ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Αντιστοίχιση παροδικών σφαλμάτων υλικού με το σημείο κώδικα στο οποίο εκδηλώθηκαν. Μελέτη στο πεδίο των ανεκτικών σε λάθη υπολογισμών.

Matching transient hardware faults with the point in code where they manifested. Study in the context of fault-tolerant computing.

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Παναγιώτης Μήτσης

Επιβλέποντες: **Δρ. Αντωνόπουλος Χρήστος**, Επίκουρος Καθηγητής, Πανεπιστήμιο Θεσσαλίας
Δρ. Μπέλλας Νικόλαος, Αναπληρωτής Καθηγητής, Πανεπιστήμιο Θεσσαλίας

ΒΟΛΟΣ

ΙΟΥΛΙΟΣ 2014

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Αντιστοίχιση παροδικών σφαλμάτων υλικού με το σημείο κώδικα στο οποίο εκδηλώθηκαν. Μελέτη στο πεδίο των ανεκτικών σε λάθη υπολογισμών.

Matching transient hardware faults with the point in code where they manifested. Study in the context of fault-tolerant computing.

Παναγιώτης Μήτσης

ΑΜ: 1085.

ΕΠΙΒΛΕΠΟΝΤΕΣ :

Δρ. Αντωνόπουλος Χρήστος, Επίκουρος Καθηγητής
Δρ. Μπέλλας Νικόλαος, Αναπληρωτής Καθηγητής

Εγκρίθηκε από την διμελή εξεταστική επιτροπή την 11/07/2014

....	....
Χρήστος Αντωνόπουλος	Νικόλαος Μπέλλας
Επίκουρος Καθηγητής	Αναπληρωτής Καθηγητής

Διπλωματική Εργασία για την απόκτηση του Διπλώματος του Μηχανικού Ηλεκτρονικών Υπολογιστών, Τηλεπικοινωνιών και Δικτύων του Πανεπιστημίου Θεσσαλίας, στα πλαίσια του Προγράμματος Προπτυχιακών Σπουδών του Τμήματος Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του Πανεπιστημίου Θεσσαλίας.

....
Παναγιώτης Μήτσης
Διπλωματούχος Μηχανικός Ηλεκτρονικών Υπολογιστών, Τηλεπικοινωνιών και Δικτύων του Πανεπιστημίου Θεσσαλίας

Copyright © Panagiotis Mitsis, 2014
Με επιφύλαξη παντός δικαιώματος. All rights reserved.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα.

Στην Οικογένεια μου

ΕΥΧΑΡΙΣΤΙΕΣ

Με τη περάτωση της παρούσας εργασίας, θα ήθελα να ευχαριστήσω θερμά τους επιβλέποντες καθηγητές της διπλωματικής εργασίας δρ. Χρήστο Αντωνόπουλο και δρ. Νικόλαο Μπέλλα για την εμπιστοσύνη που έδειξαν στο πρόσωπό μου, την άριστη συνεργασία όλα τα χρόνια που δουλέψαμε μαζί, την καθοδήγηση και τις ουσιώδεις υποδείξεις και παρεμβάσεις που διευκόλυναν την εκπόνηση της διπλωματικής εργασίας.

Επίσης θα ήθελα να ευχαριστήσω θερμά τον κύριο Παρασύρη Κωνσταντίνο για όλες τις συμβουλές και υποδείξεις που διευκόλυναν την περάτωση αυτής της διπλωματικής.

Τέλος, οφείλω ένα μεγάλο ευχαριστώ στην οικογένεια μου για την αμέριστη υποστήριξη που μου παρείχαν τόσο κατά τη διάρκεια των σπουδών μου όσο και κατά την εκπόνηση αυτής της εργασίας.

Παναγιώτης Μήτσης

ΠΕΡΙΕΧΟΜΕΝΑ

Περίληψη **16**

Abstract **17**

1 Εισαγωγή **19**

1.1 Περιγραφή του Προβλήματος 19

1.2 Διάρθρωση εργασίας 20

2 Πληροφορίες για το εκτελέσιμο αρχείο **22**

2.1 Κατανομή Μεταβλητών Μέσα σε ένα Εκτελέσιμο Αρχείο 22

2.2 Πρωτόκολλα αναπαράστασης Συμβόλων αποσφαλμάτωσης 24

2.3 Θέσεις Μεταβλητών ενός Προγράμματος 24

2.4 Περιγραφή του πρωτοκόλλου DWARF 26

2.5 Προσπέλαση των συμβόλων αποσφαλμάτωσης από τον προσομοιωτή 30

3 Εύρεση Εκτελέσιμου αρχείου κατά τον χρόνο Εκτέλεσης **32**

3.1 Ο Πυρήνας του Linux 32

3.2 Δομή Διεργασίας 33

3.3 Εύρεση του PCB σε κάθε Αρχιτεκτονική 34

3.3.1 Εύρεση του PCB σε ALPHA Αρχιτεκτονική 34

3.3.2 Εύρεση του PCB σε X86-64 Αρχιτεκτονική 34

3.4 Δομές δεδομένων για την εύρεση του εκτελέσιμου Αρχείου 35

4 Πειραματική Αξιολόγηση 40

4.1 Αξιολόγηση της εφαρμογής MonteCarlo PI 41

4.2 Αξιολόγηση της εφαρμογής Knapsack 48

5 Μελλοντική ανάπτυξη του συστήματος 55

Α΄ Κώδικας Πυρήνα για Περιγραφή Διεργασίας 56

ΒΙΒΛΙΟΓΡΑΦΙΑ 58

ΚΑΤΑΛΟΓΟΣ ΣΧΗΜΑΤΩΝ

2.1 Κατανομή θέσης αποθήκευσης με βάση την εμβέλεια 25

2.2 Δεντρική αναπαράσταση του πρωτοκόλλου DWARF 26

3.1 Διάταξη μνήμης μιας διεργασίας 33

3.2 Δομές δεδομένων για την εύρεση του μονοπατιού 36

3.3 Δομή του συστήματος αρχείων του Linux 37

4.1 Σύνολο Σημείων για την εφαρμογή MonteCarlo PI 41

4.2 Σύνολο προσομοίωσης για την εφαρμογή MonteCarlo PI με την επίδραση κάθε σφάλματος 42

4.3 Επίδραση κάθε σφάλματος στις συναρτήσεις rand/myrand της εφαρμογής MonteCarlo PI 43

4.4 Επίδραση κάθε σφάλματος στη συνάρτηση MonteCarlo Integrateτης εφαρμογής MonteCarlo PI 44

4.5 Επίδραση κάθε σφάλματος σε 5 ισοχωρισμένα εύρη τιμών του iterator "count" της εφαρμογής MonteCarlo PI 45

4.6 Επίδραση κάθε σφάλματος στη γραμμή 54 του αρχικού κώδικα της εφαρμογής MonteCarlo PI 46

4.7 Επίδραση κάθε σφάλματος στη γραμμή 56 του αρχικού κώδικα της εφαρμογής MonteCarlo PI 46

4.8 Επίδραση κάθε σφάλματος στη γραμμή 58 του αρχικού κώδικα της εφαρμογής MonteCarlo PI 47
4.9 Σύνολο προσομοίωσης για την εφαρμογή KnapSack με την επίδραση κάθε σφάλματος 49
4.10 Επίδραση κάθε σφάλματος στις συναρτήσεις της stdlib της εφαρμογής KnapSack 50
4.11 Επίδραση κάθε σφάλματος στη συνάρτηση της cal_fitness της εφαρμογής KnapSack 51
4.12 Επίδραση κάθε σφάλματος στη συνάρτηση της cross της εφαρμογής KnapSack . 52
4.13 Επίδραση όλων των σφαλμάτων στη γραμμή 46 του αρχικού κώδικα της εφαρμογής KnapSack 53
4.14 Επίδραση όλων των σφαλμάτων στη γραμμή 47 του αρχικού κώδικα της εφαρμογής KnapSack 53
4.15 Επίδραση όλων των σφαλμάτων στη γραμμή 48 του αρχικού κώδικα της εφαρμογής KnapSack 54
4.16 Επίδραση όλων των σφαλμάτων στη γραμμή 84 του αρχικού κώδικα της εφαρμογής KnapSack 54

Περίληψη

Σήμερα μια από τις προκλήσεις που έχει να αντιμετωπίσει η επιστημονική κοινότητα είναι το πως μπορούν να γίνουν αξιόπιστοι υπολογισμοί κάτω από αναξιόπιστες συνθήκες καθώς μειώνεται η αξιοπιοτία στα ηλεκτρονικά κυκλώματα. Για το λόγο αυτό δημιουργήθηκε ένα εργαλείο, από τον Γιώργο Τζιαντζιούλη[3] και εξελίχθηκε από τον Κωνσταντίνο Παρασύρη[5], για την αξιολόγηση της αξιόπιστης λειτουργίας υπολογιστικών συστημάτων μετά από παροδικά σφάλματα υλικού.Το εργαλείο που δημιουργήσανε είναι βασισμένο στον προσομοιωτή υλικού Gem5.

Εγώ με τη σειρά μου ανέπτυξα ένα εργαλείο το οποίο δίνει περισσότερη πληροφορία σχετικά με την κατάσταση που βρισκόταν το σύστημα όταν έγινε η εισαγωγή ενός σφάλματος. Πιο συγκεκριμένα βρίσκει αυτόματα από τον πυρήνα του Linux που χρησιμοποιεί ο προσομοιωτής, την εφαρμογή που προσομοιώνουμε και κάνοντας χρήση των στοιχείων αποσφαλμάτωσης, που ενσωματώνονται σε κάθε εκτελέσιμο κατά τη διάρκεια της μεταγλώττισης βρίσκει και εκτυπώνει:

1. Την τιμή όλων των μεταβλητών που είναι εντός εμβέλειας.
2. Το σημείο του κώδικα που ήταν η εξεταζόμενη εφαρμογή.
3. Ένα backtrace με τις κλήσεις συναρτήσεων που έχουν γίνει από την αρχή της εκτέλεσης.

Έχοντας όλες αυτές τις πληροφορίες μπορεί να κατανοηθεί καλύτερα η επίδραση ενός σφάλματος στην εκτέλεση μια εφαρμογής.

Abstract

In our days, one challenge that the computer science community have to solve is how a machine can do fault tolerant calculations due to the electronic circuits. In our University it was developed one tool from Georgios Tziantzoulis[3] and evolved from Konstantinos Parasyris[5] in order to Simulate systems with some Faults and make observations about how the simulating system react. This is based on the computer architecture simulator Gem5.

For the context of my thesis i have developed one tool for the Gem5 simulator, to obtain more information about the status of the simulating program the exact moment that the error occur. More precisely it can detect automatically the binary which is executing direct from the Linux Kernel and using the debug symbols that the comiler encapsulates at compilation-time we can have the following information:

1. In scope variables value.
2. The exact source code point when the error occur.
3. A backtrace with all the function calls from the start of execution.

With all this information we can have better understanding about the error effect on our simulating program.

ΚΕΦΑΛΑΙΟ 1

Εισαγωγή

1.1 Περιγραφή του Προβλήματος

Η σμίκρυνση των τρανζίστορ στα πολυπύρηνα chip έχει αυξήσει τη πιθανότητα εμφάνισης παροδικών λαθών. Αυτά έχουν μη προβλεπόμενη συμπεριφορά και απειλούν την λειτουργικότητα των συστημάτων της επόμενης γενιάς. Οι σημερινές τεχνολογίες για να αποφύγουν τις παρεμβολές χρησιμοποιούν guardbands στην συχνότητα και στην τάση τροφοδοσίας. Όμως τα guardbands κοστίζουν σε επίδοση και κατανάλωση ισχύος ενώ το επιπλέον υλικό κοστίζει (πέρα από τα παραπάνω) και σε επιφάνεια.

Επομένως το πρόβλημα που καλείται να λύσει η επιστημονική κοινότητα είναι το πως μπορούν να γίνονται αξιόπιστοι υπολογισμοί κάτω από αναξιόπιστες συνθήκες. Πολλές εφαρμογές είναι εγγενώς ανεκτικές σε λάθη(ή τουλάχιστον ένα μέρος αυτών) όπως για παράδειγμα εφαρμογές που χρησιμοποιούν εξελικτικές μεθόδους υπολογισμού (Genetic programming) όπου ένα σφάλμα δεν επηρεάζει πολύ το αποτέλεσμα. Συνεπώς θα θέλαμε να εκμεταλλευθούμε τέτοιες εφαρμογές ώστε να υποστηρίζουμε υπολογισμούς χωρίς μείωση της ποιότητας των αποτελεσμάτων. Για να επιτευχθεί αυτό θα πρέπει να δημιουργηθούν εργαλεία ανάλυσης της συμπεριφοράς των λαθών σε επίπεδο υλικού και λογισμικού. Με την ανάλυση αυτή μπορεί να δημιουργηθεί μια ιεραρχία για τα κομμάτια μιας εφαρμογής που πρέπει να εκτελεσθούν σε υλικό που εγγυάται την έλλειψη σφαλμάτων και σε υλικό που υπάρχει η πιθανότητα να παρουσιαστούν παροδικά σφάλματα, ώστε να ενισχυθούν τα κατάλληλα σημεία του συστήματος.

Για να γίνει αυτή ν ανάλυση δημιουργήθηκε ένα εργαλείο εισαγωγής λαθών στον προσομοιωτή αρχιτεκτονικής Gem5,ο GemFI [4]. Ο GemFI είναι μια τροποποιημένη έκδοση του Gem5 που δουλεύει σε ALPHA και X86-64 αρχιτεκτονικές και κατά την εκκίνηση του διαβάζει ένα

αρχείο κειμένου που υπαγορεύει το σφάλμα, της μορφής Που(στο pipeline)/Πότε(κύκλος εκτέλεσης)/Τι(η φύση του σφάλματος-αντιστροφή bits) [5]. Όμως ο GemFI αυτό που θα δείξει ως αποτέλεσμα είναι που έγινε το σφάλμα δηλαδή σε ποιό καταχωρητή ή υπολογισμό, την προγενέστερη τιμή που είχε το κομμάτι που επηρεάστηκε και την επόμενη τιμή που θα πάρει, χωρίς όμως να προσφέρει πληροφορία για το πόσο σημαντικό ήταν το σημείο εκείνο για την εφαρμογή μας καθώς εκεί μπορεί να ήταν αποθηκευμένη μια μεταβλητή ισχυρής βαρύτητας και σημασίας ή μπορεί να ήταν και ένα κομμάτι το οποίο δεν χρησιμοποιόταν καθόλου από την εφαρμογή μας. Ορμώμενοι από αυτό το εργαλείο δημιουργήσαμε μια επέκταση για τον GemFI η οποία μας προσφέρει μια καλύτερη οπτική στο σύστημα που θέλουμε να προσομοιόσουμε.

Πιο συγκεκριμένα η επέκταση αυτή μας βοηθάει να αντιστοιχήσουμε τα λάθη από το επίπεδο της αρχιτεκτονικής εικόνας που βλέπει ο προγραμματιστής, στο επίπεδο των semantics του προγράμματος (θέση στον κώδικα, συγκεκριμένες μεταβλητές / δομές δεδομένων κλπ). Για να γίνει αυτό πρέπει πρώτα να βρούμε τη θέση του εκτελέσιμου αρχείου στο δίσκο του υπό προσομοίωση συστήματος χρησιμοποιώντας το πυρήνα του Linux διότι χρειαζόμαστε άμεση πρόσβαση σε αυτό για να προσπελάσουμε τα δεδομένα των συμβόλων αποσφαλμάτωσης. Ύστερα χρησιμοποιώντας τα δεδομένα των συμβόλων αποσφαλμάτωσης που είναι ενσωματωμένα στο εκτελέσιμο αρχείο που εξετάζουμε σε συνδυασμό με την τιμή του Program Counter βρίσκουμε το κομμάτι του κώδικα τις εφαρμογής που εκτελείται καθώς και όλες τις μεταβλητές εντός εμβέλειας. Έτσι αν το σφάλμα που εισαχθεί από τον GemFI ταιριάζει με καταχωρητή ή θέση μνήμης που έχει κάποια μεταβλητή τότε ξέρουμε τι σημασία του και μπορούμε να εκτιμήσουμε πόσο σημαντικό ή όχι ήταν το λάθος. Επιπλέον μπορούμε να δούμε το στάδιο του προγράμματος όταν έγινε το σφάλμα (πχ αν είμαστε σε εμβέλεια ενός βρόγχου επανάληψης βλέπουμε σε ποιά επανάληψη είμαστε) αλλά και ποιές συναρτήσεις είχαν κληθεί εως εκείνη τη στιγμή με τη λήψη ενός backtrace από τη στοίβα του προγράμματος.

1.2 Διάρθρωση της Διπλωματικής εργασίας

Στο κεφάλαιο 2 παρουσιάζονται οι πληροφορίες που είναι χρήσιμες από ένα πρόγραμμα. Αρχικά παρουσιάζεται ο τύπος των μεταβλητών που μπορεί να οριοθεί και η εμβέλεια. Ύστερα, αναλύεται το πρότυπο συμβόλων αποσφαλμάτωσης DWARF και τέλος πως μπορεί να χρησιμοποιηθεί για τη λύση του δικού μας προβλήματος.

Στο κεφάλαιο 3 αναλύονται τα βήματα που πρέπει να ακολουθηθούν στον προσομοιωτή αρχιτεκτονικής Gem5 ώστε να μπορέσει μη έχοντας άμεση πρόσβαση στο λειτουργικό σύστημα υπό προσομοίωση (Host OS) να βρει το πλήρες μονοπάτι στο δίσκο που ανήκει το εκτελέσιμο αρχείο

που θα αναλυθεί η συμπεριφορά του σε παροδικά σφάλματα υλικού.

Στο κεφάλαιο 4 βρίσκονται τα αποτελέσματα της πειραματικής αξιολόγησης και τέλος στο Κεφάλαιο 5 προτάσεις για την μελλοντική ανάπτυξη του συστήματος.

ΚΕΦΑΛΑΙΟ 2

Πληροφορίες για το εκτελέσιμο αρχείο

2.1 Κατανομή Μεταβλητών Μέσα σε ένα Εκτελέσιμο Αρχείο

Οι μεταβλητές ενός προγράμματος είναι διάφορα συμβολικά ονόματα συνδεδεμένα με μια θέση στην οποία αποθηκεύονται οι τιμές που παράγονται, η οποία είναι άγνωστη για τον προγραμματιστή καθώς κατά τη συγγραφή ενός προγράμματος ο προγραμματιστής γνωρίζει για τις μεταβλητές του την εμβέλεια τους και τον τύπο τους και τίποτα παραπάνω. Στην C υπάρχουν οι εξής βασικοί τύποι:

1. char
2. int
3. float
4. double
5. void

καθώς και προσδιοριστικά μεγέθους όπως short/long για τα int/float/double. Επίσης μπορούν να οριστούν και άλλοι τύποι δεδομένων με τις ρίζες τους στους βασικούς τύπους όπως τα structs για παράδειγμα:

```
struct foo{
        int x;
        int y;
};
```

Ο τύπος δεδομένων foo δεν είναι βασικός, όμως ο προγραμματιστής μπορεί να δηλώσει μεταβλητές τύπου struct foo όπως ακριβώς και τις μεταβλητές βασικού τύπου άρα για να μπορέσουμε να βρούμε μια μεταβλητή στο σύστημα μας χρειαζόμαστε τις εξής πληροφορίες:

1. Ο τύπος δεδομένων που αντιπροσωπεύει.
2. Το όνομα της μεταβλητής.
3. Το μέγεθος που καταλαμβάνει στην μνήμη.
4. Το που βρίσκεται (βλ. παρ. 2.3)
5. Αν είναι σύνθετος τύπος δεδομένων πρέπει τα παραπάνω να γίνουν αναδρομικά για κάθε πεδίο που το στελεχώνει.

Για τις ανάγκες τις παρούσας διπλωματικής θα ασχοληθούμε μόνο με τους ακόλουθους τύπους δεδομένων:

1. long unsigned int
2. unsigned char
3. short unsigned int
4. unsigned int
5. signed char
6. short int
7. int long long int
8. long long unsigned int
9. long int
10. char
11. float
12. double

2.2 Πρωτόκολλα αναπαράστασης Συμβόλων αποσφαλμάτωσης

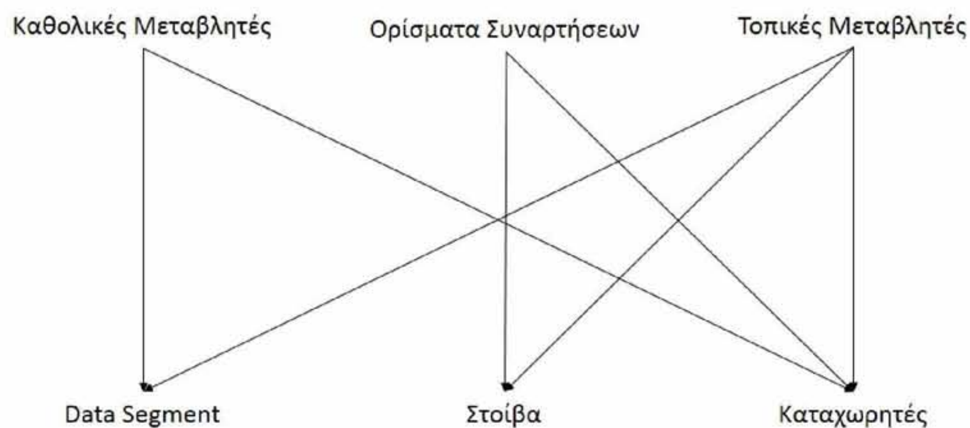
Κατά τη μεταγλώττιση ενός προγράμματος ο προγραμματιστής μπορεί να ορίσει στο μεταγλωττιστή αν επιθυμεί η όχι σύμβολα αποσφαλμάτωσης. Η πληροφορία των συμβόλων αυτών χρησιμοποιείται από τα προγράμματα αποσφαλμάτωσης (debugger) έτσι ώστε όταν ένα πρόγραμμα έχει πρόβλημα κατά την δημιουργία του ή την επέκταση/συντήρηση του να μπορεί ο προγραμματιστής να το τρέχει βήμα βήμα και να δει που εντοπίζεται το πρόβλημα. Χωρίς τα σύμβολα αναπαράστασης δεν θα μπορούσε ο debugger να γνωρίζει τις μεταβλητές ενός προγράμματος καθώς και σε ποιά γραμμή του αρχικού κώδικα βρίσκεται η τρέχουσα εκτέλεση. Σε Linux συστήματα αυτό γίνεται με το όρισμα στο μεταγλωττιστή gcc (-g). Το -g δημιουργεί πληροφορίες αποσφαλμάτωσης ανάλογα με το σύστημα που τρέχει (stabs, COFF, XCOFF, ή DWARF). Το πρωτόκολλο coff χρησιμοποιείται από τον αποσφαλματωτή SDB στα περισσότερα συστήματα System V που προηγούνται τις έκδοσεις 4 , το πρωτόκολλο xcoff χρησιμοποιείται από τον αποσφαλματωτή DBX σε συστήματα IBM RS/6000 ενώ το stabs είναι η έκδοση του GNU και υποστηρίζεται μόνο από τον GNU Debugger (gdb).

Σε Linux συστήματα το προκαθορισμένο είναι το πρωτόκολλο DWARF στο οποίο και επικεντρωθήκαμε. Το πρωτόκολλο DWARF υποστηρίζει τις γλώσσες προγραμματισμού C, C++ και Fortran και το πιο σημαντικό είναι ότι είναι ανεξάρτητο από την αρχιτεκτονική που θα εκτελεσθεί το τελικό εκτελέσιμο, πράγμα που σημαίνει ότι ο κώδικας επεξεργασίας των συμβόλων αποσφαλμάτωσης που δουλεύει σε ALPHA αρχιτεκτονική θα δουλεύει συμμετρικά και σε X86-64.

2.3 Θέσεις Μεταβλητών ενός Προγράμματος

Κάθε μεταβλητή έχει δικιά της θέση στην μνήμη καθώς και εμβέλεια. Με τον όρο εμβέλεια (scope) ορίζεται σε ποιες συναρτήσεις είναι ορατή κάθε μεταβλητή. Αν μια μεταβλητή είναι ορισμένη έξω από όλες τις συναρτήσεις λέμε ότι έχει καθολική εμβέλεια (global scope) και είναι ορατή σε όλες τις συναρτήσεις και για αυτό το λόγο καταλαμβάνει χώρο στη μνήμη του προγράμματος (Data section) η οποία ορίζεται από τον μεταγλωττιστή κατά την ώρα της μεταγλώττισης. Εκτός από τις μεταβλητές καθολικής εμβέλειας έχουμε και τις μεταβλητές τοπικής εμβέλειας (local scope), οι οποίες αποτελούνται από όλες τις μεταβλητές που ορίζονται μέσα σε μια συνάρτηση καθώς επίσης και από τα ορίσματα της συνάρτησης (formal parameters). Για το αν

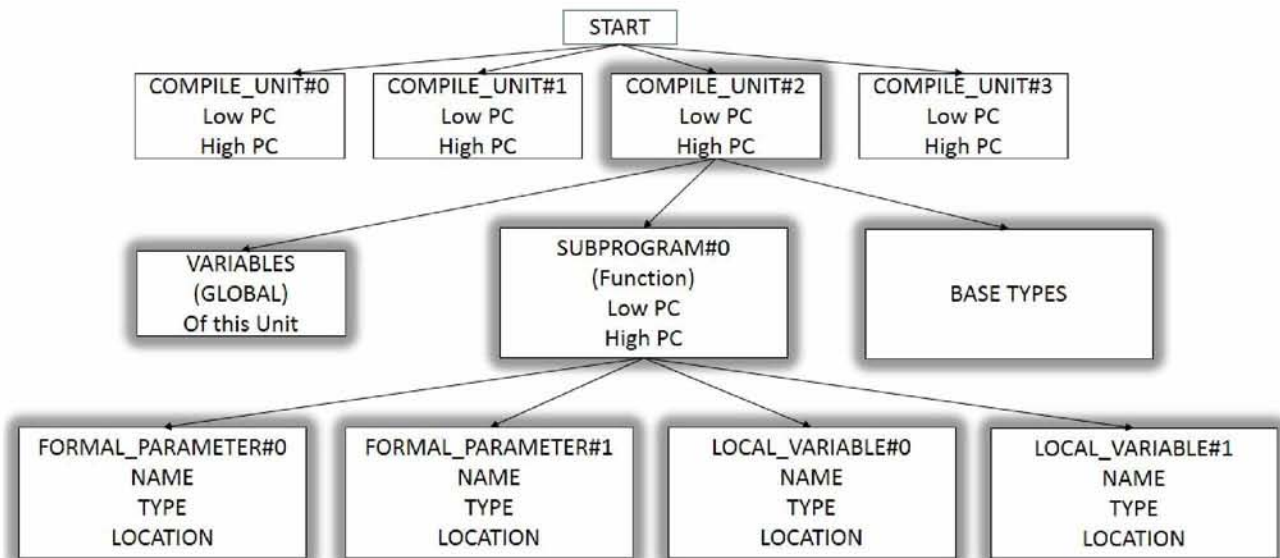
κάποια μεταβλητή αποθηκευτεί στη μνήμη (είτε στην στοίβα είτε στο Data Section ανάλογα με την εμβέλεια της) ή σε κάποιο καταχωρητή εξαρτάται από τις βελτιστοποιήσεις του μεταγλωττιστή καθώς επίσης και αν έχει οριστεί σαν Volatile. Για να βρεθεί η εμβέλεια κάθε μεταβλητής πρέπει να γίνει η κατάλληλη επεξεργασία των συμβόλων αποσφαλμάτωσης και ύστερα να γίνουν πράξεις με τα ανάλογα offsets στην στοίβα, νούμερα καταχωρητών και ιδεατές διευθύνσεις για να βρεθεί η τιμή κάθε μεταβλητής τη στιγμή που έγινε το σφάλμα στο συγκεκριμένο σημείο στον κώδικα της εφαρμογής. Συνοπτικά η εμβέλεια των μεταβλητών και οι θέσεις που μπορούν να βρίσκονται φαίνονται στο επόμενο σχήμα.



Σχήμα 2.1: Κατανομή θέσης αποθήκευσης με βάση την εμβέλεια

2.4 Περιγραφή του πρωτοκόλλου DWARF

Το πρωτόκολλο DWARF ακολουθεί δεντρική μορφή όπως φαίνεται στο ακόλουθο σχήμα:



Σχήμα 2.2: Δεντρική αναπαράσταση του πρωτοκόλλου DWARF

Το κομμάτι που πρέπει να επεξεργαστούμε από τα σύμβολα αποσφαλμάτωσης είναι το .debug_info. Αυτό αποτελείται από τα Debugging Information Entry (DIE) που κάθε ένα έχει ένα αναγνωριστικό tag που καθορίζει πάνω σε τι αναφέρετε το DIE καθώς επίσης και μια λίστα ιδιοτήτων που περιγράφουν κάθε DIE [2]. Για να επεξεργαστεί κάποιος το κομμάτι .debug_info κάθε DIE έχει (αν υπάρχουν) αδέρφια και παιδιά, ορίζοντας έτσι στο ίδιο επίπεδο τα αδέρφια και σε ένα επίπεδο πιο κάτω τα παδιά. Για να μετακινηθεί κανείς ανάμεσα σε αδέρφια και παιδιά χρησιμοποιεί τις συναρτήσεις της libdwarf με όρισμα το TAG που ψάχνει πχ αν ψάχνει μεταβλητές σε ένα επίπεδο ψάχνει το TAG DW_TAG_variable ενώ αν ψάχνει συναρτήσεις το TAG DW_TAG_subprogram. Τα Compile Units είναι το πρώτο επίπεδο στο κομμάτι .debug_info. Ορίζουν κάθε αρχείο κώδικα που συμμετέχει στο τελικό εκτελέσιμο και ανάλογα με το pc Address που θα δωθεί ελέγχετε και βρίσκεται το κατάλληλο Compile Unit. Για την παρουσίαση του πρωτοκόλλου DWARF θα χρησιμοποιήσουμε τον εξής κώδικα:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int getter=0;
4
5  void foo(int times){
6          int i,j;
7          for(i=0;i<times;i++){
8                  j=i;
9                  printf("[Val] %d\n",j);
10
11         }
12 }
13
14
15 int main(int argc, char *argv[]){
16   getter=11;
17   foo(getter);
18   return 0;
19 }
```

Αν μεταγλωτίσουμε το κώδικα αυτό παίρνουμε τις εξής πληροφορίες:

```
.debug_info

COMPILE_UNIT<header overall offset = 0x00000000>:
< 0><0x0000000b>  DW_TAG_compile_unit
                    DW_AT_producer          "GNU C 4.4.3"
                    DW_AT_language          DW_LANG_C89
                    DW_AT_name              "example.c"
                    DW_AT_comp_dir          "/home/pamitsis"
                    DW_AT_low_pc            0x00400524
                    DW_AT_high_pc           0x00400590
                    DW_AT_stmt_list         0x00000000
```

Η διαθέσιμη πληροφορία από το πιο πάνω παράδειγμα είναι ότι αν ο Program Counter του προγράμματος μας βρίσκεται ανάμεσα από τις διευθύνσεις 0x00400524 και 0x00400590 τότε ανήκει στο αρχείο example.c που βρίσκεται στην διαδρόμή /home/pamitsis/test.c και έγινε μεταγλώττιση με τον GNU Compiler χρησιμοποιόντας το πρότυπο C89. Η παραπάνω έξοδος έγινε από το εργαλείο dwarfdump· παρατηρήστε ότι δίπλα στο DW_AT_compile_unit υπάρχουν οι αριθμοί < 0><0x0000000b> που σημαίνουν ο πρώτος το επίπεδο που βρισκόμαστε (0 άρα στην αρχή της δεντρικής μορφής του DWARF) και το offset που ορίζεται το συγκεκριμένο DIE

είναι το0x0000000b. Το offset αυτό είναι το αναγνωριστικό κάθε DIE και θα χρησιμοποιηθεί από οποιοδήποτε άλλο DIE για να δείξει σε αυτό.

Ύστερα ακολουθούν οι τύποι των μεταβλητών, οι καθολικές μεταβλητές και οι συναρτήσεις. Πιο αναλυτικά οι τύποι μεταβλητών (DW_TAG_base_type) ορίζονται ως εξής:

```
< 1><0x00000050>    DW_TAG_base_type
                      DW_AT_byte_size          0x00000002
                      DW_AT_encoding           DW_ATE_signed
                      DW_AT_name               "short int"
< 1><0x00000057>    DW_TAG_base_type
                      DW_AT_byte_size          0x00000004
                      DW_AT_encoding           DW_ATE_signed
                      DW_AT_name               "int"
< 1><0x0000005e>    DW_TAG_base_type
                      DW_AT_byte_size          0x00000008
                      DW_AT_encoding           DW_ATE_signed
                      DW_AT_name               "long int"
```

Ο τύπος των ακέραιων μεταβλητών "int" ορίζεται στο DIE <0x00000057> και έχει μέγεθος 4 bytes. Αντίστοιχα ο τύπος long int ορίζεται στο DIE <0x0000005e> και έχει μέγεθος 8 bytes. Αυτό ισχύει για όλες τους τύπους μεταβλητών όχι μόνο για τις παραπάνω. Ακολούθως όταν τελειώσουμε το ψάξιμο του Compile Unit για του τύπους δεδομένων και εφόσον είμαστε στο πρώτο επίπεδο ψάχνουμε για τις global μεταβλητές με το TAG DW_TAG_variable, το οποίο χρησιμοποιούν και οι local μεταβλητές με τη διαφορά όμως του επιπέδου που βριοκόμαστε. Έτσι για την global μεταβλητή παίρνουμε:

```
< 1><0x00000113>    DW_TAG_variable
                      DW_AT_name               "getter"
                      DW_AT_decl_file          0x00000001 /home/pamitsis/example.c
                      DW_AT_decl_line          0x00000003
                      DW_AT_type               <0x00000057>
                      DW_AT_external           yes(1)
                      DW_AT_location           DW_OP_addr 0x00601030
```

Το όνομα της μεταβλητής είναι "getter" δηλώθηκε στο "/home/pamitsis/example.c" στη γραμμή 3 του αρχικού κώδικα και είναι τύπου DIE 0x00000057 δηλαδή "int" και βρίσκεται στην διεύθυνση 0x00601030 του Data Segment. Εφόσον δεν υπάρχει άλλη global μεταβλητή μπορούμε να ψάξουμε τις συναρτήσεις του Compile Unit για να δούμε σε ποιά συνάρτηση ανήκει η εντολή/υπολογισμός που εισήχθη ένα σφάλμα.

Έστω ότι το σφάλμα έγινε όταν ο Program Counter ήταν στην τιμή 0x00400528, ψάχνουμε

όλα τα TAG DW_TAG_subprogram και ελέγχουμε αν η δοσμένη τιμή του Program Counter βρίσκεται στην συνάρτηση αυτή. Στο δικό μας παράδειγμα αυτός ο έλεγχος θα μας επιστρέψει το ακόλουθο DIE:

```
< 1><0x00000083>    DW_TAG_subprogram
                      DW_AT_external              yes(1)
                      DW_AT_name                  "foo"
                      DW_AT_decl_file             0x00000001 /home/pamitsis/example.c
                      DW_AT_decl_line             0x00000005
                      DW_AT_prototyped            yes(1)
                      DW_AT_low_pc                0x00400524
                      DW_AT_high_pc               0x00400563
                      DW_AT_frame_base            <loclist with 3 entries follows>
                        [ 0]<lowpc=0x00400524><highpc=0x00400525>DW_OP_breg7+8
                        [ 1]<lowpc=0x00400525><highpc=0x00400528>DW_OP_breg7+16
                        [ 2]<lowpc=0x00400528><highpc=0x00400563>DW_OP_breg6+16
                      DW_AT_sibling               <0x000000cb>
```

Η συνάρτηση που εισήχθη το σφάλμα ήταν η "foo" που ορίζεται στην γραμμή 5 με εντολές από την θέση 0x00400524 εως 0x00400563 του τελικού assembly κώδικα και τέλος η επόμενη συνάρτηση από αυτή βρίσκεται στο DIE 0x000000cb (sibling) όπου βρίσκεται η συνάρτηση main. Από τα παραπάνω πολύ σημαντική για εμάς είναι η ιδιότητα DW_AT_frame_base καθώς για να βρεθεί μια local μεταβλητή πρέπει να βρούμε την αρχή του στιγμιότυπου της στοίβας που εκτελείται αυτή τη στιγμή και ορίζεται σε αυτό το σημείο. Να σημειωθεί εδώ ότι ο καταχωρητής που αποθηκεύεται η αρχή του στιγμιοτύπου μπορεί να είναι διαφορετικός ανάλογα με το PC Address που βρισκόμαστε καθώς μπορεί να χρησιμοποιηθεί από τον μεταγλωττιστή και σε κάποιο σημείο της συνάρτησης συνεπώς δεν είναι μόνο μια η τιμή που μπορεί να πάρουμε για την ιδιότητα DW_AT_frame_base όπως εδώ. Η δική μας PC Address είναι στην τελευταία καταχώρηση του Frame Base και το στιγμιότυπο που μας ενδιαφέρει ξεκινά από τη διεύθυνση που δείχνει ο καταχωρητής 6 προσθέτοντας τη τιμή 16.

```
<   2><0x000000a4>    DW_TAG_formal_parameter
                        DW_AT_name                "times"
                        DW_AT_decl_file           0x00000001 /home/pamitsis/example.c
                        DW_AT_decl_line           0x00000005
                        DW_AT_type                <0x00000057>
                        DW_AT_location            DW_OP_fbreg -36
< 2><0x000000b2>      DW_TAG_variable
                        DW_AT_name                "i"
                        DW_AT_decl_file           0x00000001 /home/pamitsis/example.c
```

```
                      DW_AT_decl_line          0x00000006
                      DW_AT_type               <0x00000057>
                      DW_AT_location           DW_OP_fbreg -20
< 2><0x000000be>    DW_TAG_variable
                      DW_AT_name               "j"
                      DW_AT_decl_file          0x00000001 /home/pamitsis/example.c
                      DW_AT_decl_line          0x00000006
                      DW_AT_type               <0x00000057>
                      DW_AT_location           DW_OP_fbreg -24
```

Για να ψάξουμε τα ορίσματα της συνάρτησης ψάχνουμε το TAG DW_TAG_formal_parameter και βρίσκουμε τη μεταβλητή times που είναι τύπου "int" (DIE 0x00000057) και είναι αποθηκευμένη για αυτό το στιγμιότυπο εκτέλεσης στην θέση που υπολογίστηκε η Frame Base προστιθέμενη με το offset -36 (DW_OP_fbreg -36). Η διεύθυνση που θα πρέπει να διαβάσουμε λοιπόν για την τιμή της μεταβλητής times είναι: Η τιμή που διαβάστηκε στον καταχωρητή 6 -20 (+16-36=-20). Τέλος οι local μεταβλητές "i" και "j" με τον ίδιο τρόπο ανάλυσης βρίσκονται στις θέσεις Frame Base -24 και Frame Base -20) αντίστοιχα αφού η στοίβα αυξάνεται προς τα κάτω στην αρχιτεκτονική X86-64 που χρησιμοποιούμε για το παράδειγμα αυτό.

Συνοψίζοντας λοιπόν οι global μεταβλητές αν βρίσκονται στο Data Segment στο πρωτόκολλο DWARF στην ιδιότητα DW_AT_location θα υπάρχει το όρισμα DW_OP_addr ακολουθούμενο από τη διεύθυνση που χρειαζόμαστε. Οι local μεταβλητές αν βρίσκονται στη στοίβα ορίζουν το offset από το τρέχων στιγμιότυπο με τη τιμή DW_OP_fbreg <offset> στην ιδιότητα DW_AT_location ενώ αν είναι static αποθηκεύονται στο Data Segment και στην ιδιότητα DW_AT_location θα υπάρχει το όρισμα DW_OP_addr ακολουθούμενο από τη διεύθυνση που χρειαζόμαστε όπως και οι global μεταβλητές. Τέλος αν βρίσκεται μεταβλητή οποιασδήποτε εμβέλειας σε καταχωρητή τότε η τιμή της ιδιότητας DW_AT_location είναι DW_OP_regX όπου X το νούμερο του καταχωρητή στον οποίο βρίσκεται η μεταβλητή.

2.5 Προσπέλαση των συμβόλων αποσφαλμάτωσης από τον προσομοιωτή

Για να μπορέσουμε να διαβάσουμε το πρωτόκολλο DWARF έπρεπε να εισάγουμε τη βιβλιοθήκη libdwarf. Η βιβλιοθήκη αυτή για να δουλέψει πρέπει να μπορεί να επεξεργαστεί τα δεδομένα από το εκτελέσιμο αρχείο μας, κάτι το οποίο μπορεί να γίνει με τις βιβλιοθήκες libelf και libbfd. Και οι δυο προσφέρουν συναρτήσεις πρόσβασης στο εκτελέσιμο αρχείο χωρίς να εξαρτάται από

την αρχιτεκτονική στην οποία προορίζεται όμως εμείς χρησιμοποιήσαμε την libelf καθώς αυτή χρησιμοποιεί και ο Gem5 για να έχει εγγενώς πρόσβαση σε όλα τα εκτελέσιμα αρχεία (όταν πχ βρίσκεται σε Syscall Emulation Mode ή για να διαβάσει τα σύμβολα του πυρήνα της εκάστοτε αρχιτεκτονικής που προσομοιώνεται). Για να χρησιμοποιήσουμε την libdwarf εγγενώς μέσα από το προσομοιωτή χρειαστήκαμε έναν αντίγραφο του project elftoolchain το οποίο περιέχει και τις δυο βιβλιοθήκες.

Αφού βρήκαμε τις απαραίτητες βιβλιοθήκες για να αναλύσουμε τα σύμβολα αποσφαλμάτωσης ακολουθήσαμε τα βήματα που ακολουθεί και το εργαλείο addr2line. Αυτό που κάνει το εργαλείο αυτό είναι ότι δοθείσης μια διεύθυνσης του Program Counter βρίσκει το σημείο του κώδικα το οποίο αυτή αντιστοιχεί βγάζοντας σαν πληροφορία το αρχείο του κώδικα και τη γραμμή του που αντιστοιχούν σε αυτή τη τιμή. Στην ίδια φιλοσοφία λοιπόν βρίσκουμε και εμείς το Compile Unit που εμπεριέχει την τιμή του Program Counter που έγινε το σφάλμα και το αντιστοιχούμε στο σημείο του κώδικα που εμφανίστηκε. Ύστερα αναλύουμε το Compile Unit για να βρούμε τις καθολικές μεταβλητές που ορίζονται σε αυτό καθώς και τη συνάρτηση που έγινε το σφάλμα και στο τέλος για εκείνη τη συνάρτηση βρίσκουμε τα ορίσματα και τις τοπικές μεταβλητές. Αφού βρούμε τα παραπάνω μένει μόνο να διαβάσουμε τις θέσεις που μας έχει υποδείξει το πρωτόκολλο DWARF καθώς έχουμε άμεση πρόσβαση τόσο στους καταχωρητές όσο και στην μνήμη.

ΚΕΦΑΛΑΙΟ 3

Εύρεση Εκτελέσιμου αρχείου κατά τον χρόνο Εκτέλεσης

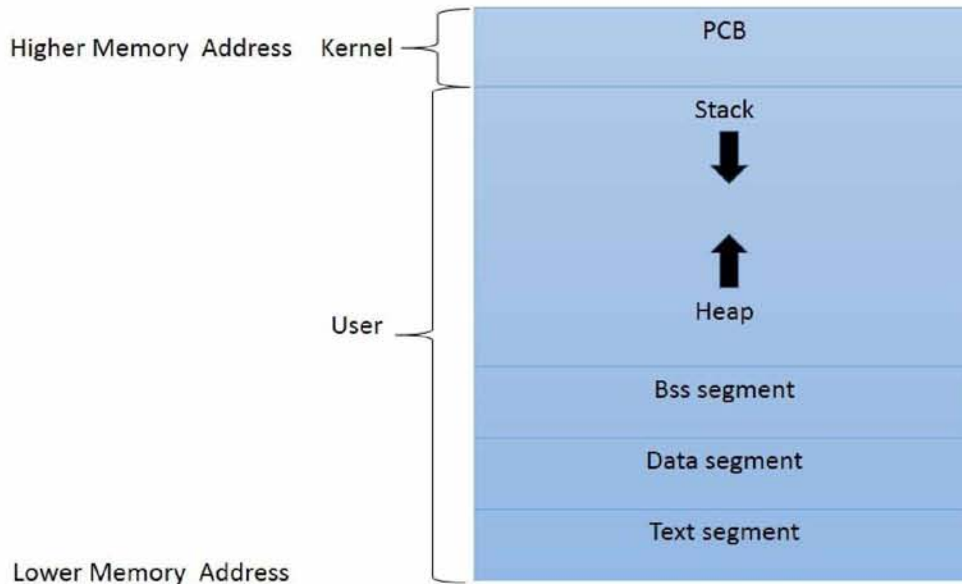
3.1 Ο Πυρήνας του Linux

Όπως αναφέραμε και στην εισαγωγή για να προσπελάσουμε τα δεδομένα των συμβόλων αποσφαλμάτωσης πρέπει να έχουμε άμεση πρόσβαση στο εκτελέσιμο αρχείο. Ο προσομοιωτής Gem5 όμως βλέπει ό,τι είναι ορατό σε επίπεδο υλικού, ενώ εμείς χρειαζόμαστε στοιχεία τα οποία είναι γνωστά από το επίπεδο του λειτουργικού και πάνω. Έτσι πρέπει να βρούμε τον τρόπο να διαβάσουμε τις πληροφορίες που θέλουμε χρησιμοποιόντας μόνο το υλικό στο οποίο και έχουμε άμεση πρόσβαση με συναρτήσεις που ελέγχουν κάθε επεξεργαστή,την μνήμη κτλ.

Το πρώτο βήμα για την εύρεση του εκτελέσιμου από τον προσομοιώτη ήταν η μελέτη της αρχιτεκτονικής του πυρήνα του Linux και πιο συγκεκριμένα πως μεταχειρίζεται τις διεργασίες. Στην παρούσα διπλωματική επειδή θέλαμε να δουλεύει το εργαλείο μας τόσο σε ALPHA όσο και σε X86-64 αρχιτεκτονική όπως και ο Gem FI χρησιμοποιούμε τις εκδόσεις 2.6.27.6 για ALPHA και 2.6.28.4 για X86-64, μεταγλωττισμένες με υποστήριξη SMP (Symmetric Multiprocessor). Αυτές είναι και οι προτεινόμενες εκδόσεις και ρυθμίσεις για να δουλεύουμε με τη σουίτα μετροπρογράμματων PARSEC-Princeton Application Repository for Shared-Memory Computers [1] στον προσομοιωτή Gem5. Η σουίτα αυτή στόχο έχει να προσομοιώνει μεγάλης κλίμακας πολυνηματικά προγράμματα ώστε να υπάρχει ένα μέτρο σύγκρισης στις προσπάθειες για βελτίωση του υλικού καθώς δημιουργούν μεγάλο φόρτο εργασίας για τον μηχάνημα υπό προσομοίωση και μπορεί να διακρίνει κανείς τις βελτιώσεις αν υπάρχουν.

3.2 Δομή Διεργασίας

Το ακόλουθο σχήμα δείχνει πως ορίζεται μια διεργασία στο Linux και τι περιλαμβάνει.



Σχήμα 3.1: Διάταξη μνήμης μιας διεργασίας

Το Process Control Block είναι μια δομή δεδομένων του πυρήνα η οποία διατηρεί διατηρεί όλα τα ¨λογιστικά¨ στοιχεία μιας διεργασίας που είναι απαραίτητα στον πυρήνα του λειτουργικού για να τη διαχειριστεί. Για παράδειγμα διατηρεί τον αναγνωριστικό αριθμό της διεργασίας που παίρνει από τον πυρήνα (PID -Process Identifier) όπως επίσης και το όνομα του εκτελέσιμου χωρίς το μονοπάτι που ανήκει στον δίσκο.

Ο πυρήνας του Linux ορίζει την δομή του Process Control Block με το task_struct (Α΄- Κώδικας Πυρήνα για Περιγραφή Διεργασίας). Μέσα σε αυτή τη δομή υπάρχουν κάποιοι δείκτες σε θέσεις μνήμης οι οποίες είναι απαραίτητες για την εύρεση της διεύθυνσης στην οποία βρίσκεται το εκτελέσιμο αρχείο μας αρά αυτό που πρέπει να γίνει είναι να βρεθεί το Process Control Block για κάθε αρχιτεκτονική και να διαβάστουν οι δείκτες στις άλλες δομές. Για να γίνει αυτό βάζουμε κάποια σταθερά ¨σύμβολα¨ στον πυρήνα μας τα οποία μπορεί να βρεί ο Gem5. Για παράδειγμα έστω ότι θέλουμε να βρούμε το αναγνωριστικό μιας διεργασίας στον πυρήνα (PID) που ορίζεται στο struct task_struct και είναι ένας ακέραιος αριθμός από το 0-32768, θα πρέπει να γίνουν τα εξής:

1. Προσθήκης της απόσταση της διεύθυνσης της PID από την αρχική διεύθυνση του task_struct δηλαδή το offset από την αρχική διεύθυνση ως εξής στον πυρήνα

```
const int task_struct_pid = offsetof(struct task_struct, pid);
```

2. Ανάγνωση από τον προσομοιωτή του παραπάνω offset (όπου tc είναι ένας δείκτης για το συγκεκριμένο νήμα πυρήνα- Kernel Thread):

```
1 FSTranslatingPortProxy &vp = tc->getVirtProxy();
2 tc->getSystemPtr()->kernelSymtab->findAddress("thread_info_size", addr));
3 int pid_off = vp.readGtoH<int32_t>(addr);
```

3. Όταν βρείς το PCB πήγαινε να διαβάσεις από τη διεύθυνση του PCB + Offset

```
uint16_t pd = vp.readGtoH<uint16_t>(task + pid_off);
```

Με τον ίδιο τρόπο δουλεύει και το σκεπτικό για την εύρεση του εκτελέσιμου αρχείο στον δίσκο χρησιμοποιώντας το PCB ή οποιαδήποτε άλλη διεύθυνση διαβαστεί από κάποιον δείκτη προσθέτοντας τα διάφορα offsets, όπως αναλύεται στην επόμενη ενότητα.

3.3 Εύρεση του PCB σε κάθε Αρχιτεκτονική

3.3.1 Εύρεση του PCB σε ALPHA Αρχιτεκτονική

Σε ALPHA Αρχιτεκτονική ο προσομοιωτής Gem 5 έχει έναν καταχωρητή ορισμένο ειδικά για να διατηρεί τη θέση από που μπορεί να βρεθεί το task _struct και είναι ο καταχωρητής AlphaISAIPR::_PALtemp23. Από εκεί διαβάζουμε την τιμή του και εφαρμόζοντας τη μάσκα ~0x3fff, η οποία μηδενίζει τα 14 τελευταία bits και έτσι πηγαίνουμε στην αρχή του PCB από που μπορούμε να βρούμε με τα κατάλληλα offsets όποια διεύθυνση θέλουμε. Ακριβώς την ίδια διαδικασία Masking ακολουθεί και ο πυρήνας του Linux για να βρει την τρέχουσα διεργασία η οποία στον κώδικα τού αναφέρεται ως "current".

3.3.2 Εύρεση του PCB σε X86-64 Αρχιτεκτονική

Σε X86-64 Αρχιτεκτονική υπάρχει ένας ένας καταχωρητής ειδικού σκοπού, ο GS στον οποίο υπάρχει ένας μόνιμος δείκτης για τη δομή δεδομένων struct x8664_pda. Σε αυτή τη δομή δεδομένων υπάρχουν διάφοροι δείκτες για την διεργασία που τρέχει κάθε στιγμή πάνω στον επεξεργαστή. Από τα πεδία του struct x8664_pda που είναι ορισμένο στο (arch/x86/include/pda.h) εμάς μας ενδιαφέρει το πεδίο:

```
unsigned long kernelstack; /* 16 top of kernel stack for current */
```

καθώς επίσης και οι τιμές THREAD_SIZE και PDA_STACKOFFSET που ορίζονται επίσης στο ίδιο αρχείο. Έχοντας τις παραπάνω τρεις ακέραιες τιμές που ορίζουν τα απαραίτητα offsets κάποιος μπορεί να ακολουθήσει τη μέθοδο που ακολουθεί και ο πυρήνας για να βρει την αρχή του PCB η οποία είναι οριομένη στο arch/x86/include/thread_info.h) και είναι η ακόλουθη:

```
1  /* how to get the thread information struct from ASM */
2  \#define GET_THREAD_INFO(reg)
3          movq \%gs:pda_kernelstack,reg ; \
4          subq $(THREAD_SIZE-PDA_STACKOFFSET),reg
5
6  \#endif
```

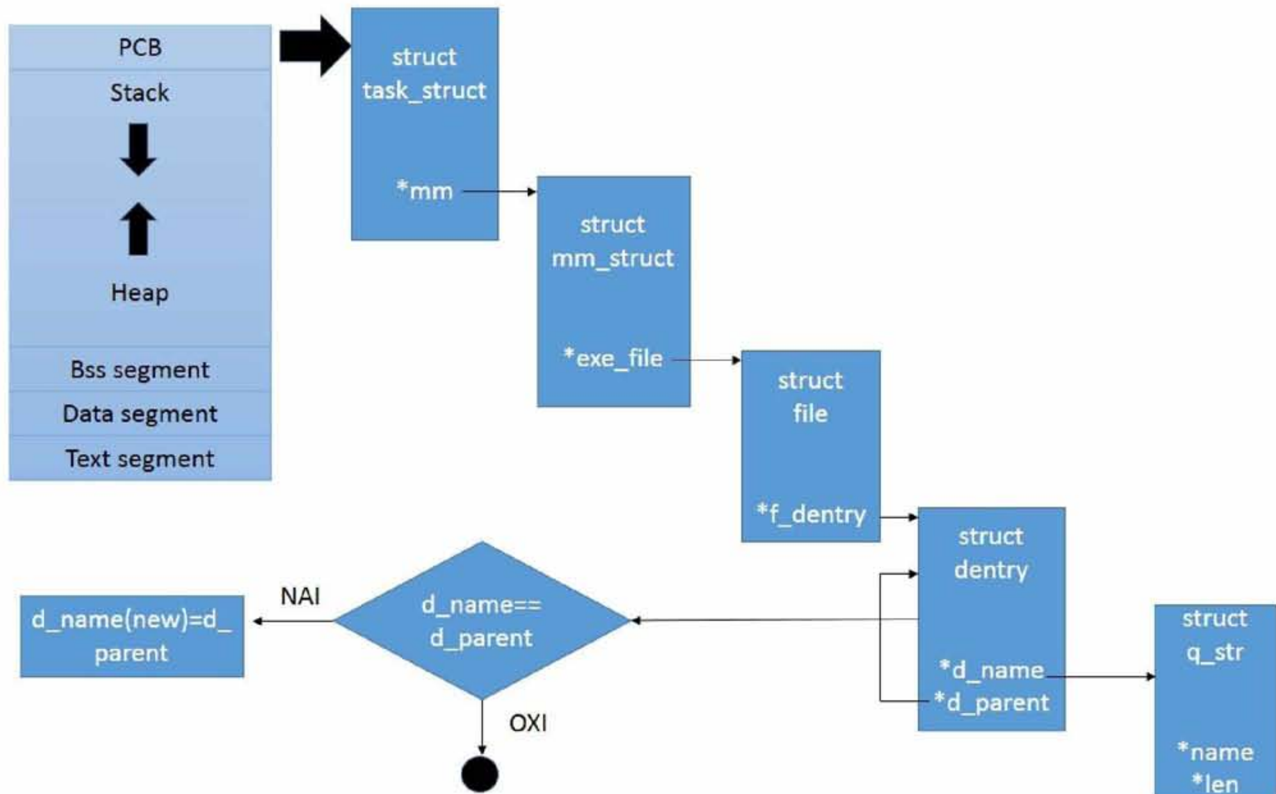
Οπότε ο κώδικας του προσομοιωτή για την εύρεση της διεργασίας γίνεται όπως δείχνεται παρακάτω:

```
1  FSTranslatingPortProxy &vp = tc->getVirtProxy();
2  #if THE_ISA == ALPHA_ISA
3          Addr ksp;
4          Addr temp_addr=tc->readMiscRegNoEffect(AlphaISA::IPR_PALtemp23);
5          PortProxy &p=tc->getPhysProxy();
6          p.readBlob(temp_addr,(uint8_t *)&ksp,sizeof(Addr));
7          Addr base = ksp & ~0x3fff;
8  #else
9          Addr ksp=tc->readMiscReg(X86ISA::MISCREG_KERNEL_GS_BASE);
10         Addr base=vp.readGtoH<Addr>(ksp+pda_kernelstack)-pda_thread_size+pda_stackoffset;
11 #endif
```

3.4 Δομές δεδομένων για την εύρεση του εκτελέσιμου Αρχείου

Εφόσον έχει βρεθεί η αρχή του PCB μένει μόνο να βρεθούν οι δείκτες που δείχνουν στο πλήρες μονοπάτι που βρίσκεται το εκτελέσιμο. Όλα τα Unix συστήματα υλοποιούν ένα ειδικό σύστημα αρχείων το ¨Procfs¨ ή ¨Proc¨ το οποίο αναπαριστά πληροφορία για κάθε διεργασία που εκτελείται. Για να διαβάσει κάποιος το procfs για κάποια διεργασία αρκεί να γνωρίζει την τιμή του PID της διεργασίας που ψάχνει. Μαζί με όλες τις άλλες πληροφορίες που μπορεί κάποιος να δει για μια διεργασίας θα βρεί και τον συμβολικό σύνδεσμο /proc/PID/exe που είναι αυτό

που ψάχνουμε καθώς περιέχει το πλήρες μονοπάτι στο δίσκο για το εκτελέσιμο αρχείο. Δυστυχώς όμως ο Gem5 δεν έχει πρόσβαση στο procfs, όμως μπορεί να ακολουθήσει τα ίδια βήματα με αυτό για να βρει το εκτελέσιμο αρχείο στο δίσκο από το PCB. Η διαδρομή που πρέπει να ακολουθηθεί στον πυρήνα για να βρεθεί το μονοπάτι σχηματικά είναι η ακόλουθη:



Σχήμα 3.2: Δομές δεδομένων για την εύρεση του μονοπατιού

Για να βρεθεί το εκτελέσιμο αρχείο στο δίσκο χρειαζόμαστε τις εξής δομές δεδομένων του πυρήνα:

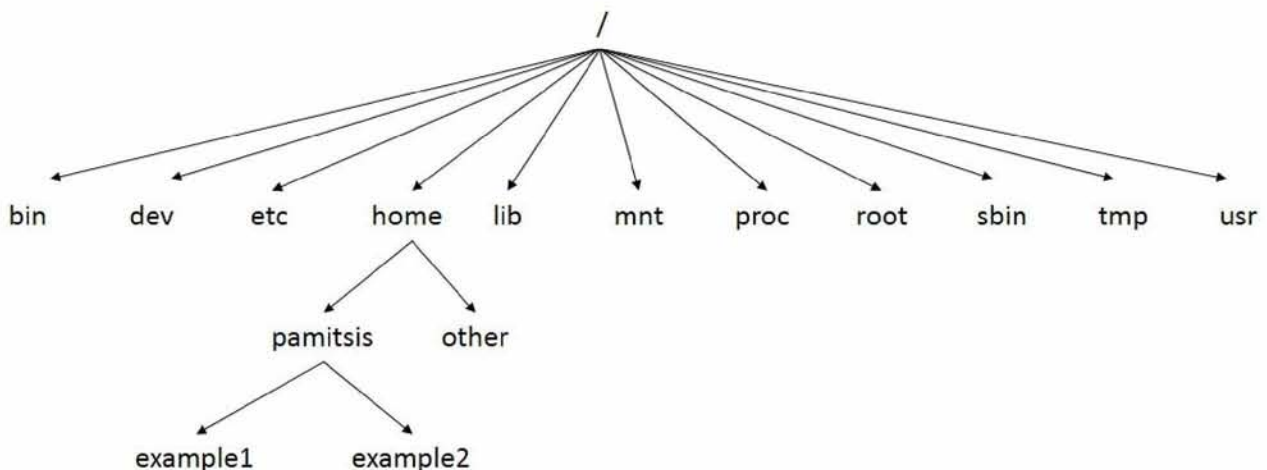
1. struct task_struct: Αναπαριστά το PCB και ορίζεται στο include/linux/sched.h .
2. struct mm_struct: Αναπαριστά την ιδεατή μνήμη μιας διεργασίας και ορίζεται στο include/linux/mm_types.h .
3. struct file: Κάθε ανοιχτό αρχείο,socket κτλ αναπαριστώνται με τη δομή struct file που ορίζεται στο include/linux/fs.h .

4. struct dentry: Υλοποιείται από το Virtual File System (VFS) για να εκτελεί λειτουργίες για τους καταλόγους όπως την εύρεση του μονοπατιού. Ορίζεται στο include/linux/dcache.h.

5. struct q_str: Αναπαριστά το όνομα και το μέγεθος της καταχώριοης του struct dentry και ορίζεται στο include/linux/dcache.h .

Για να βρει κάποιος το εκτελέσιμο αρχείο πρέπει να βρει το PCB όπως αναφέραμε στην προηγούμενη ενότητα και να βρει το offset για τον δείκτη mm ο οποίος οδηγεί στην δομή δεδομένων mm_struct που περιγράφει την ιδεατή μνήμη μιας διεργασίας. Ύστερα σε εκείνη τη δομή δεδομένων πρέπει να βρεθεί το offset για τον δείκτη exe_file ο οποίος αναπαριστά τη δομή stuct_file και σε εκείνη το offset για τον δείκτη f_dentry ο οποίος αναπαριστά τη δομή stuct_dentry που αντιπροσωπεύει μια καταχώρηση καταλόγου και τέλος σε κάθε διεύθυνση καταλόγου να διαβαστεί το μέγεθος του ονόματος και το όνομα της καταχώρησης με την δομή δεδομένων struct q_str από τις μεταβλητές len και name αντίστοιχα.

Για να γίνει καλύτερα αντιληπτό η αναδρομική διαδικασία που ακολουθείται θα αναφερθούμε στην δομή του συστήματος αρχείων (File System) του Linux το οποίο έχει δεντρική υποδομή όπως παρακάτω:



Σχήμα 3.3: Δομή του συστήματος αρχείων του Linux

Για να βρεθεί το πλήρες μονοπάτι για κάθε αρχείο αυτό που κάνει ο πυρήνας είναι να πηγαίνει για κάθε καταχώρηση καταλόγου(stuct_dentry) να διαβάζει το όνομα της μέσω της δομής (struct q_str) και να κάνει ανδρομικά την ίδια δουλειά για τον κόμβο πατέρα κάθε καταχώρησης μέχρι

να φτάσει στην κορυφή του δέντρου. Για να ελέγχει αν έφτασε στην κορυφή του δέντρου ελέγχει τον εκάστοτε κόμβο με την ακόλουθη μακροεντολή:

```
#define IS_ROOT(x) ((x) == (x)->d_parent)
```

Παράδειγμα: Έστω ότι θέλουμε να προσδιορίσουμε τη θέση του example1 στο δίσκο, το οποίο είναι φύλλο (τερματικός κόμβος στο δέντρο του συστήματος αρχείων του Linux). Αν διαβάσει κανείς τα δεδομένα του d_name η τιμή της μεταβλητής name θα είναι example1, αν διαβάσουμε την τιμή d_name του d_parent θα πάρουμε την τιμή pamitsis κτλ. Άρα για να πάρει κανείς το πλήρες μονοπάτι για το εκτελέσιμο αρχείο example1 θα πρέπει για κάθε καταχώρηση καταλόγου να διαβάζει την τιμή d_name→ name και μετά ο πατέρας του κόμβου αυτού να κάνει το ίδιο αναδρομικά ώσπου η τιμή του δείκτη d_parent να είναι ίδια με την τιμή της θέσης μνήμης της καταχώρισης καταλόγου που εξετάζουμε. Στο παράδειγμα μας κάτι τέτοιο θα επέστρεφε: example1 ⇒ pamitsis ⇒ home και στο home θα τελείωνε η εκτέλεση. Αν κάποιος ήθελε να αποθηκεύσει το μονοπάτι αυτό που θα έπρεπε να γίνει είναι να συμπληρώνει κάθε φορά το όνομα του πατέρα του καταλόγου στην αρχή του string που θα έχει ως έξοδο και θα είχε τα εξής βήματα εκτέλεσης:

1. Αρχικοποίηση κενής συμβολοσειράς: output="".
2. Προσθήκη του ¨/¨ ακολουούμενο από το d_name → name: output="/example1".
3. Έλεγχος αν είναι η ρίζα του δέντρου ⇒ Όχι άρα μεταφορά στον πατέρα.
4. Προσθήκη του ¨/¨ ακολουούμενο από το d_name → name: output="/pamitsis/example1".
5. Έλεγχος αν είναι η ρίζα του δέντρου ⇒ Όχι άρα μεταφορά στον πατέρα.
6. Προσθήκη του ¨/¨ ακολουούμενο από το d_name → name: output="/home/pamitsis/example1".
7. Έλεγχος αν είναι η ρίζα του δέντρου ⇒ Ναι άρα τερματισμός και το μονοπάτι που ανακτήσαμε είναι το "/home/pamitsis/example1" το οποίο και ψάχναμε.

Ο κώδικας του Gem5 που υλοποιεί την πιο πάνω διαδικασία είναι:

```
1 Addr start=this->dentry();
2 FSTranslatingPortProxy &vp = tc->getVirtProxy();
3 Addr top=vp.readGtoH<Addr>(this->dentry()+dentry_struct_parent);
4 string fullpath;
5 while(top!=start){
```

```
6         char current_name[256]={0};
7         CopyStringOut(tc, current_name, vp.readGtoH<Addr>(start\
8         +dentry_struct_qstr+qstr_struct_name), sizeof(current_name));
9         std::string intermediate(current_name);
10        fullpath.insert(0,current_name);
11        fullpath.insert(0,1,'/');
12        start=top;
13        top=vp.readGtoH<Addr>(start+dentry_struct_parent);
14 }
15 return fullpath;
```

ΚΕΦΑΛΑΙΟ 4

Πειραματική Αξιολόγηση

Μόλις ολοκληρώθηκε η σύνδεση της επέκτασης που δημιουργήσαμε με τον GemFI τρέξαμε ένα σύνολο πειραμάτων για να δείξουμε την επίδραση των παροδικών σφαλμάτων στις εφαρμογές MonteCarlo PI και Knapsack που είχαν ξαναδοκιμαστεί ως προς την επίδραση των παροδικών σφαλμάτων χωρίς όμως να υπάρχει αξιολόγηση των σφαλμάτων με βάση τα semantics της [4]. Τα λάθη είναι 6 ειδών και χωρίζονται σε:

1. Float Reg: Πειράματα που το λάθος μπήκε σε καταχωρητή κινητής υποδιαστολής.
2. Int Reg: Πειράματα που το λάθος μπήκε σε ακέραιο καταχωρητή.
3. PC Address: Πειράματα που το λάθος μπήκε στον Program Counter του προγράμματος.
4. Fetch: Πειράματα που το λάθος μπήκε στο Pipeline κατά τη στιγμή που κάνουμε Fetch μια εντολή.
5. Decode: Πειράματα που το λάθος μπήκε στο Pipeline κατά τη στιγμή που κάνουμε decoding τους καταχωρητές μια εντολής.
6. Execute: Πειράματα που το λάθος μπήκε στο Pipeline κατά τη στιγμή που κάνουμε τον υπολογισμό του αποτελέσματος μιας εντολής.

Ο διαχωριομός των αποτελεσμάτων γίνεται ως εξής:

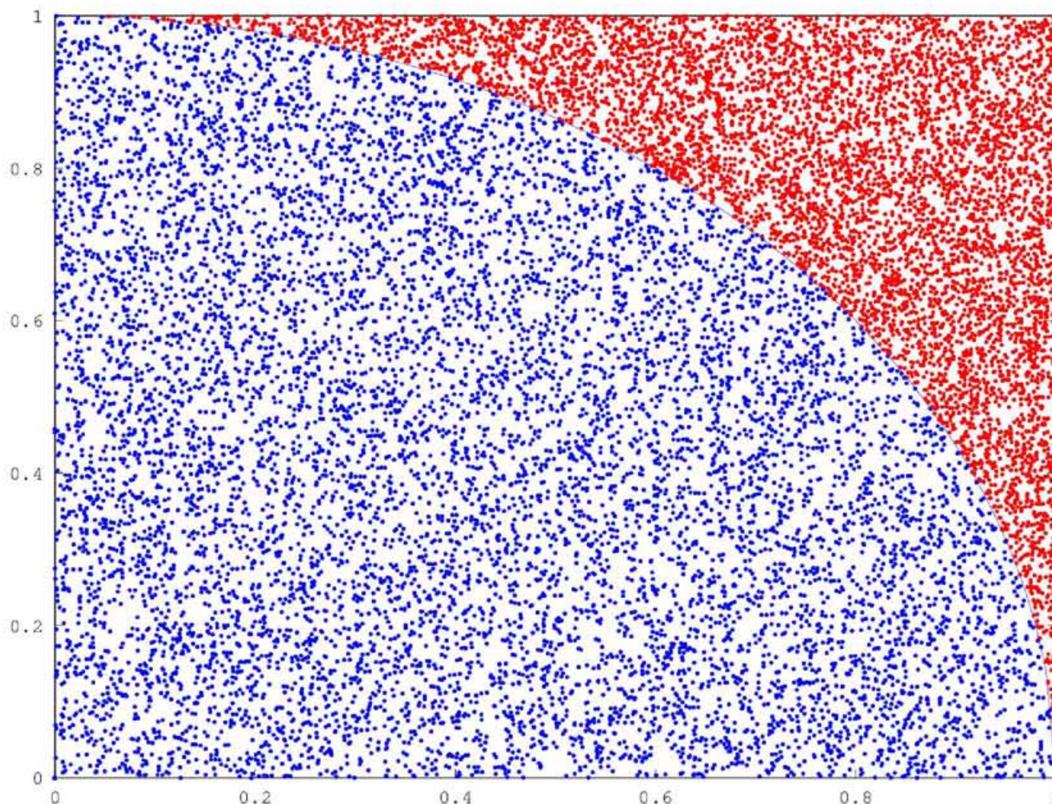
1. Sdc (Silent Data Corruption) : πειράματα που έβγαλαν τελείως διαφορετικό αποτέλεσμα από το αναμενόμενο.
2. Correct: πειράματα που έβγαλαν αποτέλεσμα μέσα σε ένα αποδεκτό εύρος τιμών.

3. Crashed: πειράματα που δεν τερμάτιοε η εκτέλεοη τους λόγο του λάθους.

4. Strict: πειράματα που έβγαλαν αποτέλεσμα σαν να μην υπήρχε λάθος.

4.1 Αξιολόγηση της εφαρμογής MonteCarlo PI

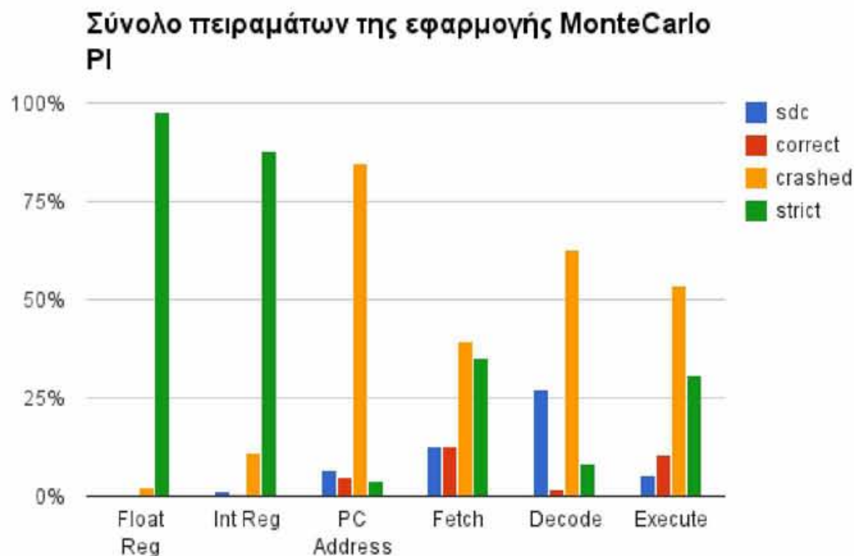
Η εφαρμογή MonteCarlo PI διαλέγει τυχαία δυο σημεία x_i,y_i με μοναδιαίο τετράγωνο. Ύστερα υπολογίζει το άθροισμα $x_i^2 + y_i^2$ και κρατάει σε μια μεταβλητή των αριθμό των σημείων που ικανοποιούν τη σχέση $x_i^2 + y_i^2 < 1$. Τέλος βρίσκει το λόγο $\rho = m/n$ όπου m είναι η μεταβλητή που υπολογίσαμε προηγουμένως και n ο αριθμός των επαναλήψεων που θα κάνουμε έτσι ώστε να προσεγγιστεί το π ως εξής: $\pi = \rho * 4$.



Σχήμα 4.1: Σύνολο Σημείων για την εφαρμογή MonteCarlo PI. Με μπλε απεικονίζονται τα αποδεκτά σημεία και με κόκκινο τα μη αποδεκτά. Πηγή: Wikipedia

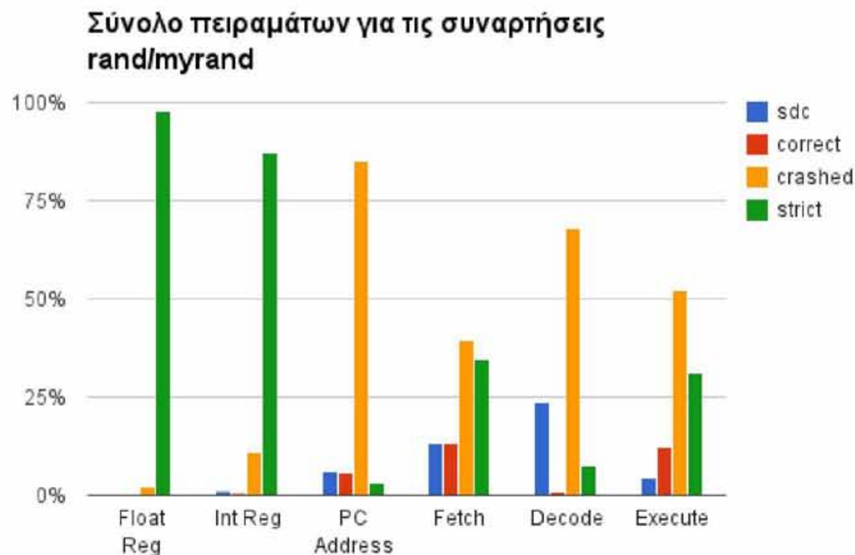
Την εφαρμογή αυτή την τρέξαμε με ένα νήμα και το εκτελέσιμο δεν είχε καθόλου βελτιστοποιήσεις για να μπορέσουμε να διαβάσουμε τις τιμές όλων των μεταβλητών όπως για παράδειγμα του iterator της συνάρτησης MonteCarlo Integrate (που είναι η συνάρτηση υπολογισμού των ρ και m) καθώς θέλουμε να συσχετίσουμε την επίδραση των σφαλμάτων και του βρόγχου επανάληψης.

Στο ακόλουθο γράφημα βλέπουμε ότι τα περισσότερα sdc πειράματα οφείλονται σε λάθος κατά το decoding των καταχωρητών, τα περισσότερα correct πειράματα οφείλονται σε λάθη όταν κάνουμε Fetch μια εντολή. και τέλος ότι τα περισσότερα πειράματα που το λάθος μπήκε στον Program Counter του προγράμματος τερμάτισαν χωρίς αποτέλεσμα. Επίσης αξίζει να σημειωθεί πως λάθη σε ακέραιους καταχωρητές αλλά και καταχωρητές κινητής υποδιαστολής έβγαλαν τα περισσότερα αποτέλεσμα σαν να μην υπήρχε λάθος.



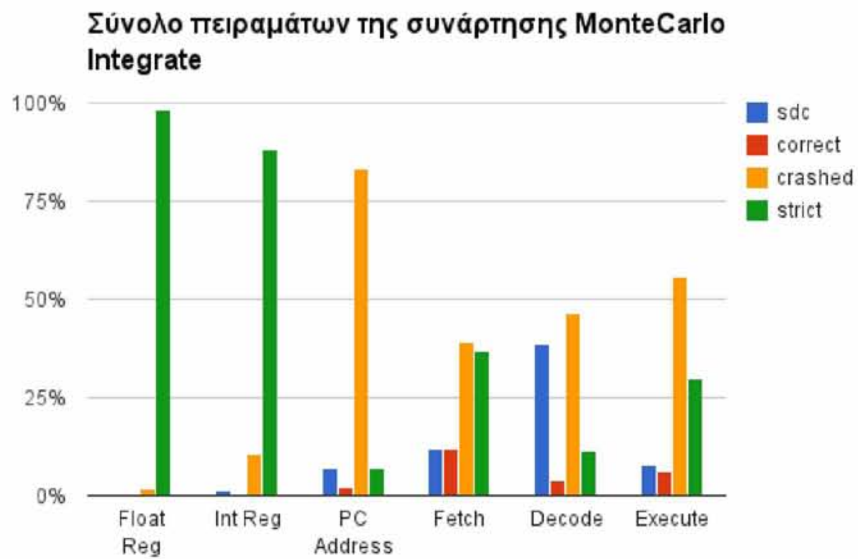
Σχήμα 4.2: Σύνολο προσομοίωσης για την εφαρμογή MonteCarlo PI με την επίδραση κάθε σφάλματος

Το γράφημα που ακολουθεί δείχνει τη συμπεριφορά των συναρτήσεων rand και myrand ομαδοποιημένες ως προς τα λάθη. Από αυτό συμπεραίνουμε είναι ότι τα περισσότερα λάθη στο pipeline και στον Program Counter κατά τη διάρκεια των συναρτήσεων αυτών οδηγούν σε crashed πειράματα.



Σχήμα 4.3: Επίδραση κάθε σφάλματος στις συναρτήσεις rand/myrand της εφαρμογής Monte-Carlo PI

Το γράφημα που ακολουθεί δείχνει τη συμπεριφορά της συνάρτησης MonteCarlo Integrate ως προς τα λάθη. Παρατηρούμε ότι περισσότερα πειράματα μπορούν να βγάλουν strict και correct πειράματα σε σχέση με τις rand και myrand πράγμα που δείχνει ότι είναι πιο ανεκτική συνάρτηση από τις άλλες.



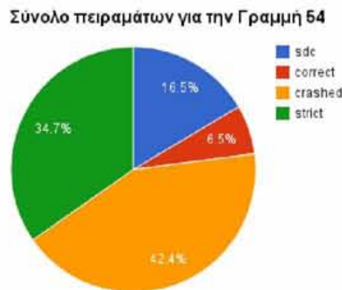
Σχήμα 4.4: Επίδραση κάθε σφάλματος στη συνάρτηση MonteCarlo Integrateτης εφαρμογής MonteCarlo PI

Το γράφημα που ακολουθεί δείχνει τη συμπεριφορά του iterator count της συνάρτησης MonteCarlo Integrate ως προς τα λάθη. Παρατηρούμε ότι σε οποιοδήποτε από τα 5 ισοχωρισμένα εύρη τιμών μπεί οποιοδήποτε λάθος έχουμε ίδιο ποσοστό με sdc αποτελέσματα. Επίσης βλέπουμε ότι τα πειράματα που ήταν crashed είναι στις αρχικές και στις τελευταίες επαναλήψεις όπως επίσης και ότι τα περισσότερα πειράματα που ήταν strict άνηκαν στις ενδιάμεσες επαναλήψεις.

Σχήμα 4.5: Επίδραση κάθε σφάλματος σε 5 ισοχωρισμένα εύρη τιμών του iterator "count" της εφαρμογής MonteCarlo PI

Τέλος υπάρχουν τα ποσοστά σφαλμάτων για τρεις σημαντικές εντολές του αρχικού κώδικα της εφαρμογής αυτής. Αυτές οι γραμμές είναι:

1. Γραμμή 54: Υπολογισμός και ανάθεση τιμής $\alpha = x_i^2 + y_i^2$
2. Γραμμή 56: Έλεγχος τιμής $\alpha < 1$
3. Γραμμή 58: Πρόσθεση του σημείου $m = m + 1$



Σχήμα 4.6: Επίδραση κάθε σφάλματος στη γραμμή 54 του αρχικού κώδικα της εφαρμογής MonteCarlo PI

Σχήμα 4.7: Επίδραση κάθε σφάλματος στη γραμμή 56 του αρχικού κώδικα της εφαρμογής MonteCarlo PI

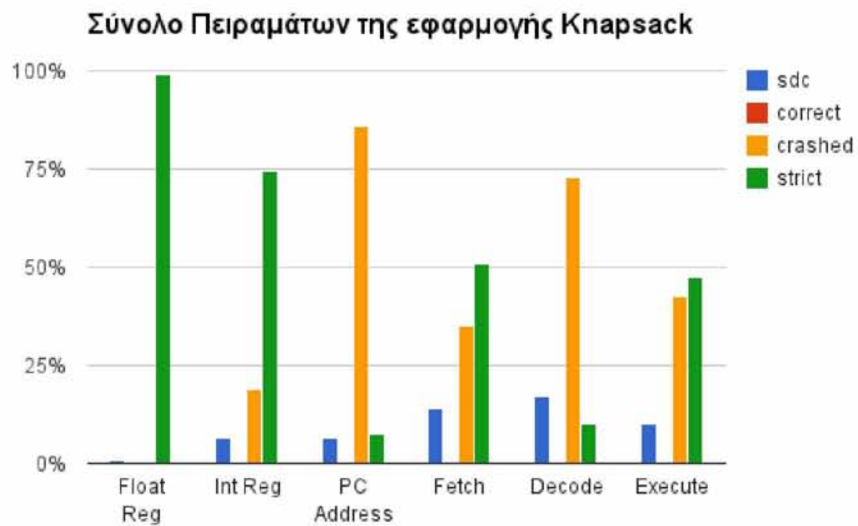
Σχήμα 4.8: Επίδραση κάθε σφάλματος στη γραμμή 58 του αρχικού κώδικα της εφαρμογής MonteCarlo PI

4.2 Αξιολόγηση της εφαρμογής Knapsack

Η εφαρμογή Knapsack είναι μια λύση του συνδυαστικού προβλήματος ¨Σακιδίου 0-1¨ χρησιμοποιώντας έναν αλγόριθμο εμπνευσμένο από τη γενετική (Genetic Algorithm). Σας είσοδο παίρνει 24 αντικείμενα με τα βάρη τους και την αξία τους, και όριο βάρους σακιδίου 500. Ύστερα με τροποποίησης του αρχικού τυχαίου πληθυσμού προσεγγίζει τη βέλτιστη λύση που είναι να έχουμε τη μέγιστη αξία από τα πράγματα που μπορούμε να κουβαλήσουμε στο σακίδιο μας.

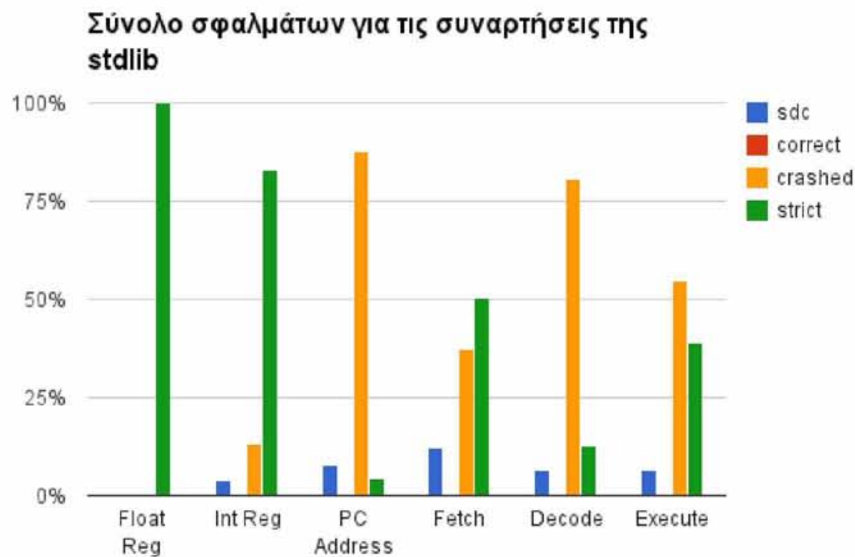
Την εφαρμογή αυτή την τρέξαμε με ένα νήμα και το εκτελέσιμο είχε όλες τις δυνατές βελτιστοποιήσεις (εκτός από τις Inline Functions καθώς δυσκολεύει την ανάγνωση του πρωτοκόλλου DWARF). Χρησιμοποιώντας όλες τις δυνατές βελτιστοποιήσεις ο μεταγλωττιστής χρησιμοποιεί όλους τους πόρους όσο πιο αποδοτικά μπορεί και έτσι ένα σφάλμα φαίνεται πιο εύκολα καθώς χωρίς βελτιστοποιήσεις κάποιοι πόροι μπορεί να μη χρησιμοποιούνται καν.

Στο ακόλουθο γράφημα για το σύνολο των σφαλμάτων που εισάγαμε βλέπουμε ότι τα περισσότερα sdc πειράματα οφείλονται σε λάθος κατά το decoding των καταχωρητών, λάθη σε ακέραιους καταχωρητές αλλά και καταχωρητές κινητής υποδιαστολής έβγαλαν τα περισσότερα strict αποτελέσματα και τέλος ότι τα περισσότερα πειράματα που το λάθος μπήκε στον Program Counter του προγράμματος τερμάτισαν χωρίς αποτέλεσμα. Αξίζει να σημειωθεί πως δεν βγαίνει κανένα correct αποτέλεσμα.



Σχήμα 4.9: Σύνολο προσομοίωσης για την εφαρμογή KnapSack με την επίδραση κάθε σφάλματος

Στο ακόλουθο γράφημα βλέπουμε τα σφάλματα που έγιναν σε συναρτήσεις των οποίων δεν έχουμε τις πληροφορίες από το πρωτόκολλο DWARF. Τέτοιες είναι για παράδειγμα η malloc και η qsort. Αυτό που διαφαίνεται από αυτό το γράφημα είναι ότι λάθη στο Pipeline και στον Program Counter σε αυτές τις συναρτήσεις οδηγούν σε crashed αποτελέσματα.



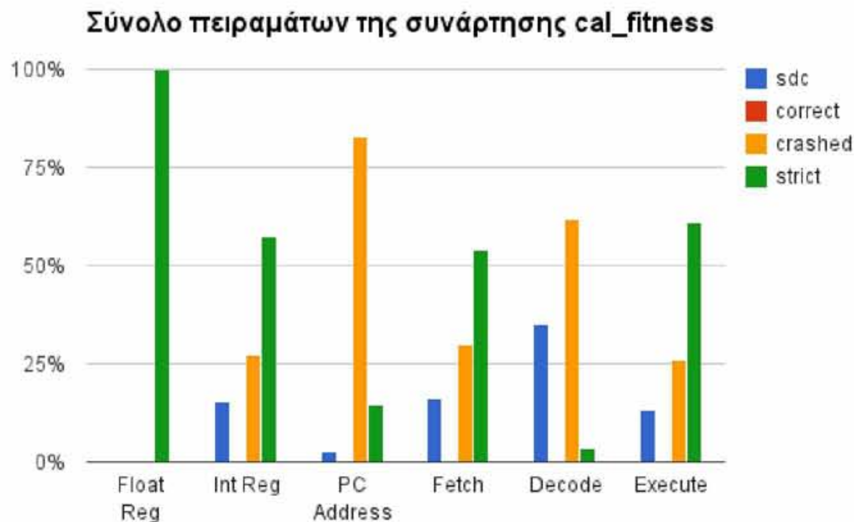
Σχήμα 4.10: Επίδραση κάθε σφάλματος στις συναρτήσεις της stdlib της εφαρμογής KnapSack

Το διάγραμμα που ακολουθεί αναφέρεται στην συνάρτηση cal_fitness του αρχικού κώδικα. Αυτή η συνάρτηση αυτό που κάνει είναι να βρίσκει την καλύτερη τιμή βάρους. Για να γίνει αυτό κάθε στοιχείο του πληθυσμού έχει μια μεταβλητή που ονομάζεται Fitness και όσο μεγαλύτερη είναι αυτή τόσο πλησιάζουμε την λύση. Έτσι λοιπόν η συνάρτηση αυτή διατρέχει όλο το πληθυσμό και για την λύση που "προτείνει" το σωματίδιο αυτό ελέγχει το βάρος των αντικειμένων με δυο περιπτώσεις:

1. Να μην έχουμε υπερβεί το όριο του σακιδίου άρα αυξάνουμε το fitness του συγκεκριμένου σωματιδίου του πληθυσμού.

2. Να μην είναι καλή λύση οπότε πρέπει να "πέσει" στις χαμηλές θέσεις του πληθυσμού κάνοντας το fitness του συγκεκριμένου σωματιδίου του πληθυσμού ίσο με το 0.

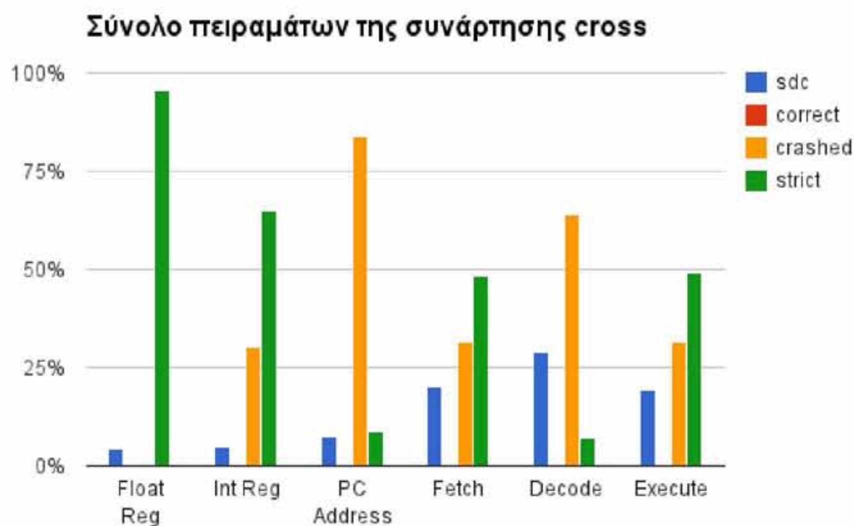
Αυτό που παρατηρούμε είναι ότι τα μεγαλύτερα ποσοστά crashed αποτελεσμάτων βρίσκονται σε σφάλματα που αναφέρονται στον Program Counter και κατά το decoding στο Pipeline.



Σχήμα 4.11: Επίδραση κάθε σφάλματος στη συνάρτηση της cal_fitness της εφαρμογής KnapSack

Το ακόλουθο διάγραμμα αναφέρεται στην συνάρτηση cross του αρχικού κώδικα. Αυτή η συνάρτηση αυτό που κάνει είναι να κοιτάζει τα σωματίδια που βρίσκονται στην κορυφή του πληθυσμού (=αυτά με το μεγαλύτερο fitness) και να δημιουργεί ένα καινούργιο πληθυσμό από τα σωματίδια της κορυφής χρησιμοποιώντας τα στοιχεία πότε του σωματιδίου α και πότε του σωματιδίου β. Τα α και β διαλέγοντα τυχαία από το πάνω μισό του πληθυσμού.

Αυτό που παρατηρούμε είναι ότι αυξάνονται τα ποσοστά crashed αποτελεσμάτων βρίσκονται σε οφάλματα στο Pipeline σε σύγκριση με την συνάρτηση cal_fitness που δείξαμε πριν.



Σχήμα 4.12: Επίδραση κάθε οφάλματος στη συνάρτηση της cross της εφαρμογής KnapSack

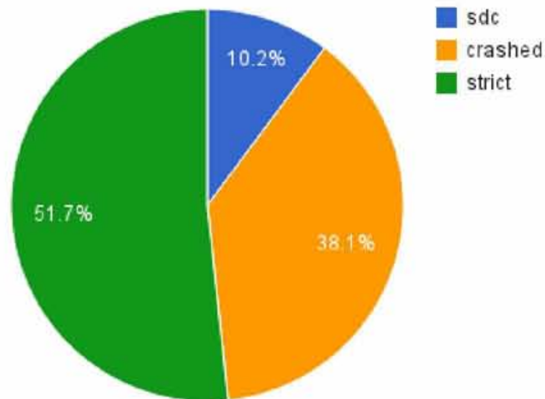
Τέλος υπάρχουν τα ποσοστά οφαλμάτων για τέσσερις σημαντικές εντολές του αρχικού κώδικα της εφαρμογής αυτής. Αυτές οι γραμμές είναι:

1. Γραμμή 46: For Loop στην συνάρτηση cal_fitness που διατρέχει όλα τα αντικείμενα που έχει δεχθεί σαν είσοδο η εφαρμογή μας.

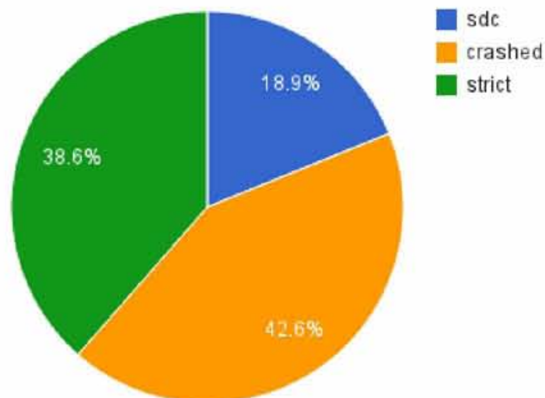
2. Γραμμή 47: Έλεγχος τιμής αν ξεπερνάει τα όρια του βάρους του σακιδίου.

3. Γραμμή 48: Αύξηση fitness.

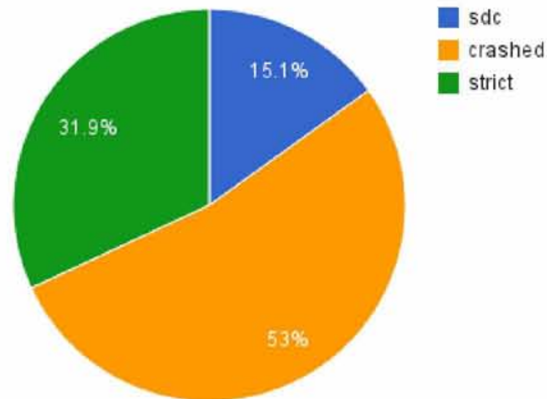
4. Γραμμή 84: Διασταύρωση δυο σωματιδίων.



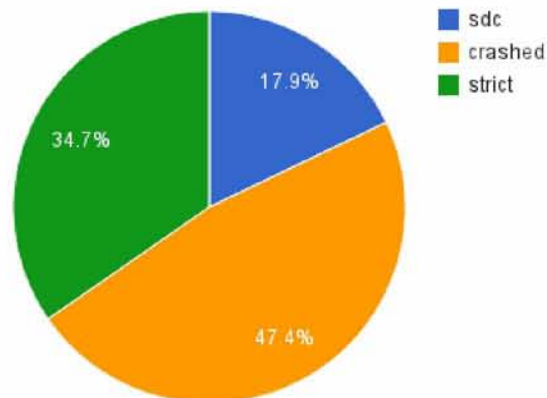
Σχήμα 4.13: Επίδραση όλων των οφαλμάτων στη γραμμή 46 του αρχικού κώδικα της εφαρμογής KnapSack



Σχήμα 4.14: Επίδραση όλων των οφαλμάτων στη γραμμή 47 του αρχικού κώδικα της εφαρμογής KnapSack



Σχήμα 4.15: Επίδραση όλων των σφαλμάτων στη γραμμή 48 του αρχικού κώδικα της εφαρμογής KnapSack



Σχήμα 4.16: Επίδραση όλων των σφαλμάτων στη γραμμή 84 του αρχικού κώδικα της εφαρμογής KnapSack

ΚΕΦΑΛΑΙΟ 5

Μελλοντική ανάπτυξη του συστήματος

Το εργαλείο έχει αρκετά περιθώρια βελτίωσης. Πρώτα από όλα το πιο σημαντικό που πρέπει να γίνει είναι να μπορεί να διαβάσει σύνθετους τύπους δεδομένων για να μπορεί να προσφέρει περισσότερη πληροφορία καθώς σε όλες τις εφαρμογές υπάρχουν σύνθετοι τύποι δεδομένων και κάποιες φορές μπορεί να είναι αρκετά σημαντικές οι τιμές που παίρνουν τα πεδία τους. Έτσι πρέπει να προστεθεί δυνατότητα ελέγχου για τιμές στα πεδία των structs καθώς και typedef. Τέλος πρέπει να μελετηθεί πως μπορεί να ενσωματωθούν και τιμές από Union και πίνακες. Αν γίνουν αυτές οι αλλαγές θα έχουμε μια πλήρη επισκόπηση της κατάστασης του προγράμματος όταν εισήχθη το σφάλμα.

Επιπροσθέτως πρέπει να μελετηθούν οι βελτιστοποιήσεις και πως μπορούν να λυθούν τα προβλήματα που μπορούν να προκύψουν στο εργαλείο μας (όπως για παράδειγμα να οριοθεί μια local non-static μεταβλητή αλλά ο μεταγλωττιστής να μην της δώσει κάποια θέση σε καταχωρητή ή στη στοίβα αλλά να υλοποιήσει αλλιώς τη λειτουργικότητα της οπότε δεν μπορούμε να βρούμε σε ποιο κομμάτι του κώδικα βρίσκεται και αν επηρεάζεται).

Σε επίπεδο αρχιτεκτονικής για την αρχιτεκτονική X86-64 πρέπει να βρεθεί ο τρόπος αποκρυπτογράφησης της στοίβας καθώς δεν μπορούμε να εξάγουμε backtrace. Αυτό οφείλεται στο γεγονός ότι ο προσομοιωτής δεν προσφέρει εγγενώς τη δυνατότητα προσπέλασης του task_struct όπως κάνει για την ALPHA αρχικτεκτονική αλλά την προσθέσαμε εμείς χωρίς όμως να γνωρίζουμε το απαραίτητο masking που πρέπει να γίνει ώστε να διατρέξουμε τα στιγμιότυπα της στοίβας.

ΠΑΡΑΡΤΗΜΑ Α΄

Κώδικας Πυρήνα για Περιγραφή Διεργασίας

```
1  /*
2   * Orismos Task_struct se pirines 2.6.26+
3   */
4   struct task_struct {
5          volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped */
6          void *stack;
7          atomic_t usage;
8          unsigned int flags;     /* per process flags, defined below */
9          unsigned int ptrace;
10
11         ....
12         struct list_head tasks;
13
14         struct mm_struct *mm, *active_mm;
15
16 /* task state */
17         struct linux_binfmt *binfmt;
18         int exit_state;
19         int exit_code, exit_signal;
20         int pdeath_signal;  /*  The signal sent when the parent dies  */
21         /* ??? */
22         unsigned int personality;
23         unsigned did_exec:1;
24         pid_t pid;
```

```
25          pid_t tgid;
26
27          .....
28           */
29          struct list_head ptraced;
30          struct list_head ptrace_entry;
31
32          /* PID/PID hash table linkage. */
33          struct pid_link pids[PIDTYPE_MAX];
34          struct list_head thread_group;
35
36          struct completion *vfork_done;          /* for vfork() */
37          int __user *set_child_tid;              /* CLONE_CHILD_SETTID */
38          int __user *clear_child_tid;            /* CLONE_CHILD_CLEARTID */
39
40          cputime_t utime, stime, utimescaled, stimescaled;
41          cputime_t gtime;
42          cputime_t prev_utime, prev_stime;
43          unsigned long nvcsw, nivcsw; /* context switch counts */
44          struct timespec start_time;             /* monotonic time */
45          struct timespec real_start_time;        /* boot based time */
46          .....
47  };
```

ΒΙΒΛΙΟΓΡΑΦΙΑ

[1] Christian Bienia. *Benchmarking Modern Multiprocessors.* PhD thesis, Princeton University, January 2011.

[2] Michael J. Eager. *Introduction to the DWARF Debugging Fo*, February 2012.

[3] Γεώργιος Τζιαντζούλης. Εισαγωγή σφαλμάτων για ανάλυση αξιόπιστης λειτουργίας σε επεξεργαστές πολλαπλών πυρήνων. Διπλωματική Εργασία, Πανεπιστήμιο Θεσσαλίας, Ιούλιος 2011.

[4] Christos D. Antonopoulos Nikolaos Bellas Konstantinos Parasyris, Georgios Tziantzioulis. Gemfi: A fault injection tool for studying the behavior of applications on unreliable substrates. In *International Conference on Dependable Systems and Networks (DSN)*, June 23-26 2014.

[5] Konstantinos Parasyris. Transient hardware faults simulation in gem5-study of the behavior of multithreaded applications under faults. Diploma Thesis, University of Thessaly, February 2013.