρία , αλλά σε άλλες περιπτώσεις ίσως κοστίσει και ζωές. Είναι σημαντικό να ελέγχεται η αξιοπιστία του συστήματος και να παίρνουν προληπτικά μέτρα.

1.2 Ένεση Σφάλματος

Η Ένεση Σφάλματος είναι μια τεχνική που καλύπτει ένα φάσμα τεχνικών για την πρόκληση βλαβών σε υπολογιστικά συστήματα για τη μέτρηση της ανταπόκρισης τους σε αυτές τις βλάβες [1][5]. Ειδικότερα , μπορεί να χρησιμοποιηθεί και στα δύο ηλεκτρονικά συστήματα υλικού και λογισμικού συστημάτων για μέτρηση της ανοχή λαθών του συστήματος. Για το υλικό, τα σφάλματα μπορεί να εγχέονται σε προσομοιώσεις του συστήματος , καθώς και την υλοποίησή του, τόσο σε pin ή εξωτερικό επίπεδο και , πρόσφατα , σε εσωτερικό επίπεδο για ορισμένα τσιπ. Για το λογισμικό , τα σφάλματα μπορούν να εγχέονται με προσομοιώσεις των συστημάτων λογισμικού , όπως στα κατανεμημένα συστήματα , ή στην λειτουργία των συστημάτων λογισμικού , σε επίπεδα όπως στα μητρώα της CPU ,στη μνήμη ,στο δίσκο ,στα δίκτυα ακόμη και σε καταχωρητές. Η ένεση σφάλματος χρησιμοποιείται καλύτερα ως μέσο για τη μέτρηση της ανοχής σφάλματος ή τη ευρωστία ενός συστήματος , ειδικά για την "άγχος δοκιμή" - όπου διεκπεραιώνεται για να διαπιστωθεί η απόδοση - ενός συστήματος που μπορεί να εμφανίσουν σφάλματα πολύ σπάνια για την κανονική δοκιμή. Αν και η θεωρία πίσω από την ένεση σφάλμα εξακολουθεί να αναπτύσσεται , οι μηχανισμοί είναι

πλήρως κατανοητοί. Κατά τον σχεδιασμό του συστήματος αυτή η τεχνική επιχειρεί να μετρηθεί ο βαθμός στον οποίο ελέγχει δηλαδή εάν ο σχεδιασμός του είναι ανθεκτικός σε βλάβες. Έγχυση βλάβης μπορεί να είναι μια χρήσιμη τεχνική για την ποσοτικοποίηση αυτής της πτυχής του σχεδιασμού.

Κεφάλαιο 2

Τίτλος Δεύτερου Κεφαλαίου

2.1 Λάθη και οι πηγές τους

Υπάρχουνε τρία είδη λαθών που κατηγοριοποιούνται σύμφωνα με τον χρόνο διάρκειας του σφάλματος στο σύστημα. [6].

Η πρώτη κατηγορία ονομάζεται **transient** όπου εμφανίζεται μια φορά το λάθος και μετά εξαφανίζεται. Οι αιτίες που προκαλούν αυτού του είδος σφάλματα μπορεί να είναι είτε ~~από~~ λόγω της θερμότητα είτε λόγω κάποιων ειδών ακτινοβολίας. Για παράδειγμα στη άλφα ακτινοβολία σωματιδίων υπάρχει περίπτωση ένα άλφα σωματίδιο να κτυπήσει ένα τρανζίστορ με αποτέλεσμα να διαταράξει το φορτίο του . Επίσης ένα άλλο παράδειγμα είναι οι ηλεκτρομαγνητικές παρεμβολές , όπου ηλεκτροφόρα σύρματα μπορούν να προκαλέσουν transient σφάλματα. Είναι ένα σφάλμα που δεν είναι πλέον παρών διότι έχει αποσυνδεθεί για σύντομο χρονικό διάστημα και στη συνέχεια αποκαταστάθηκε. Τυπικά παραδείγματα παροδικών σφαλμάτων περιλαμβάνουν:

- momentary tree contact
- bird or other animal contact
- lightning strike
- conductor clashing

Παροδικές βλάβες μπορεί να προκαλέσουν ζημιά τόσο ~~ακόμη~~ στη θέση του αρχικού σφάλματος ή αλλού στο δίκτυο, όπως ρεύμα σφάλματος.

Η δεύτερη κατηγορία ονομάζεται **permanent** ή αλλιώς **"hard faults"** όπως το όνομα υπονοεί, είναι το αποτέλεσμα της μόνιμης βλάβης η μόνωσης. Σε αυτή την περίπτωση, ο εξοπλισμός θα πρέπει να επισκευαστεί. Οι αιτίες που προκαλούν αυτού του είδος λάθη είναι οι ελαττωματικές συσκευές (π.χ φθαρμένο σύρμα) ή μπορεί λόγω θερμότητας να καεί κάποιο μέρος του υλικού .

Η Τρίτη και τελευταία κατηγορία ονομάζεται **Intermittent** , όπου είναι μια δυσλειτουργία της συσκευής ή του συστήματος που εμφανίζεται κατά διαστήματα , συνήθως ακανόνιστα , σε μια συσκευή ή σύστημα που λειτουργεί κανονικά σε άλλους χρόνους. Περιοδικά σφάλματα είναι κοινά σε όλους τους κλάδους της τεχνολογίας, συμπεριλαμβανομένου του λογισμικού ηλεκτρονικών υπολογιστών. Μια διαλείπουσα βλάβη προκαλείται από διάφορους παράγοντες που συμβάλλουν, μερικοί από τους οποίους μπορεί να είναι αποτελεσματικά τυχαία και συμβαίνουν ταυτόχρονα.. Όσο πιο πολύπλοκο είναι το σύστημα ή ο μηχανισμός που εμπλέκεται , τόσο μεγαλύτερη είναι η πιθανότητα ύπαρξης ενός **intermittent** σφάλματος.

Τι είδους λάθη μπορούν να υπάρξουν κατά την διάρκεια της εκτέλεσης του εκτελέσιμου αρχείου? Υπάρχουν πολλοί διαφορετικοί τύποι «run time errors» . Είναι πολύ πιο δύσκολο να απαλλαγούμε από τα compiler λάθη . Υπάρχουν ακόμη και ειδικά κομμάτια του λογισμικού που ονομάζονται «εντοπισμός σφαλμάτων» και έχουν γραφτεί για να βοηθήσουν τους ανθρώπους στο να ανακαλύψουν τα σφάλματα χρόνου εκτέλεσης. Λάθη εκτέλεσης του χρόνου μπορεί να είναι λογικά λάθη που προκαλούν το πρόγραμμά να παράγει την κακή απόδοση ή μπορεί να αποβούν σε μοιραία λάθη που θα συντρίψει το πρόγραμμά όταν τρέξει.[7] [8].

Κάποια σημαντικά μοιραία λάθη κατά την διάρκεια εκτέλεσης αναλύονται πιο κάτω:
Floating exception(core dumped): Αυτό προκαλείται από την διαίρεση με το μηδέν στο πρόγραμμά.

Segmentation fault (core dumped): Επίσης ονομάζεται παραβίαση ή απλά SEGV . Αυτό συμβαίνει όταν αποκτηθεί πρόσβαση σε μνήμη που δεν ανήκει στο πρόγραμμά ή δεν υπάρχει , ακόμη και η εικονική μνήμη έχει τα όρια της.

Logic errors (infinite loop) : Συμβαίνει όταν το πρόγραμμα δεν εκτελείτε όπως θα έπρεπε.

SIGBUS : Προκαλείται λόγω της μη εναρμόνισης με την λειτουργία της CPU (π.χ., προσπαθεί να διαβάσει ένα τύπο μεταβλητής από μια διεύθυνση που δεν είναι πολλαπλάσιο του 4)

2.1 Χρήσιμοι Ορισμοί και Μετρήσεις

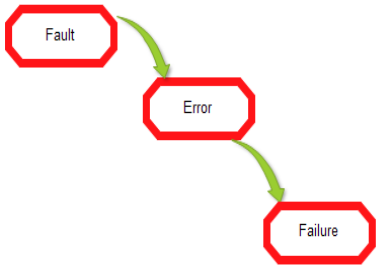
Πιο κάτω υπάρχουνε σημαντικοί ορισμοί για την ένεση σφάλματος στο λογισμικό.

Fault Tolerance: Είναι η ικανότητα ενός συστήματος να ανταποκριθεί σε μια απροσδόκητη αποτυχία υλικού ή λογισμικού (software ή hardware). Υπάρχουν πολλά επίπεδα ανοχής σε σφάλματα, η χαμηλότερη είναι η δυνατότητα να συνεχίσει τη λειτουργία σε περίπτωση διακοπής ρεύματος. [6][7][8]

Fault: Ένα fault είναι το φυσικό ελάττωμα όπως για παράδειγμα η φθορά των συρμάτων. Ένα fault έχει την δυνατότητα για την παραγωγή του error.

Error: Το Error είναι όταν ένα flip bit γίνεται από 1 σε 0 ή από 0 σε 1 λόγω του fault.

Failure: Η αιτία του failure είναι το error και μπορεί να διευκρινιστεί ο όρος του failure όταν το σύστημα αποκλίνει από τις προδιαγραφές του και είναι ορατό στον χρήστη ότι υπάρχει σφάλμα (βλέπε σχήμα 2.1) [6][7]



Σχήμα 2.1

Πως θα μπορούμε να αξιολογήσουμε την αξιοπιστία ενός συστήματος μετά από την ένεση σφάλματος χωρίς μετρήσεις. Πιο κάτω περιγράφω σημαντικές μετρήσεις που χρησιμοποιούνται για την αξιολόγηση της ένεσης σφάλματος στο λογισμικό:

Για να μελετήσουμε το software fault injection θα χρειαστούμε κάποιες μετρήσεις για να ελέγξουμε κατά πόσο αξιόπιστο μπορεί να είναι το faul ttolerance κάποιου συστήματος. [8][9][10]

Είδη Μετρήσεων

MTBF, MTTR, MTTF και **FIT** είναι όροι αξιοπιστίας και βασίζονται σε μεθόδους και σε διαδικασίες για τον κύκλο ζωής για ένα προϊόν (προβλέψεις).

- MTBF(Mean Time Between Failures)
- MTTR(Mean Time To Repair)
- MTTF(Mean Time to Failure)
- FIT(Failure In Time)

MTTF : Μετρά τον μέσο χρόνο μέχρι την πρώτη αποτυχία και αυτή η μέτρηση χρησιμοποιείται για να διαπιστώσουμε κατά πόσο ένα σύστημα είναι αξιόπιστο. Είναι ένα βασικό μέτρο της αξιοπιστίας των συστημάτων. Είναι ο μέσος αναμενόμενος χρόνος μέχρι την πρώτη αποτυχία του συστήματος . Το MTTF είναι μια στατιστική αξία και προορίζεται να είναι ο μέσος όρος για μια μακρά χρονική περίοδο και για ένα μεγάλο αριθμό μονάδων.

MTBF : Είναι ένας όρος που χρησιμοποιείται για την αξιοπιστία στο να παράσχει το ποσό των αποτυχιών ανά εκατομμύριο ωρών για ένα σύστημα. Αυτή είναι η πιο κοινή έρευνα σχετικά με τη διάρκεια ζωής ενός συστήματος .

MTTR : Είναι ουσιαστικά ο χρόνος που απαιτείται για να επισκευάσει ένα σφάλμα του συστήματος .

AVF (**Architecture Vulnerability Factor**): Τυπικά υπολογίζονται ανά βασική λειτουργική μονάδα. Λαμβάνει υπόψη την επίδραση των πλεοναζόντων εντολών, και χαρακτηριστικά της λειτουργικής μονάδας. Η AVF για το branch predictor είναι μηδέν. Για τα lunches είναι περίπου 50%.. Αυτό ποικίλλει ευρέως από 10% έως 70% για όλες τις άλλες μονάδες. Πώς υπολογίζεται το AVF; Για τις μονάδες, όπως το προγνωστικό υποκατάστημα, μπορεί να υπολογιστεί θεωρητικά .Σε αντίθετη περίπτωση, εκτιμάται με προφίλ και τρέχει για ένα σύνολο σημείων αναφοράς

Software Reliability: Πιο σημαντικότερη και κυριότερη μέτρηση για την ένεση σφάλματος ~~και~~ υλικού και λογισμικού[11][12][13]. Είναι το "Κλειδί" για την ποιότητα λογισμικού. και κατηγοριοποιείται σε τρία μέρη:

1. Λογισμικό μοντελοποίησης αξιοπιστίας
2. Μέτρηση Αξιοπιστίας λογισμικού
3. Βελτίωση Αξιοπιστίας Λογισμικού

Για καλύτερη κατανόηση πιο κάτω επεξηγούνται τα τρία αυτά μέρη:

- Λογισμικό μοντελοποίησης αξιοπιστίας

  Έχει αναπτυχθεί σε σημείο που ουσιαστικά τα αποτελέσματα μπορούν να ληφθούν με εφαρμογή κατάλληλων μοντέλων για κάποιο πρόβλημα[9][10]. Υπάρχουνε πολλά μοντέλα , αλλά υπάρχει ένα ενιαίο μοντέλο που μπορεί να συλλάβει μια απαραίτητη

ποσότητα των χαρακτηριστικών του λογισμικού . Προϋποθέσεις και αφαιρέσεις πρέπει να γίνουν για να απλοποιηθεί το πρόβλημα. Όμως είναι λογικό ότι δεν υπάρχει ένα ενιαίο μοντέλο που να είναι και καθολικό (universal) για όλες τις καταστάσεις.

Υπάρχουνε 2 υποκατηγορίες

1.1 Prediction Model

Χρησιμοποιεί ιστορικά τα δεδομένα. Αναλύονται τα προηγούμενα δεδομένα και γίνονται ορισμένες παρατηρήσεις. Έγιναν συνήθως πριν την ανάπτυξη και την τακτική φάσεις δοκιμής. Το μοντέλο ακολουθεί τη φάση της σύλληψης και της κατηγοριοποίησης από το μελλοντικό χρόνο.

1.2 Estimation Model

Χρησιμοποιούνται τα τρέχοντα δεδομένα από την τρέχουσα προσπάθεια ανάπτυξης λογισμικού και δεν χρησιμοποιεί τις σχεδιαστικές φάσεις ανάπτυξης .Μπορεί να εκτιμηθεί οποιαδήποτε στιγμή.

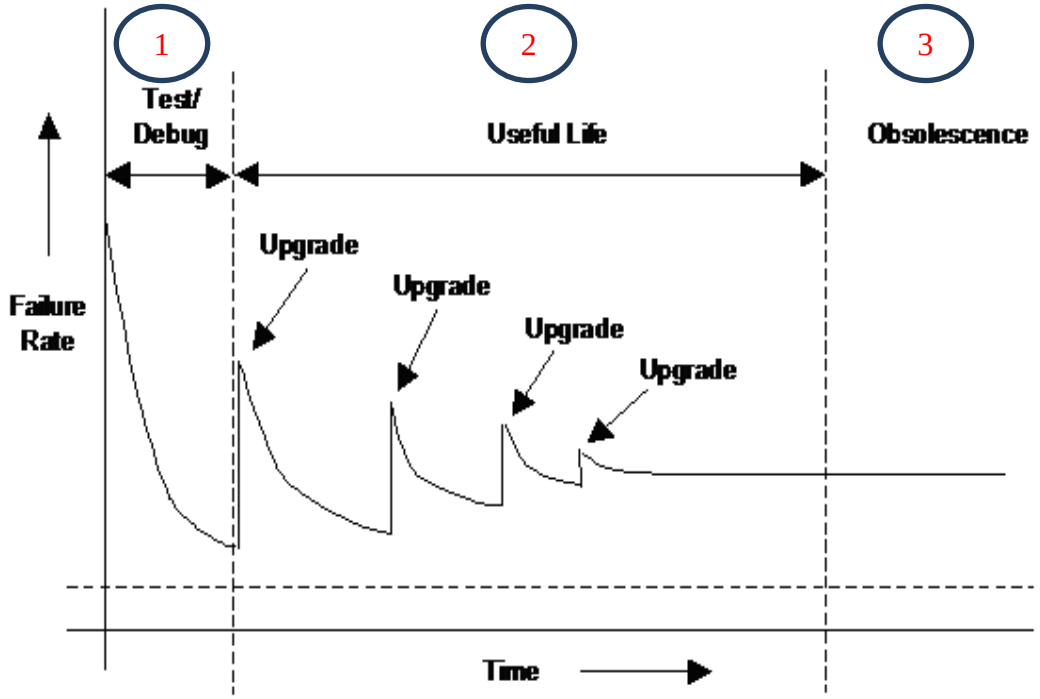
- Μέτρηση Αξιοπιστίας λογισμικού

  Είναι αφελής εφόσον η μέτρηση αξιοπιστίας λογισμικού είναι πολύ διαδεδομένη στο λογισμικό όπως και σε άλλους τομείς του engineering field, «πόσο καλό θα είναι ένα λογισμικό συνολικά , ή ποσοτικά». Ακόμη δεν υπάρχει μια καλή και ενδεικτική απάντηση σε αυτό το ερώτημα . Η αξιοπιστία του λογισμικού δεν μπορεί να μετρηθεί άμεσα , έτσι ώστε οι άλλοι παράγοντες που σχετίζονται με την μέτρηση για την εκτίμηση της αξιοπιστίας του λογισμικού να είναι συγκρίσιμοι μεταξύ των προϊόντων - συστημάτων ολικά. Οι Αναπτυξιακές διαδικασίες faults και failures που διαπιστωθήκαν είναι όλοι παράγοντες που σχετίζονται με την αξιοπιστία του λογισμικού.

- Βελτίωση Αξιοπιστίας Λογισμικού

  Είναι αρκετά δύσκολο εφόσον η δυσκολία του προβλήματος προκύπτει από ανεπαρκή κατανόηση της αξιοπιστίας του λογισμικού και σε γενικές γραμμές , τα χαρακτηριστικά του λογισμικού. Μέχρι τώρα δεν υπάρχει καλός τρόπος για να «κατακτηθεί» το πρόβλημα της πολυπλοκότητας του λογισμικού. Πλήρης δοκιμή μιας μέτριας πολύπλοκης μονάδας λογισμικού είναι ανέφικτο. Ελάττωμα χωρίς λογισμικό δεν μπορεί να διασφαλιστεί. Ρεαλιστικός περιορισμός του χρόνου και του προϋπολογισμού περιορίζει σημαντικά την προσπάθεια που καταβάλλεται για την βελτίωση της αξιοπιστίας του λογισμικού.

Η αξιοπιστία του λογισμικού χωρίζεται σε τρεις φάσεις όπου , κατά την πρώτη φάση (1), δηλαδή κατά την δοκιμή και την ολοκλήρωση θα υπάρξει , όπως είναι φυσικό, υψηλός αριθμός σφαλμάτων, αλλά μετά την αφαίρεση των σφαλμάτων, υπάρχει μόνο ένας μικρός αριθμός σφαλμάτων και αυτή η διαδικασία της απομάκρυνσης των σφαλμάτων συνεχίζεται σε βραδύτερο ρυθμό (2). Τα προϊόντα λογισμικού δεν θα φθείρονται με το χρόνο και τη χρήση, (3) αλλά μπορεί να είναι ξεπερασμένα σε μεταγενέστερο στάδιο. (Για καλύτερη κατανόηση βλέπε σχήμα 2.1)[11][14]



Σχήμα 2.1

Λογικό είναι οποιοσδήποτε να διερωτηθεί, γιατί χρειαζόμαστε αξιοπιστία λογισμικού και αξιοπιστία υλικού. Όλοι μας είμαστε χρήστες , αγοραστές μικρών ή μεγάλων συστημάτων και είναι φυσικό ότι η ποιότητα του συστήματος παίζει σημαντικό ρόλο σε όλους μας .Και τι ακριβώς εννοούμε ποιότητα συστήματος? Αυτό που χρειαζόμαστε ως χρήστες είναι την ποιότητα του υλικού και της ποιότητας-λογισμικού. Γνωρίζουμε ότι η ποιότητα του υλικού είναι σταθερά υψηλή- ειδικά σε μικρές συσκευές (smartphones , tablets, laptops). Άρα επικεντρωνόμαστε στην ποιότητα του λογισμικού. Υπάρχουνε διάφοροι τρόποι να μετρηθεί η ποιότητα του λογισμικού και αυτές οι μετρήσεις έχουν ως στόχο να χρησιμοποιούνται για τη βελτίωση της αξιοπιστίας του συστήματος .Οι διάφοροι τύποι των μετρήσεων του λογισμικού που χρησιμοποιούνται για βελτίωση της αξιοπιστίας είναι οι εξής:

1. Μετρήσεις απαιτήσεων Αξιοπιστίας
2. Μετρήσεις κώδικα και σχεδιασμό Αξιοπιστίας
3. Μετρήσεις για πειράματα Αξιοπιστίας

Πριν περιγράψω τους διάφορους τύπους μετρήσεων του λογισμικού να τονίσω πως μπορεί να αυξηθεί η αξιοπιστία κάνοντας μετρήσεις. Για οποιοδήποτε προϊόν πρέπει η ποιότητα του λογισμικού να περάσει από όλα τα στάδια του κύκλου ζωής. Ποια είναι αυτά τα στάδια?

1) Φάση Απαίτησης

   Σε αυτό το στάδιο πρέπει να εξασφαλίζεται εάν το προϊόν πληρεί τις προδιαγραφές του τελικού προϊόντος ή όχι.

2) Φάση κωδικοποίησης

   Η πιο σημαντική φάση εφόσον πρέπει να διασφαλιστεί ότι ο παραγόμενος κώδικας του συστήματος μπορεί να υποστηρίξει την συντήρηση για αποφυγή τυχόν επιπρόσθετων σφαλμάτων.

3) Φάση Πειράματος

   Σε αυτή την φάση γίνεται μια βεβαίωση ότι όλες οι απαιτήσεις που καθοριστήκαν στο πρώτο στάδιο είναι ικανοποιητικές ή όχι .

Εφόσον αναφέραμε πως μπορεί να αυξηθεί η αξιοπιστία του συστήματος πλέον θα ορίσουμε και πως θα βελτιωθεί η αξιοπιστία του συστήματος σύμφωνα με τις μετρήσεις που ανάφερα προηγουμένως .[14]

1. Μετρήσεις απαιτήσεων Αξιοπιστίας

Τι θα διαθέτει το λογισμικό και τι πρέπει να περιέχει.? Δηλαδή ποιές θα είναι οι απαιτήσεις οι οποίες πρέπει να περιέχουν έγκυρες δομές για να αποφευχθεί η απώλεια πολύτιμων πληροφοριών. Άρα στην ουσία γίνεται ορισμός και κατηγοριοποίησης τρόπων αστοχίας του λογισμικού. Γίνεται προσδιορισμός αναγκών αξιοπιστίας πελατών - χρηστών και τέλος καθορισμός στόχων για την αξιοπιστία του προϊόντος.

Ο στόχος «Αξιοπιστία» αποδεικνύεται από την εξέταση των αναγκών των πελατών , καθώς και τους περιορισμούς της τεχνολογίας μηχανικής λογισμικού , τις δυνατότητες των προγραμματιστών , καθώς και άλλων περιορισμων , όπως το οργανωτικό μοντέλο των επιχειρήσεων ~~και~~ , το κόστος ανάπτυξης και των χρονοδιαγραμμάτων .

2. Μετρήσεις κώδικα και σχεδιασμό Αξιοπιστίας

Οι παράγοντες της ποιότητας που υπάρχουν στο σχεδιασμό και την κωδικοποίηση είναι η πολυπλοκότητα και το μέγεθος και modularity. Άρα γίνεται ορισμός της λειτουργίας και της αποτίμησης στοιχείων αξιοπιστίας του λογισμικού που θα επαναχρησιμοποιηθούν. Παρακολούθηση και διαχείριση των διάφορων σφαλμάτων και διαδικασία να ανακτηθεί το λογισμικό όταν υπάρξουν σφάλματα

3. . Μετρήσεις για πειράματα Αξιοπιστίας

Θα πρέπει να γίνονται δοκιμές για επιμέρους συστήματα και εξαρτήματα με τέτοιο τρόπο ώστε η συνολική αντικειμενική αξιοπιστία του συστήματος να επιτυγχάνεται Δηλαδή να γίνεται ανοχή σε κάποια σφάλματα , είτε να διαχειρίζονται τα διάφορα faults έτσι ώστε να μην υπάρχουνε αποτυχίες στο λογισμικό. Άρα μέσα από πολλές και διάφορες δοκιμές καθορίζουν πως θα επικαλύψουν ένα σφάλμα ή αν θα το ανεχτούν. Ιδιαίτερη προσοχή πρέπει να δοθεί στην παραγωγή των περιπτώσεων δοκιμών που αντιμετωπίζουν κρίσιμα και ειδικά θέματα .

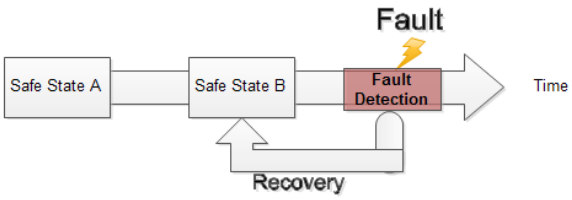
2.3 Τρόποι Για Ανάκτηση από Σφάλματα

Τώρα το ερώτημα είναι πώς μπορεί το σύστημα να ανταποκριθεί από τα σφάλματα και εάν είναι δυνατόν να συμβεί αυτό. Για την ανάκτηση σφάλματος, λαμβάνονται διορθωτικά μέτρα κατά την εμφάνιση ενός σφάλματος. Ανάκτηση σφάλματος γενικά κατηγοριοποιούνται είτε ως προς τα πίσω αποκατάστασης σφάλματος (backward error recovery)ή προς τα εμπρός αποκατάσταση σφαλμάτων (forward error recovery).Ανάκτηση σφάλματος είναι μια πτυχή του fault tolerance.[15]

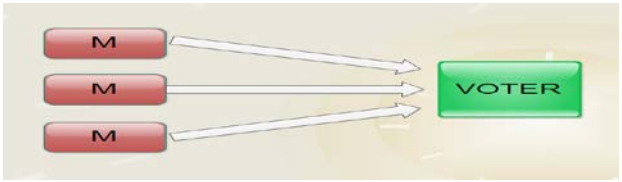
Backward error recovery συνεπάγεται την επιστροφή του συστήματος σε ασφαλή κατάσταση, κάτι το οποίο δεν είναι πάντα εφικτό με συστήματα πραγματικού χρόνου. Απαιτεί κάποιο είδος σημείων ελέγχου ή να αποθηκεύουν την κατάσταση που βρίσκεται το σύστημα όταν λειτουργά σωστά , checkpoints (βλέπε σχήμα 2.2). Πάραυτα σε ορισμένες περιπτώσεις, να προκαλέσει άλλα προβλήματα μέσω της χρήσης επίκαιρων πληροφοριών (outdatedinformation). Μπορεί επίσης να εμφανιστεί μέσω ματαίωσης λειτουργιών που έχουν παράγει ένα λάθος.

Forward error recovey περιλαμβάνει τη διόρθωση του σφάλματος, χωρίς να καταφεύγουν σε αντιστροφή προηγούμενων εργασιών .Για τα συστήματα πραγματικού χρόνου, όπως είναι τα συστήματα ελέγχου πτήσης, εμπρός αποκατάσταση σφαλμάτων μπορεί να είναι η μόνη επιλογή. Για να γίνει πιο κατανοητό ένα παράδειγμα forward error recovery θεωρείται μια τεχνική που ονομάζεται Triple Error Redundancy(σχέδιο 2.2) όπου υπάρχουνε 3 modules που εκτελούν την ίδια διαδικασία και το majority voting system (voter) θα παράξει ένα αποτέλεσμα σαν έξοδο. Σε περίπτωση που ένα από τα συστήματα αποτύχει τα άλλα δύο θα το διορθώσουν και θα καταλήξουν στο masking του fail (κάλυψη του σφάλματος). Είναι αξιόπιστο και η έννοια του Triple Error Redudancy (TMR) μπορεί να εφαρμοστεί σε πολλές

μορφές πλεονασμού όπως software redundancy σε μορφή N-version programming. Αν αποτύχει ωστόσο ο voter θα αποτύχει όλο το σύστημα (βλέπε σχήμα 2.3).



Σχήμα 2.2



Σχήμα 2.3

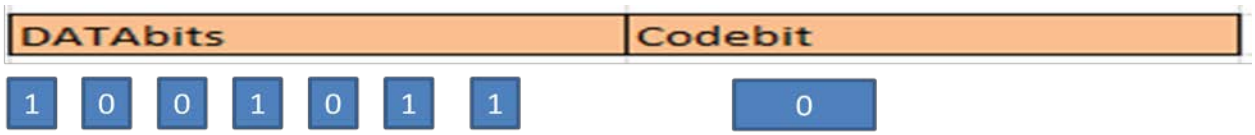
Σημαντικό να αναφέρω πως εντοπίζεται από την μνήμη ότι ένα block έχει λάθος bits.

Διόρθωσης σφαλμάτων μνήμης κωδικού (ECC) είναι ένας τύπος αποθήκευσης δεδομένων υπολογιστή που μπορεί να ανιχνεύσει και να διορθώσει τα πιο κοινά είδη της εσωτερικής διαφθοράς δεδομένων[16]. ECC μνήμη χρησιμοποιείται στους περισσότερους υπολογιστές, όπου η διαφθορά των δεδομένων δεν μπορεί να γίνει ανεκτή σε καμία περίπτωση.

Η ECC μνήμη διατηρεί ένα σύστημα μνήμης ανοσία σε μόνο-bit λάθη: τα δεδομένα που διαβάζονται από κάθε λέξη είναι πάντα το ίδιο με τα δεδομένα που είχαν γραφτεί σε αυτό, ακόμη και αν έχουν ένα ή περισσότερα bit πράγματι αποθηκεύεται έχουν γυρίσει σε λάθος κατάσταση . Ποιά είναι όμως τα πλεονεκτήματα του ECC?

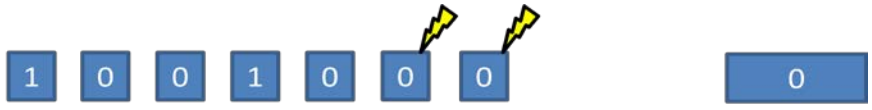
Το ECC μειώνει επίσης τον αριθμό των σφαλμάτων , ιδιαίτερα σε server εφαρμογές multi-user συστήματα με μέγιστη διαθεσιμότητα . Οι περισσότερες μητρικές πλακέτες και πολλοί επεξεργαστές για λιγότερο κρίσιμες αιτήσεις δεν έχουν σχεδιαστεί για την υποστήριξη ECC , για οικονομία .Η ECC μνήμη κοστίζει περισσότερο , καθώς κάθε τράπεζα απαιτεί 9 τσιπ μνήμης σε σύγκριση με το 8 για τη μνήμη μη ECC . Σε ορισμένες περιπτώσεις, ο δείκτης τιμών μειώνει σε 9/8 , για παράδειγμα , στις 11/30/2008 , στις Crucial.com , ένα EKK CL = 5 unbuffered 2GB DDR2 - 667 DIMM κόστος $ 30, ενώ το αντίστοιχο τμήμα non-ECC κοστίζουν 28 δολάρια , μια διαφορά 1/15 , ωστόσο, ορισμένες μονάδες ECC με διπλάσιο κόστος ως μη ECC ισοδύναμά τους [Crucial CT12872Z40B και CT12864Z40B Ιαν 2009,]) . [αρχική έρευνα;] ECC - στήριξη motherboards , chipsets , και οι μεταποιητές μπορεί επίσης να είναι πιο ακριβά..Το ECC μπορεί να μειώσει την απόδοση της μνήμης κατά περίπου 2-3 τοις εκατό σε μερικά συστήματα, ανάλογα με την εφαρμογή και την υλοποίηση , λόγω του πρόσθετου χρόνου που απαιτείται για την ECC ελεγκτές μνήμης για γίνει έλεγχος για λάθη . Ωστόσο , τα σύγχρονα συστήματα ενσωματώνουν δοκιμές ECC στη CPU , δεν δημιουργούν καμία πρόσθετη καθυστέρηση στη μνήμη προσπελάσεις . Τελικά , υπάρχει ένα trade -off του

μικρού αριθμού των περιπτώσεων απώλειας της ακεραιότητας και συντρίβεται κατά της πληρωμής ένα υψηλότερο ασφάλιστρο δεδομένων .Υπάρχουνε επίσης και άλλοι τρόποι για να διαπιστωθεί εάν κάποιο error στο λογισμικό. Υπάρχουν τεχνικές που ονομάζονται Error Detection και Error Correction. Μια τεχνική από αυτές ονομάζεται parity check[17]. Το parity check είναι η διαδικασία που εξασφαλίζει την ακριβή μετάδοση δεδομένων μεταξύ των κόμβων κατά την επικοινωνία. Ένα bit ισοτιμίας επισυνάπτεται στα αρχικά bits δεδομένων για να δημιουργήσετε ένα μονό ή ζυγό αριθμό bit. ο αριθμός των bits με μία τιμή. Η πηγή μεταδίδει τα δεδομένα μέσω ενός συνδέσμου, και τα bits και ελέχθησαν στον τόπο προορισμού[16]. Δεδομένων θεωρείται ακριβής, εάν ο αριθμός των δυαδικών ψηφίων (άρτιο ή περιττό) ταιριάζει με τον αριθμό που μεταδίδεται από την πηγή. Εάν έχουμε ένα block από 32bits , στο odd parity check μετριούνται πόσα bit είναι ο αριθμός 1, εάν ο αριθμός των bits είναι περιττός αριθμός τότε το parity bit θα είναι 1 αλλιώς θα είναι 0. Για παράδειγμα εάν έχουμε τα πιο κάτω bits σε ένα block:



Έστω εισαχθεί ένα λάθος και γίνει bit flip σε ένα από τα bits:

Όπως παρατηρούμε το parity check αλλάζει και αυτό μπορεί να προκαλέσει error και για κάποια πιθανότητα failure. Όμως σε περίπτωση που γίνει ξανά ένα bit flip το parity check θα είναι το εξής:



Όπως θα παρατηρήσετε το parity check είναι και πάλι μηδέν και δεν μπορεί να εντοπιστεί αν υπήρχε λάθος. Φυσικά είναι μικροσκοπική η πιθανότητα να υπάρξουν δύο λάθη σε ένα block αλλά αυτό αποδεικνύει ότι αυτή η τεχνική error detection είναι λιγότερο αξιόπιστη και γίνεται μόνο για ανίχνευση λαθών εφόσον δεν μπορεί να γνωρίζει πιο συγκεκριμένο bit έχει αλλάξει.Μια άλλη τεχνική που μπορεί να κάνει error detection και error correction ονομάζεται hamming code. Μπορεί να ανιχνεύσει μέχρι και δύο λάθη -bit ή τη διόρθωση σφαλμάτων ενός bit χωρίς ανίχνευση των εναπομενόντων σφαλμάτων . Αντίθετα , το odd parity check που δεν μπορεί να διορθώσει τα λάθη του , και μπορεί να ανιχνεύσει μόνο ένα μονό αριθμό bits σε λάθος . Κώδικες Hamming είναι κώδικες , που μπορεί να επιτύχει την υψηλότερη δυνατή τιμή για κώδικες με μήκος το μπλοκ τους και την ελάχιστη απόσταση

3[16][17].Σε μαθηματικούς όρους , κώδικες Hamming είναι μια κατηγορία δυαδικών γραμμικών κωδίκων. Για κάθε ακέραιο r ≥ 2 υπάρχει ένας κώδικας με μήκος μπλοκ n = 2 ^ r-1 και το μήνυμα μήκους k = 2 ^ r-r - 1 . Επομένως ο ρυθμός των κωδικών Hamming είναι R = k / n = 1 -R / (2 ^ r- 1) , η οποία είναι υψηλότερη δυνατή για κώδικες με ελάχιστη απόσταση 3 (δηλαδή ο ελάχιστος αριθμός των αλλαγών bit που απαιτούνται για να πάει από οποιαδήποτε λέξη κώδικα σε οποιαδήποτε άλλη κώδικα λέξη είναι 3) και το μήκος του μπλοκ 2 ^ r - 1 . Η μήτρα ελέγχου ισοτιμίας ενός κώδικα Hamming κατασκευάζεται με όλες τις στήλες επιχείρηση μήκους r που είναι μη - μηδέν

Η μήτρα ελέγχου ισοτιμίας έχει την ιδιότητα ότι κάθε δύο στήλες ανά δύο γραμμικά ανεξάρτητες .Λόγω του περιορισμένου πλεονασμού που κωδικοποιεί Hamming προσθέτονται τα δεδομένα , μπορούν μόνο να εντοπίζουν και να διορθώνουν τα λάθη , όταν το ποσοστό σφάλματος είναι χαμηλό . Αυτή είναι η περίπτωση στη μνήμη του υπολογιστή (μνήμη ECC) , όπου τα σφάλματα bit είναι εξαιρετικά σπάνια και οι κωδικοί Hamming χρησιμοποιείται ευρέως . Σε αυτό το πλαίσιο , η επέκταση κώδικα Hamming έχει ένα επιπλέον bit ισοτιμίας χρησιμοποιείται συχνά . Κώδικες Hamming Extended επιτυγχάνεται απόσταση Hamming 4 , η οποία επιτρέπει στον αποκωδικοποιητή να γίνει διάκριση μεταξύ , όταν συνέβη το πολύ ένα λάθος bit και όταν συνέβη δύο σφάλματα bit .

Κεφάλαιο 3

Fault Injection

3.1 Σκοπός

Ο στόχος των πειραμάτων ένεσης σφάλματος είναι για να αξιολογεί στατιστικά αξιολογεί με ακρίβεια όσο το δυνατόν την κάλυψη σφαλμάτων. Ιδανικά θα μπορούσε κάποιος να υπολογίσει την κάλυψη σφαλμάτων με την παρατήρηση του συστήματος κατά τη διάρκεια της κανονικής λειτουργίας του για ένα άπειρο χρόνο και τον υπολογισμό του ορίου μεταξύ του αριθμού των φορών ότι ο μηχανισμός κάλυψης σφαλμάτων καλύπτει ένα σφάλμα και το συνολικό αριθμό των σφαλμάτων που προέκυψε στο σύστημα. Καθώς η ασφάλεια είναι αδιαπραγμάτευτη προϋπόθεση για τα λογισμικά συστήματα , η ένεση σφάλματος στα λογισμικά προσπαθεί να εξασφαλιστεί ότι δεν θα πρέπει να οδηγήσει σε σοβαρούς κινδύνους η ύπαρξη σφαλμάτων , ειδικά σε συστήματα που δεν είναι ανεκτικά τα σφάλματα διότι υπάρχει κίνδυνος για καταστροφικά αποτελέσματα .

3.2 Ένεσης Σφάλματος στο Λογισμικό

Είναι ουσιαστικά η ανάλυση της ανταπόκρισης του συστήματος σε εξαιρετικές περιπτώσεις (υπό την παρουσία λαθών). Δηλαδή γίνεται εισαγωγή μη φυσιολογικών καταστάσεων κατά την διάρκεια της κανονικής λειτουργίας του συστήματος έτσι ώστε με την παρακολούθηση των αποτελεσμάτων να γίνεται διεξαγωγή συμπερασμάτων[2]. Ο λόγος της χρήσης σχετίζεται άμεσα με τις τεχνικές που αναπτυχθήκαν έτσι ώστε να μειωθούν τα σφάλματα που εμφανίζονταν στα συστήματα. Κατά την ανάπτυξη της τεχνολογίας έχοντας ως στόχο την μείωση ενέργειας, την αύξηση της απόδοσης κτλ , υπήρχε η τάση οι διαστάσεις των τρανζίστορ να μειώνονται σημαντικά για μεγαλύτερη απόδοση πάραυτα εμφανίστηκαν σφάλματα στα συστήματα λόγω της σμίκρυνσης τους. Στα μεγάλα υπολογιστικά συστήματα συνήθως δεν γίνεται ανοχή των σφαλμάτων διότι μπορεί να έχουν σοβαρές επιπτώσεις. Υπάρχουνε διάφορες χρήσεις του software fault injectionn πιο κυριότερη και εμφανές είναι για την εύρεση ελαττωμάτων στο λογισμικό και γίνονται διάφορες δοκιμές για πόσο αυθεντικό είναι το σύστημα υπό την παρουσία των σφαλμάτων στην κανονική ροή του

συστήματος, δηλαδή ποια η συμπεριφορά του συστήματος πως θα αντιμετωπιστούν τα σφάλματα κτλ .[19] Επίσης αναμενόμενο είναι ότι γίνεται και έλεγχος ασφάλειας αλλά και αξιολόγηση της σε σχέση με άλλα συστήματα. Ποιοί είναι όμως οι περιορισμοί της ένεσης σφάλματος στο λογισμικό ? Για παράδειγμα, ορισμένα εργαλεία SWFI απαιτούν code instrumentation. Αυτό μπορεί να προκαλέσει περιττή επιβάρυνση στο σύστημα και κατά συνέπεια της επιδόσεις του συστήματος .Ως αποτέλεσμα, ο κώδικας που εκτελείται κατά τη διάρκεια της δοκιμής δεν θα είναι κατ 'ανάγκη ο ίδιος κώδικα που θα έτρεχε σε ένα ρεαλιστικό περιβάλλον. Υπάρχει περίπτωση να μιμηθεί λανθάνουσα κατάσταση. Ως εκ τούτου, οι προγραμματιστές λογισμικού και οι δοκιμαστές δεν θα πρέπει να βασίζεστε αποκλειστικά στο Software Fault Injection για ακριβές έλεγχο της λειτουργίας του συστήματος. Επιπλέον, το software fault injection έχει επικεντρωθεί στον καθορισμό του λογισμικού πως να συμπεριφέρεται με την παρουσία μιας σειράς των σφαλμάτων που παράγονται σε μη ιδανικό περιβάλλον.

Για το software fault injection η εισχώρηση του σφάλματος γίνεται με τους εξή δύο τρόπους:

- Compile-time
- Run time

Στο compile- time injection σημαίνει ότι το σφάλμα είναι αυστηρά κωδικοποιημένο στον πηγαίο κώδικα[19]. Με αυτό τον τρόπο μπορεί κανείς να μιμηθεί το υλικό, το λογισμικό και παροδικές βλάβες. Το injection μπορεί να προκαλέσει ένα λανθασμένη προγράμματος οποίο όταν το σύστημα το εκτελεί προκαλεί ένα σφάλμα. Αυτή η μέθοδος ΔΕΝ απαιτεί κανένα άλλο λογισμικό και δεν προκαλεί διατάραξη κατά τη διάρκεια της εκτέλεσης .Η εφαρμογή είναι πολύ απλή, αλλά δεν επιτρέπει την έγχυση των βλαβών καθώς το πρόγραμμα τρέχει την εργασία του. Όπως εξηγήθηκε, αυτή η μέθοδος είναι απλή, αλλά έχει τα μειονεκτήματά του, όπως να καθυστερεί πολύ και αυτός ο χρόνος οφείλεται σε διάφορους εκτελέσεις που απαιτούνται για διαφορετικές εισόδους και αστοχίες. Η μέθοδος αυτή ως εκ τούτου σπάνια χρησιμοποιείται σε εμπορικά έργα.

Ενώ στο runtime injection είναι σε θέση να εισάξει σφάλματα κατά τη διάρκεια της εκτέλεσης ,ένας μηχανισμός είναι που απαιτείται για να ενεργοποιήσουν το fault injection[19][20]. Η ενεργοποίηση του μηχανισμού αυτού μπορεί να είναι

•Ένα time-out, είναι μια από τις απλούστερες τεχνικές, περιμένοντας ένα προκαθορισμένου χρόνου πριν από το fault injection. Πιο ακριβέστερα το time-out προκαλεί μία διακοπή. Ο χρονοδιακόπτης μπορεί να είναι του λογισμικού ή από τη φύση του υλικού, αλλά και για ένα χρονόμετρο υλικού για να λειτουργήσει πρέπει να συνδέεται με διάνυσμα διακοπής του χειριστή του συστήματος.

Αυτοί οι τύποι των βλαβών είναι καλοί στην προσομοίωση απροστάτευτο από βλάβες επειδή συμβαίνουν σε σχέση μετ ο χρόνο και όχι κάποιο είδος της εκδήλωσης του

συστήματος. Αυτό το καθιστά πρακτικό να μιμηθεί παροδικά και διαλείπουσα σφάλματα υλικού.

• Exception/trap, προκαλεί το injection να γίνει σε ένα συγκεκριμένο γεγονός ή σε συνθήκες που επικρατούν. Ένα exception του υλικού ή παγίδα λογισμικό θα μεταφέρει τον έλεγχο στο fault injector Το λογισμικό παγίδα μπορεί να τεθεί σε ένα πρόγραμμα που να αποτελεί αντικείμενο επίκλησης, όταν μια συγκεκριμένη εντολή θα εκτελείται. Αυτό θα δημιουργήσει μια διακοπή που μεταφέρει τον έλεγχο στο χειριστή διακοπών.
•Dynamic Code Insertion,είναι η τεχνική που εισάγει οδηγίες του κώδικα, ώστε το fault injection μπορεί να γίνει πριν από ορισμένες οδηγίες. Αυτό μοιάζει πολύ με code-mutation εκτός από το ότι code insertion μπορεί εισάγει σφάλματα κατά το run time και οι εντολές Να προστεθούν στον κώδικα.

Ποιά είναι τα οφέλη όμως της ένεσης σφάλματος στο λογισμικό? Αυτή η τεχνική μπορεί να στοχεύει σε εφαρμογές και λειτουργικά συστήματα, που είναι δύσκολο να γίνει χρησιμοποιώντας ένεση βλάβη υλικού[6]. Τα πειράματα μπορούν να εκτελεστούν σε σχεδόν πραγματικό χρόνο, επιτρέποντας για τη δυνατότητα διεξαγωγής ενός μεγάλου αριθμού σφάλματος πειράματα έγχυσης. Το τρέξιμο των πειραμάτων για ένεση σφάλμα σχετικά με το υλικό που εκτελεί το λογισμικό έχει το πλεονέκτημα συμπεριλαμβανομένων τυχόν σχεδιαστικά σφάλματα που μπορεί να είναι παρόντες στην πραγματική υλικού και λογισμικού σχεδιασμό. Δεν απαιτείται κανένα ειδικό εξοπλισμό ούτε ειδικό υλικό χαμηλή πολυπλοκότητα, χαμηλή ανάπτυξη και χαμηλό κόστος υλοποίησης. Μπορεί να επεκταθεί για νέες κατηγορίες βλαβών. Παρόλα αυτά τα μειονεκτήματα είναι ότι δεν υπάρχει περιορισμένο σύνολο των στιγμών της ένεσης: Σε επίπεδο οδηγίες συναρμολόγησης, μόνο. Δεν μπορεί να εισφέρει βλάβες σε περιοχές που είναι σε προσιτή στο λογισμικό. Απαιτεί τροποποίηση του πηγαίου κώδικα για να υποστηρίξει την ένεση σφάλματος, πράγμα που σημαίνει ότι ο κώδικας που εκτελεί κατά τη διάρκεια του πειράματος βλάβης δεν είναι το ίδιο κωδικό που θα τρέξει στο πεδίο[21]. Μια έρευνα σχετικά με τεχνικές Fault Injection Περιορισμένη παρατήρησης και ελέγχου. Στην καλύτερη περίπτωση, θα μπορούσε κανείς να διαφθείρει τους εσωτερικούς καταχώρητες του επεξεργαστή (καθώς και θέσεις εντός του χάρτη μνήμης) που είναι ορατά στον προγραμματιστή, παραδοσιακά αναφέρεται ως μοντέλο του προγραμματιστή του επεξεργαστή. Έτσι σφάλματα δεν μπορεί να εγχυθεί στον αγωγό επεξεργαστή ή ουρά διδασκαλίας για παράδειγμα. Πολύ δύσκολο να διαμορφώσει μόνιμες βλάβες. Σχετίζονται με τέσσερα, η εκτέλεση του λογισμικού εγχύσεως σφάλματος θα μπορούσε να επηρεάσει τον προγραμματισμό του

συστήματος εργασίες με τέτοιο τρόπο ώστε να προκαλέσουν σκληρά, σε πραγματικό χρόνο τις προθεσμίες που πρέπει να χαθεί, η οποία παραβιάζει παραδοχή δύο.

Επίσης σημαντικό είναι σε ποιά σημεία κώδικα να εισαχθούν τα σφάλματα. Σύμφωνα με το άρθρο "Application Resiliency Analyzer for Transient Faults " δεν χρειάζεται το σφάλμα να εισαχθεί σε κάθε σημείο του κώδικα εφόσον εάν εισαχθεί το σφάλμα σε ένα συγκεκριμένο σημείο μπορεί αυτό το σφάλμα να καλύψει ένα σύνολο σφαλμάτων όπου θα έχουν την ίδια συμπεριφορά στο σύστημα. Άρα δεν χρειάζεται να εισάγονται περισσότερα σφάλματα στο σύστημα. Σύμφωνα με το άρθρο αυτό είναι πολύ σημαντικό εφόσον δεν χρειάζεται να είναι χρονοβόρα η διαδικασία του fault injection και μπορούν να εισαχθούν λιγότερα σφάλματα τα οποία θα αντιπροσωπεύουν ένα σύνολο σφαλμάτων. Υπάρχει συγκεκριμένη εξίσωση που υπολογίζει την σημειακή εκτίμηση για την κάλυψη σφαλμάτων του συστήματος.

$$\mathbf{C}=\frac{\mathbf{1}}{\boldsymbol{n}} \times \sum_{\boldsymbol{i}=\mathbf{1}}^{\boldsymbol{n}} \boldsymbol{y}(\boldsymbol{ti}, \boldsymbol{\Delta i}, \boldsymbol{\sigma i}, \boldsymbol{li}, \boldsymbol{f})$$

Το C είναι η εκτίμηση σημείου για κάλυψη σφαλμάτων του συστήματος. Το n είναι ο αριθμός των πειραμάτων για fault injction. Από αυτή την εξίωση μπορούμε να αναλύσουμε ότι η εκτίμηση της κάλυψης σφαλμάτων εξαρτάται από το

- Fault type
- Location
- Time
- Duration
- Workload profile

Κάθε σύστημα έχει περίπου 100 δισεκατομμύρια τοποθεσίες για να γίνει fault injection και είναι αδύνατο να γίνουν σε όλες τις τοποθεσίες και για αυτό γίνεται αυτή η στατική προσέγγιση . Δεν είναι όμως εγγυημένη αυτή η εξίσωση.

3.3 Ένεσης Σφάλματος στο Υλικό

Η ένεση σφάλματος στο υλικό πραγματοποιείται σε ειδικά σχεδιασμένες εγκαταστάσεις δοκιμής υλικού για να επιτραπεί την έγχυση βλαβών στο σύστημα και να εξετάζονται οι συνέπειες[22]. Γίνεται χρήση επιπλέον υλικού για εισαγάγει σφάλματα στο στόχο του υλικού του συστήματος.[23] Ανάλογα με τις βλάβες και τις θέσεις τους, οι μέθοδοι hardware implemented fault injection μέθοδοι εμπίπτουν σε δύο κατηγορίες και είναι οι εξής:

1) Ένεση Σφάλματος στο Υλικό με επαφή

Η συσκευή έγχυσης έχει άμεση φυσική επαφή με το σύστημα έχοντας ως στόχο την παραγωγή τάσης ή αλλαγές ρεύματος στον στοχευόμενο τσιπ. Για παραδείγματα είναι οι

μέθοδοι που χρησιμοποιούν pin level. Το Pin level βασίζεται στην ιδέα να δημιουργεί διαταραχές ενσωματώνοντας κυκλώματα με ελαττώματα. [23]

2) Ένεση Σφάλματος στο Υλικό χωρίς επαφή

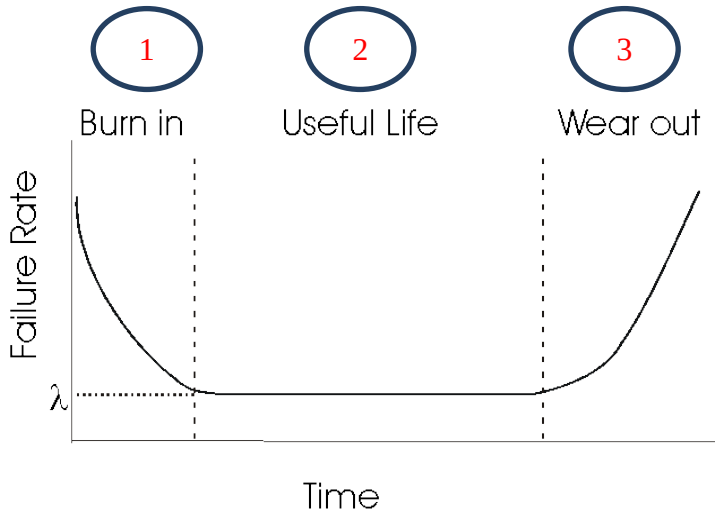
Η έγχυση σφαλμάτων δεν έχει άμεση φυσική επαφή με το σύστημα. Αντ. 'αυτού , μια εξωτερική πηγή παράγει κάποια φυσικά φαινόμενα , όπως τα βαρέα ιόνταακτινοβολίας και τις ηλεκτρομαγνητικές παρεμβολές ,προκαλώντας παρασιτικά ρεύματα στο εσωτερικό του chip (targetchip).Προσομοιώσεις υλικού συμβαίνουν συνήθως σε υψηλό επίπεδο του κυκλώματος. Αυτή η υψηλού επιπέδου περιγραφή μετατράπηκε σε ένα επίπεδο περιγραφής τρανζίστορ του κυκλώματος ,και τα σφάλματα εγχέονται στο κύκλωμα .Η λογισμική προσομοίωση πιο συχνά χρησιμοποιείται για την ανίχνευση της απόκρισης προς τις κατασκευαστικές ατέλειες . Το σύστημα κατόπιν προσομοιώνεται και γίνεται αξιολόγηση την απόκρισης του κυκλώματος προς το συγκεκριμένο σφάλμα . Δεδομένου ότι αυτό είναι μια προσομοίωση , ένα νέο σφάλμα μπορεί τότε εύκολα να εισαχθεί , και η προσομοίωση να επαναληφθεί και να γίνει μέτρηση στη νέα βλάβη. Αυτό ωστόσο θα καταναλώνει χρόνο για να κατασκευάσει το μοντέλο ,να εισχωρήσουν τα λάθη , και στη συνέχεια η προσομοίωση του κυκλώματος .

Τα οφέλη που μπορούν να αποκτηθούν με την ένεση σφάλμα στο υλικό είτε με επαφή είτε χωρίς επαφή είναι ότι γίνεται περισσότερο «ρεαλιστικά» η εισχώρηση των σφαλμάτων εφόσον προκαλούνται ηλεκτρομαγνητικές παρεμβολές είτε προκαλώντας θερμότητα είτε εισχώρηση φθαρμένων συρμάτων, μπορεί να παρατηρηθεί πως το κύκλωμα θα αντεπεξέλθει σε αυτή την κατάσταση. Έχει πρόσβαση σε περιοχές που είναι δύσκολο να υπάρχει πρόσβαση από αλλού. Πάραυτα το κόστος αυτής της διαδικασίας παίζει σημαντικό ρόλο εφόσον είναι αρκετά ακριβή διαδικασία και επίσης όσα σφάλματα να εκχωρηθούν για να εξεταστεί και η ποιότητα του υλικού μέσα στα συστήματα πάντοτε θα υπάρχουνε σφάλματα.[24]

Η αξιοπιστία του συστήματος για το υλικό στην ουσία κατηγοριοποιείται σε 3 φάσεις:

1. Burn –in
2. Operation
3. Wear – out

Στην αξιοπιστία του υλικού, αρχικά κατά την (1) πρώτη φάση της κατασκευής ,μπορεί να υπάρχει ένας μεγάλος αριθμός από σφάλματα, όμως,(2)μετά την ανακάλυψη αυτών των σφαλμάτων ο αριθμός αυτός μπορεί να μειωθεί και σταδιακά κατά τη δεύτερη φάση(διάρκεια ζωής), υπάρχει μόνο ένας μικρός αριθμός σφαλμάτων. Έπειτα από αυτή τη φάση, θα φτάσει στην (3) φάση φθοράς κατά την οποία, το φυσικό συστατικό φθείρεται λόγω του χρόνου και της συνεχής χρήσης τότε θα έχει ως αποτέλεσμα ο αριθμό των βλαβών θα αυξηθεί και πάλι[14].(βλέπε σχήμα 3.1)



Σχήμα 3.1

Κεφάλαιο 4

Εργαλεία για ένεση σφάλματος

Σε αυτό το σημείο θα αναφέρω μερικά εργαλεία που χρησιμοποιούνται για την ένεση σφαλμάτων στο λογισμικό.

4.1 JACA

Το JACA είναι ένα νέο εργαλείο λογισμικού Injection Fault , γραμμένο σε Java . Χρησιμοποιεί Javassist , ένα πρωτόκολλο Meta - Object , προκειμένου να διοχετεύσουν τα σφάλματα μέσα σε ένα σύστημα που δοκιμάζεται . Παρακολουθεί , επίσης, ότι το σύστημα να επαληθεύσει αν δίνει τα αναμενόμενα αποτελέσματα , ακόμη και με την παρουσία των ρηγμάτων [25].Το JACA μπορεί να εκτελέσει την ένεση σφάλμα υψηλού επιπέδου σε αντικειμενοστρεφή συστήματα . Δηλαδή , μπορεί να εισφέρει ελαττώματα που συνδέονται με atributes και τις μεθόδους των αντικειμένων ενός προγράμματος Java. JACA μπορεί επίσης να επεκταθεί εύκολα για να εκτελέσει την ένεση σφάλματος χαμηλού επιπέδου , που επηρεάζουν τα στοιχεία Συνέλευση γλώσσας (καταχωρητές της CPU , λεωφορεία , κλπ) .Το JACA σχεδιάστηκε για να είναι εξαιρετικά προσαρμόσιμο και φορητές συσκευές : να τρέχει σε κάθε πλατφόρμα τρέχει το Java Virtual Machine (JVM) .Το JACA δεν χρειάζεται τον πηγαίο κώδικα μιας εφαρμογής : όργανα που είναι αναγκαία για την ένεση βλάβης και την παρακολούθηση εισαχθούν στο επίπεδο bytecode .Μια διεπαφή χρήστη παρέχει μενού με γνώμονα την πρόσβαση στις λειτουργίες του εργαλείου .Σε ένα κοινό σχέδιο με το Πανεπιστήμιο της Κοΐμπρα , Jaca βελτιώθηκε , δίνοντας καταγωγής JacaC3 . Μεταξύ άλλων , Jaca μπορούν τώρα να δημιουργήσω εκθέσεις με βάση τη μορφή αρχείου . CVS . Οι εκθέσεις αυτές επιτρέπουν στατιστική ανάλυση και απαιτούν λιγότερη προσπάθεια για να

χειριστεί τα αποτελέσματα των πειραμάτων . Ικανότητες παρακολούθησης Jaca παρέχουν λεπτομερείς πληροφορίες για να παρατηρούν εξαιρετική συμπεριφορά κατά τη διάρκεια της εκτέλεσης της εκστρατείας ένεση βλάβης .[26]

4.2 LIBFIU

Το libfiu είναι μια βιβλιοθήκη C για ένεση σφάλμα. Έχει ως στόχο να απλοποιήσει αυτοματοποιημένη δοκιμή της συμπεριφοράς του λογισμικού σε σενάρια αποτυχίας.

Παρέχει λειτουργίες για να σηματοδοτήσει "σημεία της αποτυχίας" στο εσωτερικό του κώδικα ενός προγράμματος(ο πυρήνας API), και λειτουργεί για ενεργοποίηση / απενεργοποίηση της αποτυχίας αυτών των σημείων (το API ελέγχου). [27]

Ο πυρήνας API χρησιμοποιείται μέσα στον κώδικα που επιθυμείται να γίνει η ένεση αποτυχίας. Το API ελέγχου χρησιμοποιείται μέσα στον κώδικα δοκιμής, προκειμένου να ελέγχει την έγχυση των αποτυχιών. Επίσης, έρχεται με κάποια εργαλεία που μπορούν να χρησιμοποιηθούν για την εκτέλεση της ένεσης σφάλματος στο POSIX API, χωρίς να χρειάζεται τροποποίηση του πηγαίου κώδικα της εφαρμογής, που μπορούν να βοηθήσουν στον έλεγχο της αποτυχίας χειρισμό σε μια εύκολη και επιλήψιμο τρόπο. Είναι στο δημόσιο τομέα, εντελώς ανοικτή πηγή, ώστε να μπορεί οποιοσδήποτε να εκτελέσει το λογισμικό οπουδήποτε, και να συνδέσει αυτή τη βιβλιοθήκη με ό, τι θέλει .

4.5 Holodeck

Το Holodeck χρησιμοποιεί τεχνικές βλάβης - ένεση για να εισάγει την εφαρμογή σε προσομοίωση σεναρίων που προκύπτουν ως αποτέλεσμα τα « σφάλματα » στο περιβάλλον , όπως από τις συνθήκες της μνήμης , διεφθαρμένα αρχεία , κακά στοιχεία του μητρώου .

Συμπυκνώνει την εφαρμογή στόχου μέσα σε ένα προσομοιωμένο , tester - είναι ελεγχόμενο περιβάλλον που υπάρχει μεταξύ της εφαρμογής και του λειτουργικού συστήματος . Ως εκ τούτου, είναι σε θέση να εισαχθεί και να δοκιμαστεί η αντίδραση της εφαρμογής σε διάφορες ανωμαλίες του περιβάλλοντος αλλάζοντας την προσομοίωση χωρίς να πραγματοποιεί το λειτουργικό βάσης ή το σύστημα αρχείων [28].Ενώ το Holodeck αναχαιτίζει όλες τις κλήσεις συστήματος και API κλήσεις σας , αλλά επίσης την παρακολούθηση και την καταγραφή τους . Αυτό δίνει τη δυνατότητα να εξετάσει τις αλληλεπιδράσεις χαμηλού επιπέδου και πακέτα δικτύου - εντοπίζοντας και εύκολα να δημιουργήσει εκ νέου τα γεγονότα bug που παράγονται με ευκολία . Εν τω μεταξύ , το ολοκληρωμένο πρόγραμμα εντοπισμού σφαλμάτων δημιουργεί ένα « minidump " που συμβαίνουν σφάλματα και παρέχει την ακριβή γραμμή του κώδικα όπου συνέβη το σφάλμα . Με αυτές τις πληροφορίες , κατάλληλο για χρήση απευθείας σε ένα IDE όπως το Visual Studio .

4.4 GNU Debugger

Το GNU Debugger, που συνήθως ονομάζεται απλά GDB είναι το πρότυπο πρόγραμμα εντοπισμού σφαλμάτων για το λειτουργικό σύστημα GNU. Ωστόσο, η χρήση του δεν περιορίζεται αυστηρά στο λειτουργικό σύστημα GNU [29][30]. Είναι ένα φορητό πρόγραμμα εντοπισμού σφαλμάτων που τρέχει σε πολλά συστήματα Unix και λειτουργεί για πολλές γλώσσες προγραμματισμού, συμπεριλαμβανομένων Ada, C, C + +, Objective-C, Δωρεάν Pascal, Fortran, Java. Λίγα λόγια για την ιστορία του GDB .Το GDB γράφτηκε για πρώτη φορά από τον Richard Stallman το 1986 ως μέρος του συστήματος του GNU, μετά την GNU Emacs του ήταν "αρκετά σταθερή". Το GDB είναι ελεύθερο λογισμικό που διατίθεται βάσει της GNU General Public License (GPL). Διαμορφώθηκε μετά από το πρόγραμμα εντοπισμού σφαλμάτων DBX, η οποία ήρθε με Berkeley Unix διανομές. Από το 1990 έως το 1993 διατηρήθηκε από τον John Gilmore, ενώ εργάστηκε για Cygnus Solutions. [Παραπομπή που απαιτείται] . Το GDB προσφέρει εκτεταμένες εγκαταστάσεις για τον εντοπισμό και την αλλαγή της εκτέλεσης των προγραμμάτων ηλεκτρονικών υπολογιστών. Ο χρήστης μπορεί να παρακολουθεί και να τροποποιήσετε τις τιμές των εσωτερικών μεταβλητών προγραμμάτων », ζητώντας ακόμη και να λειτουργεί ανεξάρτητα από την κανονική συμπεριφορά του προγράμματος. Αυτός είναι και ο σκοπός χρήσης του GDB για το τρέξιμο των διάφορων πειραμάτων για ένεση σφαλμάτων στο λογισμικό για ένα C εκτελέσιμο[36]. Μέσα στο GDB θα αλλάζουν οι τιμές των 8 καταχωρητών. Αυτό είναι το εργαλείο το οποίο εργάστηκα για να βγάλω αποτελέσματα στα διάφορα πειράματα που έγιναν.

Αρχικά να αναφέρω ποιοί καταχωρητές υπάρχουν στην πλατφόρμα όπου ολοκληρωθήκαν τα διάφορα πειράματα. Εργάστηκα σε Intel x 86 πλατφόρμα . Κάθε καταχωρητής μπορεί να κρατάει από 32 bit μέχρι και 64 bit δεδομένα. Οι καταχωρητές είναι σαν μεταβλητές ενσωματωμένοι στον επεξεργαστή[31][32][33]. Χρησιμοποιώντας καταχωρητές αντί για τη μνήμη για την αποθήκευση τιμών καθιστά τη διαδικασία πιο γρήγορη και καθαρότερη. Το πρόβλημα με τους x86 επεξεργαστές είναι ότι υπάρχουν λίγοι καταχωρητές που μπορούν να χρησιμοποιηθούν. Ορισμένες λειτουργίες είναι απολύτως αναγκαίες σε κάποιο

είδος καταχωρητών , αλλά οι περισσότεροι από αυτούς μπορούν να χρησιμοποιηθούν πιο ελεύθερα. Οι καταχωρητές αυτής της πλατφόρμας είναι οι εξής :

Γενικοί καταχωρητές

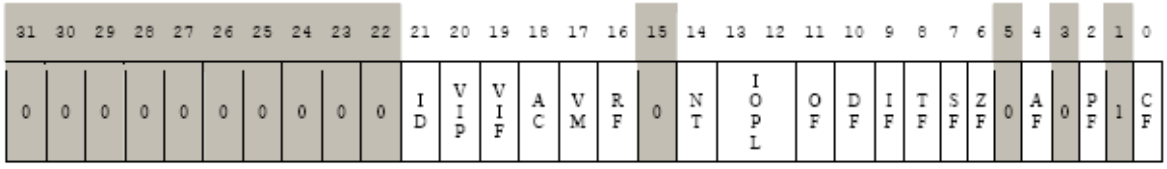
Σε λειτουργία 64-bit, υπάρχουν 16 καταχωρητές γενικού σκοπού και το προεπιλεγμένο μέγεθος του καταχωρητή είναι 32 bits. Ωστόσο, καταχωρητές γενικού σκοπού είναι σε θέση να εργαστούν είτε με 32-bit ή τελεστές 64-bit. Αν ένα μέγεθος τελεστή καθορίζεται 32-bit: EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP, R8d - R15d είναι διαθέσιμα. Αν ένα μέγεθος τελεστή καθορίζεται 64-bit: RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, R8-R15 είναι διαθέσιμα. R8D-R15D/R8-R15 αντιπροσωπεύουν οκτώ νέους καταχωρητές γενικής χρήσης. Όλοι αυτοί οι καταχωρητές μπορούν να έχουν πρόσβαση σε byte, λέξη, dword, και το επίπεδο QWORD. REX προθέματα χρησιμοποιούνται για να παράγουν 64-bit μεγέθη τελεστές ή καταχωρητές αναφοράς R8-R15 [32][33][34].

Καταχωρητές τμήματος

Οι καταχωρητές αυτοί κατέχουν το 16-bits επιλογέα τμήματος. Ένας επιλογέας τμήμα είναι ένας ειδικός δείκτης που προσδιορίζει ένα τμήμα στη μνήμη. Για να αποκτηθεί πρόσβαση σε ένα συγκεκριμένο τμήμα της μνήμης, ο επιλογέας τμήμα για το εν λόγω τμήμα πρέπει να είναι παρόντες στο κατάλληλο καταχωρητή τομέα. Οι τέσσερις καταχωρητές τμήματος είναι CS, DS, ES[32][33][34].

Καταχωρητές EFLAGS

Οι 32bit καταχωρητές περιέχουν μια ομάδα των σημαιών κατάστασης, σημαία ελέγχουν και μια ομάδα από σημαίες συστήματος. Πιο κάτω είναι ο πίνακας με τα είδη καταχωρητών EFLAG (βλέπε σχήμα 4.1) και τι αντιπροσωπεύουν (βλέπε σχήμα 4.2) [35].



Σχήμα 4.1

Flag	Mean	Type
ID	ID Flag	X
VIP	Virtual Interrupt Pending	X
VIF	Virtual Interrupt Flag	X
AC	Alignment Check	X
VM	Virtual 8086 Mode	X
RF	Resume Flag	X
NT	Nested Task	X
IOPL	IO Privilege Level	X
OF	Overflow Flag	X
DF	Direction Flag	C
IF	Interrupt Enable Flag	X
TF	Trap Flag	X
SF	Sign Flag	S
ZF	Zero Flag	S
AF	Auxiliary Carry Flag	S
PF	Parity Flag	S
CF	Carry Flag	S
	X - System Flags	
	C - Control Flags	
	S - Status Flags	

ΚΑΤΑΧΩΡΗΤΗΣ	ΜΕΓ BITS

Στα πειράματα ήρθα σε επαφή με τους πιο κάτω επεξεργαστές. Μελετώντας την λειτουργία του κάθε καταχωρητή μπόρεσα να ολοκληρώσω ορθά κάποια πειράματα [32][33][34][35].

Σχήμα 4.2

AL, AH/AX/EAX	8,8/16/32	Χρησιμοποιείται σε αριθμητικούς υπολογισμούς. Επίσης γνωστό ως συσσωρευτής, δεδομένου ότι κατέχει τα αποτελέσματα των αριθμητικών πράξεων και τιμές επιστροφής από συναρτήσεις.
BL, BH/BX/EBX	8,8/16/32	Καταχωρητές βάσης. Δείκτης με τα δεδομένα στην κατηγορία DS. Χρησιμοποιείται για να αποθηκεύσει τη διεύθυνση βάσης του προγράμματος.
CL, CH/CX/ECX	8,8/16/3	Καταχωρητές Μέτρησης. Χρησιμοποιείται συχνά για να κρατήσει μια τιμή που αντιπροσωπεύει τον αριθμό των φορών που η διαδικασία αυτή πρέπει να επαναληφθεί. Χρησιμοποιείται για εργασίες βρόχο.
DL, DH/DX/EDX	8,8/16/32	Καταχωρητές γενικής χρήσης. Επίσης χρησιμοποιείται για τις πράξεις I / O. Βοηθά να επεκτείνει EAX σε 64-bit.
SI/ESI	16/32	Καταχωρητής δείκτη κώδικα . Δείκτης με τα δεδομένα στην κατηγορία που υποδεικνύεται από τον καταχωρητή DS. Χρησιμοποιείται ως μια διεύθυνση offset σε ένα string και πράξεις πινάκων . Κατέχει τη διεύθυνση από την οποία να διαβάσει τα δεδομένα.
DI/EDI	16/32	Καταχωρητής δείκτης προορισμού . Δείκτης με τα στοιχεία (ή του προορισμού) στο τμήμα που υποδεικνύεται από το καταχωρητή ES. Χρησιμοποιείται ως μια διεύθυνση offset σε ένα string και πράξεις πινάκων.
BP/EBP	16/32	Καταχωρητής δείκτης βάσης . Δείκτης σε δεδομένα σχετικά με την στοίβα (στο τμήμα SS). Επισημαίνει στον πυθμένα του τρέχοντος πλαισίου στοίβας. Χρησιμοποιείται για την αναφορά σε τοπικές μεταβλητές
SP/ESP	16/32	Καταχωρητής Δείκτης στοίβας(στο τμήμα SS). Επισημαίνει στην κορυφή του τρέχοντος πλαισίου στοίβας. Χρησιμοποιείται για την αναφορά σε τοπικές μεταβλητές.

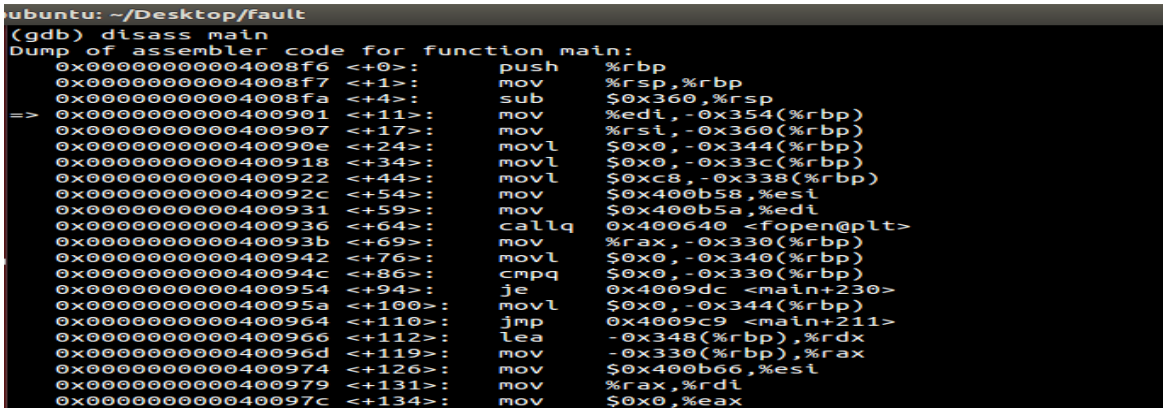
Αν ένα μέγεθος τελεστή καθορίζεται 32-bit είναι οι πιο κάτω καταχωρητές. (βλέπε σχήμα 4.3).

Αν ένα μέγεθος τελεστή καθορίζεται 64-bit είναι οι πιο κάτω καταχωρητές. (βλέπε σχήμα 4.4).

ΚΑΤΑΧΩΡΗΤΗΣ	ΜΕΓΕΘΟΣ ΣΕ BITS	ΛΕΙΤΟΥΡΓΙΑ
RIP	64	Χρησιμοποιείται σαν δείκτης που δείχνει στην τρέχουσα εντολή
RSP	64	Χρησιμοποιείται σαν δείκτης στοίβα;
RSP	64	Δείκτης Frame. Αποθηκευμένος δείκτης στοίβας σε μια παράμετρο της στοίβας
RDI	64	Παράμετρος συνάρτησης 1.
RDX	64	Παράμετρος συνάρτησης 2.
RCX	64	Παράμετρος συνάρτησης 3.
R8	8/16/32/64	Παράμετρος συνάρτησης 4.
R9	8/16/32/64	Παράμετρος συνάρτησης 5.
RAX	64	Επιστροφή τιμής από συνάρτηση
R10/R11	8/16/32/64	Προσωρινοί καταχωρητές (δεν χρειάζεται να αποθηκευτούν πριν χρησιμοποιηθούν)
RBX /R12/R13/R14/R15	64	Προσωρινοί καταχωρητές, πρέπει να αποθηκευτούν πριν χρήση και αποκατάσταση πριν από την επιστροφή του από την τρέχουσα συνάρτηση. (συνήθως με pop/push εντολές)

Σχήμα 4.4

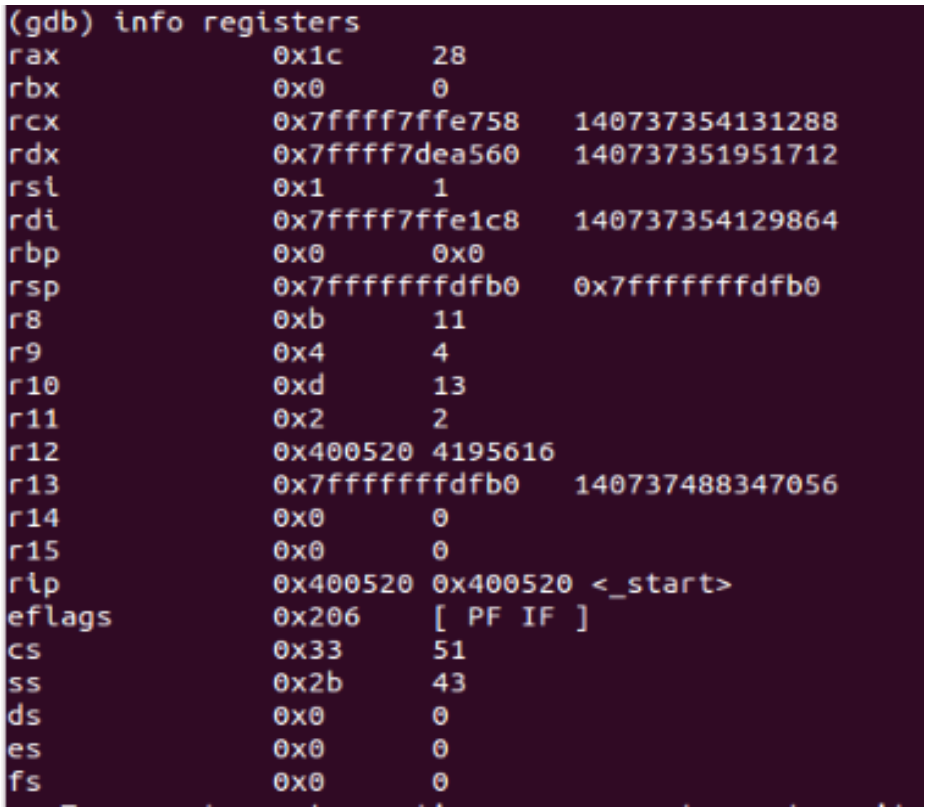
Ενώ το GDB Debugger σχεδιάστηκε για εντοπισμό σφαλμάτων ενός εκτελέσιμου , με τις εντολές που προσφέρει εύκολα μπορεί να χρησιμοποιηθεί σαν run time injector. Μπορεί να γίνει breakpoint σε οποιοδήποτε μέρος του εκτελέσιμου και να γίνουν διάφορες αλλαγές στο επίπεδο μεταβλητών ή στο χαμηλότερο επίπεδο, των καταχωρητών! Για περισσότερη κατανόηση πως οι καταχωρητές δουλεύουν στο σύστημα μελέτησα την assembly ενός εκτελέσιμου αρχείου. Με την εντολή disass main, μετά όταν γίνει breakpoint κάπου στο πρόγραμμα και τρέξει ως εκείνο το σημείο το εκτελέσιμο , θα εμφανιστεί η assembly του εκτελέσιμου. (βλέπε σχήμα 4.5)



Σχήμα 4.5

Μπορούμε να μελετήσουμε ποιός καταχωρητής χρησιμοποιείται σε συγκεκριμένη εντολή , σε ποιά διεύθυνση βρίσκεται , ποιά εντολή assembly χρησιμοποιείται και τέλος ποιοί καταχωρητές εκτελούν πράξεις . Μπορούμε επίσης να μελετήσουμε την λειτουργία του κάθε καταχωρητή όπως όρισα και πιο πάνω. Είναι σημαντικό για την ανάλυση της συμπεριφοράς του εκτελέσιμου αρχείου να ξέρουμε ποιοί καταχωρητές χρησιμοποιούνται κατά την διάρκεια του προγράμματος και ίσως μπορούν να επηρεάσουν και το τελικό αποτέλεσμα .

Επίσης κατά διάρκεια του εκτελέσιμου όταν γίνει breakpoint σε οποιοδήποτε εντολή του προγράμματος με την εντολή gdb info registers μπορούμε να δούμε ποιοί καταχωρητές γενικοί υπάρχουν. Η πρώτη στήλη δηλώνει όνομα καταχωρητή , δεύτερη στήλη δηλώνει ποιά διεύθυνση μνήμης χρησιμοποιούν και επίσης η τρίτη στήλη δηλώνει εάν υπάρχει τιμή που κρατά ο συγκεκριμένος καταχωρητής (βλέπε σχήμα 4.6)



Σχήμα 4.6

Σημαντικό είναι να γίνει κατανόηση πως οι καταχωρητές χρησιμοποιούνται κατά την διάρκεια εκτέλεσης έτσι ώστε όταν κάνουμε fault injection στο εκτελέσιμο να μπορέσουμε να αναλύσουμε τα αποτελέσματα που θα προκύψουν.

Κεφάλαιο 5

Μεθοδολογία κα Αποτελέσματα

5.1 Μεθοδολογία

Για την πραγματοποίηση των σφαλμάτων έχω επιλέξει να εργαστώ, όπως ανάφερα και πιο πριν, με το GNU Debugger . Ο λόγος είναι ότι θα μπορέσω να κάνω bit flip σε κάποιο καταχωρητή κατά την διάρκεια εκτέλεσης του προγράμματος. Η ένεση σφάλμα θα γίνει στο χαμηλότερο επίπεδο , στους καταχωρητές και θα γίνονται αλλαγές στους 16 καταχωρητές αλλάζοντας 1 bit από τον κάθε καταχωρητή και ξανατρέχοντας το εκτελέσιμο αρχείο. Για να γίνει αυτό το bit flip πρέπει να γίνουν XOR τα binary bits που έχει σαν τιμή ο κάθε καταχωρητής. Εφόσον κάθε καταχωρητής κρατάει 64 bits από δεδομένα επί τους 8 καταχωρητές μπορούν να υπάρξουν 1024 πειράματα για μια μόνο γραμμή εντολής του προγράμματος. Φανταστείτε το πρόγραμμα να περιέχει χιλιάδες γραμμές κώδικα. Για αυτό και είναι μια μακροχρόνια διαδικασία και αδύνατο να επικαλύπτει κάθε είδος σφάλματος.
Αρχικά στο πρώτο πείραμα θα γίνει επικύρωση ότι πράγματι υπάρχουνε επιπτώσεις στο αποτέλεσμα ενός εκτελέσιμου αρχείου όταν γίνουν αλλαγές και στους 16 καταχωρητές για κάθε bit-flip που θα αλλάξει. Έπειτα όταν γίνει η επικύρωση ότι βάζοντας λάθη σε ένα bit σε ένα καταχωρητή τότε θα έχει επιπτώσεις στο αποτέλεσμα τότε θα προχωρήσω στα επόμενα πειράματα. Στα επόμενα πειράματα θα γίνουν αλλαγές στους καταχωρητές για ένα breakpoint , δηλαδή για μια ένεση σφάλματος, έπειτα για δύο ενέσεις σφάλματος και τέλος για τρεις ενέσεις σφάλματος.

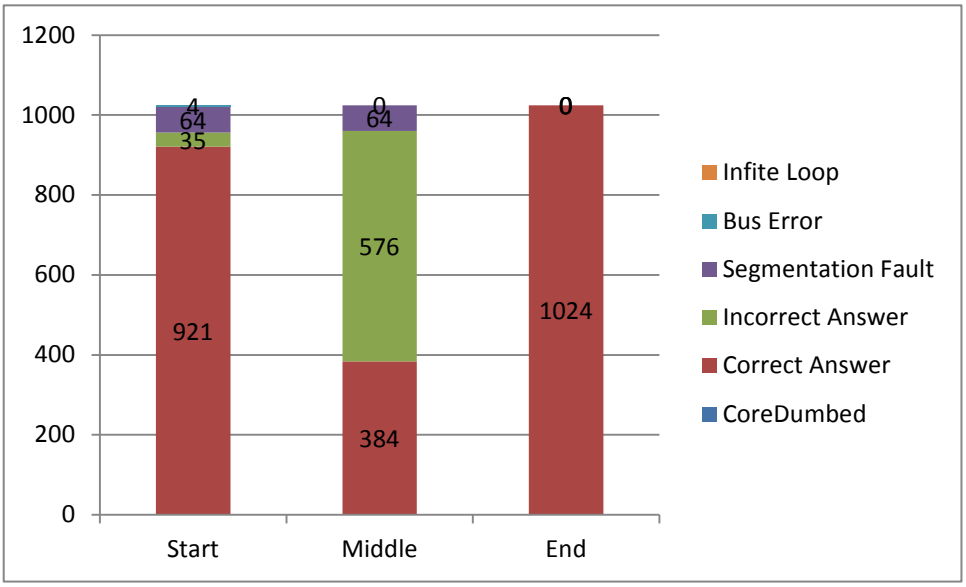
5.1 Αποτελέσματα

Στο πρώτο πείραμα όπως ανάφερα και πριν θα γίνει επικύρωση ότι πράγματι εάν αλλάξει ένα bit σε ένα καταχωρητή θα υπάρχουνε επιπτώσεις στο τελικό αποτέλεσμα του εκτελέσιμου αρχείου. Το εκτελέσιμο για το πρώτο πείραμα είναι πολύ απλό. Σε ένα c αρχείο γίνεται ανάθεση σε μια μεταβλητή ο αριθμός 4 και έπειτα τυπώνεται το αποτέλεσμα. Στο πρώτο πείραμα έγινε ένεση σφάλματος στην αρχή του προγράμματος για όλους του καταχωρητές αλλάζοντας 1 bit κάθε φορά και παρατηρούσα τα αποτελέσματα , μετά έγινε ένεση σφάλματος στην μέση του προγράμματος και συγκεκριμένη όταν γινόταν ανάθεση στην μεταβλητή x και το τελευταίο πείραμα στο τέλος του προγράμματος.

Το πρόγραμμα στην c που χρησιμοποίησα για εκτελέσιμο είναι το εξής:

```
#include <stdio.h>

void main()
{

    int x=4;
    printf("Result: %d \n",x);
}
```

Τα αποτελέσματα αυτών των πειραμάτων φαίνονται πιο κάτω. (βλέπε γραφική 5.1)



Σχήμα 5.1

Πιο πάνω αναπαριστώνται τα αποτελέσματα των πειραμάτων που περίγραψα πιο πάνω. Στον άξονα του x υποδεικνύει σε ποιά εντολή έγινε ένεση σφάλματος κατά την διάρκεια μιας εκτέλεσης του εκτελέσιμου αρχείου. Δηλαδή το Start- στην αρχή του προγράμματος, στο Middle- στη μέση του προγράμματος όπου γινόταν η ανάθεση της τιμής στην μεταβλητή και το End - στο τέλος του προγράμματος. Στον άξονα των y υποδεικνύει τον αριθμό των πειραμάτων. Τα χρώματα αναπαριστούν τα διάφορα αποτελέσματα των πειραμάτων (correct answer/fault/segmentation/bus error).

Όπως αποδεικνύω πράγματι υπήρξαν σφάλματα όταν έβαλα ένεση σφάλματος σε ένα bit από τα 64 bits ,στους 16 καταχωρητές. Στο πρώτο πείραμα στην αρχή του προγράμματος υπήρξαν συνολικά 35 εκτελέσιμα με λάθος αποτελέσματα, 921 εκτελέσιμα με σωστά αποτελέσματα, 4 εκτελέσιμα με bus error και 64 εκτελέσιμα με segmentation fault.

Ό λόγος που υπάρχουνε 64 segmentation είναι εφόσον συγκεκριμένα ο ένας καταχωρητής , ο $RIP, είναι καταχωρητής που χρησιμοποιείται σαν δείκτης. Δείχνει στην τρέχουσα εντολή του προγράμματος και όταν εισαχθεί λάθος σε αυτό τον καταχωρητή θα δείξει σε διευθύνσεις που είτε δεν υπάρχουνε στην μνήμη είτε δεν υπάρχει πρόσβαση σε αυτές. Επίσης έστω και εάν γίνουν αλλαγές σε κάποιους καταχωρητές σύμφωνα με τον assembly κώδικα κάποιοι αρχικοποιούνται ξανά ή σε κάποιους ανατίθεται μια τιμή στις επόμενες εντολές προγράμματος. Τα λανθασμένα αποτελέσματα είναι ένας μικρός αριθμός , μόλις 35

Στο δεύτερο πείραμα στην μέση του προγράμματος , όπου γινόταν ανάθεση στην μεταβλητή μια τιμή, υπήρξαν συνολικά 576 εκτελέσιμα με λάθος αποτελέσματα, 384 εκτελέσιμα με σωστά αποτελέσματα, και 64 εκτελέσιμα με segmentation fault.

Ό λόγος που υπάρχουνε 64 segmentation είναι εφόσον συγκεκριμένα ο ένας καταχωρητής , ο $RIP, είναι καταχωρητής που χρησιμοποιείται σαν δείκτης όπως ανάφερα και πιο πριν. Πραγματικά ενδιαφέρον τα αποτελέσματα εφόσον 576 εκτελέσιμα προέκυψαν σφάλματα ενώ μόλις 384 εκτελέσιμα έβγαλαν ορθά αποτελέσματα. Ο λόγος είναι εφόσον μερικοί από τους καταχωρητές που αλλάζω δεν επηρεάζουν τελικό αποτέλεσμα προγράμματος. Πιο κάτω φαίνονται μερικά από τα λανθασμένα αποτελέσματα.

Ο κώδικας του εκτελέσιμου προγράμματος στην c που χρησιμοποίησα είναι ο εξής:

```
#include<stdio.h>
 #include<stdlib.h>
#define NUMBERS 200
void quicksort(int *x,int first,int last){
 int pivot,j,temp,i;
 if(first<last){
        pivot=first;
        i=first;
        j=last;
 while(i<j){
            while(x[i]<=x[pivot]&&i<last)
               i++;
            while(x[j]>x[pivot])
               j--;
            if(i<j){
               temp=x[i];
                x[i]=x[j];
                x[j]=temp;
            }
        }
temp=x[pivot];
        x[pivot]=x[j];
        x[j]=temp;
        quicksort(x,first,j-1);
        quicksort(x,j+1,last);

    }
}
```

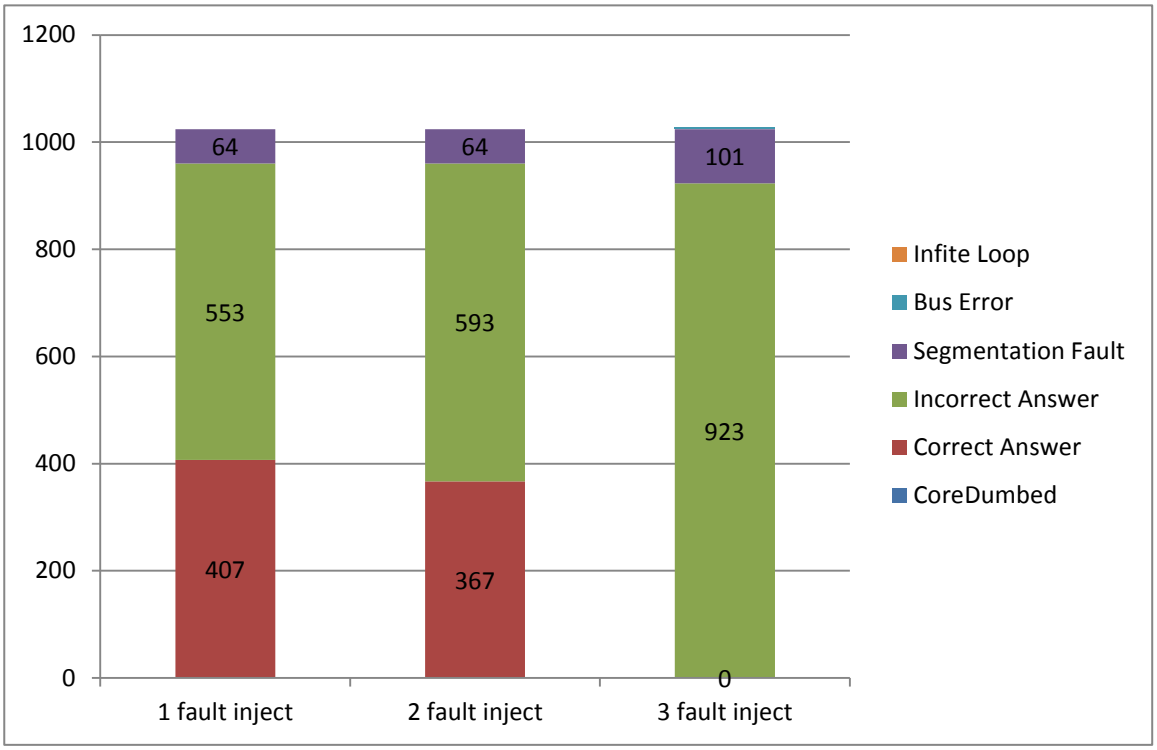
```
void main(int argc,char **argv){
 int x[NUMBERS],size,i=0,j=0;
 size=NUMBERS; int c;
 //x=(int *)malloc(size *sizeof(int));
 FILE* file;

    file = fopen ("numbers.txt", "r");
 int num_elements=0;
   if ( file != NULL )
            {
            i=0;
            while(!feof(file))
            {
                    fscanf(file, "%d", &c);
                    if(c!='\n')
                    {
                            if((c!=' ') || (c!='\0'))
                            {
                                    x[i]=c;
                                    num_elements++;
                            }
                            i++;
                    }
            }
   }
 size=num_elements-1;
 int first=0;
 quicksort(x,first,size-1);

 FILE *output;
 output = fopen ("sorted.txt", "a");
 if (output == NULL) {
       printf("I couldn't open results.dat for writing.\n");
       exit(0);
    }
//file = fopen ("sorted.txt", "w");
 for(i=0;i<size;i++)
       fprintf(output, "%d \t",x[i]);
      fprintf(output, "\n");
fclose(file);
fclose(output);
 printf("ELements have been sorted \n");
}
```

Το αναμενόμενο αποτέλεσμα ήταν το : Result: 4 αλλά με το fault injection τα περισσότερα αποτελέσματα ήταν λανθασμένα (βλέπε σχήμα 5.2)

```
22 Result: 5
23 [Inferior 1 (process 13724) exited with code 013]
24 (gdb) Starting program: /home/xanthi/Desktop/fault/simple_program
25
26 Breakpoint 1, 0x000000000040053f in main () at simple_program.c:7
27 7               printf("Result: %d \n",x);
28 (gdb) (gdb) (gdb) (gdb) Continuing.
29 Result: 6
30 [Inferior 1 (process 13728) exited with code 013]
31 (gdb) Starting program: /home/xanthi/Desktop/fault/simple_program
32
33 Breakpoint 1, 0x000000000040053f in main () at simple_program.c:7
34 7               printf("Result: %d \n",x);
35 (gdb) (gdb) (gdb) (gdb) Continuing.
36 Result: 7
37 [Inferior 1 (process 13729) exited with code 013]
38 (gdb) Starting program: /home/xanthi/Desktop/fault/simple_program
39
40 Breakpoint 1, 0x000000000040053f in main () at simple_program.c:7
41 7               printf("Result: %d \n",x);
42 (gdb) (gdb) (gdb) (gdb) Continuing.
43 Result: 0
```

Τέλος στο τρίτο πείραμα στο τέλος του προγράμματος όλα τα αποτελέσματα των εκτελέσιμων ήταν σωστά. Δηλαδή 1024 εκτελέσιμα είχαν το επιθυμητό αποτέλεσμα εφόσον οι καταχωρητές πλέον δεν έχουν τιμές που να επηρεάζουν επιθυμητό αποτέλεσμα.

Στο δεύτερο σύνολο πειραμάτων όπως ανάφερα και πριν γίνονται πειράματα έτσι ώστε να δείξω συμπεριφορά εκτελέσιμου αρχείου όταν υπάρξουν περισσότερα από μια ένεση σφαλμάτων . Για τα συγκεκριμένα πειράματα έτρεξε ένα εκτελέσιμο στην c όπου γίνεται ταξινόμηση αριθμών με την συνάρτηση quick_sorting().Στο πρώτο πείραμα έγινε ένεση ενός σφάλματος σε μια εντολή προγράμματος για όλους του καταχωρητές αλλάζοντας 1 bit κάθε φορά και παρατηρούσα τα αποτελέσματα , μετά έγινε δύο ένεσες σφάλματος σε άλλη εντολή προγράμματος μερικά δευτερόλεπτα πιο μετά και τέλος στο τελευταίο πείραμα έγιναν 3 ενέσεις πειραμάτων στο εκτελέσιμο σε 3 διαφορετικές εντολές στο ίδιο εκτελέσιμο αρχείο . Τα αποτελέσματα αυτών των πειραμάτων φαίνονται πιο κάτω. (βλέπε γραφική 5.2)



Σχήμα 5.2

Πιο πάνω αναπαριστώνται τα αποτελέσματα των πειραμάτων που περίγραψα πιο πάνω. Στον άξονα του x υποδεικνύει πόσα breakpoints έγιναν κατά την διάρκεια μιας εκτέλεσης του εκτελέσιμου αρχείου. Στον άξονα των y υποδεικνύει τον αριθμό των πειραμάτων. Τα χρώματα αναπαριστούν τα διάφορα αποτελέσματα των πειραμάτων (correct answer/fault/segmentation). Για κάθε breakpoint έτρεξαν 16 καταχωρητές όπου κάθε καταχωρητής έχει 64bits , άρα συνολικά έτρεξαν 1024 πειράματα .

Πάνω στην γραφική αναγράφονται ο αριθμός των πειραμάτων που παρουσιάστηκαν είτε λάθη, είτε σωστά είτε προκλήθηκε segmentation fault. Ενδιαφέρον είναι ότι όσα περισσότερα breakpoints βάλω τόσα περισσότερα faults εμφανίζονται εφόσον βάζω περισσότερο από ένα λάθη στο εκτελέσιμο.
Στο πρώτο πείραμα όταν εισάχθηκε ένα σφάλμα στο εκτελέσιμο υπήρξαν συνολικά 553 λανθασμένα αποτελέσματα , 407 σωστά αποτελέσματα και 64 segmentation faults. Ο λόγος που υπήρξε segmentation fault το ανάλυσα στα προηγούμενα πειράματα. Όπως παρατηρήθηκε η συχνότητα των λαθών αυξάνεται, όσο αυξάνονται οι ενέσεις σφαλμάτων που βάζω στην εκτέλεση του c αρχείου . Αυτό είναι λογικό εφόσον εάν ένας καταχωρητής αρχικοποιήθηκε πιο μετά σε άλλη γραμμή εντολής τότε εισχωρώντας του λάθος υπάρχει μεγαλύτερη πιθανότητα να επηρεαστεί το αποτέλεσμα του εκτελέσιμου αρχείου.
Υποδεικνύεται στο δεύτερο πείραμα όπου σε ένα εκτελέσιμο γίνονται δύο ενέσεις σφαλμάτων στο εκτελέσιμο και παρατηρήθηκε ότι υπήρξαν 367 εκτελέσιμα με σωστό αποτέλεσμα, 593 εκτελέσιμα με λανθασμένο αποτέλεσμα και 64 εκτελέσεις είχαν segmentation fault. Στο τρίτο πείραμα όταν εισάχθηκαν 3 ενέσεις σφαλμάτων σε ένα εκτελέσιμο για τα 1024 πειράματα παρατηρήθηκε ότι υπήρξαν 101 segmentation faults ενώ τα υπόλοιπα 923 είχαν σφάλματα στο τελικό αποτέλεσμα! Εάν ένας καταχωρητής δεν επηρεάστηκε αισθητά στο πρώτο breakpoint στα επόμενα ίσως να επηρεάστηκε μετέπειτα. Μερικά από τους καταχωρητές όπου έκανα ένεση σφάλμα συσχετίζονται με παραμέτρους συναρτήσεων ή επιστροφή τιμών από συναρτήσεις και if δηλώσεις. Στο συγκεκριμένο c εκτελέσιμο υπήρχαν επαναληπτικοί βρόγχοι, υπήρχε συνάρτηση με παραμέτρους και επιστροφή τιμής, ανάθεση τιμών σε μεταβλητές. Οι καταχωρητές που χρησιμοποίησα όπως ανάφερα και στο προηγούμενο κεφάλαιο συσχετίζονται με όλα τα πιο πάνω και για αυτό σε οποιοδήποτε καταχωρητή έκανα ένεση σφάλματος δεν υπήρχε εκτελέσιμο που να βγάλει σωστό αποτέλεσμα.

Κεφάλαιο 6

Συμπεράσματα

6.1 Συμπεράσματα

Στα διάφορα πειράματα συνειδητοποίησα ότι έστω και εάν εισαχθεί ένα σφάλμα σε ένα καταχωρητή υπάρχει περίπτωση να επηρεαστούνε κανένας , ένας ή περισσότεροι καταχωρητές. Εξαρτάται πάντοτε από το πρόγραμμα και ποιός καταχωρητής επηρεάζει τελικό αποτέλεσμα. Είναι σημαντικό να προσθέσω ότι η μνήμη από μόνη της με το ECC όπου μπορεί να ανακαλύψει εάν πράγματι υπήρχε σφάλμα και η τιμή κάποιου καταχωρητή δεν είναι σωστή και για αυτό ίσως στα τελευταία πειράματα στο ένα breakpoint ή στα δύο δεν υπήρχαν περισσότερα λάθη. Φανταστείτε χωρίς το ECC πόσα περισσότερα σφάλματα θα υπήρχαν. Αυτά τα πειράματα αποδεικνύουν ότι έστω και με ένα bit-flip στην τιμή του καταχωρητή εάν δεν γίνει επικάλυψη (masking) από το ECC, μπορεί να υπάρχει σαν αποτέλεσμα να υπάρξουνε λάθη . Εάν συνέβαινε αυτό σε μεγάλα συστήματα και δεν υπήρχε τρόπος για επικάλυψη ή αποκατάσταση από το σφάλμα θα υπήρχαν καταστροφικά αποτελέσματα (χρηματικά συνήθως). Σημαντικό είναι για οποιοδήποτε σύστημα να ελέγχεται με την έγχυση σφαλμάτων σε αυτά και να αναπτύξουν δυνατότερους μηχανισμούς για προστασία των δεδομένων μέσα στους καταχωρητές ή μνήμη πριν τα συστήματα αυτά βγουν στην αγορά. Είναι σημαντικός τομέας της πληροφορικής και επίσης ενδιαφέρον. Με την ένεση σφάλματος στο λογισμικό παίρνονται προληπτικά μέτρα για το πως μπορεί να αποκατασταθεί το σύστημα αυτόματα από αυτά ή να τα επικαλύψει. Αυτός ο τομέας αναπτύσσεται και γίνονται όλο και περισσότερες έρευνες για αυτόν , εφόσον πλέον αναγνωρίζεται ότι ένα σύστημα πρέπει να είναι και αξιόπιστο. Τέλος ,η ένεση των σφαλμάτων στο λογισμικό, μεταλλάσσοντας το εκτελέσιμο κώδικα ενός προγράμματος , επιτρέπει την πειραματική αξιολόγηση της αξιοπιστίας των συστημάτων για τα οποία ο πηγαίος κώδικας δεν είναι διαθέσιμος . Η προσέγγιση αυτή όμως απαιτεί δομές προγραμματισμού που χρησιμοποιείται στον πηγαίο κώδικα και πρέπει να προσδιοριστεί με την εξέταση στο δυαδικό κώδικα , δεδομένου ότι η ένεση γίνεται σε αυτό το επίπεδο .

Δυστυχώς , αυτό είναι δύσκολο έργο να εισφέρει λάθη στο δυαδικό κώδικα που να μιμούνται σωστά ελαττώματα λογισμικού στον πηγαίο κώδικα . Η ακρίβεια του δυαδικού επιπέδου σε τεχνικές ένεσης σφάλμα λογισμικού είναι , συνεπώς, μια σημαντική ανησυχία για την έγκρισή τους σε πραγματικά σενάρια

6.2 Μελλοντική Εργασία

Μελλοντικά θα μελετήσω περισσότερα το fault injection για μεγαλύτερα και πιο περίπλοκα προγράμματα. Επίσης τα σφάλματα να εισάγονται πιο τυχαία για πιο" ρεαλιστική" εικόνα πως θα υπήρχαν σφάλματα στο λογισμικό. Δηλαδή με την μέθοδο runtime exception/trap αντί με την μέθοδο timeout που χρησιμοποίησα. Μπορεί να υπάρξει διαφορετική συμπεριφορά του εκτελέσιμου προγράμματος. Τα τυχαία αυτά σφάλματα θα μπορούσαν να εισάγονται είτε μετά από κάποια δευτερόλεπτα όταν τρέχει το εκτελέσιμο είτε όταν φτάσει σε ένα συγκεκριμένο σημείο του προγράμματος. Για παράδειγμα έπειτα από 100 εμφανίσεις μιας τιμής να εισάγει το λάθος. Σημαντικό θα ήταν να δοκιμαστεί και διαφορετικές προγραμματιστικές γλώσσες όπως για παράδειγμα java ή c++ διότι υπάρχουνε διαφορές στους μεταγλωττιστές κάθε προγραμματιστικής γλώσσας .Θα υπάρξουνε διαφορετικά αποτελέσματα στην ένεση σφαλμάτων σε κάθε γλώσσα προγραμματισμού και θα ήταν ενδιαφέρον να παρατηρήσω την συμπεριφορά προγράμματος. Τέλος τα διάφορα πειράματα να εκτελεστούν σε διαφορετικές αρχιτεκτονικές και να παρατηρηθούν διαφορετικές συμπεριφορές λογισμικού σε σχέση με την Intel x86 core i7 όπου χρησιμοποίησα

Βιβλιογραφία

[1] Robert Thorhuus, "Software Fault Injection Testing", Stockholm February 2000

[2] http://en.wikipedia.org/wiki/Fault_injection

[3] http://en.wikipedia.org/wiki/Ariane_5

[4] http://www.ima.umn.edu/~arnold/disasters/ariane5rep.html

[5] Kalynnda Berens, "Software Fault Injection, Science Applications International Corporation NASA Glenn Research Center"

[6] Haissan Ziade,Rafic Ayioubi, Raoul Velanzo, " A Survey on Fault Injection Techniques TIMA Laboratory France"

[7] George A.Reis, Jonathan Chang, Neil Vachharajani, "Software-Controlled Fault Tolerance"

[8] http://www.ida.liu.se/~TDDB37/lecture-notes/lect9-10.frm.pdf

[9] http://www.imcnetworks.com/Assets/DocSupport/WP-MTBF-0311.pdf

[10] Sebastian G.Elbaum and John C.Munson ,"Getting a Handle of the Fault Injection Process: Valdation of Measurement Tools, University of Idaho Moscow"

[11] http://users.ece.cmu.edu/~koopman/des_s99/sw_reliability/

[12] http://www.theriac.org/DeskReference/viewDocument.php?id=126

[13] http://en.wikipedia.org/wiki/List_of_software_reliability_models

[14] http://www.ijcaonline.org/volume10/number5/pxc3871990.pdf

[15] Guohong Cao, Student Member, IEEE, and Mukesh Singhal, Member, IEEE, "On Coordinated Checkpointing in Distributed Systems,1996"

[16] http://people.ee.duke.edu/~sorin/ece254/

[17] R.W.Hamming, "The Bell System Technical Journal, Error Detecting and Error Correcting Codes, April 1950 "

[18] Alfredo Benso and Paolo Prinetto, "Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation, Boston 2003"

[19] Robert Slater "Fault Injection, Carnegie Mellon University, Sprint 1998"

[20] http://www.cs.bu.edu/teaching/cpp/debugging/errors/

[21] http://www.cs.umd.edu/~atif/Teaching/Fall2009/Jonathan.pdf

[22] http://www.cs.uiuc.edu/class/fa05/cs598yyz/slides/presentation/radu_fault_injection.pdf

[23] Pedro Gil, Sara Blanc and Juan Serrano, "PIN-LEVEL Hardware Fault Injection Techniques"

[24] http://www.sp.se/en/index/services/felinjic/sidor/default.aspx

[25] http://www.ic.unicamp.br/~eliane/JACA.html

[26] Nelson G.M.Lerne , Eliane Martings , Cecilia .M.F.Runbira, "A Software Fault Injection Pattern System, Brazil"

[27] http://blitiri.com.ar/p/libfiu/

[28] http://download.cnet.com/Holodeck-Enterprise-Edition/3000-2218_4-10551609.html

[29] http://en.wikipedia.org/wiki/GNU_Debugger

[30] http://www.cs.cmu.edu/~gilpin/tutorial/

[31] http://www.eecg.toronto.edu/~amza/www.mindsec.com/files/x86regs.html

[32] http://en.wikibooks.org/wiki/X86_Assembly/X86_Architecture

[33] http://www.cs.virginia.edu/~evans/cs216/guides/x86.html

[34] https://software.intel.com/en-us/articles/introduction-to-x64-assembly

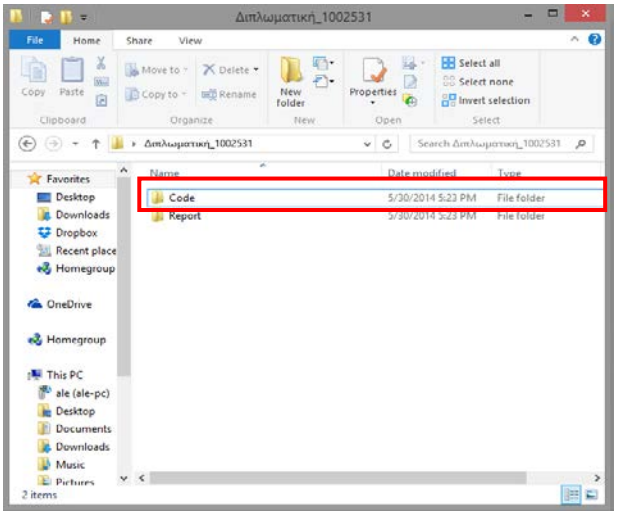
[35] http://www.tenouk.com/Bufferoverflowc/Bufferoverflow1a.html

[36] Mike Loukides and Andy Oram, "Getting to Know GDB, September 1996"

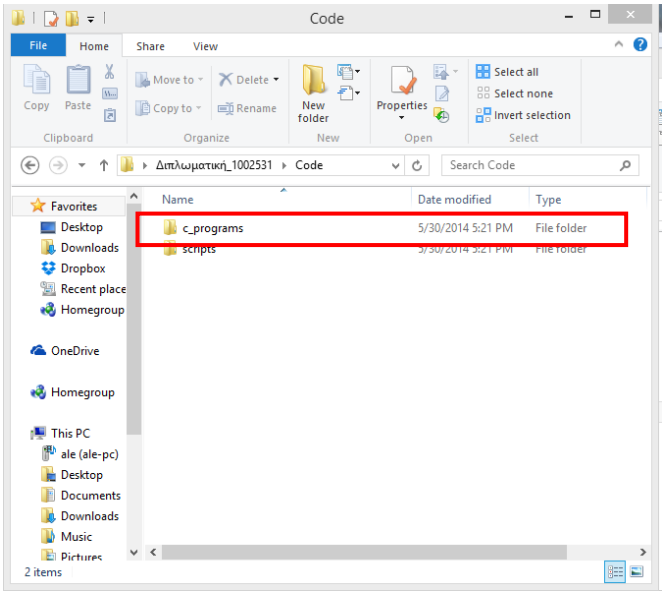
[37] Siva Kumar Hari, Helia Naeimi,Pradeep Ramachandran και Sarita V. Adve, "Relyzer: Application Resiliency Analyzer for Transient Faults, University og Illinois"

Παράρτημα Α

Ο κώδικας για τα διάφορα πειράματα βρίσκονται στο cd στον φάκελο Code.



Ο κώδικας για τα c προγράμματα βρίσκονται στον φάκελο c_programs

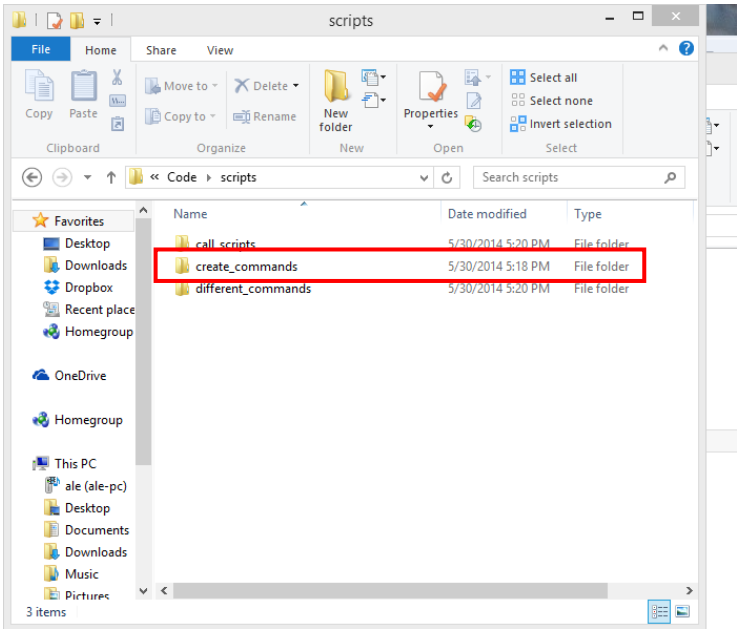


Το simple_program.c χρησιμοποιήθηκε για να γίνει validation ότι πράγματι αλλάζουν τα αποτελέσματα όταν γίνει fault injection

Το quick_sorting.c χρησιμοποιήθηκε για τα επόμενα πειράματα.

Παράρτημα Β

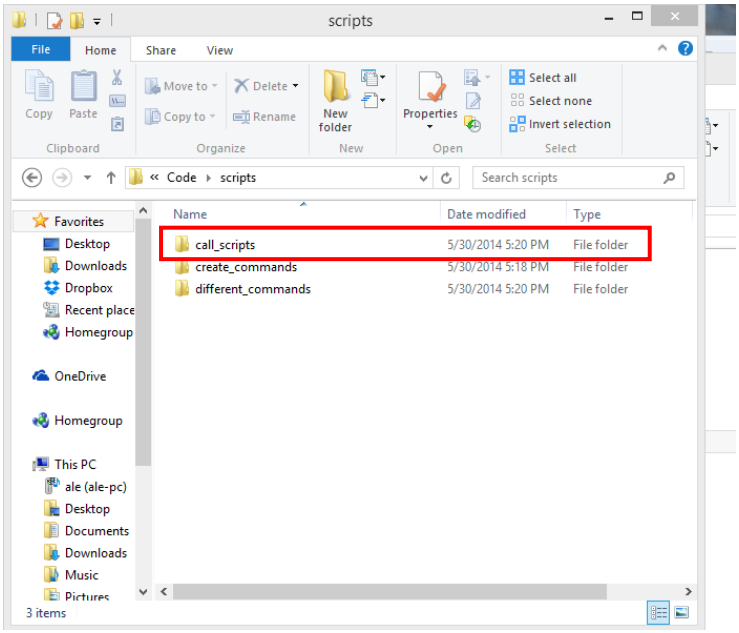
Ο κώδικας για την δημιουργία των εντολών που χρησιμοποιούνται στο gdb για να γίνει fault injection είναι στον φάκελο create_commands



Μέσα στον φάκελο υπάρχουνε αρχεία όπου δημιουργούνται σε bash script οι εντολές για να γίνονται fault injection σε ένα εκτελέσιμο αρχείο.

Παράρτημα Γ

Ο κώδικας για το κάλεσμα αυτών των εντολών υλοποιείται σε bash script . Βρίσκονται στον φάκελο call_scripts



Είναι τα scripts τα οποία ελέγχουν την απάντηση εάν είναι σωστή, εάν υπάρχει segmentation fault ή γενικά υπάρχει λάθος στο αποτέλεσμα εκτελέσιμου.