Πολυτεχνείο Κρήτης

# Υλοποίηση DIFT στον Επεξεργαστή MIPS για την Αντιμετώπιση Κακόβουλων Επιθέσεων

Άγγελος Κούκουνας

Φεβρουάριος 2018

# Περίληψη

Ο τομέας της ασφάλειας των υπολογιστικών συστημάτων αποτελεί έναν ιδιαίτερα ευαίσθητο και σημαντικό τομέα και είναι αναγκαία η συστηματική μελέτη του, προκειμένου να επιτευχθεί η προστασία του συστήματος και των δεδομένων που μεταφέρονται. Τα 'τρωτά' σημεία του λογισμικού, εκμεταλλευόμενα κατάλληλα από έναν επιτιθέμενο, μπορούν να οδηγήσουν σε μη εξουσιοδοτημένες ενέργειες, στην τροποποίηση της διάρθρωσης του συστήματος και τελικά στην πλήρη πρόσβαση σε αυτό. Για το λόγο αυτό, αναπτύσσονται συνεχώς τεχνικές που αποσκοπούν στην πρόληψη, ανίχνευση και αντιμετώπιση των κακόβουλων ενεργειών.

Μία σχετικά πρόσφατη τεχνική είναι και αυτή του *Dynamic Information Flow Tracking (DIFT)* που αποτελεί έναν μηχανισμό, ο οποίος στηρίζεται στην παρακολούθηση της ροής δεδομένων που προέρχονται από μη έμπιστες πηγές. Ο μηχανισμός λειτουργεί κατά την διάρκεια εκτέλεσης του προγράμματος και η παρακολούθηση επιτυγχάνεται με την προσθήκη δύο extra bits, επεκτείνοντας τους καταχωρητές και την μνήμη έτσι ώστε να αποθηκεύουν παραπάνω πληροφορία. Η παραγωγή των δύο επιπρόσθετων bits, το καθένα από τα οποία έχει και μία συγκεκριμένη λειτουργία, διέπεται από κανόνες για τον εκάστοτε τύπο εντολής που εκτελείται.

Στην δική μας προσέγγιση, με την τροποποίηση του MIPS για την εφαρμογή του μηχανισμού DIFT, επεκτείνουμε τον επεξεργαστή κατάλληλα, με αποτέλεσμα να προστατεύει από τις *low level* επιθέσεις (όπως το buffer overflow ή format string attack). Τέλος, αξίζει να σημειωθεί πως με μελλοντική εργασία, είναι δυνατή και η προστασία του συστήματος από *high level* επιθέσεις (όπως SQL Injections).

**Λέξεις Κλειδιά**: Ασφάλεια Υπολογιστικών Συστημάτων, Dynamic Information Flow Tracking, επεξεργαστής MIPS

# Περιεχόμενα

# Κατάλογος Σχημάτων

# Κατάλογος Πινάκων

# Κεφάλαιο 1

# Εισαγωγή

## 1.1 Βασικές Αρχές

Η ασφάλεια υπολογιστών, που αποτελεί γνωστικό πεδίο της επιστήμης της πληροφορικής, ασχολείται με την προστασία των υπολογιστών, των δικτύων που τους διασυνδέουν και των δεδομένων σε αυτά τα συστήματα, αποτρέποντας τη μη εξουσιοδοτημένη πρόσβαση ή χρήση τους. Συνιστά έναν πολύ κρίσιμο τομέα και λόγω αυτού, είναι απαραίτητη η ενδελεχής έρευνά του, προκειμένου να αντιμετώπιζονται, όσο το δυνατόν πιο αποτελεσματικά, τα ζητήματα που προκύπτουν. Στηρίζεται πάνω σε τρεις βασικές αρχές:

1. **Ακεραιότητα (Integrity)**: αναφέρεται στη διατήρηση των δεδομένων ενός πληροφοριακού συστήματος σε μια γνωστή κατάσταση χωρίς ανεπιθύμητες τροποποιήσεις, αφαιρέσεις ή προσθήκες από μη εξουσιοδοτημένα άτομα, καθώς και την αποτροπή της πρόσβασης ή χρήσης των υπολογιστών και δικτύων του συστήματος από άτομα χωρίς άδεια.

2. **Διαθεσιμότητα (Availability)**: η διαθεσιμότητα των δεδομένων και των υπολογιστικών πόρων είναι η εξασφάλιση ότι οι υπολογιστές, τα δίκτυα και τα δεδομένα θα είναι στη διάθεση των χρηστών όποτε απαιτείται η χρήση τους.

3. **Εμπιστευτικότητα (Confidentiality)**: είναι η διασφάλιση της πληροφορίας από οποιονδήποτε δεν έχει το δικαίωμα να την δει ή να κρατήσει αντίγραφό της. Με άλλα λόγια, σημαίνει ότι ευαίσθητες πληροφορίες δεν θα έπρεπε να αποκαλύπτονται σε μη εξουσιοδοτημένα άτομα.

Ωστόσο, παρά την συνεχόμενη ανάπτυξη που συντελείται στον τομέα της ασφάλειας, οι τεχνικές που υφίστανται δεν παρέχουν πάντοτε απόλυτη

προστασία, καθώς αναπτύσσονται παράλληλα και οι τρόποι έκθεσης των αδυναμιών που υπάρχουν, από την απέναντι πλευρά, αυτή του επιτιθέμενου. Οι υπάρχοντες μηχανισμοί δεν προσφέρουν αξιόπιστη προστασία έναντι κακόβουλων επιθέσεων, που στοχεύουν όχι απαραίτητα μόνο μία αδυναμία, αλλά σε αρκετές περιπτώσεις πολλαπλά τρωτά σημεία του συστήματος. Έτσι, πέρα από τις low-level επιθέσεις όπως είναι το *buffer overflow* ή το *format string* είναι επιτακτική και η αναχαίτιση των high level επιθέσεων όπως το *path traversal, sql injection* ή *cross-site scripting*.

## 1.2 Εισαγωγή στο μηχανισμό DIFT

Στη συγκεκριμένη περίπτωση, εστιάζουμε στις κακόβουλες ενέργειες που συμβαίνουν κατά την διαδικασία εισαγωγής δεδομένων από κάποια εξωτερική πηγή και αναλύουμε τους τρόπους με τους οποίους οι κακόβουλοι χρήστες εκμεταλλεύονται τα αδύναμα σημεία, για να αποκτήσουν πρόσβαση σε ευαίσθητες πληροφορίες ή να προκαλέσουν κατάρρευση του συστήματος.

Σε κάθε υπολογιστικό σύστημα, χρησιμοποιούνται εφαρμογές που επεξεργάζονται δεδομένα εισόδου, τα οποία προέρχονται από μη αξιόπιστες πηγές. Αυτό έχει ως συνέπεια, όταν οι πηγές είναι κακόβουλες, τις λεγόμενες Input Validation επιθέσεις, που επιτυγχάνονται με την κατάλληλη τροποποίηση των δεδομένων που εισάγονται, για την εκμετάλλευση των ευάλωτων σημείων και την επίτευξη του επιθυμητού στόχου από την πλευρά του επιτιθέμενου.

Ανάμεσα στις διάφορες μεθόδους που υπάρχουν για την προστασία από τις προαναφερθείσες επιθέσεις, υπάρχει και μια πολλά υποσχόμενη τεχνική, η οποία χρησιμοποιείται για την ανίχνευση μεγάλου εύρους κακόβουλων ενεργειών. Πρόκειται για το μηχανισμό Dynamic Information Flow Tracking ή DIFT [25], η βασική ιδέα λειτουργίας του οποίου είναι η εξής: παρακολουθεί τις ύποπτες ροές πληροφορίας, συσχετίζοντας ένα ή περισσότερα tag bit με κάθε byte μνήμης/καταχωρητή. Αυτό το tag ορίζεται για κάθε μη έμπιστη πληροφορία που εισάγεται στο σύστημα. Στη συνέχεια, τα tags παράγονται και διαδίδονται ακολουθώντας συγκεκριμένους κανόνες, ανάλογα με τις εντολές που εκτελούνται. Αν η αναξιόπιστη πληροφορία χρησιμοποιείται επισφαλώς, δημιουργείται μια εξαίρεση ασφαλείας, περιορίζοντας ή αποτρέποντας τη χρήση των δεδομένων.

## 1.3 Σκοπός και Συνεισφορά της Διπλωματικής Εργασίας

Σκοπός της εργασίας είναι η ανάπτυξη ενός ολοκληρωμένου συστήματος, στο επίπεδο του υλικού, το οποίο θα περιλαμβάνει την εφαρμοφή του μηχανισμού DIFT στον τροποποιήμενο, γι΄ αυτό το σκοπό, επεξεργαστή MIPS. Για την επίτευξη του παραπάνω στόχου, μελετήθηκαν:

1. η δομή και λειτουργία του επεξεργαστή MIPS.
2. ο μηχανισμός του Dynamic Information Flow Tracking.
3. το πρωτόκολλο ασφαλείας που εφαρμόστηκε (PI policy).

Όλα τα παραπάνω, σε συνδυασμό με την μελέτη των αδυναμιών και των υφιστάμενων μηχανισμών ασφαλείας, είναι απαραίτητες γνώσεις για την ολοκλήρωση της εργασίας. Αποτέλεσμα της μελέτης όσων προαναφέρθηκαν είναι:

→ η επέκταση όλων των απαραίτητων καταχωρητών/μνημών και σημάτων εντός του pipeline του επεξεργαστή MIPS.

→ η υλοποίηση των propagation/check module στον επεξεργαστή MIPS για την υποστήριξη της λειτουργικότητας του μηχανισμού.

→ η υλοποίηση μιας μεθόδου για την κάλυψη της λειτουργικότητας DIFT στο επίπεδο του λογισμικού, μέσω της προσθήκης extra DIFT εντολών στα τροποποιημένα binaries για την ανάθεση των επιθυμητών τιμών στα tags.

Όλα τα παραπάνω, οδηγούν στην ανάπτυξη της τεχνικής DIFT στον επεξεργαστή MIPS και αποτελούν την συνεισφορά της παρούσας Διπλωματικής Εργασίας.

## 1.4 Οργάνωση της Διπλωματικής Εργασίας

Στο *κεφάλαιο 2* παρουσιάζονται αναλυτικά οι ποικίλες αδυναμίες που υπάρχουν και ο τρόπος έκθεσής τους από τους κακόβουλους χρήστες. Επιπλέον, παρατίθενται και οι υπάρχουσες τεχνικές άμυνας, πλην του μηχανισμού DIFT, που χρησιμοποιούνται για την αποτροπή του επιτιθεμένου από την επίτευξη των στόχων του.

Στο *κεφάλαιο 3* γίνεται εκτενής αναφορά στο μηχανισμό του *Dynamic Information Flow Tracking*, παραθέτοντας όλες τις παραμέτρους και διεργασίες που απαιτούνται να υλοποιηθούν, προκειμένου η τεχνική αυτή να εκτελεστεί

στο βέλτιστο βαθμό. Επιπλέον, παρουσιάζονται οι εναλλακτικοί σχεδιασμοί που υπάρχουν, βάσει των προτάσεων που έχουν αναλυθεί μέχρι στιγμής, και εκθέτονται ποικίλες περιπτώσεις συστημάτων DIFT, εστιάζοντας επιγραμματικά στην λειτουργικότητα της καθεμιάς πλατφόρμας.

Στο *κεφάλαιο 4* παρουσιάζεται ο επεξεργαστής MIPS και γίνεται σύντομη αναφορά στα πιο σημαντικά δομικά τμήματα και τη λειτουργία τους. Η κατανόηση αυτών είναι ιδιαίτερα σημαντική για το επόμενο στάδιο που αφορά την εφαρμογή της τεχνικής, μέσω της τροποποίησης των δομικών τμημάτων, στον επεξεργαστή.

Στο *κεφάλαιο 5* παρατίθεται η υλοποίηση του μηχανισμού *Dynamic Information Flow Tracking* στον επεξεργαστή MIPS, αναφέροντας παράλληλα όλες τις παραμέτρους και τις τροποποιήσεις που είναι απαραίτητες να καθοριστούν και να γίνουν αντίστοιχα (καθορισμένη πολιτική ασφαλείας, επέκταση συγκεκριμένων δομικών τμημάτων για την υποστήριξη των tags, modules για την παραγωγή, διάδοση και έλεγχο των tags), προκειμένου να εφαρμοστεί η τεχνική στον επεξεργαστή.

Τέλος, στο *κεφάλαιο 6* παρουσιάζεται ο επίλογος της εργασίας κάνοντας μια ανασκόπηση του μηχανισμού DIFT και των παραμέτρων/στοιχείων εκείνων που απαιτείται να τροποποιηθούν-προστεθούν για να υλοποιηθεί η τεχνική στον επεξεργαστή MIPS.

# Κεφάλαιο 2

# Επιθέσεις

Σε αυτό το κεφάλαιο παρουσιάζονται οι αδυναμίες και τα τρωτά σημεία που υπάρχουν κατά τη διαδικασία εισαγωγής δεδομένων από εξωτερική πηγή σε ένα πρόγραμμα, τα οποία εκμεταλλευόμενα κατάλληλα μπορούν να οδηγήσουν σε ανεπιθύμητα αποτελέσματα, όπως κατάρρευση του προγράμματος ή πλήρη πρόσβαση σε αυτό. Οι επιθέσεις που περιγράφονται παρακάτω έχουν ως κοινό παρονομαστή την προσπάθεια του επιτιθέμενου να διαφοροποιήσει, ανάλογα με την περίπτωση, τα δεδομένα που στέλνει ως είσοδο σε ένα πρόγραμμα προκειμένου να αποκτήσει πρόσβαση σε αυτό. Η προαναφερθείσα διαδικασία, που περιλαμβάνει την αλλαγή της συμβολοσειράς εισόδου, στηρίζεται κάθε φορά στην εκμετάλλευση της αδυναμίας που υπάρχει.

Σημειώνεται ότι, η μετατροπή της συμβολοσειράς εισόδου για να επωφεληθεί ο κακόβουλος χρήστης, μπορεί να περιλαμβάνει, από την παραγωγή κάποιων επιπρόσθετων εντολών έως την χρησιμοποίηση πολλών δεκάδων γραμμών κώδικα για την αποδιοργάνωση του προγράμματος και την πρόσβαση στους πόρους του συστήματος.

Παρακάτω, λοιπόν, περιγράφεται συνοπτικά το τρωτό σημείο κάθε περίστασης -με την παράθεση και ενός παραδείγματος για την πλήρη κατανόηση του προβλήματος και πώς αυτό μπορεί να χρησιμοποιηθεί κατάλληλα από τον επιτιθέμενο, καθώς επίσης και οι τεχνικές που χρησιμοποιούνται για να αποτρέψουν την εκάστοτε επίθεση.

Τέλος, αναφέρεται ότι οι επιθέσεις, που προκύπτουν από τα ευάλωτα σημεία κάθε φορά, αφορούν είτε *low level* επιθέσεις όπως buffer overflow ή format string attacks είτε *high level* επιθέσεις όπως SQL injection ή Cross-site Scripting attacks.

## 2.1 Buffer Overflow

### 2.1.1 Συνοπτική Περιγραφή

Η υπερχείλιση του buffer (*buffer overflow ή buffer overrun*) είναι μια ανωμαλία κατά την οποία, ένα πρόγραμμα καθώς εκχωρεί δεδομένα σε έναν buffer, υπερχειλίζει τα όριά του με αποτέλεσμα να πανωγράφει σε γειτονικές περιοχές της μνήμης. Με άλλα λόγια, η 'υπέρβαση' των ορίων του buffer συμβαίνει όταν ένα πρόγραμμα ή μια διαδικασία επιδιώκει να γράψει περισσότερα δεδομένα σε ένα καθορισμένου μήκους, τμήμα της μνήμης. Επομένως, από τη στιγμή που το buffer δημιουργήθηκε για να κρατά συγκεκριμένη ποσότητα δεδομένων, η επιπρόσθετη πληροφορία θα εκχωρηθεί σε περιοχές της μνήμης προσκείμενες σε αυτόν.

Η συγκεκριμένη κατάσταση μπορεί να ενεργοποιηθεί από εισόδους δεδομένων που σκοπό έχουν την εκτέλεση κώδικα ή την αλλαγή της κανονικής ροής του προγράμματος. Έτσι, το πρόγραμμα παρουσιάζει 'αλλοπρόσαλλη' συμπεριφορά όπως λανθασμένα αποτελέσματα, σφάλματα κατά την πρόσβαση στη μνήμη ή καταρρέει.

Για την αποφυγή όλων των παραπάνω υπάρχουν κάποιες -πιο σύγχρονες γλώσσες προγραμματισμού (όπως η C# και η Java) οι οποίες μειώνουν την πιθανότητα σφαλμάτων στον κώδικα που μπορεί να οδηγήσουν σε υπερχείλιση. Αντίθετα, γλώσσες προγραμματισμού όπως η C και η C++, είναι ευάλωτες σε buffer overflow επιθέσεις καθώς δεν έχουν ενσωματωμένους μηχανισμούς που να αποτρέπουν συγκεκριμένες ενέργειες που οδηγούν στην υπερχείλιση. Στον πίνακα που ακολουθεί παρουσιάζεται μια λίστα με μη ασφαλείς συναρτήσεις, στη γλώσσα C, η χρήση των οποίων μπορεί να οδηγήσει στην υπερχείλιση του buffer.

| Συνάρτηση | **Ενδεχόμενη ανωμαλία** |
|---|---|
| strcpy(char *dest, const char *src) | Πιθανή υπερχείλιση του dest buffer |
| gets(char *str) | Πιθανή υπερχείλιση του str buffer |
| strcat(char *dest, const char *src) | Πιθανή υπερχείλιση του dest buffer |
| realpath(const char *path, char *resolved_path) | Πιθανή υπερχείλιση του path buffer |
| sprintf(char *str, const char *format, ...) | Πιθανή υπερχείλιση του str buffer |

Πίνακας 2.1: Λίστα με κάποιες μη ασφαλείς συναρτήσεις οι οποίες ανάλογα με το πώς χρησιμοποιούνται ενδεχομένως να οδηγήσουν σε υπερχείλιση του buffer.

Στη συνέχεια, παρατίθεται ένα παράδειγμα σε κώδικα C που παρουσίαζει το 'τρωτό' σημείο στον κώδικα, το οποίο ενδεχομένως προκαλέσει υπέρβαση των ορίων του buffer.

```
#include <stdio.h>
#define BUFSIZE 10

void func()
{
    char buf[BUFSIZE];
        gets(buf);       // no bounds checking  }

int main(int argc, char **argv)
{
    func();    }
```

Σχήμα 2.1: Τμήμα προγράμματος γραμμένο σε κώδικα C το οποίο ενδεχομένως προκαλέσει υπερχείλιση του buffer. Η αδυναμία υπάρχει εξαιτίας της χρήσης της συνάρτησης gets() η οποία δεν ελέγχει την ποσότητα των δεδομένων που εισάγονται.

Το παραπάνω τμήμα κώδικα καταδεικνύει το *buffer overflow vulnerability*, παρουσιάζοντας ένα πολύ απλό παράδειγμα στο οποίο δεν γίνεται έλεγχος στην ποσότητα δεδομένων που εκχωρούνται στον buffer, μέσω της συνάρτησης *gets()*. Πιο συγκεκριμένα, ο κώδικας χρησιμοποιεί τη συνάρτηση gets() για να διαβάσει μια αυθαίρετη ποσότητα δεδομένων και να τα εκχωρήσει στον buffer που έχει οριστεί. Επειδή επομένως, δεν υπάρχει τρόπος περιορισμού της ποσότητας των δεδομένων που διαβάζει η συνάρτηση, η ασφάλεια του κώδικα εξαρτάται από τον χρήστη, που πρέπει κάθε φορά να εισάγει λιγότερους χαρακτήρες από το *BUFSIZE*.

### 2.1.2 Τεχνικές Αντιμετώπισης

Σε αυτό το σημείο περιγράφονται οι διάφορες τεχνικές που υπάρχουν για την αντιμετώπιση του buffer overflow, έτσι ώστε να μην παρουσιάζονται ανεπιθύμητα αποτελέσματα κατά την εκτέλεση του προγράμματος. Περιληπτικά, οι μηχανισμοί αυτοί είναι οι παρακάτω:

- **Address Space Layout Randomization**: τεχνική η οποία βασίζεται στην μικρή πιθανότητα που έχει ο επιτιθέμενος να μαντέψει σωστά, **τυχαία** τοποθετημένες περιοχές δεδομένων [1]. Με αλλά λόγια, το A-SLR τυχαιοποιεί διάφορα μέρη του χώρου διευθύνσεων με σκοπό την δύσκολη πρόβλεψη της θέσης τους. Για παράδειγμα, σε μια ενδεχόμενη προσπάθεια διάχυσης κώδικα στη στοίβα, ο κακόβουλος χρήστης πρέπει αρχικά να βρει την στοίβα κάτι που ο μηχανισμός καθιστά πολύ δύσκολο. Η ασφάλεια επομένως αυξάνεται καθώς αυξάνεται ο χώρος αναζήτησης. Η συγκεκριμένη τεχνική έχει αναπτυχθεί τόσο σε συστήματα Linux [2] όσο και σε αντίστοιχα Windows [3].

- **NX bit (No-eXecute bit)**: μηχανισμός ο οποίος απομονώνει περιοχές της μνήμης που χρησιμοποιούνται είτε για αποθήκευση εντολών (πηγαίος κώδικας), είτε για αποθήκευση δεδομένων. Με αλλά λόγια, ένα λειτουργικό που υποστηρίζει την τεχνολογία του NX bit, μαρκάρει συγκεκριμένες περιοχές της μνήμης ως 'μη εκτελέσιμες' με αποτέλεσμα ο επεξεργαστής να μην επιτρέπει την εκτέλεση κώδικα εντός αυτών των περιοχών. Αποτελεί υποκατηγορία της γενικής τεχνικής executable space protection η οποία αποτρέπει κακόβουλο λογισμικό να πάρει τον έλεγχο του υπολογιστή εγχύοντας κώδικα στην περιοχή της μνήμης που αποθηκεύονται τα δεδομένα και εκτελώντας τον στην συνέχεια εντός αυτής. Τέλος αξίζει να αναφερθεί πως, ως NX bit, ορίζονται όλες οι παρεμφερείς τεχνικές που εφαρμόζονται στους επεξεργαστές αν και σε ορισμένες περιπτώσεις η ονομασία μπορεί να διαφοροποιείται ({Intel - *XD bit, eXecute Disable*}, {ARM - *XN bit, eXecute Never*}).

- **Canaries**: η συγκεκριμένη μέθοδος χρησιμοποιεί τυχαίες τιμές για να ανιχνεύει πότε τα δεδομένα πανωγράφονται. Πιο αναλυτικά, τα canaries τοποθετούνται πριν από την αρχή των προστατευόμενων δεδομένων και η τιμή τους επαληθεύεται κάθε φορά που τα συγκεκριμένα δεδομένα χρησιμοποιούνται [4, 6]. Επομένως, στην περίπτωση μιας buffer overflow επίθεσης, η πρώτη τιμή που αλλάζει συνήθως είναι αυτή του canary (λόγω της υπέρβασης των ορίων του buffer), η επαλήθευση είναι αποτυχημένη και τελικά ανιχνεύται η επίθεση. Οι compilers τόσο σε Linux [7] όσο και σε Windows [3] υποστηρίζουν την προαναφερθείσα τεχνική για την αποτροπή υπερχείλισης της στοίβας.

- **Bounds Checking**: μέθοδος η οποία ελέγχει αν μια μεταβλητή είναι εντός κάποιων καθορισμένων ορίων. Χρησιμοποιείται για να εξασφαλίσει είτε ότι ένας αριθμός ταιριάζει σε ένα συγκεκριμένο τύπο (range checking) είτε ότι μια μεταβλητή που χρησιμοποιείται ως δείκτης σε έναν πίνακα είναι εντός των ορίων του πίνακα (index checking).

## 2.2 Format String

### 2.2.1 Συνοπτική Περιγραφή

Η επίθεση αυτή εκμεταλλεύεται μια συγκεκριμένη αδυναμία που μπορεί να εμφανιστεί σε προγράμματα γραμμένα σε C, Perl κλπ. Αυτή η αδυναμία έγκειται στον τρόπο που 'τυπώνονται' στο standard output κάποιες μεταβλητές του προγράμματος και υλοποιείται, αν αυτές οι μεταβλητές που 'τυπώνονται', ζητήθηκαν από τον χρήστη (ενδεχομένως κακόβουλο) κατά την διάρκεια εκτέλεσης του προγράμματος ή δόθηκαν ως παράμετροι κατά την κλήση του προγράμματος. Ο κακόβουλος χρήστης επωφελείται από το format string vulnerability μέσω της 'έξυπνης' χρησιμοποίησης των format specifiers.

Συνέπεια των παραπάνω, είναι η εξαγωγή ευαίσθητων πληροφοριών, οι οποίες κανονικά, δεν θα έπρεπε να είναι προσβάσιμες.

Στη συνέχεια, παρατίθεται ένα παράδειγμα σε κώδικα C, για την πλήρη κατανόηση της αδυναμίας του format string καθώς και ένας από τους τρόπους που αυτή μπορεί να εκμεταλλευθεί από τον επιτιθέμενο και να οδηγήσει σε διάφορα ανεπιθύμητα αποτελέσματα.

```
printf ("%s%s%s%s%s%s%s%s%s%s%s%s");
```

Σχήμα 2.2: Ένας από τους τρόπους έκθεσης της αδυναμίας του format string που οδηγεί σε κατάρρευση του προγράμματος.

1. Για κάθε %s, η συνάρτηση printf() θα προσκομίσει έναν αριθμό από τη στοίβα, θα χρησιμοποιήσει αυτόν τον αριθμό σαν διεύθυνση και στη συνέχεια θα εκτυπώσει, σαν συμβολοσειρά, τα περιεχόμενα της μνήμης που ορίζονται από την διεύθυνση αυτή. Η εκτύπωση τερματίζει μόλις προσπελαστεί ο χαρακτήρας τερματισμού.
2. Αν ο αριθμός που προσκομισθεί από την printf() δεν αντιστοιχεί σε κάποια διεύθυνση, η μνήμη που ορίζεται από τον αριθμό δεν υφίσταται και τότε το πρόγραμμα καταρρέει.
3. Στην περίπτωση που ο αριθμός τυχαίνει να αντιστοιχεί σε σωστή διεύθυνση αλλά ο συγκεκριμένος χώρος είναι μη προσπελάσιμος, τότε και πάλι το πρόγραμμα θα καταρρεύσει.

### 2.2.2 Τεχνικές Αντιμετώπισης

Σε αυτό το σημείο περιγράφονται οι διάφορες τεχνικές που υπάρχουν για την αντιμετώπιση του format string, έτσι ώστε να μην γνωστοποιούνται ευαίσθητα δεδομένα στον κακόβουλο χρήστη κατά την εκτέλεση του προγράμματος. Περιληπτικά, οι μηχανισμοί αυτοί είναι οι παρακάτω:

- **Address Randomization**: ο μηχανισμός είναι ο ίδιος με αυτόν που αναφέρθηκε και προηγουμένως για την προστασία από buffer overflow επιθέσεις. Όπως εκεί, έτσι και σε αυτήν την περίσταση, η τεχνική τυχαιοποιεί διάφορα μέρη του χώρου διευθύνσεων με σκοπό την δύσκολη πρόβλεψη της θέσης τους. Έτσι, καθίσταται δύσκολο για τον επιτιθέμενο να βρει την διεύθυνση στην οποία θέλει να γράψει ή να διαβάσει κάποια πληροφορία. Βέβαια, ένα σημαντικό μειονέκτημα είναι ότι δεν μπορεί να αποτρέψει τον κακόβουλο χρήστη από το να προκαλέσει κατάρρευση του συστήματος καθώς δεν γίνεται κάποιος συσχετισμός ανάμεσα στα format specifiers και τα ορίσματα που δέχεται κάθε φορά η συνάρτηση π.χ. η printf.

- **Format Guard**: εργαλείο που αναπτύχθηκε για να παρέχει προστασία, κατά τη διάρκεια εκτέλεσης του προγράμματος, από την αδυναμία format string [8]. Επιγραμματικά, το Format Guard συγκρίνει τον αριθμό των ορισμάτων που δέχεται μια format function με τον αριθμό των ορισμάτων που το format specifier αναμένει από την στοίβα. Αν ο τελευταίος αριθμός είναι μεγαλύτερος από τα ορίσματα που δέχεται η συνάρτηση τότε το εργαλείο ταξινομεί την περίπτωση αυτή ως επίθεση, την καταγράφει και διακόπτει την εκτέλεση του προγράμματος.

- **Libsafe**: παρακολουθεί τις κλήσεις των format functions και πραγματοποιεί ασφαλείς ελέγχους στα ορίσματά τους [9]. Αρχικά, ελέγχει την διεύθυνση επιστροφής και τον frame pointer, εξασφαλίζοντας ότι κανένα από τα παραπάνω δεν πανωγράφεται. Το σενάριο αυτό συμβαίνει όταν χρησιμοποιείται λανθασμένα ένας %n format specifier και γι αυτό, όταν η εφαρμογή συναντήσει μια τέτοια παράμετρο, ελέγχει την διεύθυνση στην οποία πρόκειται να γράψει. Αν η διεύθυνση αντιστοιχεί σε διεύθυνση επιστροφής ή frame pointer, τότε καταγράφει την προσπάθεια ως επίθεση και διακόπτει την εκτέλεση. Ο επόμενος έλεγχος γίνεται για να αποτρέψει τον επιτιθέμενο να διαβάσει τιμές εκτός των περιοχών που έχει ορίσει η εφαρμογή Libsafe. Έτσι, ο κακόβουλος χρήστης δεν έχει πρόσβαση σε δευθύνσεις προορισμού και frame pointers και οποιαδήποτε προσπάθειά του να διαβάσει τιμές από αυτές τις περιοχές, οδηγούν σε σφάλμα.

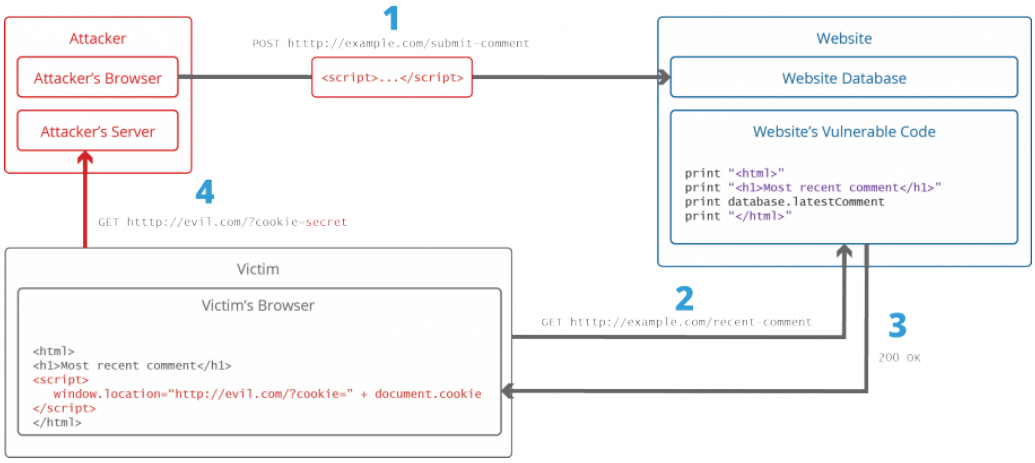
## 2.3 Cross-site scripting

### 2.3.1 Συνοπτική Περιγραφή

Χαρακτηριστικό της επίθεσης αυτής (XSS), που ανήκει στη γενικότερη κατηγορία επιθέσεων που πραγματοποιούνται με διάχυση κώδικα (code injection attacks), είναι η χρησιμοποίηση κακόβουλων scripts τα οποία διαχέονται σε επιλεγμένες ιστοσελίδες. Ο επιτιθέμενος εισάγει κώδικα HTML ή Javascript ο οποίος, επειδή δεν 'μεταχειρίζεται' σωστά από τον ιστοχώρο, προκαλεί προβλήματα στο διαχειριστή ή επισκέπτη του ιστοχώρου [12]. Τα προβλήματα αυτά αφορούν:

- Κλοπή κωδικών/λογαριασμών και λοιπών προσωπικών δεδομένων
- Αλλαγή ρυθμίσεων του ιστοχώρου
- Κλοπή των cookies
- Ψεύτικη διαφήμιση (μέσω, π.χ., ενός συνδέσμου)

Στη συνέχεια, παρατίθεται ένα παράδειγμα που παρουσιάζει βήμα προς βήμα πώς γίνεται η συγκεκριμένη επίθεση, καταδεικνύοντας επίσης πως απαιτούνται τρεις βασικοί 'πρωταγωνιστές' προκειμένου αυτή να ολοκληρωθεί επιτυχώς: *η ιστοσελίδα, το θύμα και ο επιτιθέμενος*.




Σχήμα 2.3: Βήμα προς βήμα παρουσίαση μιας cross-site scripting επίθεσης κατά την οποία γίνεται κλοπή του cookie από τον κακόβουλο χρήστη.

1. Ο επιτιθέμενος διαχέει κακόβουλο κώδικα στη βάση δεδομένων του ιστοτόπου εκμεταλλευόμενος ένα από τα τρωτά σημεία αυτού.
2. Το θύμα αιτείται την ιστοσελίδα.
3. Ο ιστότοπος εξυπηρετεί την αίτηση παρέχοντας στον περιηγητή του θύματος τη σελίδα με τον κακόβουλο εισαγμένο κώδικα ως μέρος του κώδικα HTML.
4. Ο περιηγητής του θύματος εκτελεί το κακόβουλο script, στέλνοντας το cookie του θύματος στον server του επιτιθέμενου και ο τελευταίος το εξάγει όταν φτάσει η αίτηση στον server.

Τέλος, αναφέρουμε πως οι περισσότεροι ειδικοί διακρίνουν τις ευπάθειες από XSS επιθέσεις σε δυο βασικές κατηγορίες: *μη μόνιμες* και *μόνιμες*. Επίσης, δύο άλλες κατηγορίες που μπορούν να χωριστούν είναι, σε *παραδοσιακές επιθέσεις* (που προκαλούνται από την πλευρά του εξηπηρετητή) και σε *επιθέσεις βασισμένες σε DOM*[1] (που προκαλούνται από την πλευρά του πελάτη).

- *Μη μόνιμες*: σε αυτήν την περίσταση, τα ίδια τα θύματα στέλνουν (άθελά τους) το κακόβουλο payload στη σελίδα. Η σελίδα κατασκευάζει την έξοδό της, ενσωματώνοντας αυτόν τον κώδικα, έτσι ο κώδικας επιστρέφει στο θύμα και εκτελείται από τον browser. Η παραπάνω διαδικασία συμβαίνει για παράδειγμα, όταν το θύμα πεισθεί (εξαπατηθεί) να πατήσει πάνω σε κάποιο ειδικά διαμορφωμένο link, το οποίο έχει ενσωματωμένο στο URL του, κακόβουλο κώδικα. Στη συγκεκριμένη περίπτωση, ο επιτιθέμενος ποντάρει στην άγνοια του θύματος. Μια μη μόνιμη επίθεση πραγματοποιείται μόνο όταν το θύμα πατήσει στο μολυσμένο link.
- *Μόνιμες*: σε αυτήν την περίσταση, το κακόβουλο payload δεν εκτελείται μόνο σε κάποιον μεμονωμένο χρήστη, αλλά σε όλους τους επισκέπτες της μολυσμένης σελίδας. Αυτό συμβαίνει γιατί το payload αποθηκεύεται στη βάση δεδομένων του ιστοχώρου και φορτώνεται αυτόματα από τον κώδικα των ιστοσελίδων. Οι ευπάθειες σε μόνιμες XSS επιθέσεις είναι πολύ πιο καταστροφικές καθώς τα δεδομένα που στέλνονται από τον επιτιθέμενο, αποθηκεύονται στον εξυπηρετητή.

[1] DOM ή Document Object Model: αποτελεί έναν τρόπο αναπαράστασης του περιεχομένου μιας σελίδας HTML, ο οποίος τηρεί την ιεραρχία των στοιχείων που την απαρτίζουν

- *επιθέσεις βασισμένες σε DOM*: στο συγκεκριμένο τύπο XSS, ο κακόβουλος κώδικας του επιτιθέμενου εκτελείται ως αποτέλεσμα της τροποποίησης του DOM περιβάλλοντος στον περιηγητή του θύματος. Προϋπόθεση για κάτι τέτοιο είναι η αξιοποίηση των μεθόδων DOM από τη σελίδα, για τη λήψη του URL στο οποίο εμφανίζονται. Για παράδειγμα, στην περίπτωση των μεθόδων document.URL και document.location, αν στο address bar του περιηγητή εισάγουμε κάποιο κακόβουλο payload, αυτές οι μέθοδοι DOM θα το διαβάσουν, θα το ενσωματώσουν στη σελίδα και τελικά θα καταλήξει να εκτελεστεί από τον περιηγητή.

### 2.3.2 Τεχνικές Αντιμετώπισης

Οι cross-site scripting επιθέσεις είναι από τις πιο διαδεδομένες επιθέσεις στο διαδίκτυο και η ανίχνευσή τους καθίσταται δύσκολη, καθώς είναι δύσκολο να διαχωριστεί ποια τμήματα, από ένα HTML document, προέρχονται από ασφαλείς πηγές και ποια εμπεριέχουν δεδομένα μη αξιόπιστης προέλευσης.

Όπως σε κάθε απόπειρα διάχυσης κώδικα (code injection), έτσι και εδώ, η αποτελεσματικότερη άμυνα είναι ο διεξοδικός έλεγχος όλων των δεδομένων που εισάγονται. Οι τεχνικές που χρησιμοποιούνται [13] για την αποτροπή του XSS είναι:

- **Encoding/escaping of string input**: αποτελεί το βασικό μηχανισμό άμυνας έναντι του XSS. Υπάρχουν διάφορες στρατηγικές escaping που χρησιμοποιούνται ανάλογα με το που τοποθετείται, εντός του HTML document, η μη αξιόπιστη συμβολοσειρά (HTML entity encoding, Javascript escaping, CSS escaping, URL encoding). Στον πίνακα της επόμενης σελίδας παρουσιάζονται κάποιες εκ των μεθόδων που χρησιμοποιούνται, για την αυτόματη κωδικοποίηση των δεδομένων που είσαγονται από το χρήστη. Εδώ σημειώνεται πως, πραγματοποιώντας escaping μόνο στους πέντε σημαντικούς χαρακτήρες δεν είναι πάντοτε αρκετό για την αποτροπή των ενεργειών XSS[2]. Σε πολλές περιπτώσεις, οι εφαρμογές χρησιμοποιούν το escaping για να εξαλείψουν σε μεγάλο βαθμό τον κίνδυνο των cross-site scripting επιθέσεων.

- **Validation**: η τεχνική αφορά το φιλτράρισμα της εισόδου του χρήστη έτσι ώστε να απομακρύνονται τα τμήματα εκείνα, που ενδεχομένως οδηγήσουν σε κακόβουλες ενέργειες. Μια από τις πιο γνωστές περιπτώσεις

[2]Οι πέντε ειδικοί χαρακτήρες είναι:
double quotation ( " ) = quot, ampersand ( & ) = amp, apostrophe ( ' ) = apos, less-than sign ( < ) = lt, greater-than sign ( > ) = gt

επικύρωσης εισόδου είναι η έγκριση κάποιων στοιχείων HTML (όπως το  , ) και η απόρριψη άλλων, πιθανώς επικίνδυνων (όπως το <script>).

Υπάρχουν δύο κύρια χαρακτηριστικά επικύρωσης που διαφέρουν ανάλογα με την υλοποίησή τους:

1. **Η είσοδος ταξινομείται χρησιμοποιώντας blacklist ή whitelist**: Στην περίπτωση του blacklisting, ορίζεται ένα απαγορευμένο πρότυπο που δεν πρέπει να εμφανίζεται στην είσοδο. Αν μια συμβολοσειρά ταιριάξει με αυτό, τότε μαρκάρεται ως μη έγκυρη. Η προσέγγιση του whitelisting, ορίζει ένα επιτρεπόμενο πρότυπο βάσει του οποίου αποφασίζει στη συνέχεια αν η συμβολοσειρά είναι έγκυρη ή μη έγκυρη. Η τελευταία μέθοδος είναι το αντίθετο αυτής του blacklisting.
2. **Η είσοδος που αναγνωρίζεται ως κακόβουλη, είτε απορρίπτεται είτε φιλτράρεται**: Στην πρώτη περίπτωση, η είσοδος απορρίπτεται, αποτρέποντας τη χρήση της οπουδήποτε αλλού μέσα στον ιστότοπο. Στην δεύτερη περίπτωση, απομακρύνονται τα κακόβουλα τμήματα και η υπόλοιπη είσοδος χρησιμοποιείται κανονικά.

| | **Μέθοδος Κωδικοποίησης** |
|---|---|
| HTML element content | *node*.textContent = userInput |
| HTML attribute value | *element*.setAttribute(*attribute*, userInput) |
| URL query value | window.encodeURIComponent(userInput) |
| CSS value | *element*.style.property = userInput |

Πίνακας 2.2: Παρουσίαση κάποιων ενσωματωμένων μεθόδων/συναρτήσεων για την αυτόματη κωδικοποίηση των δεδομένων, όταν αυτή συμβαίνει από την πλευρά του client, χρησιμοποιώντας Javascript.

## 2.4 SQL Injection

### 2.4.1 Συνοπτική Περιγραφή

Στη συγκεκριμένη κατάσταση, ο επιτιθέμενος εκτελεί κακόβουλα queries με σκοπό να αποκτήσει πρόσβαση στη βάση δεδομένων μιας εφαρμογής. Επωφελούμενος από την αδυναμία αυτή (SQL Injection vulnerability) και δίνοντας τα κατάλληλα queries, ο κακόβουλος χρήστης μπορεί να παρακάμψει τους μηχανισμούς πιστοποίησης και επαλήθευσης, ανακτώντας τα δεδομένα ολόκληρης της βάσης δεδομένων. Επιπλέον, έχει τη δυνατότητα να τροποποιήσει τα δεδομένα, προσθέτοντας ή αφαιρώντας στοιχεία στη βάση της εφαρμογής, επηρεάζοντας τελικώς την ακεραιότητά τους.

Πιο συγκεκριμένα, για να πραγματοποιηθεί αυτού του είδους η επίθεση, ο ευάλωτος ιστότοπος πρέπει να πιστοποιεί την είσοδο του χρήστη μέσα από ένα κατάλληλο query. Αρχικός στόχος του επιτιθέμενου λοιπόν, είναι η εύρεση ενός τετοιού ιστοτόπου και στη συνέχεια η εισαγωγή της παραλλαγμένης συμβολοσειράς για να αποκτήσει τον έλεγχο της βάσης.

Το SQL Injection συμβαίνει συνήθως όταν ζητούνται από τον χρήστη στοιχεία εισόδου (username, userid) και αντ΄ αυτών, ο χρήστης προσκομίζει μια συνθήκη sql η οποία εκτελείται ακούσια στη βάση δεδομένων [10].

Παρακάτω, παρατίθεται ένα παράδειγμα για την πλήρη κατανόηση του αδύναμου σημείου στον κώδικα, αλλά και ένας από τους τρόπους που η αδυναμία μπορεί να εκτεθεί από τον επιτιθέμενο και να οδηγήσει σε ανεπιθύμητα αποτελέσματα.

```
txtUserId = getRequestString("UserId");
txtSQL = "SELECT * FROM Users WHERE UserId = " + txtUserId;
```

Σχήμα 2.4: Sql κώδικας που παρουσιάζει την αδυναμία του Sql Injection. Ο κακόβουλος χρήστης, παραλλάσσοντας την συμβολοσειρά εισόδου, και αναμειγνύοντας κώδικα Sql και δεδομένα είναι ικανός να ανακτήσει τα στοιχεία της βάσης δεδομένων, προσβάλλοντας την ακεραιότητά τους.

Το παραπάνω παράδειγμα, μέσω της συνθήκης *SELECT*, επιλέγει όλες τις στήλες από τον πίνακα *Users* για την σειρά, της οποίας το *UserId* ταυτίζεται με την μεταβλητή *txtUserId*. Η τιμή στην μεταβλητή δίνεται από το χρήστη κατά την διαδικασία εισόδου (*getRequestString*). Αν δεν υπάρχει κάποιος μηχανισμός που να αποτρέπει το χρήστη να προσκομίσει 'οποιαδήποτε' είσοδο, τότε αυτός μπορεί να εισάγει μια 'έξυπνη' είσοδο όπως: *UserId: 13 or 1=1*. Τότε, το SQL query του παραδείγματος θα είναι το εξής:

*SELECT * FROM Users WHERE UserId = 13 OR 1=1;*

Σχήμα 2.5: Το Sql statement με εισαγμένη την έξυπνη είσοδο από τον κακόβουλο χρήστη.

Η παραπάνω συνθήκη είναι έγκυρη και επιστρέφει όλες τις στήλες του πίνακα *Users* αφού η συνθήκη *OR 1=1* είναι πάντοτε αληθής. Με άλλα λόγια, όταν το query εκτελείται, παρακάμπτεται η είσοδος και είναι πιθανό η εφαρμογή να παράξει πρόσβαση στον επιτιθέμενο με τον πρώτο λογαριασμό που επιστρέφει η συνθήκη και ο οποίος είναι συνήθως αυτός του χρήστη - διαχειριστή.

### 2.4.2 Τεχνικές Αντιμετώπισης

Σε αυτό το σημείο περιγράφονται οι διάφορες τεχνικές [11] που υπάρχουν για την αντιμετώπιση του SQLi έτσι ώστε να μην καθίσταται δυνατή η πρόσβαση στη βάση και η προσβολή της ακεραιότητας των δεδομένων της. Περιληπτικά, οι μηχανισμοί αυτοί είναι οι παρακάτω:

- **Prepared Statements**: Η βάση μιας SQLi επίθεσης είναι η ανάμειξη κώδικα και δεδομένων, επομένως, η ιδέα πάνω στην οποία βασίζεται η τεχνική αυτή είναι η ξεχωριστή αποστολή του query και των δεδομένων, στον SQL server. Πιο αναλυτικά, στην περίπτωση των Prepared Statements το πρόγραμμά μας παραμένει άθικτο καθώς, πρώτα γίνεται η αποστολή του query (όπου τα δεδομένα έχουν αντικατασταθεί από μεταβλητές συγκεκριμένου τύπου) στον server και στη συνέχεια στέλνονται, με δεύτερη αίτηση, τα δεδομένα, εντελώς διαχωρισμένα από τον κώδικα. Σε ενδεχόμενη προσπάθεια, λοιπόν, του επιτιθέμενου να εισάγει ένα κακόβουλο query, το τελευταίο θα φτάσει στον server ως δεδομένο και δεν θα εκτελεστεί ως εντολή. Επιπλέον, στην περίπτωση που η μεταβλητή έχει ορισθεί ως αριθμός (κάποιο id για παράδειγμα) και το query είναι κακόβουλο (συμβολοσειρά ή κάποιος μή έγκυρος τύπος)

τότε, θα παραχθεί σφάλμα ή μηδενική τιμή λόγω του διαφορετικού τύπου μεταβλητής κάθε φορά.

- **Stored Procedures**: τεχνική, της οποίας η λειτουργία είναι παρόμοια με αυτήν της προαναφερθείσας και βασίζεται στην χρήση παραμετροποιημένων queries. Οι Stored Procedures διαχωρίζουν τον κώδικα από τα δεδομένα, γράφοντας το query εκ των προτέρων και μαρκάρωντας τις παραμέτρους, έτσι ώστε αργότερα να εισαχθούν τα δεδομένα. Αφού έχει γίνει ο διαχωρισμός αυτών των δύο, τα κακόβουλα δεδομένα δε χρειάζεται να 'φιλτραριστούν' καθώς η βάση ήδη γνωρίζει ότι πρόκειται για δεδομένα και δεν θα τα αντιμετωπίσει ως εντολές.

- **White List Input Validation**: μηχανισμός ελέγχου των δεδομένων που εισάγονται, κατά τον οποίο καθορίζονται εκ των προτέρων ονόματα πινάκων/στηλών της βάσης δεδομένων και μόνο αυτά γίνονται αποδεκτά. Σε περίπτωση που δοθεί κάτι διαφορετικό από αυτά που έχουν προσδιορισθεί, τότε προκύπτει κάποιο exception.

- **Escaping**: τεχνική η οποία προτιμάται μόνο όταν οι υπόλοιποι μηχανισμοί δεν είναι εφικτοί. Η βασική ιδέα αυτής είναι η διαφυγή (Escaping) των χαρακτήρων εκείνων που έχουν ειδική σημασία στην SQL. Οι οδηγίες του SQL DBMS παρέχουν γνώση για το ποιοι είναι αυτοί οι ειδικοί χαρακτήρες, επιτρέποντας έτσι την κατηγοροιοποίησή τους, προκειμένου αργότερα να χρησιμοποιηθεί η τεχνική. Για παράδειγμα, κάθε εμφάνιση του συμβόλου ( ′ ) σε μία παράμετρο πρέπει να αντικαθίσταται από ( ″ ) για να αποτελεί μια έγκυρη SQL συμβολοσειρά. Γενικά, η τεχνική είναι επιρρεπής σε λάθη καθώς είναι εύκολο να ξεχαστεί να γίνει διαφυγή σε κάποιον από τους ειδικούς χαρακτήρες.

- **Hex-encoding all input**: ο μηχανισμός μπορεί να θεωρηθεί υποκατηγορία της παραπάνω τεχνικής καθώς κωδικοποιεί δεκαεξαδικά, ολόκληρη τη συμβολοσειρά που δέχεται ως είσοδο (μπορεί να λογιστεί επομένως ως escaping του κάθε χαρακτήρα). Η τεχνική θα πρέπει να έχει κωδικοποιήσει (hex-encode) τα δεδομένα εισόδου του χρήστη πριν τα συμπεριλάβει στο SQL query. Στη συνέχεια, τα δεδομένα συγκρίνονται και στην περίπτωση που ο επιτιθέμενος έχει διαβιβάσει κάποιο κακόβουλο query, η επίθεση αναχαιτίζεται.

  Για παράδειγμα, το παρακάτω:

  *SELECT * FROM Users WHERE hex_encode (UserId) = '616263313233';*

  αντιστοιχεί σε δεκαεξαδικά κωδικοποιημένο query στο οποίο δόθηκε είσοδος από το χρήστη το *UserId = abc123.*

Αν ο επιτιθέμενος προσπαθήσει να διαβιβάσει κακόβουλο query που περιέχει το χαρακτήρα ( ′ ) και στη συνέχεια την υπόλοιπη επίθεσή του, τότε η συνθήκη SQL που θα προκύψει μετά την κωδικοποίηση θα είναι η:

*SELECT * FROM Users WHERE hex_encode (UserId) = '27.....';*

Η τιμή 27 είναι ο κώδικας ASCII σε δεκαεξαδικό, του συμβόλου ( ′ ), το οποίο κωδικοποιείται κανονικά όπως οι υπόλοιποι χαράκτηρες. Επομένως, στην τελική συνθήκη εμπεριέχονται μόνο *τα ψηφία 0-9 και τα γράμματα a-f* και κανένας ειδικός χαρακτήρας που μπορεί να προκαλέσει SQL injection.

## 2.5 Path Traversal

### 2.5.1 Συνοπτική Περιγραφή

Μια τέτοιου είδους επίθεση, που είναι επίσης γνωστή και ως *Directory Traversal*, επιτρέπει στον επιτιθέμενο να αποκτήσει πρόσβαση σε απόρρητα directories και να εκτελέσει εντολές εκτός του root directory του web server. Ο κακόβουλος χρήστης εκμεταλλεύεται την αδυναμία αυτή, παραποιώντας μεταβλητές που αναφέρονται σε αρχεία με τη χρήση της ακολουθίας (../) και των παραλλαγών της ή χρησιμοποιώντας ολόκληρα file paths. Αποκτά έτσι αυθαίρετα, πρόσβαση σε αρχεία και directories που μπορεί να περιέχουν τον πηγαίο κώδικα της εφαρμογής ή άλλα σημαντικά στοιχεία για τον τρόπο διαμόρφωσης και σύνθεσης της.

Στη συνέχεια, παρατίθεται ένα παράδειγμα για την πλήρη κατανόηση του ευπαθούς σημείου στον κώδικα αλλά και ένας από τους τρόπους που η αδυναμία μπορεί να εκτεθεί από τον επιτιθέμενο, παρεμβαίνοντας κατάλληλα, προκειμένου να αποκτήσει πρόσβαση σε σημαντικές και απόρρητες πληροφορίες.

Με το παραπάνω URL, ο περιηγητής αιτείται την σελίδα show.asp από τον server και επίσης στέλνει την παράμετρο view μαζί με την τιμή του oldarchive.html. Όταν η αίτηση εκτελείται, το show.asp ανακτά το αρχείο

```
GET http://example.com/show.asp?view=oldarchive.html HTTP/1.1
Host: example.com
```

Σχήμα 2.6: Παράδειγμα HTTP αίτησης μέσω της συνθήκης GET.

oldarchive.html από το file system του server και το επιστρέφει στον περιηγητή ο οποίος το εμφανίζει στο χρήστη. Ο επιτιθέμενος υποθέτοντας ότι το show.asp ανακτά αρχεία από το file system στέλνει το ακόλουθο url:

```
GET http://example.com/show.asp?view=../../../../../Windows/ system.ini
Host: example.com
```

Σχήμα 2.7: Παραποιημένο url που εκμεταλλεύεται την αδυναμία του Path Traversal, με τη χρήση της ακολουθίας (../) και των παραλλαγών της

Το παραπάνω έχει ως αποτέλεσμα, η σελίδα show.asp να ανακτήσει το αρχείο system.ini από το file system και να το εμφανίσει στον χρήστη. Η έκφραση (../) δίνει εντολή στο σύστημα να προχωρήσει στο ανώτερο, ιεραρχικά, directory. Επομένως ο επιτιθέμενος το μόνο που χρειάζεται να κάνει, είναι να μαντέψει πόσα directories πρέπει να 'ανέβει' για να προσπελάσει τον φάκελο των Windows στο σύστημα, πράγμα όμως που είναι αρκετά εύκολο καθώς μπορεί να επιτευχθεί μέσα απο συνεχόμενες δοκιμές.

### 2.5.2 Τεχνικές Αντιμετώπισης

Σε αυτό το σημείο περιγράφονται οι διάφορες τεχνικές που υπάρχουν για την αντιμετώπιση του Path Traversal έτσι ώστε να μην καθίσταται δυνατή η πρόσβαση σε αρχεία ή directories που εμπεριέχουν σημαντικές πληροφορίες για τη λειτουργία του συστήματος.

- **Access Control Lists**: Η ACL αποτελεί μια λίστα που χρησιμοποιεί ο διαχειριστής του web server για να υποδεικνύει ποιοι χρήστες ή ομάδες χρηστών έχουν την δυνατότητα πρόσβασης, τροποποίησης και εκτέλεσης συγκεκριμένων αρχείων στον server, καθώς επίσης και άλλα δικαιώματα. Με άλλα λόγια, η ACL αφορά μια λίστα με τις εξουσιοδοτήσεις που έχουν οριστεί για κάθε αρχείο, προσδιορίζει ποιοι χρήστες έχουν πρόσβαση σε συγκεκριμένα αρχεία και ευαίσθητες πληροφορίες

καθώς και τι είδους διεργασίες επιτρέπεται να εκτελεί ο κάθε χρήστης. Για παράδειγμα, αν ένα αρχείο έχει ACL που περιέχει (Alice: read,write; Bob: read), αυτό σημαίνει ότι η Alice έχει άδεια για να διαβάσει και να γράψει το αρχείο ενώ ο Bob μόνο για να το διαβάσει. Παράδειγμα εφαρμογής του συγκεκριμένου τρόπου σε λειτουργικό σύστημα Linux αποτελεί το SELinux[14].

- **Root Directory**: Το Root Directory είναι ένα συγκεκριμένο directory στο file system του server πάνω (ιεραρχικά) από το οποίο, ο χρήστης δεν μπορεί να προσπελάσει απολύτως τίποτα. Αποτρέπει έτσι τον κακόβουλο χρήστη από την πρόσβαση σε ευαίσθητες πληροφορίες όπως το cmd στα Windows και το passwd αρχείο στα Linux. Για παράδειγμα, το root directory του IIS στα Windows είναι C:\Inetpub\wwwroot και λόγω αυτού, ένας χρήστης δεν έχει πρόσβαση στο C:\Windows αλλά προσπελαύνει το C:\Inetpub\wwwroot\news και οποιοδήποτε άλλο directory ή αρχείο υπάρχει ιεραρχικά κάτω από αυτό το root directory.

## 2.6 Command Injection

### 2.6.1 Συνοπτική Περιγραφή

Η επίθεση αυτή συμβαίνει όταν ο επιτιθέμενος επιχειρεί να πραγματοποιήσει διάχυση εντολών του συστήματος (command injection) μέσω μιας ευάλωτης web εφαρμογής. Μια εφαρμογή θεωρείται έκθετη σε command injection επιθέσεις όταν, εκμεταλλευόμενη έξυπνα από τον κακόβουλο χρήστη, επιτρέπει την εκτέλεση μη εξουσιοδοτημένων εντολών συστήματος. Όπως συμβαίνει και με τις υπόλοιπες επιθέσεις που σχετίζονται με τη διάχυση κώδικα, έτσι και εδώ αναμειγνύεται μη έγκυρος κώδικας, από την πλευρά του επιτιθέμενου, με τις 'νόμιμες' εντολές της εφαρμογής. Συνήθως χρησιμοποιούνται συγκεκριμένοι ειδικοί χαρακτήρες ( | ), ( // ), ( ; ) για τη συγχώνευση των εντολών συστήματος και τη δημιουργία των αντίστοιχων κακόβουλων. Το αποτέλεσμα είναι η πρόσβαση σε απόρρητα αρχεία ή η τροποποίηση και διαγραφή τους, προσβάλλοντας έτσι την ακεραιότητα των δεδομένων του θύματος.

Στο παράδειγμα του σχήματος παρατίθεται μια περίπτωση ευπαθούς κώδικα σε command injection επίθεση και αναλύεται ένας από τους τρόπους

```
<?php

$email_subject = "some subject";
if (isset($_GET{'email'}))
{
    system("mail " + $_GET{'email'}) + " -s '" + $email_subject

    +"' < /tmp/email_body", $return_val);
}

?>
```

Σχήμα 2.8: Παράδειγμα εφαρμογής που δέχεται από το χρήστη ένα email (διεύθυνση email και κείμενο), η οποία είναι ευάλωτη σε command injection επίθεση.

έκθεσής του από την πλευρά του επιτιθέμενου. Πιο αναλυτικά, το παρακάτω τμήμα κώδικα δέχεται ως είσοδο την διεύθυνση email του χρήστη στην παράμετρο *"email"* και την τοποθετεί απευθείας στην συνάρτηση *system*. Σημειώνεται πως σκοπός του επιτιθέμενου είναι η διάχυση των κακόβουλων εντολών στην παράμετρο *"email"* με τρόπο τέτοιο που να εξασφαλίζει την σωστή σύνταξη πριν και μετά την παράμετρο αυτή.

Το ευάλωτο σημείο έγγυται στη συνάρτηση *system* η οποία δέχεται απευθείας, χωρίς να πραγματοποιεί κάποιο φιλτράρισμα, την παράμετρο email (παράμετρος GET). Επομένως, από την πλευρά του επιτιθέμενου στόχος είναι η ορθή τοποθέτηση των κατάλληλων εντολών στο

*mail [MISSING PUZZLE PIECE] –s 'some subject' < /tmp/email_body*

έτσι ώστε αυτές να εκτελεστούν ακούσια από την εφαρμογή. Για παράδειγμα, το mail -help θα εκτελεστεί και θα τερματιστεί κανονικά. Ως συνέχεια αυτού, ο κακόβουλος χρήστης προσθέτει και άλλες εντολές, διαχωρίζοντάς αυτές με ερωτηματικό ( ; ). Υποθέτοντας ότι η τελική μορφή της κακόβουλης εισόδου είναι η παράκατω:

*mail –help; wget http://evil.org/attack_program; ./attack_program*
*# -s 'some subject' < /tmp/email_body*

παρατηρείται η χρήση του συμβόλου # το οποίο καθιστά, οτιδήποτε υπάρχει μετά από αυτό, σχόλιο και επομένως μη εκτελέσιμο. Άρα, το παραπάνω ισοδυναμεί με:

*mail –help; wget http://evil.org/attack_program; ./attack_program*

Τελικά, εκτελείται το mail -help, γίνεται λήψη του attack_program από τη διεύθυνση evil.org και πραγματοποιείται η εκτέλεσή του.

### 2.6.2 Τεχνικές Αντιμετώπισης

Για την αντιμετώπιση των command injection επιθέσεων θα πρέπει να γίνεται η χρήση των υπάρχοντων APIs για την εκάστοτε γλώσσα προγραμματισμού. Για παράδειγμα, στην περίπτωση της Java θα πρέπει να γίνεται η χρήση του Java API javax.mail προκειμένου να μοντελοποιηθεί ένα σύστημα mail [20]. Αν η προαναφερθείσα τεχνική δεν είναι διαθέσιμη τότε θα πρέπει να επικυρώνεται η είσοδος με τη χρήση Regex ή λίστας επιτρεπόμενων τιμών (whitelist).

## 2.7 Σύνοψη

Σε αυτό το κεφάλαιο έγινε προσπάθεια παρουσίασης των ποικίλων αδυναμιών που υπάρχουν όταν εισάγονται δεδομένα σε μια εφαρμογή, αναλύθηκαν οι τρόποι με τους οποίους οι αδυναμίες αυτές εκτείθενται από τους κακόβουλους χρήστες και παρατέθηκαν οι υπάρχοντες αμυντικοί μηχανισμοί που χρησιμοποιούνται για την αποτροπή των κακόβουλων ενεργειών. Για την πλήρη προστασία, οι προγραμματιστές πρέπει να πραγματοποιούν τους κατάλληλους ελέγχους (bounds checking για buffer overflow, SQL escaping για Sql Injection) πριν χρησιμοποιήσουν τα 'μη έμπιστα' δεδομένα σε κρίσιμες, από πλευράς ασφάλειας, πράξεις και διεργασίες. Στον πίνακα που ακολουθεί παρατίθενται συγκεντρωτικά οι μηχανισμοί ασφαλείας που υπάρχουν για την εκάστοτε επίθεση, έτσι όπως παρουσιάστηκαν στο κεφάλαιο αυτό. Ωστόσο, οι τεχνικές που υπάρχουν δεν εγγυώνται πάντοτε την απόλυτη αποτελεσματικότητα, άλλες υστερούν σε παραμέτρους όπως το κατά πόσο είναι πρακτική η χρήση τους, ενώ υπάρχουν και επιθέσεις για τις οποίες δεν υφίσταται κάποια μέθοδος αντιμετώπισης τους.

Πρόσφατες έρευνες έδειξαν ότι ο μηχανισμός DIFT αποτελεί μια πολλά υποσχόμενη τεχνική προς την κατεύθυνση μιας ολοκληρωμένης προσέγγισης

για την αποτροπή των *input validation* επιθέσεων. Στο επόμενο κεφάλαιο γίνεται λεπτομερής παρουσίαση του μηχανισμού μαζί με τις βασικές προϋποθέσεις που πρέπει να υλοποιηθούν και τις παραμέτρους που πρέπει να ληφθούν υπόψιν προκειμένου η τεχνική του DIFT να εφαρμοστεί αποτελεσματικά και στο βέλτιστο βαθμό.

| **Επίθεση** | **Τεχνική Αντιμετώπισης** |
|---|---|
| *Buffer Overflow* | →Address Space Layout Randomization<br>→NX bit (No-eXecute bit)<br>→Canaries<br>→Bounds Checking |
| *Format String* | →Address Space Layout Randomization<br>→Format Guard<br>→Libsafe |
| *Cross-site Scripting* | →Encoding/escaping of input string<br>→Validation |
| *Sql Injection* | →Prepared Statements<br>→Stored Procedures<br>→Whitelist Input Validation<br>→Escaping<br>→Hex-Encoding all input |
| *Path Traversal* | →Access Control Lists<br>→Root Directory |
| *Command Injection* | →Χρήση υπάρχοντων APIs για την εκάστοτε γλώσσα προγραμματισμού |

Πίνακας 2.3: Συγκεντρωτικός πίνακας με τις διάφορες τεχνικές αντιμετώπισης για κάθε είδος επίθεσης.

# Κεφάλαιο 3

# Dynamic Information Flow Tracking

Οι επιτιθέμενοι, συνήθως, εκθέτουν τα ευάλωτα σημεία που υφίστανται σε ένα πρόγραμμα έτσι ώστε να αποκτήσουν μη εξουσιοδοτημένη πρόσβαση στο σύστημα. Οι πιο συχνά εκτιθέμενες αδυναμίες είναι αυτές του buffer overflow και format string που επιτρέπουν στον επιτιθέμενο να πανωγράψει περιοχές της μνήμης με επιβλαβή κώδικα. Δυστυχώς, είναι εξαιρετικά δύσκολη η προστασία ενός προγράμματος, σταματώντας το πρώτο βήμα μιας επίθεσης που είναι η έκθεση του ευάλωτου σημείου, με σκοπό να πανωγραφούν περιοχές της μνήμης. Για την επιτυχή αντιμετώπιση μεγάλου εύρους επιθέσεων οι κακόβουλες ενέργειες πρέπει να αντικρούονται στο τελικό τους στάδιο, που είναι η αυθαίρετη μεταφορά του σημείου ελέγχου προκειμένου να εκτελεστεί ο επιθυμητός, από την πλευρά του επιτιθέμενου, κώδικας. Σε αντίθεση με τα όσα αναφέρθηκαν παραπάνω για τους ποικίλους τρόπους αντιγραφής περιοχών της μνήμης, στην περίπτωση μεταφοράς του σημείου ελέγχου υπάρχουν συγκεκριμένοι τρόποι που επιτυγχάνουν αυτόν το σκοπό. Έτσι, για την κάλυψη ενός μεγάλου φάσματος επιθέσεων, είναι αρκετά πιο εύκολη η προστασία του σημείου ελέγχου, με μοναδική αλλά σημαντική προϋπόθεση την διάκριση ανάμεσα στις κακόβουλες μεταφορές και εκείνες που 'νομίμως' συμβαίνουν κατά την εκτέλεση του προγράμματος.

Το Dynamic Information Flow Tracking [25, 26] είναι ένας μηχανισμός προστασίας από κακόβουλες ενέργειες ο οποίος παρακολουθεί τις ύποπτες ροές πληροφορίας, συνήθως συσχετίζοντας ένα tag bit με κάθε byte μνήμης και κάθε καταχωρητή. Αυτό το tag ορίζεται για κάθε μη έμπιστη πληροφορία που εισάγεται στο σύστημα. Οι εντολές του προγράμματος παράγουν και διαδίδουν τα tag bits, με συγκεκριμένους κανόνες, κατά τη διάρκεια της εκτέλεσης του προγράμματος. Αν η αναξιόπιστη πληροφορία χρησιμοποιείται

επισφαλώς τότε δημιουργείται μια εξαίρεση ασφαλείας, περιορίζοντας ή αποτρέποντας τη χρήση των δεδομένων.

Χαρακτηριστικό της τεχνικής είναι η διάκριση που υφίσταται ανάμεσα στο μηχανισμό και την πολιτική που ακολουθείται κάθε φορά. Πιο συγκεκριμένα, οι πλατφόρμες DIFT παρέχουν τη βασική δομή και οργάνωση για την εκτέλεση των ελέγχων (tag check) και της διάδοσης (tag propagation) των tags στις εφαρμογές. Οι πολιτικές DIFT, που εκτελούνται από τις πλατφόρμες, ορίζουν τις προδιαγραφές και τους κανόνες, βάσει των οποίων θα πραγματοποιηθούν οι έλεγχοι και η παραγωγή των tags, προκειμένου να αντιμετωπισθεί επιτυχώς οποιαδήποτε κακόβουλη προσπάθεια[1]

Εν κατακλείδι, η τεχνική του DIFT αποτελεί ένα πολύ δυνατό εργαλείο έναντι των επιζήμιων επιθέσεων και παρέχει τις απαραίτητες εγγυήσεις για την ασφαλή εκτέλεση ευάλωτων προγραμμάτων.

## 3.1 Η αξιολόγηση βάσει 4 σημείων κλειδιών

Οι βασικές προϋποθέσεις που πρέπει να πληρούνται έτσι ώστε ένας μηχανισμός άμυνας, απέναντι σε κακόβουλες ενέργειες, να θεωρείται υλοποιημένος στο βέλτιστο βαθμό έχουν να κάνουν με τα παρακάτω τέσσερα σημεία κλειδιά. Με βάση αυτά αξιολογούνται όλες οι τεχνικές προστασίας και στη συγκεκριμένη περίπτωση ο μηχανισμός του DIFT.

- **Ασφάλεια**: Σχετίζεται με το κατά πόσο ο μηχανισμός μπορεί να παρακαμφθεί και να καταστρατηγηθεί όταν έρχεται αντιμέτωπος με έναν ικανό επιτιθέμενο.
- **Ταχύτητα**: Αφορά το πόσο γρήγορη είναι η τεχνική, δηλαδή πόσο υψηλή επιβάρυνση υπάρχει στην απόδοση, με την υλοποίησή της.
- **Ευελιξία**: Αφορά στην δυνατότητα εφαρμογής του μηχανισμού σε ένα μεγάλο εύρος επιθέσεων και την αποτροπή έκθεσης ποικίλων αδυναμιών.
- **Πρακτικότητα**: Σχετίζεται με το κατα πόσο είναι εύκολη η χρήση του μηχανισμού σε διαφορετικού τύπου εφαρμογές και στο κατά πόσο είναι απαραίτητη η σε βάθος γνώση της λειτουργικότητας του εσωτερικού της εφαρμογής.

[1] Ως κακόβουλη προσπάθεια εννοείται κάθε ενέργεια που επιχειρεί να εκθέσει τις αδυναμίες που υπάρχουν *κατά τη διαδικασία εισόδου των δεδομένων* (input validation επιθέσεις).

Όσον αφορά την περίπτωση του Dynamic Information Flow Tracking, ο μηχανισμός αποτελεί ένα πολλά υποσχόμενο και ισχυρό εργαλείο απέναντι στις επιθέσεις, καθώς είναι η πρώτη και ίσως μοναδική τεχνική που μπορεί να καλύψει όλες τις προαναφερθείσες προϋποθέσεις. Πιο αναλυτικά, ο DIFT μπορεί να είναι *ασφαλής*, αφού μπορεί να προστατεύσει ολοκληρωτικά από μια επίθεση, εμποδίζοντας μη έμπιστα δεδομένα από την εκτέλεση επισφαλών πράξεων και λειτουργιών. Για παράδειγμα, η τεχνική μπορεί να παρέχει πλήρη προστασία από επιθέσεις διάχυσης κώδικα SQL αφού ο DIFT παρακολουθεί την ροή των αναξιόπιστων πληροφοριών και προσδιορίζει ποια bytes σε ένα query προέρχονται από μη έμπιστη πηγή. Αυτό συντελείται με τη βοήθεια των tags τα οποία προσδιορίζουν με σαφήνεια αν μια πληροφορία είναι αναξιόπιστη. Επιπλέον, εμποδίζουν το χρήστη να παρακάμψει την υφιστάμενη πολιτική τροποποιώντας κατάλληλα την συμβολοσειρά εισόδου ώστε να μοιάζει αβλαβής.

Επίσης, ο DIFT μπορεί να αποτελέσει ένα *ευέλικτο* εργαλείο καθώς οι πολιτικές που έχουν αναπτυχθεί καλύπτουν την προστασία ενός μεγάλου εύρους κακόβουλών ενεργειών, από low level επιθέσεις όπως format string και buffer overflow έως high level επιθέσεις όπως SQL Injection και Path Traversal. Κανένας άλλος υπάρχον μηχανισμός δεν καλύπτει τόσο μεγάλο φάσμα κακόβουλων ενεργειών.

Επιπλέον, οι πλατφόρμες DIFT μπορεί να είναι *γρήγορες* και σημειώνεται πως έχουν υλοποιηθεί πολλά, υψηλής απόδοσης συστήματα, τόσο σε επίπεδο υλικού όσο και σε επίπεδο λογισμικού. Στην ιδεατή περίπτωση, οι πολιτικές DIFT που εφαρμόζονται για την αποτροπή των επιθέσεων πρέπει να έχουν αμελητέες επιβαρύνσεις στην απόδοση και να μην επιβάλλουν περιορισμούς στις εφαρμογές.

Τέλος, ένα σύστημα DIFT μπορεί να είναι και *πρακτικό* καθώς δεν εξαρτάται από τη γνώση της λειτουργικότητας του εσωτερικού της εφαρμογής ή του σχεδιασμού του προγράμματος. Ο μηχανισμός απλά παρακολουθεί και περιορίζει τις ροές των πληροφοριών κατά τη διάρκεια εκτέλεσης του προγράμματος. Αυτός ο σχεδιασμός επιτρέπει την λειτουργία του DIFT σε μη τροποποιημένες, σε επίπεδο κώδικα, εφαρμογές.

Στη συνέχεια, δίνουμε ένα απλό παράδειγμα παρουσιάζοντας την παρακολούθηση της ροής πληροφορίας (*information flow tracking*). Όπως φαίνεται στον Πίνακα 3.1, το a είναι μια μεταβλητή που προέρχεται από μια μη αξιόπιστη πηγή, έστω ότι στην προκειμένη περίπτωση αυτή η πηγή είναι το δίκτυο. Στην αμέσως επόμενη γραμμή, η πληροφορία του a 'ρέει (flows)' στο b. Μαζί με αυτήν, ρέει στο b και η επιπλέον πληροφορία που καταδεικνύει

αν το b είναι και αυτό αναξιόπιστο. Όταν εκτελεστεί η εντολή τύπου jump της τελευταίας γραμμής, και το πρόγραμμα διακλαδωθεί στην θέση που ορίζεται από τη c, θα προκύψει μια εξαίρεση αν τα δεδομένα της c είναι μη αξιόπιστα.

| **Πρόγραμμα** | **Παρακολούθηση Ροής Πληροφορίας** |
|---|---|
| receive(&a); | *Tag(a) = 1* $\rightarrow$ μη έμπιστο καθώς προέρχεται από το δίκτυο |
| b=a; | *Tag(b) = Tag(a)* |
| ⋮ | ⋮ |
| jmp c; | *if(Tag(c) == 1) then raise exception* |

Πίνακας 3.1: Απλό παράδειγμα που παρουσιάζει την διαδικασία της παρακολούθησης της ροής πληροφορίας. Ο συμβολισμός *Tag()* αναφέρεται στην επιπλέον πληροφορία που σχετίζεται με το αν τα δεδομένα προέρχονται από μη έμπιστη πηγή. Μια εξαίρεση προκύπτει όταν η c που χρησιμοποιείται στην εντολή τύπου jump είναι μη ασφαλής.

## 3.2 Πλατφόρμες DIFT

Σε αυτό το σημείο παραθέτονται οι εφαρμογές του μηχανισμού σε επίπεδο υλικού και λογισμικού και αξιολογούνται οι πολιτικές που χρησιμοποιούνται στην εκάστοτε εφαρμογή με βάση τα τέσσερα σημεία - κλειδιά που προαναφέρθηκαν. Από τα παρακάτω γραφόμενα γίνεται αντιληπτό ότι η τεχνική του DIFT αν και έχει την προοπτική να καλύψει όλες τις προϋποθέσεις που απαιτούνται ωστόσο, αρκετές φορές, υστερεί σε κάποια παράμετρο με αποτέλεσμα να μην πληροί όλα τα κριτήρια ταυτόχρονα.

### 3.2.1 Hardware Platforms

Στο επίπεδο του υλικού, έχουν υλοποιηθεί πολλά συστήματα DIFT [22, 24]. Σε αυτά, το runtime περιβάλλον είναι η κεντρική μονάδα επεξεργασίας και η μνήμη αναπαρίσταται από τις CPU caches και την RAM. Η υλοποίηση βασίζεται στην επέκταση των καταχωρητών και των μνημών με ένα ή περισσότερα tag bits, ανάλογα με την πολιτική που ακολουθείται. Η παραγωγή

και διάδοση των tags πραγματοποιείται εντός της CPU παράλληλα με την εκτέλεση των εντολών. Ως προς την αξιολόγηση αυτών των συστημάτων, βάσει των τεσσάρων σημείων που αναφέρθηκαν προηγουμένως, οι πλατφόρμες DIFT είναι γρήγορες, καθώς η αποθήκευση και διάδοση των tags παρέχεται από το υλικό και συμβαίνει σε παραλληλία με την εκτέλεση των εντολών. Ωστόσο, στην περίπτωση του *buffer overflow*, οι υπάρχουσες προσεγγίσεις χρησιμοποιούν μία, προκαθορισμένη πολιτική για την αποτροπή της υπερχείλισης κάτι που καθιστά το σχεδιασμό 'άκαμπτο' ως προς το εύρος των επιθέσεων που αντιμετωπίζει. Τέλος, οι πολιτικές DIFT για τις κακόβουλες ενέργειες υπερχείλισης του buffer δεν είναι πρακτικές και ασφαλείς εξαιτίας των false positives/negatives.

### 3.2.2 Programming Language Platforms

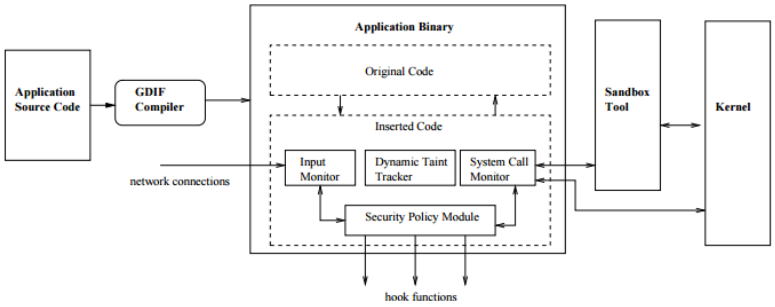
Στις υλοποιήσεις του μηχανισμού DIFT που σχετίζονται με τις γλώσσες προγραμματισμού, προσθέτονται DIFT δυνατότητες στον διερμηνέα. Έχουν προταθεί σχετικές υλοποιήσεις για πολλές γλώσσες προγραμματισμού όπως η C, η Python [15], και η PHP [16]. Επιπλέον, η ιδέα του DIFT χρησιμοποιείται ήδη σε κάποιες περιπτώσεις (taint mode) από ορισμένες γλώσσες όπως η Perl [17, 18] και η Ruby [19]. Ως προς τη λειτουργία της τεχνικής, το runtime περιβάλλον είναι ο διερμηνέας της γλώσσας και η μνήμη αποτελείται από τις μεταβλητές της, οι οποίες έχουν επεκταθεί για την υποστήριξη των tags με σκοπό την παρακολούθηση των μολυσμένων δεδομένων. Τα συστήματα αυτά είναι ευέλικτα και χρησιμοποιούνται με μεγάλη επιτυχία για την αντιμετώπιση high level επιθέσεων, με ελάχιστη επιβάρυνση στην απόδοση. Ερευνητές έχουν αναπτύξει πλατφόρμες DIFT, τροποποιώντας τους διερμηνευτές των σύγχρονων γλωσσών προγραμματισμού με σκοπό την αποφυγή της έκθεσης των σφαλμάτων επικύρωσης εισόδου όπως SQL Injection, Path Traversal και Command Injection. Ως προς την ταχύτητα, τα συστήματα είναι γρήγορα καθώς ο διερμηνευτής μπορεί να βελτιστοποιήσει την αποθήκευση, τον έλεγχο και τη διάδοση των tags. Από την άλλη, οι πλατφόρμες υστερούν ως προς τις άλλες δύο παραμέτρους. Δεν είναι πρακτικές στην περίπτωση που ο χρήστης επιδιώκει να αμυνθεί κατά των αδυναμιών που συμβαίνουν σε ένα εύρος γλωσσών προγραμματισμού, αφού η τεχνική παρέχει ασφάλεια μόνο για την εκάστοτε γλώσσα.

### 3.2.3 Compiler-based DIFT

Μια διαφορετική περίπτωση, στην οποία υλοποιείται η τεχνική DIFT, αφορά και η τροποποίηση κάποιων μεταγλωττιστών, οι οποίοι προσθέτουν την

λειτουργικότητα DIFT, αυτόματα, κατά την διαδικασία της μεταγλώττισης. Πιο συγκεκριμένα, αναλύεται η περίπτωση της ολοκληρωμένης εφαρμογής - compiler Aussum [28], η ανάπτυξη της οποίας βασίζεται στην τεχνολογία του GIFT (General dynamic Information Flow Tracking). Η τελευταία επιτρέπει στον προγραμματιστή να συσχετίσει συγκεκριμένα tags με δεδομένα εισόδου, εξοπλίζει την εφαρμογή ώστε να διαδώσει τα tags αυτά σε όλα τα άλλα δεδομένα που εξαρτώνται από τα αντίστοιχα προηγούμενα και καλεί ειδικές διεργασίες για τα δεδομένα εξόδου αναλόγως με την τιμή που έχει το tag αυτών.

Βάσει των παραπάνω, σχεδιάστηκε ο Aussum compiler ο οποίος αξιοποιώντας τις δυνατότητες του GIFT, μαρκάρει τα περιεχόμενα εκείνα που λαμβάνονται από το Internet. Όταν τα περιεχόμενα αυτά γράφονται σε ένα αρχείο, ο Aussum 'σημαδεύει' το αρχείο με τέτοιο τρόπο, ώστε οποιαδήποτε επακόλουθη εκτέλεση χρησιμοποιεί το αρχείο αυτό να ελέγχεται - περιορίζεται (sandboxed[2]). Επιπλέον, αν κάποιες εφαρμογές χρησιμοποιούν μαρκαρισμένα δεδομένα ως ορίσματα σε συγκεκριμένες κλήσεις συστήματος, όπως για παράδειγμα η *exec*, τότε αυτές οι κλήσεις συστήματος επίσης περιορίζονται - ελέγχονται.




Σχήμα 3.1: Ο Aussum μεταγλωττιστής, η ανάπτυξη του οποίου βασίζεται στην τεχνολογία του GIFT (General dynamic Information Flow Tracking).

Στο σχήμα παρουσιάζεται ο Aussum μεταγλωττιστής. Το *Input Monitor* εντοπίζει τα δεδομένα που διαβάζονται από μια σύνδεση δικτύου και τα μαρκάρει σύμφωνα με μια πολιτική ασφαλείας. Το *Dynamic Taint Tracker* διαδίδει την ένδειξη - σημάδι σε όλο το πρόγραμμα. Το *System Call Monitor* μαρκάρει τα αρχεία και τα ορίσματα σε κλήσεις συστήματος που προέρχονται από 'ύποπτες' πηγές.

[2]Sandbox: μηχανισμός ασφαλείας που παρέχει ένα άκρως ελεγχόμενο περιβάλλον σε επίπεδο πόρων (file descriptors, memory, file system space) για την εκτέλεση 'ύποπτων' προγραμμάτων.

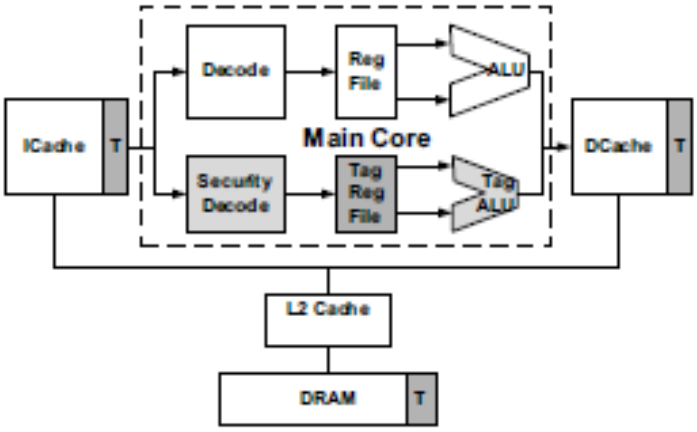
### 3.2.4 Dynamic Information Flow Control

Σε αυτήν την περίπτωση, η εφαρμογή του μηχανισμού DIFT διαφοροποιείται ως προς τις επιθέσεις που καλείται να αντιμετωπίσει. Η τεχνική χρησιμοποιείται όχι για την αντιμετώπιση των *input validation* επιθέσεων αλλά για τις περιπτώσεις που οι ίδιες οι εφαρμογές είναι κακόβουλες. Πιο συγκεκριμένα, η προστασία από *input validation* επιθέσεις σχετίζεται με μια ευάλωτη, αλλά όχι κακόβουλη εφαρμογή, η οποία επεξεργάζεται αναξιόπιστα δεδομένα εισόδου. Οι επιτιθέμενοι λοιπόν, εκτός από την έκθεση των ευάλωτων σημείων, συχνά επιτίθονται και με κακόβουλο λογισμικό ή ιούς. Ωστόσο, από την οπτική του λειτουργικού συστήματος, καθεμία από αυτές τις επιθέσεις είναι παρόμοιες αφού και οι μεν και οι δε στοχεύουν στην εκτέλεση κλήσεων του συστήματος με σκοπό την παράκαμψη και καταστρατήγηση της ασφάλειάς του. Συνεπώς, έχουν προταθεί εφαρμογές βασισμένες σε DIFT για την αποτροπή μη έμπιστων εφαρμογών από την αποκάλυψη, διαρροή ή τροποποίηση ευαίσθητων πληροφοριών χωρίς εξουσιοδότηση. Η τεχνική, που ονομάζεται Dynamic Information Flow Control [21], εφαρμόζει την ιδέα του DIFT στις διεργασίες του λειτουργικού συστήματος. Πιο αναλυτικά, τα προγράμματα αντιμετωπίζονται ως μια ακολουθία από κλήσεις του συστήματος. Το λειτουργικό συσχετίζει ένα tag με κάθε διεργασία. Για παράδειγμα, μια διεργασία που διαβάζει αναξιόπιστες πληροφορίες από το διαδίκτυο επισημαίνεται ως μη έμπιστη ενώ μια διεργασία που διαβάζει το password file επισημαίνεται ως ευαίσθητη. Η προέλευση μιας διεργασίας μπορεί να επηρεάσει το tag, επομένως ένα εκτελέσιμο αρχείο που έχει ληφθεί από το Internet επισημαίνεται εξ'αρχής ως αναξιόπιστο. Σε ένα σύστημα DIFC οι συνδέσεις δικτύου και I/O αρχείων χρησιμεύουν ως tag sources ενώ η επικοινωνία ανάμεσα στις διεργασίες οδηγούν στην διάδοση των tags. Οι έλεγχοι των tags (tag checks) πραγματοποιούνται πριν από οποιαδήποτε διαδικασία μπορεί να οδηγήσει στην μεταφορά απόρρητων πληροφοριών εκτός του συστήματος ή πριν από διαδικασίες που μπορεί να τροποποιήσουν κρίσιμα αρχεία του συστήματος. Οι πολιτικές DIFC που χρησιμοποιούνται συντελούν στην αποτροπή επικοινωνίας, μέσω Internet, μιας εφαρμογής που έχει διαβάσει απόρρητες πληροφορίες ή στην αποτροπή εκτέλεσης μη εξουσιοδοτημένων τροποποιήσεων σε κρίσιμα αρχεία του συστήματος, από μια εφαρμογή που έχει επεξεργαστεί αναξιόπιστα δεδομένα εισόδου.

## 3.3 Εναλλακτικοί Σχεδιασμοί DIFT Συστημάτων

Τα διάφορα συστήματα DIFT που έχουν υλοποιηθεί και προταθεί βασίζονται σε έναν από τους επόμενους τρεις βασικούς σχεδιασμούς: *In-core DIFT, Offloading DIFT* και *Off-core DIFT*. Μια περιεκτική περιγραφή, όπως αυτή παρουσιάζεται στο [23], παρατίθεται παρακάτω.

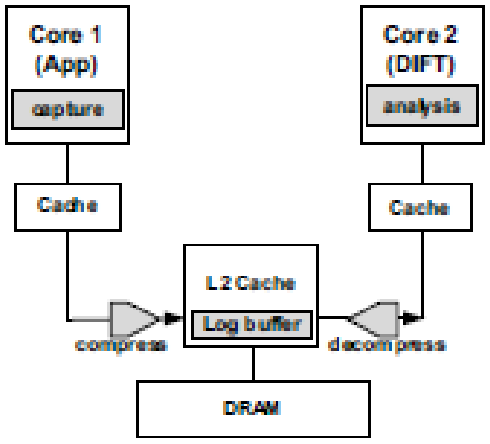
***In-core DIFT***: Σε αυτόν το σχεδιασμό είναι βασισμένες οι περισσότερες υλοποιήσεις συστημάτων DIFT που έχουν προταθεί. Πιο αναλυτικά σε αυτήν την περίπτωση, η λειτουργικότητα της τεχνικής, δηλαδή η διάδοση και ο έλεγχος των tags, συμβαίνει εντός του pipeline του επεξεργαστή και σε παραλληλία με την εκτέλεση των εντολών που φυσιολογικά εκτελεί. Πρόκειται λοιπόν για έναν σχεδιασμό που ενσωματώνει το μηχανισμό DIFT με την εκτέλεση των εντολών, σε διοχέτευση, που πραγματοποιεί ο επεξεργαστής. Ένα σημαντικό μειονέκτημα αυτής της προσέγγισης είναι η σημαντική τροποποίηση που πρέπει να γίνει στον πυρήνα του επεξεργαστή. Όλα τα επίπεδα της διοχέτευσης πρέπει να μεταβληθούν ώστε να αποθηκεύουν τα tags που σχετίζονται με τις εντολές των οποίων εκρεμμεί η εκτέλεση. Επιπλέον, το αρχείο καταχωρητών και οι μνήμες πρέπει να επεκταθούν ώστε να αποθηκεύουν την επιπρόσθετη πληροφορία των tags και για τις εντολές και για τα δεδομένα.




Σχήμα 3.2: *In-core DIFT* σχεδιασμός, με τον μηχανισμό του DIFT να υλοποιείται εντός του πυρήνα του επεξεργαστή και παράλληλα με την pipeline εκτέλεση των εντολών.

Γενικά, οι τροποποιήσεις που επιδέχεται ο πυρήνας του επεξεργαστή είναι σημαντικές και μπορεί να έχουν αρνητικές επιδράσεις στο χρόνο που χρειάζεται για να υλοποιηθεί ο σχεδιασμός. Από την άλλη μεριά σε αυτήν την προσέγγιση δεν απαιτείται επιπρόσθετος πυρήνας για την υποστήριξη της λειτουργικότητας DIFT και κατά συνέπεια δεν υπάρχει κάποιο overhead για το συγχρονισμό των πυρήνων. Στο παραπάνω σχήμα παρουσιάζεται μια γενική εικόνα της υλοποίησης του μηχανισμού DIFT με τον *in-core DIFT* σχεδιασμό.

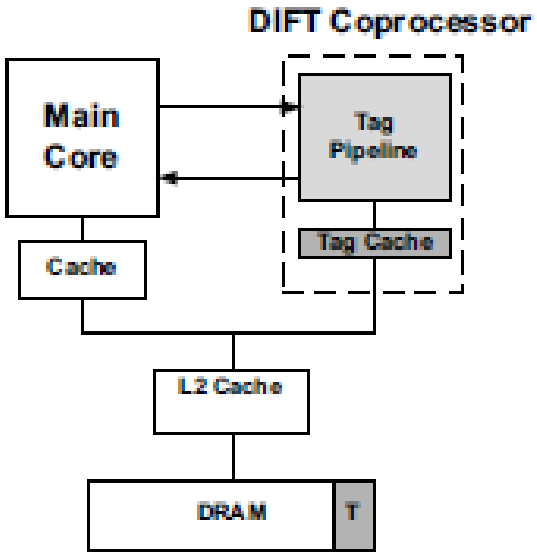
***Offloading DIFT***: Σε αυτή την εναλλακτική προσέγγιση, η λειτουργικότητα DIFT υλοποιείται σε έναν ξεχωριστό πυρήνα [32, 27]. Η εφαρμογή εκτελείται στον ένα πυρήνα ενώ ένας άλλος πυρήνας πραγματοποιεί την ανάλυση του DIFT για την προστασία της εφαρμογής. Το πλεονέκτημα αυτής της προσέγγισης είναι ότι δεν χρειάζεται σαφής και ακριβής γνώση των tags και των πολιτικών DIFT, από το υλικό. Επιπλέον, σε αυτήν την περίπτωση με τους ξεχωριστούς πυρήνες μπορούν να υποστηριχτούν και άλλες τεχνικές δυναμικής ανάλυσης όπως αυτή του memory profiling. Οι δύο πυρήνες συγχρονίζονται μόνο στις κλήσεις του συστήματος. Ένα άλλο προτέρημα, έναντι του ενσωματωμένου σχεδιασμού που παρουσιάστηκε προηγουμένως, είναι η όχι τόσο σημαντική τροποποίηση που απαιτείται να γίνει προκειμένου να υλοποιηθεί αποτελεσματικά ο σχεδιασμός.




Σχήμα 3.3: *Offloading DIFT* σχεδιασμός, με τον μηχανισμό του DIFT να υλοποιείται σε ξεχωριστό πυρήνα από αυτόν που εκτελείται η εφαρμογή.

Πιο αναλυτικά, ο πυρήνας που 'τρέχει' την εφαρμογή τροποποιείται κατάλληλα ώστε να δημιουργεί και να συμπιέζει ένα αρχείο καταγραφής με τις εντολές που έχουν εκτελεστεί. Ο πυρήνας πρέπει να επιλέξει τα γεγονότα εκείνα που θα συμπεριλάβει στο αρχείο, να συγκεντρώσει τις απαιτούμενες πληροφορίες (PC, καταχωρητές και μνήμη που λειτουργούν ως τελεστέοι στις πράξεις που ορίζονται από τις εντολές) και να τις συμπιέσει στο υλικό. Στη συνέχεια, το αρχείο καταγραφής διαμοιράζεται μέσω των L2/L3 caches. Τέλος, ο 'DIFT πυρήνας' αποσυμπιέζει το αντίγραφο στο υλικό και το εκθέτει στο λογισμικό. Το σημαντικότερο μειονέκτημα αυτής της περίπτωσης έγγυται στην χρησιμοποίηση ενός ακόμα πυρήνα για την εφαρμογή της τεχνικής του DIFT. Συνεπώς, περιορίζεται ο αριθμός των διαθέσιμων πυρήνων για άλλα προγράμματα και διπλασιάζεται η ενέργεια που καταναλώνεται εξαιτίας της εφαρμογής που τίθεται υπό ανάλυση. Στο *Σχήμα 3.3* παρατίθεται, σε γενική μορφή, ο τρόπος υλοποίησης του σχεδιασμού, με τον μηχανισμό του DIFT να εκτελείται σε έναν ξεχωριστό πυρήνα.

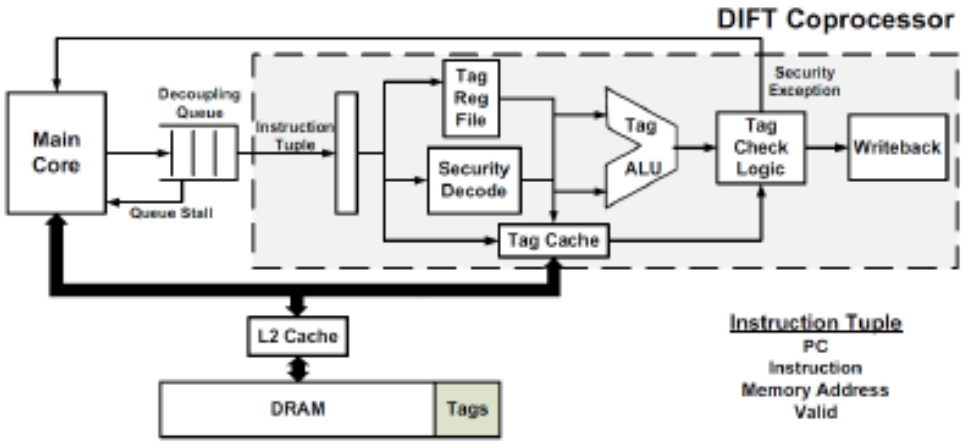
***Off-core DIFT***: Ενδιάμεση περίπτωση των δύο προηγούμενων.




Σχήμα 3.4: *Off-core DIFT* σχεδιασμός, με τον μηχανισμό του DIFT να υλοποιείται σε έναν συνημμένο coprocessor ο οποίος συγχρονίζεται με τον κύριο επεξεργαστή μόνο στις κλήσεις του συστήματος.

Πιο συγκεκριμένα, χρησιμοποιείται ένας μικρός, συνημμένος coprocessor ο οποίος εκτελεί την λειτουργικότητα του μηχανισμού DIFT για τον κύριο πυρήνα και συγχρονίζεται μαζί του μόνο στις κλήσεις του συστήματος. Ο DIFT coprocessor περιλαμβάνει όλο το υλικό που απαιτείται για την υλοποίηση του μηχανισμού. Συγκρινόμενο με τον αμέσως προηγούμενο σχεδιασμό, στον οποίο γίνεται χρήση ενός ακόμα πυρήνα για την εκτέλεση της τεχνικής, δεν απαιτούνται αλλαγές στον επεξεργαστή και στις μνήμες για τον διαμοιρασμό του αντιγράφου. Συγκρινόμενο με τον πρώτο σχεδιασμό, αυτόν του ενσωματωμένου DIFT, ο coprocessor εξαλείφει την ανάγκη για τροποποιήσεις στη σχεδίαση, τη διοχέυτεση και τη διάταξη του κύριου πυρήνα. Συνεπώς, δεν υπάρχει επιβάρυνση στον χρόνο σχεδίασης του συστήματος ή τη συχνότητα του ρολογιού στον κύριο πυρήνα. Ο σχεδιασμός ο οποίος παρέχει τις ίδιες εγγυήσεις ασφάλειας με τις περιπτώσεις του ενσωματωμένου DIFT παρουσιάζεται, στη γενική του μορφή, στο *Σχήμα 3.4*.

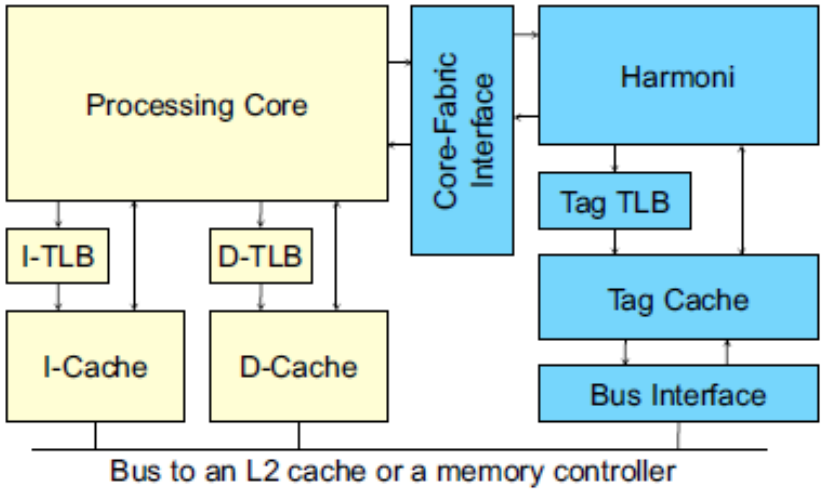
Χαρακτηριστικό παράδειγμα του *Off-core* σχεδιασμού αποτελεί η μελέτη και δημοσίευση του H. Kannan [31] η οποία αφορά μια αρχιτεκτονική που εκτελεί όλη τη λειτουργικότητα DIFT σε έναν μικρό, συνημμένο co-processor ο οποίος συγχρονίζεται με τον κύριο επεξεργαστή στις κλήσεις συστήματος. Το *Σχημα 3.5* απεικονίζει το pipeline διάγραμμα έτσι όπως αυτό προτάθηκε από τον H. Kannan.




Σχήμα 3.5: *Το pipeline διάγραμμα για τον DIFT co-processor έτσι όπως προτάθηκε από τον H. Kannan.*

Μια άλλη, παρεμφερής περίπτωση του παραπάνω σχεδιασμού αποτελεί η αρχιτεκτονική *Harmoni*[30] που μπορεί να υποστηρίξει ένα ευρύ φάσμα

μηχανισμών παρακολούθησης της ροής δεδομένων στο υλικό, ανάμεσα στους οποίους και τον DIFT. Το Block Diagram της αρχιτεκτονικής παρουσιάζεται παρακάτω.




Σχήμα 3.6: *High Level Block Diagram της Αρχιτεκτονικής Harmoni.*

Επιγραμματικά, η αρχιτεκτονική *Harmoni* σχεδιάστηκε σε έναν παράλληλο, διαχωρισμένο co-processor και υποστηρίζει διάφορες τεχνικές παρακολούθησης, προσθέτοντας την απαιτούμενη υποστήριξη, στο επίπεδο του υλικού, στον πυρήνα προκειμένου να προωθήσει ένα 'αντίγραφο' (opcode εντολής, δείκτες για τους καταχωρητές προέλευσης και προορισμού, κλπ.) από τον κύριο πυρήνα στη *Harmoni*.

## 3.4 Σύνοψη

Σε αυτό το κεφάλαιο παρουσιάστηκε ο μηχανισμός του DIFT, δίνοντας βάση σε όλες τις σημαντικές πληροφορίες που σχετίζονται με αυτόν. Επιγραμματικά, αφού επισημάνθηκαν οι λόγοι για τους οποίους η συγκεκριμένη τεχνική θεωρείται πολλά υποσχόμενη, παρατέθηκαν οι διάφορες υφιστάμενες πλατ-

φόρμες που υποστηρίζουν το μηχανισμό (είτε σε επίπεδο λογισμικού είτε σε επίπεδο υλικού) και οι διαφορετικοί σχεδιασμοί που έχουν προταθεί για την υλοποίηση της τεχνικής αυτής.

Συμπερασματικά, ο DIFT ναι μεν αποτελεί ένα πολύ ισχυρό εργαλείο έναντι μεγάλου εύρους επιθέσεων και τη λύση στην εξάλειψη ενός πλήθους αδυναμιών, αλλά οι υλοποιήσεις του πολλές φορές έχει σημαντικά μειονεκτήματα που σχετίζονται με τα τέσσερα σημεία - κλειδιά που αναφέρονται στο κεφάλαιο αυτό.

# Κεφάλαιο 4

# Ο επεξεργαστής MIPS

Ο MIPS σχεδιάστηκε το 1981 από ερευνητές στο Stanford University και εισήχθη για πρώτη φορά στην αγορά το 1984 από την MIPS Computer Systems. Προτάθηκε από τον David Patterson σαν εναλλακτική επιλογή των μέχρι τότε CISC συστημάτων. Είναι ένας RISC (Reduced Instruction Set Computer)[1] επεξεργαστής, που λόγω του απλού συνόλου εντολών, αποτελεί ένα πολύ καλό εκπαιδευτικό εργαλείο. Η αρχική του έκδοση ήταν 32-bits ενώ στη συνέχεια βγήκαν και άλλες εκδόσεις αρκετά πιο προχωρημένες και βελτιστοποιημένες όπως ο MIPS R10000 στα 64-bits.

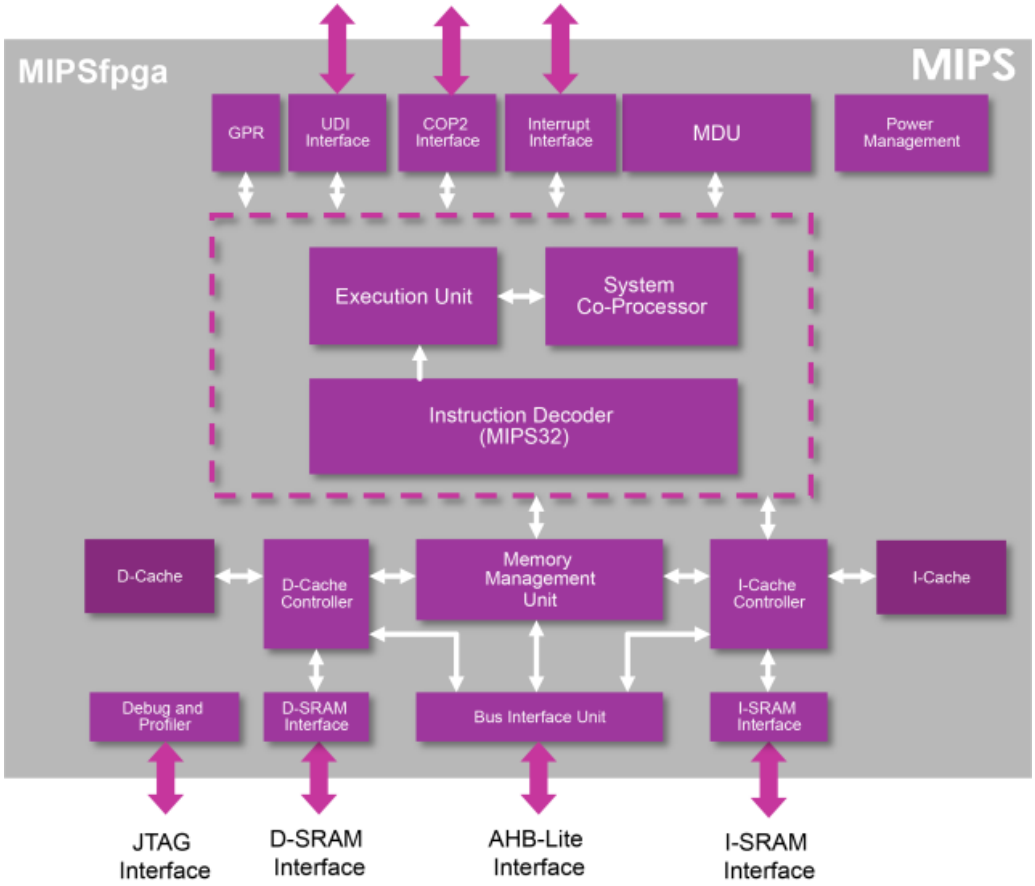
## 4.1 Πυρήνας MIPSfpga

Ο πυρήνας MIPSfpga ορίζεται από τη γλώσσα περιγραφής υλικού Verilog. Αποκαλείται και soft core processor καθώς προσδιορίζεται στο software (Verilog) αντί να κατασκευάζεται σε τσιπάκι. Έχει τα ακόλουθα χαρακτηριστικά:

- microAptiv UP core [33] που υποστηρίζει την αρχιτεκτονική συνόλου εντολών MIPS32, με 5 επίπεδα διοχεύτεσης (pipeline).
- 4KB συσχετιστική μνήμη συνόλου 2 δρόμων για εντολές (instruction cache) και δεδομένα (data cache).
- Μονάδα διαχείρισης μνήμης (MMU) με TLB 16 καταχωρήσεων.
- EJTAG programmer/debugger.
- CorExtend για εντολές που ορίζονται από το χρήστη.
- Μετρητές απόδοσης.

[1] RISC: Τύπος αρχιτεκτονικής μικροεπεξεργαστών που χρησιμοποιεί ένα μικρό, εξαιρετικά βελτιστοποιημένο σύνολο εντολών.

Στη συνέχεια, παρατίθεται το block diagram [33] του MIPSfpga επεξεργαστή. Το κεντρικό τμήμα του επεξεργαστή αποτελεί την Execution Unit στην οποία εκτελούνται οι πράξεις που ορίζονται από τις εντολές, όπως μια πρόσθεση ή αφαίρεση. Η MDU (multiply/divide unit) πρόκειται για μια επέκταση της μονάδας αυτής που εκτελεί τις πράξεις του πολλαπλασιασμού και της διαίρεσης. Το Instruction Decoder παραλαμβάνει τις εντολές από την instruction cache και παράγει τα κατάλληλα σήματα προκείμενου η μονάδα εκτέλεσης να πραγματοποιήσει την πράξη που ορίζει η εκάστοτε εντολή. Η μονάδα System Co-Processor παρέχει σήματα διεπαφής όπως το ρολόι του συστήματος και το reset. Η μονάδα GPR (general purpose registers) περιλαμβάνει τους καταχωρητές οι οποίοι χρησιμοποιούνται αργότερα ως τελεστέοι (καταχωρητής προορισμού, προέλευσης) στην πράξη που ορίζει η εντολή.




Σχήμα 4.1: Block diagram του πυρήνα MIPSfpga.

Τα υπόλοιπα τμήματα που παρουσιάζονται στην πάνω πλευρά του *Σχήμα-*

τος 4.1 (UDI Interface, COP2 Interface, Interrupt Interface) επιτρέπουν στον επεξεργαστή να υποστηρίζει, ορισμένες από το χρήστη, εντολές, να διασυνδέεται με μονάδα coprocessor 2 και να λαμβάνει εξωτερικές διακοπές, αντίστοιχα. Η μνήμη για τις εντολές (instruction cache) και τα δεδομένα (data cache) είναι συνδεδεμένη με τους αντίστοιχους ελεγκτές (I-Cache Controller και D-Cache Controller) και με τη μονάδα διαχείρισης μνήμης (MMU). Η τελευταία πραγματοποιεί τις μεταφράσεις των διευθύνσεων και προσκομίζει τις εντολές ή τα δεδομένα από τη μνήμη όταν αυτά δεν είναι διαθέσιμα στην instruction cache και data cache αντίστοιχα. Τέλος, η μονάδα Debug and Profiler παρέχει την δυνατότητα του EJTAG περιβάλλοντος για αποσφαλμάτωση, παρακολούθηση της απόδοσης του επεξεργαστή και λήψη κώδικα σε αυτόν.

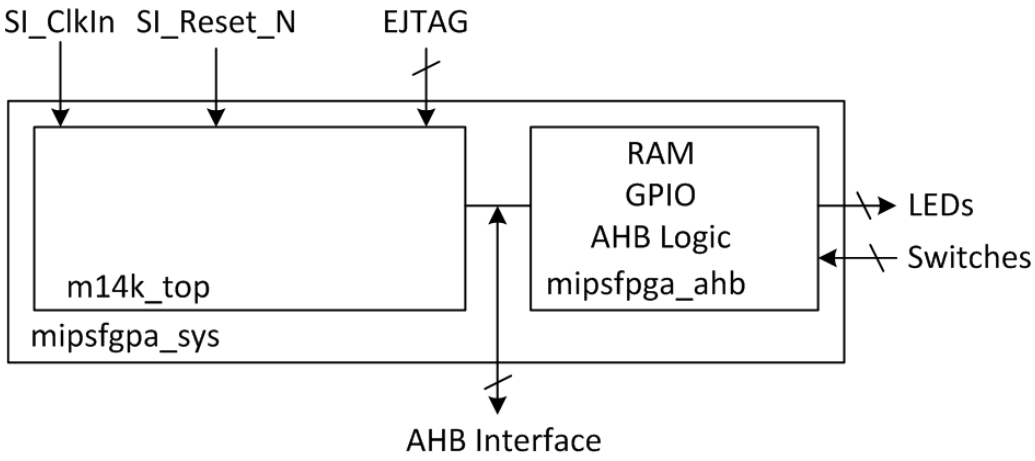
Ο MIPSfpga core έχει πέντε επίπεδα διοχέτευσης (αναλυτική περιγραφή στην ενότητα 4.4). Στον *Πίνακα 4.1* που ακολουθεί παρουσιάζονται τα στάδια αυτά με μια σύντομη περιγραφή της λειτουργίας του κάθε επιπέδου.

| | **Επίπεδο** | **Όνομα επιπέδου** | **Περιγραφή** |
|---|---|---|---|
| 1 | I | Instruction Fetch | Ο επεξεργαστής προσκομίζει μια εντολή |
| 2 | E | Execute | Ο επεξεργαστής προσκομίζει τους καταχωρητές - τελεστέους από το αρχείο καταχωρητών και εκτελεί μια πράξη ALU (π.χ. πρόσθεση, αφαίρεση, υπολογισμός διεύθυνσης μνήμης) |
| 3 | M | Memory | Προσπελάζεται η μνήμη, αν απαιτείται |
| 4 | A | Align | Αν απαιτείται, τα δεδομένα ευθυγραμμίζονται εντός των ορίων μιας λέξης |
| 5 | WB | Writeback | Αν απαιτείται, ανανεώνεται το αρχείο καταχωρητών γράφοντας τα αποτελέσματα στους καταχωρητές προορισμού |

Πίνακας 4.1: Τα 5 στάδια διοχέτευσης (pipeline) του επεξεργαστή MIPS.

Στη συνέχεια, στο *Σχήμα 4.2* παρουσιάζονται τα σημαντικά τμήματα του MIPSfpga συστήματος. Το σύστημα παίρνει τιμές για το ρολόι, το reset και τα EJTAG σήματα είτε από την πλακέτα fpga είτε από το αρχείο testbench κατά την προσομοίωση. Εντός του mipsfpga_sys τμήματος, εμπεριέχεται το στοιχείο m14k_top (περιλαμβάνει τον πυρήνα και τα δομικά στοιχεία που αναφέρθηκαν προηγουμένως) και το τμήμα mipsfpga_ahb που περιέχει τη RAM και την διεπαφή GPIO.[2]

[2]General Purpose Input/Output: pin στην πλακέτα το οποίο είτε πρόκειται για είσοδο είτε για έξοδο, είναι ελεγχόμενο από το χρήστη κατά τη διάρκεια της εκτέλεσης.




Σχήμα 4.2: Τα βασικά μέρη του MIPSfpga system.

## 4.2 Περιγραφή των βασικών μονάδων

### 4.2.1 General Purpose Registers

Ο πυρήνας microAptiv UP Core περιλαμβάνει 32 καταχωρητές γενικού σκοπού (general purpose, GPR), χωρητικότητας 32-bit που χρησιμοποιούνται σε αριθμητικές πράξεις και πράξεις υπολογισμού διεύθυνσης μνήμης. Προαιρετικά, παρέχεται η δυνατότητα για προσθήκη ενός, τριών, εφτά ή δεκαπέντε επιπρόσθετων set καταχωρητών - shadow sets (το καθένα από τα οποία περιέχει άλλους 32 καταχωρητές) για την ελαχιστοποίηση του overhead κατά την επεξεργασία των διακοπών/εξαιρέσεων. Το αρχείο καταχωρητών αποτελείται από δύο θύρες ανάγνωσης και μια θύρα εγγραφής.

### 4.2.2 System Control Coprocessor

Στην αρχιτεκτονική MIPS, το CP0 είναι υπεύθυνο για τις μεταφράσεις των διευθύνσεων από εικονικές σε φυσικές, τα πρωτόκολλα των μνημών, το σύστημα διαχείρισης εξαιρέσεων, την κατάσταση λειτουργίας (kernel, user, debug) καθώς και για το αν επιτρέπονται ή μη οι διακοπές. Επίσης, πληροφορίες

που αφορούν την διαμόρφωση κάποιων μονάδων του συστήματος, όπως το μέγεθος της μνήμης ή ο αριθμός των συνόλων σε μια συσχετιστική μνήμη, είναι διαθέσιμες και παρέχονται με πρόσβαση στο CP0. Για την ανάγνωση και εγγραφή σε αυτούς τους καταχωρητές χρησιμοποιούνται δύο εντολές, οι οποίες είναι οι εξής: *MFC0 = Move from Coprocessor 0* και *MTC0 = Move to Coprocessor 0*.

### 4.2.3 Execution Unit

Η μονάδα αυτή εφαρμόζει την αρχιτεκτονική load/store[3] με πράξεις ALU που εκτελούνται σε έναν κύκλο ρολογιού (λογικές πράξεις, πρόσθεση, αφαίρεση, shift) και περιλαμβάνει και μια αυτόνομη μονάδα για την εκτέλεση των πράξεων του πολλαπλασιασμού και της διαίρεσης (MDU - Multiply/Divide Unit). Επιγραμματικά, περιλαμβάνει μεταξύ άλλων, την αριθμητική - λογική μονάδα (ALU) για την εκτέλεση αριθμητικών και λογικών πράξεων, συγκριτές για τις συνθήκες branch και trap, μονάδα για τον υπολογισμό του επόμενου PC, μονάδα ανίχνευσης των leading Zeros/Ones για την υποστήριξη των εντολών CLZ και CLO αντίστοιχα.

### 4.2.4 Multiply/Divide Unit

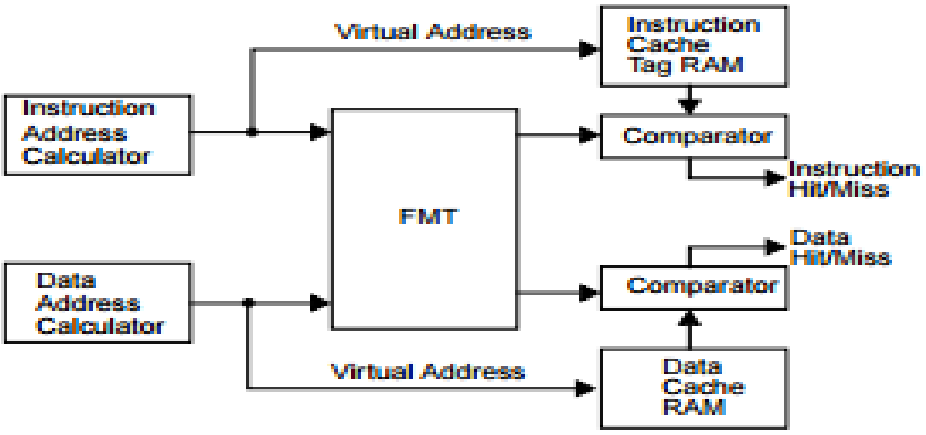
Στη μονάδα αυτή πραγματοποιούνται οι πράξεις του πολλαπλασιασμού και της διαίρεσης σε ένα ξεχωριστό pipeline το οποίο επιτρέπει την ταυτόχρονη λειτουργία με αυτό της ALU και δεν καθυστερεί όταν το pipeline της integer unit καθυστερεί. Το αποτέλεσμα των πράξεων που προκύπτει στην MDU αποθηκεύεται σε ξεχωριστούς καταχωρητές (HI/LO registers) και με την χρήση των ειδικών εντολών mfhi (move from Hi) και mflo (move from Lo), τα αποτελέσματα μεταβιβάζονται στους καταχωρητές γενικού σκοπού, στο αρχείο καταχωρητών.

### 4.2.5 Memory Management Unit

Η μονάδα διαχείρισης μνήμης αποτελεί την διασύνδεση ανάμεσα στην Execution Unit και τους Cache Controllers και είναι υπεύθυνη για τις μεταφράσεις των διευθύνσεων. Ο πυρήνας microAptiv UP Core προσφέρει δύο τρόπους υλοποίησης της MMU: Fixed Mapping Translation (FMT) και Translation Lookaside Buffer (TLB).

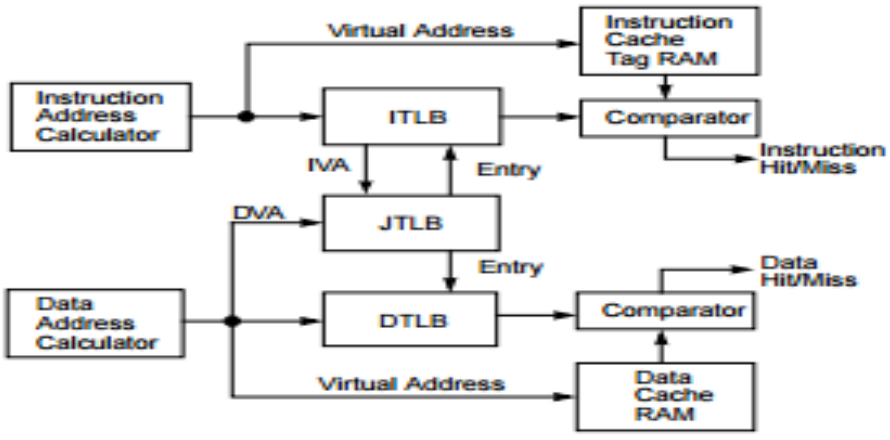
[3] Η αρχιτεκτονική load/store ταξινομεί τις εντολές σε δύο κατηγορίες: σε αυτές που προσπελαύνουν την μνήμη (load, store ανάμεσα στη μνήμη και τους καταχωρητές) και στις πράξεις ALU που εκτελούνται μόνο ανάμεσα σε καταχωρητές.

Το FMT είναι μικρότερο και πιο απλό από το TLB και όπως το τελευταίο, έτσι και το πρώτο, εκτελεί τις μεταφράσεις των διευθύνσεων από εικονικές σε φυσικές. Στο παρακάτω σχήμα παρουσιάζεται πώς γίνονται οι μεταφράσεις στην περίπτωση του FMT-based MMU.




Σχήμα 4.3: Address Translation During Cache Access with FMT Implementation

Μια TLB-based MMU αποτελείται από τρεις buffers (ITLB, JTLB, DTLB) και ο τρόπος που γίνεται η μετάφραση των διευθύνσεων φαίνεται στο παρακάτω σχήμα.




Σχήμα 4.4: Address Translation During a Cache Access with TLB Implementation

### 4.2.6 Caches

#### 4.2.6.1 Cache Controllers

Οι cache controllers για τις εντολές και τα δεδομένα, που παρέχονται στον πυρήνα microAptiv UP Core, υποστηρίζουν κρυφές μνήμες διαφόρων μεγεθών και ποικίλων δρόμων συνολο-συσχετιστικότητας. Για παράδειγμα, η data cache μπορεί να είναι χωρητικότητας 2Kbytes και συσχετιστική συνόλου 2 δρόμων ενώ η instruction cache μπορεί να έχει μέγεθος 8Kbytes και να αποτελεί μια συσχετιστική μνήμη συνόλου 4 δρόμων. Κάθε ελεγκτής είναι συνδεδεμένος με την αντίστοιχη κρυφή μνήμη. Βασική λειτουργία των ελεγκτών είναι, κάθε φορά που ο επεξεργαστής αιτείται κάποια πληροφορία από την μνήμη, να ελέγχουν πρώτα αν το συγκεκριμένο τμήμα πληροφορίας βρίσκεται στην κρυφή μνήμη. Αν βρίσκεται σε αυτήν, τότε η αίτηση ικανοποιείται με πρόσβαση στην κρυφή μνήμη κάτι που εξοικονομεί αρκετό χρόνο σε σχέση με αυτόν που θα απαιτούνταν αν προσπελαζόταν η κύρια μνήμη.

#### 4.2.6.2 Instruction Cache

Η instruction cache είναι ένα μπλοκ μνήμης έως και 64Kbytes. Λόγω της εικονικής δεικτοδότησης, η μετάφραση των διευθύνσεων από εικονικές σε φυσικές συμβαίνει παράλληλα με την πρόσβαση στη μνήμη αντί να υπάρχει αναμονή για την πραγματοποίηση της μετάφρασης αυτής. Η ετικέτα (tag) περιλαμβάνει 22bits φυσικής διεύθυνσης, το ψηφίο εγκυρότητας (valid bit)[4] και ένα lock bit. Τα bits που σχετίζονται με την μέθοδο αντικατάστασης LRU[5] αποθηκεύονται σε έναν ξεχωριστό πίνακα. Το microAptiv UP Core υποστηρίζει επίσης και το instruction cache locking δίνοντας τη δυνατότητα τμήματα δεδομένων να 'κλειδώνονται' μέσα στην μνήμη αποτρέποντας οποιαδήποτε αντιγραφή τους.

#### 4.2.6.3 Data Cache

Η data cache είναι ένα μπλοκ μνήμης έως και 64Kbytes. Όπως και στην κρυφή μνήμη για τις εντολές, η εικονική δεικτοδότηση επιτρέπει να γίνονται παράλληλα η μετάφραση της διεύθυνσης από εικονική σε φυσική και η πρόσβαση στην μνήμη. Το πεδίο της ετικέτας καταλαμβάνει 22bits της φυσικής διεύθυνσης, ένα έγκυρο bit και ένα lock bit.

[4] έγκυρο bit: ένα πεδίο στους πίνακες μιας ιεραρχίας μνήμης, που δείχνει ότι το σχετικό μπλοκ της ιεραρχίας περιέχει έγκυρα δεδομένα.

[5] LRU (Least Recently Used): μέθοδος αντικατάστασης στην οποία το μπλοκ που αντικαθίσταται είναι αυτό που έχει μείνει αχρησιμοποίητο για το μεγαλύτερο χρονικό διάστημα.

Υπάρχει ένας επιπλέον πίνακας ο οποίος αποθηκεύει τα dirty bits[6] και τα bits που σχετίζονται με την μέθοδο αντικατάστασης LRU. Ο πυρήνας microAptiv UP Core υποστηρίζει το μηχανισμό του data cache locking όπως συμβαίνει και στην περίπτωση της instruction cache. Έτσι, κρίσιμα τμήματα δεδομένων 'κλειδώνονται' στην μνήμη. Τα περιεχόμενα αυτά ενημερώνονται σε ενδεχόμενη ευστοχία εγγραφής αλλά δεν επιλέγονται προς αντικατάσταση σε περίπτωση αστοχίας της κρυφής μνήμης.

#### 4.2.6.4 Cache Memory Configuration

Στον πίνακα που ακολουθεί παρουσιάζονται τα χαρακτηριστικά καθεμιάς εκ των δύο κρυφών μνημών που υποστηρίζει ο πυρήνας microAptiv UP Core.

| **Παράμετρος** | **Instruction Cache** | **Data Cache** |
|---|---|---|
| Μέγεθος | 0-64Kbytes | 0-64Kbytes |
| Οργάνωση | Συσχετιστική συνόλου 1-4 δρόμων | Συσχετιστική συνόλου 1-4 δρόμων |
| Μέγεθος γραμμής | 16 bytes | 16 bytes |
| Read Unit | 32bits | 32bits |
| Πολιτικές Εγγραφών | - | write through, write allocate<br>write through, no write allocate<br>write back, write allocate |
| Cache Locking | ανά γραμμή | ανά γραμμή |

Πίνακας 4.2: Τα βασικά γνωρίσματα και οι ιδιότητες των μνημών που υποστηρίζονται από το microAptiv UP Core.

#### 4.2.6.5 Cache Protocols

Ο πυρήνας microAptiv UP Core υποστηρίζει τα παρακάτω πρωτόκολλα:

- *Uncached*: Σε μια περιοχή της μνήμης, οι διευθύνσεις που αναφέρονται ως uncached δεν διαβάζονται από την κρυφή μνήμη. Οι εγγραφές σε αυτές τις διευθύνσεις γράφονται κατευθείαν στην κύρια μνήμη, χωρίς να τροποποιούνται τα περιεχόμενα της cache.

[6] dirty bit: bit το οποίο σχετίζεται με ένα μπλοκ μνήμης και υποδεικνύει αν το συγκεκριμένο τμήμα έχει τροποποιηθεί.

- *Ταυτόχρονη εγγραφή με μη κατανομή σε εγγραφή*: Στις εντολές τύπου load καθώς και στις προσκομίσεις εντολών, αρχικά γίνεται αναζήτηση στην κρυφή μνήμη. Η κύρια μνήμη διαβάζεται μόνο αν τα επιθυμητά δεδομένα δεν βρίσκονται στην κρυφή μνήμη. Στην περίπτωση των store εντολών, αρχικά γίνεται αναζήτηση στην κρυφή μνήμη για να ελεγχθεί αν η ζητούμενη διεύθυνση βρίσκεται στην cache. Αν βρίσκεται τότε ενημερώνονται τα περιεχόμενα της cache και γράφεται ταυτόχρονα και η κύρια μνήμη (ταυτόχρονη εγγραφή). Αν προκύψει αστοχία ενημερώνεται το μπλοκ της κύριας μνήμης χωρίς να προσκομίζεται το μπλοκ αυτό στην κρυφή μνήμη (μη κατανομή σε μνήμη).

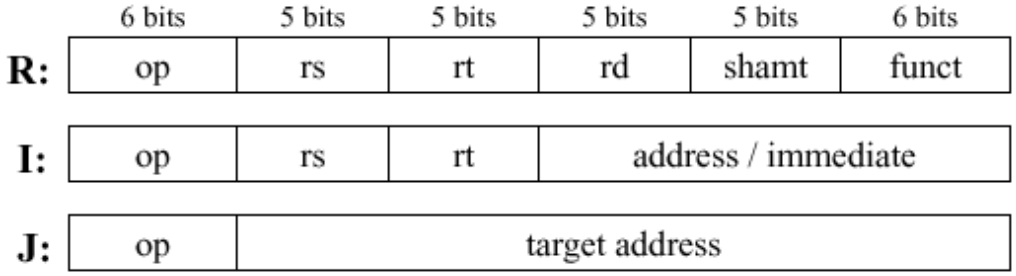
- *Ταυτόχρονη εγγραφή με κατανομή σε εγγραφή*: Στις εντολές τύπου load καθώς και στις προσκομίσεις εντολών, αρχικά γίνεται αναζήτηση στην κρυφή μνήμη. Η κύρια μνήμη διαβάζεται μόνο αν τα επιθυμητά δεδομένα δεν βρίσκονται στην κρυφή μνήμη. Στην περίπτωση των store εντολών, αρχικά γίνεται αναζήτηση στην κρυφή μνήμη για να ελεγχθεί αν η ζητούμενη διεύθυνση βρίσκεται στην cache. Αν βρίσκεται, ενημερώνονται τόσο η κρυφή όσο και η κύρια μνήμη. Στην περίπτωση αστοχίας διαβάζεται η κύρια μνήμη και προσκομίζεται το μπλοκ στην κρυφή μνήμη. Επιπλέον, ενημερώνεται και η κύρια μνήμη.

- *Ετερόχρονη εγγραφή με κατανομή σε εγγραφή*: Στις εντολές τύπου load καθώς και στις προσκομίσεις εντολών, αρχικά γίνεται αναζήτηση στην κρυφή μνήμη. Η κύρια μνήμη διαβάζεται μόνο αν τα επιθυμητά δεδομένα δεν βρίσκονται στην κρυφή μνήμη. Στην περίπτωση των store εντολών, αρχικά γίνεται αναζήτηση στην κρυφή μνήμη για να ελεγχθεί αν η ζητούμενη διεύθυνση βρίσκεται στην cache. Αν βρίσκεται, ενημερώνονται οι τιμές μόνο στο μπλοκ της κρυφής μνήμης και ορίζεται το dirty bit (που υποδυκνύει τα τροποποιημένα δεδομένα) για το μπλοκ αυτό (ετερόχρονη εγγραφή). Στην περίπτωση αστοχίας ενημερώνεται το μπλοκ της κύριας μνήμης και προσκομίζεται στην κρυφή μνήμη.

## 4.3 Σύνολο Εντολών MIPS

Κατά τη διαδικασία σχεδίασης ενός επεξεργαστή ο ορισμός και η δημιουργία του συνόλου εντολών του είναι μία από τις πρωταρχικές και πιο σημαντικές ενέργειες. Με τον όρο σύνολο εντολών περιγράφουμε όλες τις εντολές που ο εκάστοτε επεξεργαστής είναι ικανός να εκτελέσει. Στην περίπτωση του MIPS οι εντολές του είναι 32 bits και χωρίζονται σε τρείς μεγάλες κατηγορίες: εντολές τύπου R, εντολές τύπου I και εντολές τύπου J.

Η κατηγοριοποίηση αυτή προκύπτει, όχι από την λειτουργικότητα των εντολών, αλλά από το ελαφρώς διαφοροποιημένο μοτίβο κωδικοποίησης τους. Πιο συγκεκριμένα στις εντολές τύπου R περιλαμβάνονται λογικές και αριθμητικές πράξεις που γίνονται μεταξύ δύο καταχωρητών με το αποτέλεσμα της πράξης αυτής να αποθηκεύεται σε έναν καταχωρητή προορισμού. Εντολές τύπου I είναι εντολές που εκτελούν πράξεις αντίστοιχες με αυτές των R τύπων αλλά οι τελεστέοι είναι ένας καταχωρητής και μία αριθμητική σταθερά, ενώ το αποτέλεσμα αποθηκεύεται στη μνήμη ή σε έναν καταχωρητή προορισμού. Οι εντολές τύπου J είναι εντολές που πραγματοποιούν άλματα στον κώδικα αλλάζοντας έτσι την σειριακή εκτέλεσή του. Οι εντολές αυτές συχνά προκαλούν αλλαγές στον PC (Program counter) ο οποίος δείχνει την επόμενη, προς εκτέλεση, εντολή.




op: basic operation of the instruction (opcode)
rs: first source operand register
rt: second source operand register
rd: destination operand register
shamt: shift amount
funct: selects the specific variant of the opcode (function code)
address: offset for load/store instructions (+/-$2^{15}$)
immediate: constants for immediate instructions

Σχήμα 4.5: Οι κατηγοριοποιήσεις του συνόλου εντολών MIPS σε τρία διαφορετικά formats: *R-type, I-type, J-type* ανάλογα με το μοτίβο κωδικοποίησής τους.

⇒ ***Load/Store Εντολές***: Οι Load και Store εντολές είναι εντολές που ανήκουν στο *I-type* format και η υλοποίηση τους συνεπάγεται μετακίνηση δεδομένων ανάμεσα στη μνήμη και τους καταχωρητές γενικού σκοπού.

⇒ ***Computational Εντολές***: Οι εντολές αυτές μπορεί να ανήκουν είτε στο *R-type* format στο οποίο και οι δύο τελεστέοι είναι καταχωρητές, είτε

στο *I-type* format στο οποίο ο ένας τελεστέος είναι μια 16-bit αριθμητική σταθερά. Οι πράξεις στις οποίες αναφέρεται η κατηγορία αυτή είναι αριθμητικές, λογικές, shift καθώς και πράξεις πολλαπλασιασμού και διαίρεσης.

⇒ ***Branch/Jump Εντολές***: Εντολές αυτού του τύπου αλλάζουν τη ροή ενός προγράμματος. Όλες οι branch και jump εντολές συμβαίνουν με καθυστέρηση μιας εντολής.

⇒ ***Control Εντολές***: Είναι πάντοτε *R-type* εντολές που επιτρέπουν στο λογισμικό να ξεκινά διαδικασίες παγίδων.

⇒ ***Coprocessor Εντολές***: Οι εντολές αυτές εκτελούν πράξεις στους καταχωρητές του System Control Coprocessor για να χειρίζονται λειτουργίες διαχείρισης μνήμης και χειρισμού εξαιρέσεων.

⇒ ***Εντολές/Βελτιώσεις στην Αρχιτεκτονική MIPS***

- **CLZ/CLO (Count Leading Zeros/Ones)**: Οι εντολές αυτές υπολογίζουν τον αριθμό των πρόσθιων μηδενικών και άσσων αντίστοιχα σε μια λέξη. Σκανάρεται η 32-bit λέξη στον καταχωρητή προέλευσης rs από το MSB στο LSB και ο αριθμός των μηδενικών, άσσων που υπολογίζεται αντίστοιχα, γράφεται στον καταχωρητή προορισμού, rd. Αν και τα 32 bit στον rs έχουν οριστεί τότε η τιμή που γράφεται στον rd είναι 32 στην περίπτωση της εντολής CLO. Επιπλέον, αν και τα 32 bit στον rs δεν είναι ορισμένα (32 μηδενικά) τότε η τιμή που γράφεται στον rd είναι 32 στην περίπτωση της εντολής CLZ.
- **MADD/MSUB (Multiply and Add/Subtract Word)**: Οι εντολές αυτές πολλαπλασιάζουν δύο λέξεις και προσθέτουν/αφαιρούν το αποτέλεσμα με το ζευγάρι των HI/LO καταχωρητών. Η 32-bit τιμή στον rs πολλαπλασιάζεται με την 32-bit τιμή στον rt. Και οι δύο τελεστέοι αντιμετωπίζονται ως τιμές με πρόσημο και έχουν ως αποτέλεσμα μια 64-bit τιμή. Το γινόμενο προστίθεται με/αφαιρείται από, την τιμή των καταχωρητών HI, LO και η νέα τιμή που προκύπτει γράφεται πάλι στους καταχωρητές αυτούς.
- **MADDU/MSUBU (Multiply and Add/Subtract Unsigned Word)**: Η λειτουργικότητα των εντολών είναι όμοια με των παραπάνω με την διαφορά ότι οι τελεστέοι αντιμετωπίζονται ως τιμές χωρίς πρόσημο.
- **MUL (Multiply Word)**: Η εντολή πολλαπλασιάζει δύο λέξεις και γράφει το αποτέλεσμα στους καταχωρητές γενικού σκοπού. Η 32-bit τιμή της λέξης στον rs πολλαπλασιάζεται με την 32-bit τιμή

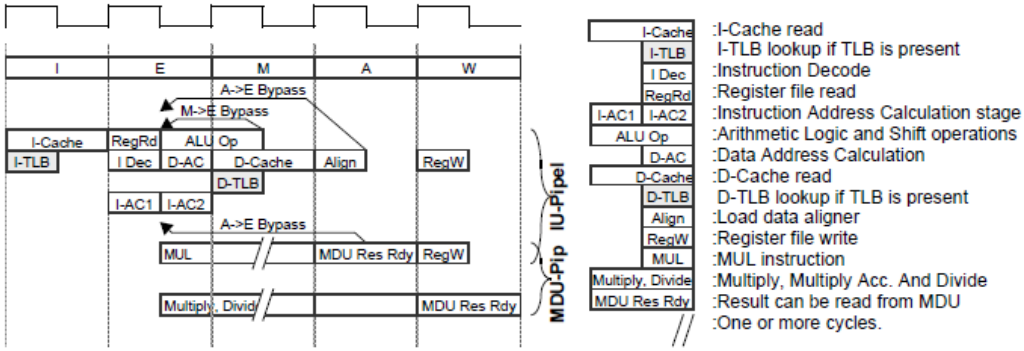
στον rt για την παραγωγή ενός 64-bit αποτελέσματος. Και οι δύο τελεστέοι αντιμετωπίζονται ως τιμές με πρόσημο. Τα λιγότερο σημαντικά 32 bits γράφονται στον rd καταχωρητή.

– **SSNOP (Superscalar Inhibit NOP)**: Η εντολή αντιμετωπίζεται ως μια κανονική NOP εντολή.

## 4.4 Στάδια Διοχέτευσης

Σε αυτήν την ενότητα παρουσιάζουμε τα στάδια της διοχέτευσης του επεξεργαστή. Ο μηχανισμός της διοχέτευσης επιτρέπει στον επεξεργαστή να επιτυγχάνει μεγαλύτερη διεκπεραιωτική ικανότητα ενώ παράλληλα ελαχιστοποιεί την πολυπλοκότητα της διάταξης, μειώνοντας το κόστος και την κατανάλωση ισχύος. Πιο συγκεκριμένα, ο *microAptiv UP Core* αποτελείται από 5 επίπεδα διοχέτευσης:

- Instruction (I Stage)
- Execution (E Stage)
- Memory (M Stage)
- Align (A Stage)
- Writeback (W Stage)




Σχήμα 4.6: Οι πράξεις που εκτελούνται σε κάθε στάδιο διοχέτευσης.

Παρακάτω ακολουθεί μια σύντομη περιγραφή της λειτουργικότητας του κάθε επιπέδου.

- Κατά την διάρκεια του *Instruction Fetch Stage* προσκομίζεται μια εντολή από την I-Cache. Το I-TLB εκτελεί μια μετάφραση διεύθυνσης από εικονική σε φυσική.
- Κατά την διάρκεια του *Execution Stage* προσκομίζονται οι καταχωρητές-τελεστέοι από το αρχείο καταχωρητών. Η ALU ξεκινά την αριθμητική ή λογική πράξη για εντολές τύπου R-type και I-type. Επίσης, η ALU υπολογίζει την εικονική διεύθυνση μνήμης στην περίπτωση των εντολών store/load και προσδιορίζει πότε η συνθήκη μιας branch εντολής είναι αληθής, υπολογίζοντας στη συνέχεια την διεύθυνση που προκύπτει από την διακλάδωση. Τέλος, όλες οι πράξεις πολ/σμου-διαίρεσης ξεκινούν σε αυτό το στάδιο.
- Κατά την διάρκεια του *Memory Fetch Stage* η αριθμητική πράξη της ALU ολοκληρώνεται. Πραγματοποιείται η πρόσβαση και αναζήτηση στην D-Cache και έπειτα ο προσδιορισμός της αστοχίας/ευστοχίας της μνήμης. Τέλος, στο στάδιο αυτό συντελείται η πράξη του πολ/σμού και της διαίρεσης στην μονάδα MDU.
- Κατά την διάρκεια του *Align Stage* κάθε δεδομένο που προέρχεται από εντολές τύπου load 'ευθυγραμμίζεται'(align) εντός των ορίων μιας λέξης. Η πράξη του πολ/σμου-διαίρεσης ενημερώνουν τους ειδικούς καταχωρητές HI/LO. Στη συνέχεια, το αποτέλεσμα των πράξεων αυτών γίνεται διαθέσιμο για εγγραφή στο αρχείο καταχωρητών.
- Κατά την διάρκεια του *Writeback Stage* το αποτέλεσμα, στην περίπτωση των R-type/I-type εντολών αλλά και των εντολών τύπου load, γράφεται στο αρχείο καταχωρητών.

# Κεφάλαιο 5

# Υλοποίηση του μηχανισμού DIFT στον επεξεργαστή MIPS

Σε αυτό το κεφάλαιο γίνεται παρουσίαση του τρόπου εφαρμογής του μηχανισμού DIFT στον επεξεργαστή, αναλύοντας μεταξύ άλλων την πολιτική ασφαλείας που καθορίστηκε, τις απαιτούμενες τροποποιήσεις που συντελέστηκαν στον κώδικα του επεξεργαστή και την προσθήκη επιπρόσθετων εντολών για την υποστήριξη της λειτουργικότητας του μηχανισμού ελλείψει της αντίστοιχης, σε επίπεδο λογισμικού.

## 5.1 Πολιτικές DIFT

Η ανάπτυξη ενός ολοκληρωμένου DIFT συστήματος απαιτεί την λεπτομερή εξέταση και ανάλυση μιας σειράς σημαντικών σχεδιαστικών παραμέτρων όπως το κόστος, η επιβάρυνση στην απόδοση και η πολιτική που κάθε φορά ακολουθείται. Η τελευταία παράμετρος θα παρουσιασθεί σε αυτήν την ενότητα, καθώς η σχεδίαση μιας DIFT policy για κάθε επίθεση, αποτελεί έναν ιδιαίτερα σημαντικό παράγοντα κατά την ανάπτυξη μιας DIFT πλατφόρμας. Αφού επιλεγεί η κατάλληλη πολιτική ασφαλείας, στη συνέχεια σειρά έχει η εφαρμογή της αρχιτεκτονικής DIFT με τρόπο τέτοιο που να υποστηρίζει επιτυχώς την πολιτική που προηγουμένως επιλέχθηκε. Στο τελικό στάδιο, μέσω της DIFT policy, διατυπώνεται και ο κατάλληλος έλεγχος που εκτελείται, για να προστατέψει τις εφαρμογές από επιθέσεις, που σκοπό έχουν την έκθεση των αδυναμιών τους. Για την υλοποίηση της πλατφόρμας στη δικιά μας περίπτωση, ακολουθήσαμε τα παραπάνω βήματα προκειμένου να σχεδιάσουμε ένα πλήρες σύστημα με δυνατότητα περαιτέρω μελέτης στο μέλλον.

### 5.1.1 Η Πολιτική που ακολουθείται

Για την εφαρμογή του μηχανισμού DIFT στον επεξεργαστή MIPS επιλέξαμε το *Pointer Injection Policy* καθώς μετά από μελέτη σχετικών εργασιών και αναφορών [35, 36], καταλήξαμε ότι αυτή είναι η καλύτερη επιλογή. Για την υλοποίηση αυτής, χρησιμοποιήθηκαν δύο *tag bits:*

- το *Taint bit* ή *T()* που ορίζεται ως αληθές όταν τα δεδομένα που εκτελούνται προέρχονται από μη έμπιστες πηγές και παράγεται σε όλες τις αριθμητικές, λογικές και data movement εντολές. Η ύπαρξη ενός τουλάχιστον 'μολυσμένου' καταχωρητή προέλευσης συνεπάγεται την παραγωγή ενός 'μολυσμένου' καταχωρητή προορισμού ή θέσης μνήμης.
- το *Pointer bit* ή *P()* αρχικοποιείται και ορίζεται ως αληθές για όλους τους 'νόμιμους' δείκτες της εφαρμογής και παράγεται κατά την τέλεση έγκυρων πράξεων ανάμεσα σε δείκτες. Η αρχικοποίηση του P bit, η οποία συντελείται από το λειτουργικό σύστημα, αποτελεί ένα σημαντικό παράγοντα για την ορθή εφαρμογή της *pointer injection* πολιτικής και για το λόγο αυτό αναλύεται ξεχωριστά στη συνέχεια του κεφαλαίου.

Οι κανόνες που διέπουν την παραγωγή του Taint bit για κάθε είδος εντολής περιγράφονται στον Πίνακα 5.1. Όπως αναφέρθηκε προηγουμένως, οποιαδήποτε πράξη περιέχει τουλάχιστον έναν καταχωρητή προέλευσης 'μολυσμένο' (Taint bit αληθές) θα έχει ως αποτέλεσμα ο καταχωρητής προορισμού ή η θέση μνήμης να περιέχουν επίσης 'μολυσμένα' δεδομένα. Για παράδειγμα, στην περίπτωση μιας sub εντολής που αφαιρεί τα περιεχόμενα των καταχωρητών, αν ένας από τους δύο καταχωρητές περιέχει δεδομένα των οποίων το Taint bit έχει οριστεί ως αληθές, τότε και το Taint bit του καταχωρητή προορισμού ορίζεται ως αληθές. Ανάλογα, σε μια data movement εντολή, όπως load ή store το Taint bit είτε του καταχωρητή προορισμού είτε της θέσης μνήμης είναι αληθές όταν η θέση μνήμης ή ο καταχωρητής προέλευσης ειναι 'μολυσμένοι', αντίστοιχα. Τέλος, στην περίπτωση των πολλαπλασιασμών/διαιρέσεων (mult/div εντολές), η παραγωγή του taint bit γίνεται με τον ίδιο τρόπο καθώς πέρα από τους καταχωρητές γενικού σκοπού, έχουν επεκταθεί κατάλληλα και οι ειδικοί καταχωρητές HI/LO. Σε όλες τις υπόλοιπες πράξεις που εκτελούνται στην ALU (πρόσθεση, αφαίρεση, λογικές πράξεις), η διαδικασία παραγωγής του taint bit είναι παρόμοια με τις αντίστοιχες παραπάνω. Οι περιπτώσεις των εντολών and και lui παρουσιάζονται ξεχωριστά λόγω της ιδιαιτερότητάς τους ως προς την παραγωγή του Pointer bit, κάτι που θα αναλυθεί παρακάτω.

Συμπερασματικά, η παραγωγή του Taint bit γίνεται με την εκτέλεση της λογικής πράξης *OR* στην περίπτωση των αριθμητικών, λογικών και data movement εντολών.

| **Εντολή** | **Παράδειγμα** | **Σημασία** | **Taint Bit Propagation** |
|---|---|---|---|
| Jump | jalr r1 + imm, r2 | r2 ← PC; PC ← r1 + imm | T(r2) ← 0 |
| Load | lw r2, offset(r1) | r2 ← MEM[r1+offset] | T(r2) ← T([r1+offset]) |
| Store | sw r1, offset(r2) | MEM[r2+offset] ← r1 | T([r2+offset]) ← T(r1) |
| Add/Sub/Or | or r3, r1, r2 | r3 ← r1 ∨ r2 | T(r3) ← T(r1) ∨ T(r2) |
| All other ALU | mult r1, r2 | (LO,HI) ← r1 × r2 | T(LO) ← T(r1) ∨ T(r2)<br>T(HI) ← T(r1) ∨ T(r2) |
| And | and r3, r1, r2 | r3 ← r1 ∧ r2 | T(r3) ← T(r1) ∨ T(r2) |
| Lui | lui r1, imm | r1 ← imm | T(r1) ← 0 |

Πίνακας 5.1: Οι κανόνες που διέπουν την διαδικασία παραγωγής του *Taint bit* για τον εκάστοτε τύπο εντολών. Ο συμβολισμός T() αναφέρεται στο *Taint bit* των καταχωρητών, εντολών ή θέσεων μνήμης.

Οι κανόνες που διέπουν την παραγωγή του Pointer bit για κάθε είδος εντολής περιγράφονται παρακάτω στον Πίνακα 5.2. Πιο συγκεκριμένα, το Pointer bit ορίζεται μόνο στις περιπτώσεις έγκυρων πράξεων μεταξύ pointers όπως η αριθμητική δεικτών. Για παράδειγμα, το P bit πρέπει να παραχθεί στις εντολές μεταφοράς δεδομένων (store/load) καθώς κάθε φορά που αντιγράφεται ένας pointer αντιγράφεται και το P bit.

Επιπλέον, το P bit πρέπει να παραχθεί στην περίπτωση της εντολής *lui*(*load upper immediate*) καθώς είναι η μοναδική εντολή του συνόλου εντολών mips [34] που μπορεί να χρησιμοποιηθεί για την αρχικοποίηση ενός pointer σε μια έγκυρη διεύθυνση. Πιο συγκεκριμένα, η εντολή lui, η οποία συντάσσεται ως *lui $rt, immed* θέτει τα 16-περισσότερο σημαντικά bits του καταχωρητή $rt με τον 16-bit immed και τα εναπομείναντα 16 με μηδενικά. Αν η επόμενη εντολή είναι μια *or/addiu*[1] εντολή με καταχωρητή προέλευσης τον $rt της lui τότε είναι δυνατή η εκχώρηση μιας 32-bit τιμής (pointer) σε έναν καταχωρητή. Για τον παραπάνω λόγο παράγεται το P bit τόσο στην περίπτωση της εντολής lui όπως αναφέρθηκε παραπάνω, όσο και στην περίπτωση της αντίστοιχης or/addiu με τον τρόπο που φαίνεται στον πίνακα 5.2.

[1]addiu: το πρόβλημα έγκειται στην περίπτωση μιας αρνητικής σταθεράς. Επιλέγεται, λοιπόν, η addiu αντί της addi καθώς προσθέτει, κάνοντας zero-extend την 16-bit σταθερά αντί του sign-extend στην περίπτωση της addi.

Στην περίπτωση της πρόσθεσης/αφαίρεσης η παραγωγή του P bit γίνεται με την τέλεση της λογικής πράξης or ανάμεσα στα P bit των καταχωρητών προέλευσης. Οι υπόλοιπες πράξεις της ALU (mult/div ops κλπ.) δεν πρέπει να εκτελούνται σε pointers και επομένως οι πράξεις αυτές έχουν ως αποτέλεσμα να μην ορίζεται το P bit ως αληθές.

| **Εντολή** | **Παράδειγμα** | **Σημασία** | **Pointer Bit Propagation** |
|---|---|---|---|
| Jump | jalr r1 + imm, r2 | r2 $\leftarrow$ PC; PC $\leftarrow$ r1 + imm | P(r2) $\leftarrow$ 1 |
| Load | lw r2, offset(r1) | r2 $\leftarrow$ MEM[r1+offset] | P(r2) $\leftarrow$ P([r1+offset]) |
| Store | sw r1, offset(r2) | MEM[r2+offset] $\leftarrow$ r1 | P([r2+offset]) $\leftarrow$ P(r1) |
| Add/Sub/Or | or r3, r1, r2 | r3 $\leftarrow$ r1 $\vee$ r2 | P(r3) $\leftarrow$ P(r1) $\vee$ P(r2) |
| All other ALU | mult r1, r2 | (LO,HI) $\leftarrow$ r1 $\times$ r2 | P(LO) $\leftarrow$ 0<br>P(HI) $\leftarrow$ 0 |
| And | and r3, r1, r2 | r3 $\leftarrow$ r1 $\wedge$ r2 | P(r3) $\leftarrow$ P(r1) $\oplus$ P(r2) |
| Lui | lui r1, imm | r1 $\leftarrow$ imm | P(r1) $\leftarrow$ P[instr] |

Πίνακας 5.2: Οι κανόνες που διέπουν την διαδικασία παραγωγής του *Pointer bit* για τον εκάστοτε τύπο εντολών. Ο συμβολισμός P() αναφέρεται στο *Pointer bit* των καταχωρητών, εντολών ή θέσεων μνήμης.

Τελευταίος και σημαντικότερος είναι ο έλεγχος ασφαλείας που συντελείται και καταδεικνύει αν κάτι αναξιόπιστο χρησιμοποιείται επισφαλώς. Οι κανόνες εκείνοι που ορίζουν την δημιουργία μιας εξαίρεσης ασφαλείας φαίνονται στον πίνακα 5.3. Πιο αναλυτικά, εξαίρεση δημιουργείται αν η διεύθυνση που χρησιμοποιείται σε μια load, store ή jump εντολή είναι αναξιόπιστη και μη έγκυρος pointer ταυτόχρονα.

| **Εντολή** | **Παράδειγμα** | **Έλεγχος Ασφαλείας** |
|---|---|---|
| Jump | jalr r1 + imm, r2 | T(r1) & $\neg$ P(r1) |
| Load | lw r2, offset(r1) | T(r1) & $\neg$ P(r1) |
| Store | sw r2, offset(r1) | T(r1) & $\neg$ P(r1) |

Πίνακας 5.3: Οι έλεγχοι ασφαλείας που εκτελούνται στην πολιτική του Pointer Injection.

## 5.2 Αρχικοποίηση των tags

Μια ιδιαίτερα σημαντική παράμετρος στην πολιτική του pointer injection είναι η ακριβής αναγνώριση των έγκυρων pointers στον κώδικα της εφαρμογής προκειμένου να γίνει η αρχικοποίηση του P bit για αυτές τις θέσεις μνήμης. Πιο συγκεκριμένα, η αρχικοποίηση του P bit πρέπει να συμβαίνει μόνο στην περίπτωση των *root pointer assignments*, όπου ένας pointer έχει οριστεί σε μια έγκυρη διεύθυνση μνήμης η οποία δεν προέρχεται από άλλο pointer. Η παραπάνω διαδικασία διακρίνεται σε δύο περιπτώσεις: a) την περίπτωση που ένας pointer αρχικοποιείται με μια έγκυρη διεύθυνση σε *στατική κατανομή μνήμης* και b) την περίπτωση που ένας pointer αρχικοποιείται με μια έγκυρη διεύθυνση σε *δυναμική κατανομή μνήμης*. Τέλος, στην περίπτωση αρχικοποίησης δείκτη σε υπάρχουσα μεταβλητή (δλδ ανάθεση τιμής που προέρχεται από υπάρχον δείκτη) το Pointer Bit Propagation είναι εκείνο που θα ορίσει κατάλληλα, βάσει των κανόνων που ορίστηκαν, την τιμή του P bit.

### 5.2.1 Δείκτες σε Δυναμικά Κατανεμημένη Μνήμη

Για την κατανομή μνήμης κατά τη διάρκεια εκτέλεσης του προγράμματος, χρησιμοποιούνται συγκεκριμένες κλήσεις συστήματος. Πιο αναλυτικά, σε ένα σύστημα με λειτουργικό Linux υπάρχουν τα παρακάτω πέντε system calls για δυναμική κατανομή μνήμης: *mmap, mmap2, brk, mremap, shmat*. Η τιμή επιστροφής αυτών των κλήσεων (στην περίπτωση επιτυχούς κλήσης) είναι κάθε φορά ένας δείκτης προς την περιοχή της μνήμης που κατανέμεται. Επομένως, με κατάλληλη τροποποίηση του πυρήνα του Linux είναι δυνατός ο ορισμός του P bit των τιμών επιστροφής των παραπάνω system calls για κάθε επιτυχή κλήση συστήματος που κατανέμει δυναμικά μνήμη. Τέλος, θέτουμε το P bit για τον stack pointer register στην αρχή του προγράμματος.

### 5.2.2 Δείκτες σε Στατικά Κατανεμημένη Μνήμη

Στην περίπτωση της στατικής κατανομής μνήμης, όλες οι αναθέσεις τιμών - διευθύνσεων μνήμης σε δείκτες περιέχονται στα τμήματα που σχετίζονται με τα δεδομένα (data segment - .data) και σε εκείνα που αφορούν τον κώδικα του προγράμματος (code segment - .code) ενός object file. Για την αρχικοποίηση του P bit πρέπει να γίνει έλεγχος των παραπάνω τμημάτων του εκτελέσιμου αρχείου κατά την έναρξη του προγράμματος. Πιο συγκεκριμένα, το τμήμα .data περιέχει δείκτες που αρχικοποιούνται σε στατικά

κατανεμημένες διευθύνσεις μνήμης. Το code segment περιέχει εντολές που χρησιμοποιούνται για να αρχικοποιήσουν δείκτες σε στατικά κατανεμημένη μνήμη κατά τη διάρκεια της εκτέλεσης του προγράμματος. Επομένως, είναι απαραίτητος ο έλεγχος κάθε data segment για την ύπαρξη 32-bit τιμών που είναι εντός του εύρους εικονικών διευθύνσεων του τρέχοντος εκτελέσιμου και ο ορισμός του P bit για κάθε τέτοια σταθερά. Στην περίπτωση του code segment πρέπει να ελεχθεί ο κώδικας για την ύπαρξη του ζεύγους εντολών lui/or ή lui/addiu. Όπως αναφέρθηκε και παραπάνω η εντολή lui αρχικοποιεί τα 16-περισσότερο σημαντικά bits ενός καταχωρητή θέτοντας τα υπόλοιπα 16 με μηδενικά. Παράλληλα, η χρήση της or/addiu (αν απαιτείται) αφορά την αρχικοποίηση των 16-λιγότερο σημαντικών bits, με το συνδυασμό τους να έχει ως αποτέλεσμα τη δημιουργία μιας 32-bit διεύθυνσης. Συμπερασματικά λοιπόν, στην περίπτωση του ελέγχου του .code τμήματος, αν ο immediate όρος της εντολής lui προσδιορίζει μια σταθερά εντός του εύρους εικονικών διευθύνσεων του τρέχοντος εκτελέσιμου, ορίζεται το P bit αυτής της εντολής.

## 5.3 Προσθήκη extra εντολών

Σε αυτήν την ενότητα αρχικά παρουσιάζουμε το σύνολο των εντολών που υποστηρίζονται από την αρχιτεκτονική MIPS, παραθέτοντας σε έναν συγκεντρωτικό πίνακα όλα τα χρησιμοποιούμενα opcodes του συνόλου εντολών του επεξεργαστή και στη συνέχεια τις επιπρόσθετες εντολές που υλοποιήσαμε ώστε να θέτουμε τις τιμές των tags κάθε φορά που θέλουμε να υποδείξουμε ότι κάτι προήλθε από μια αξιόπιστη ή μη πηγή.

| instruction opcode | | | op[28:26] | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| | | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| | 000 | 0 | SPECIAL | REGIMM | J | JAL | BEQ | BNE | BLEZ | BGTZ |
| | 001 | 1 | ADDI | ADDIU | SLTI | SLTIU | ANDI | ORI | XORI | LUI |
| | 010 | 2 | COP0 | COP1 | COP2 | COP1X | BEQL | BNEL | BLEZL | BGTZL |
| op[31:29] | 011 | 3 | DADDI | DADDIU | LDL | LDR | ***WR_MEM2T*** | ***RD_MEMT2R*** | ***WR_R2T*** | ***RD_T2R*** |
| | 100 | 4 | LB | LH | LWL | LW | LBU | LHU | LWR | LWU |
| | 101 | 5 | SB | SH | SWL | SW | SDL | SDR | SWR | - |
| | 110 | 6 | LL | LWC1 | LWC2 | PREF | LLD | LDC1 | LDC2 | LD |
| | 111 | 7 | SC | SWC1 | SWC2 | - | SCD | SDC1 | SDC2 | SD |

Πίνακας 5.4: Όλα τα χρησιμοποιούμενα opcodes των εντολών και με έντονα γράμματα εκείνων που υλοποιήσαμε.

Στον παραπάνω πίνακα φαίνονται τα opcodes όλων των εντολών και με έντονα γράμματα εκείνων που υλοποιήσαμε προκειμένου να καλύψουμε την έλλειψη της λειτουργικότητας του μηχανισμού DIFT στο λογισμικό.

Με άλλα λόγια, λόγω της υλοποίησης της τεχνικής DIFT μόνο στο επίπεδο του υλικού, προσθέτουμε extra εντολές στο σύνολο εντολών του επεξεργαστή προκειμένου να χειριζόμαστε τα tag των δεδομένων και να είμαστε σε θέση να προσομοιώσουμε το σύστημα χρησιμοποιώντας binaries με επιπρόσθετα DIFT δεδομένα. Αυτή η μέθοδος, μας επιτρέπει να προσομοιώνουμε πολλά διαφορετικά binary αρχεία χωρίς να απαιτείται κάθε φορά η τροποποίηση της σχεδίασης στο επίπεδο του υλικού. Για το σκοπό αυτό, τροποποιήσαμε το decode stage του επεξεργαστή όπου γίνονται και οι αποκωδικοποιήσεις των εντολών, παρεμβαίνοντας στο αρχείο *mpc_dec.v*. Ο παρακάτω πίνακας παρουσιάζει τις νέες εντολές που προστέθηκαν στο MIPS ISA.

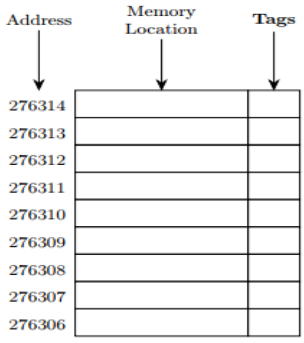
| Εντολή | opcode | Παράδειγμα | Σημασία |
|---|---|---|---|
| Write Register Tag | 011110 | wr_r2t $r1, $r2 | Tag(r1) ← r2 |
| Read Register Tag | 011111 | rd_t2r $r1, $r2 | r1 ← Tag(r2) |
| Write Memory Tag | 011100 | wr_mem2t [$r1], [$r2] | Tag(MEM[r1]) ← MEM[r2] |
| Read Memory Tag | 011101 | rd_memt2r [$r1], [$r2] | r1 ← Tag(MEM[r2]) |

Πίνακας 5.5: Οι επιπρόσθετες εντολές που υλοποιήθηκαν ελλείψει της λειτουργικότητας DIFT στο επίπεδο του λογισμικού.

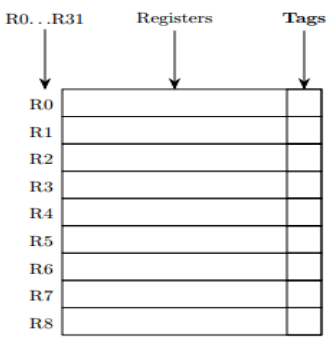
Σκοπός, λοιπόν, είναι η διαχείριση των tags τόσο στην *Register File* όσο και στην *Cache*, επομένως υλοποιήσαμε 4 επιπρόσθετες εντολές εκ των οποίων οι δύο αφορούν το αρχείο καταχωρητών (reading/writing των tags από και προς το αρχείο καταχωρητών - Read Register Tag/Write Register Tag) και οι άλλες δύο την Cache εκτελώντας παρόμοιες λειτουργίες με τις προηγούμενες αλλά από και προς την κρυφή μνήμη (Read Memory Tag/Write Memory Tag). Ως τελευταίο στάδιο, για την επιτυχή αναγνώριση των εντολών ήταν αναγκαία και η αλλαγή του pipeline του επεξεργαστή ώστε αυτές να ενταχθούν ως 'έγκυρες' στο σύνολο εντολών του.

## 5.4 Υλοποίηση DIFT στον MIPS

Σε αυτήν την ενότητα αναφέρουμε όλες τις επεκτάσεις που συντελέστηκαν στον επεξεργαστή με σκοπό την επιτυχή εφαρμογή της τεχνικής DIFT, παρουσιάζοντας παράλληλα και τα τροποποιημένα *datapaths* και *block diagrams* του επεξεργαστή. Βασικές τροποποιήσεις που πρέπει να γίνουν είναι a) του αρχείου καταχωρητών, στο οποίο κάθε καταχωρητής επεκτείνεται κατά δύο bits για την υποστήριξη της extra πληροφορίας και b) της μνήμης/Cache, όπως φαίνεται στο σχήμα που ακολουθεί.




α′) Η κατάλληλα τροποποιημένη μνήμη για την υποστήριξη της λειτουργικότητας DIFT στον επεξεργαστή.

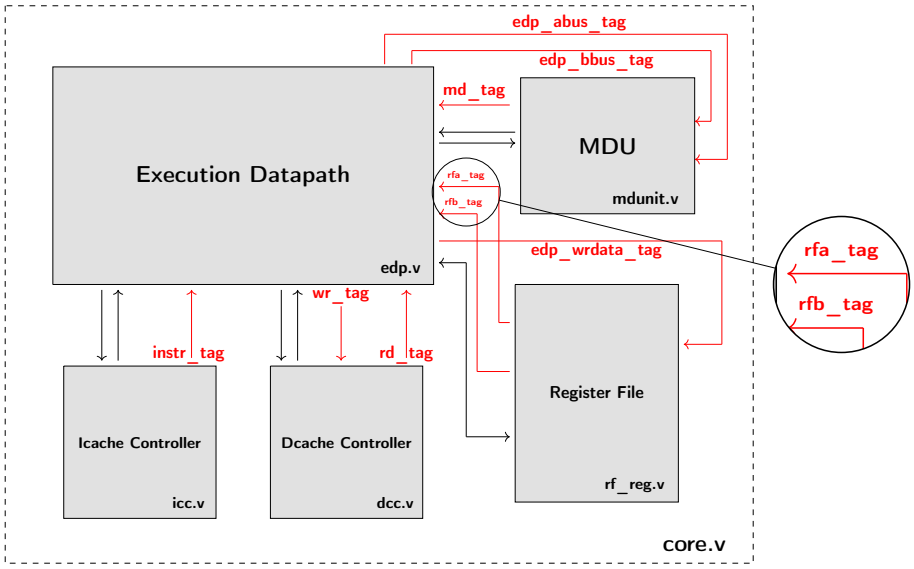



β′) Η επέκταση των καταχωρητών για την υποστήριξη της extra πληροφορίας που φέρουν τα tags.

Σχήμα 5.1: Οι τροποποιήσεις που πρέπει να γίνουν στον επεξεργαστή MIPS προκειμένου να υποστηρίζει τη λειτουργικότητα του μηχανισμού DIFT.

Οι τροποποιήσεις, λοιπόν, αφορούν τις δομικές μονάδες του πυρήνα του επεξεργαστή όπως το αρχείο καταχωρήτων που προαναφέρθηκε, τη μονάδα πολλαπλασιασμού/διαίρεσης (MDU), τις caches με τους αντίστοιχους cache controllers καθώς επίσης και το execution datapath στο οποίο πραγματοποείται το μεγαλύτερο μέρος της λειτουργίας του επεξεργαστή. Στο *σχήμα 5.2* που ακολουθεί παρουσιάζονται οι βασικές μονάδες που επεκτάθηκαν, για την υποστήριξη του μηχανισμού, όπως αυτές είναι οργανωμένες στο αρχείο *m14k_core.v*. Με κόκκινο χρώμα έχουν επισημανθεί τα tag bits από και προς το κάθε τμήμα. Τέλος, σημειώνεται πως οι αλλαγές που έγιναν αφορούν όλα τα αρχεία, που είναι κατώτερα ιεραρχικά, για καθεμιά από τις μονάδες που παρουσιάζονται στο σχήμα 5.2 αλλά είναι αδύνατο να απεικονιστούν σε ένα διάγραμμα. Επιγραμματικά, τα top levels των βασικών τμημάτων που τροποποιήθηκαν είναι:

1. *m14k_rf_reg.v* → *Αρχείο Καταχωρητών*
2. *m14k_edp.v* → *Execution Unit*
3. *m14k_icc.v* → *I-cache controller*
4. *m14k_dcc.v* → *D-cache controller*
5. *m14k_mdl.v* → *Μονάδα Πολ/σμου-Διαίρεσης*




Σχήμα 5.2: Το block diagram των βασικών δομικών μονάδων του πυρήνα του επεξεργαστή και τα επιπρόσθετα inputs/outputs (κόκκινο χρώμα) που σχετίζονται με τα tags από και προς τις μονάδες αυτές.

Όσον αφορά τις τροποποιήσεις του αρχείου καταχωρητών, εκεί πέρα από την επέκταση των καταχωρητών, προστέθηκαν τα σήματα *rfa_tag*, *rfb_tag* που αντιστοιχούν στα tag bits των καταχωρητών ανάγνωσης rfa και rfb αντίστοιχα καθώς επίσης και το σήμα *edp_wrdata_tag* το οποίο αφορά το tag του καταχωρητή προορισμού και σχετίζεται με το αν τα δεδομένα που γράφονται στην Register File είναι αναξιόπιστα. Συμπερασματικά, προστέθηκαν ως έξοδοι (προς το execution datapath - αρχείο *m14k_edp.v*) τα tags των καταχωρητών που κάθε φορά διαβάζονται και ως είσοδος (από το *m14k_edp.v*) το tag του καταχωρητή προορισμού.

Στη συνέχεια παραθέτουμε δύο πίνακες που παρουσιάζουν τις εισόδους/εξόδους του αρχείου καταχωρητών και με έντονα γράμματα τα επιπλέον σήματα που προστέθηκαν σε αυτές για την αναπαράσταση της extra πληροφορίας.

**Inputs**

| Όνομα | Πλάτος | Περιγραφή |
|---|---|---|
| mpc_dest_w | 9 bits | Destination reg |
| mpc_rega_i | 9 bits | Src A reg |
| mpc_regb_i | 9 bits | Src B reg |
| edp_wrdata_w | 32 bits | Write data |
| ***wrdata_w_tag*** | ***2 bits*** | ***Write tag*** |
| mpc_rfwrite_w | 1 bit | RF write enable |
| mpc_rega_cond_i | 1 bit | Src A reg cond |
| mpc_regb_cond_i | 1 bit | Src B reg cond |

**Outputs**

| Όνομα | Πλάτος | Περιγραφή |
|---|---|---|
| rf_adt_e | 32 bits | Read port A |
| ***rfa_tag*** | ***2 bits*** | ***Read A tag*** |
| rf_bdt_e | 32 bits | Read port B |
| ***rfb_tag*** | ***2 bits*** | ***Read B tag*** |

Πίνακας 5.6: Το τροποποιημένο αρχείο καταχωρητών (*m14k_rf_reg.v*) με τις επιπρόσθετες εισόδους/εξόδους, που σχετίζονται με τα tag bits, να επισημαίνονται με έντονα γράμματα.

Όσον αφορά τη μονάδα πολ/σμου-διαίρεσης, οι αλλαγές που έγιναν πέρα από την επέκταση των ειδικών καταχωρητών HI/LO, αφορούν την προσθήκη σημάτων που σχετίζονται με το tag, τόσο των σημάτων που εισάγονται στην MDU για πολ/σμο-διαίρεση (*edp_abus_tag, edp_bbus_tag*) όσο και του αποτελέσματος που εξάγεται από την μονάδα αυτή και προωθείται στη συνέχεια στο Execution Datapath (*mdu_tag*).

Στον πίνακα που ακολουθεί παρουσιάζονται οι βασικές είσοδοι/έξοδοι που σχετίζονται με τη λειτουργία της MDU και με έντονα γράμματα αυτές που προστέθηκαν και σχετίζονται με το μηχανισμό του DIFT.

**Inputs**

| Όνομα | Πλάτος | Περιγραφή |
|---|---|---|
| edp_abus_e | 32 bits | Src A bus |
| edp_bbus_e | 32 bits | Src B bus |
| ***edp_abus_tag*** | ***2 bits*** | ***Src A tag*** |
| ***edp_bbus_tag*** | ***2 bits*** | ***Src B tag*** |
| mpc_srcvld_e | 1 bit | rs/rt are valid |
| mpc_irval_e | 1 bit | Instr is valid |

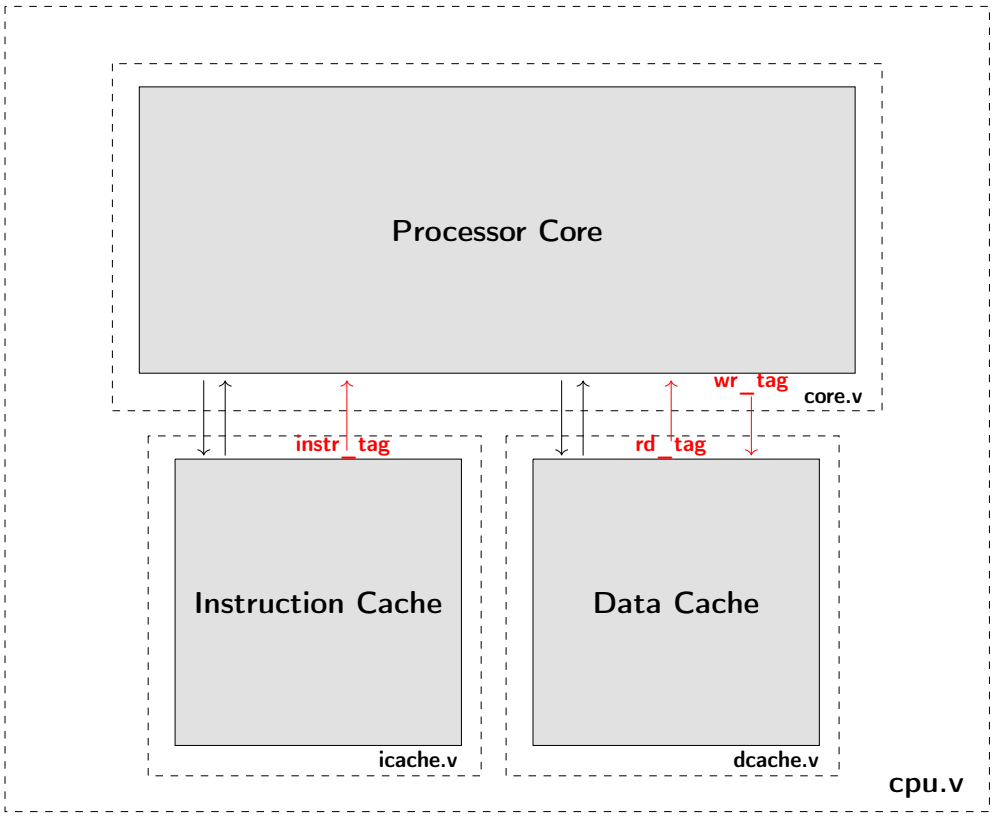
**Outputs**

| Όνομα | Πλάτος | Περιγραφή |
|---|---|---|
| mdu_res_w | 32 bits | Reg rd to RF |
| ***mdu_res_tag*** | ***2 bits*** | ***Reg's tag*** |
| mdu_busy | 1 bit | MDU stall |
| mdu_stall | 1 bit | cmds stall |

Πίνακας 5.7: Η τροποποιημένη μονάδα πολ/σμου-διαίρεσης με τις επιπρόσθετες εισόδους/εξόδους, που σχετίζονται με τα tag bits, να επισημαίνονται με έντονα γράμματα.

Κατόπιν, παραθέτουμε το block diagram του ανώτερου, ιεραρχικά, επιπέδου (m14k_cpu.v) από αυτό που περιγράφτηκε παραπάνω, στο οποίο φαίνεται η διασύνδεση ανάμεσα στις επιμέρους μονάδες του επιπέδου αυτού. Με κόκκινο χρώμα επισημαίνονται και πάλι τα επιπρόσθετα inputs/outputs που σχετίζονται με τα tags και την extra πληροφορία που αυτά φέρουν. Επιγραμματικά, στο επίπεδο αυτό παρουσιάζεται η διασύνδεση ανάμεσα στις caches και τον πυρήνα του επεξεργαστή, με τα αρχεία που τροποποιούνται να είναι τα παρακάτω:

1. *m14k_ic.v* → *Instruction Cache*
2. *m14k_dc.v* → *Data Cache*
3. *m14k_core.v* → *Processor Core*




Σχήμα 5.3: Το block diagram του πυρήνα του επεξεργαστή και η σύνδεσή του με τις caches. Με κόκκινο χρώμα τα επιπρόσθετα inputs/outputs που σχετίζονται με τα tags από και προς τις μονάδες αυτές.

Στη συνέχεια, παραθέτουμε έναν πίνακα με τις εισόδους/εξόδους της Data Cache έχοντας επισημάνει με έντονα γράμματα αυτές που προστέθηκαν

και αφορούν τα tag bits. Πιο αναλυτικά, ως επιπλέον είσοδος προστίθεται το σήμα *wr_data_tag* το οποίο σχετίζεται με το tag των δεδομένων που πρόκειται να γραφούν στην Dcache στην περίπτωση εντολών τύπου store. Επιπλέον, ως έξοδος προς τον πυρήνα του επεξεργαστή (m14k_core.v → m14k_edp.v) προστίθεται ένα νέο σήμα, το *rd_data_tag*, το οποίο αφορά το tag των δεδομένων που είναι αποθηκευμένα στην Data Cache και προορίζονται για εγγραφή σε κάποιον καταχωρητή (εντολές τύπου load). Το πλάτος σε αυτήν την περίπτωση -όπως συμβαίνει και με τα δεδομένα που είναι 64 bit- είναι το διπλάσιο (4 bits) λόγω της 2-δρόμων συνολο-συσχετιστικότητας της κρυφής μνήμης.

**Inputs**

| **Όνομα** | **Πλάτος** | **Περιγραφή** |
|---|---|---|
| tag_addr | 10 bits | Index-Tag array |
| tag_wr_en | 2 bits | Write enable |
| tag_rd_str | 1 bit | Tag Rd Strobe |
| tag_wr_str | 1 bit | Tag Wr Strobe |
| tag_wr_data | 24 bits | Data-Tag write |
| ws_addr | 10 bits | Index-WS array |
| ws_wr_mask | 14 bits | Dirty & LRU |
| ws_rd_str | 1 bit | WS Rd Strobe |
| ws_wr_str | 1 bit | WS Wr Strobe |
| ws_wr_data | 14 bits | Data-WS write |
| data_addr | 12 bits | Index-Data array |
| wr_mask | 9 bits | Byte mask-writes |
| data_rd_str | 1 bit | Data Rd Strobe |
| data_wr_str | 1 bit | Data Wr Strobe |
| wr_data | 32 bits | Data in |
| ***wr_data_tag*** | ***2 bits*** | ***Data in tag*** |

**Outputs**

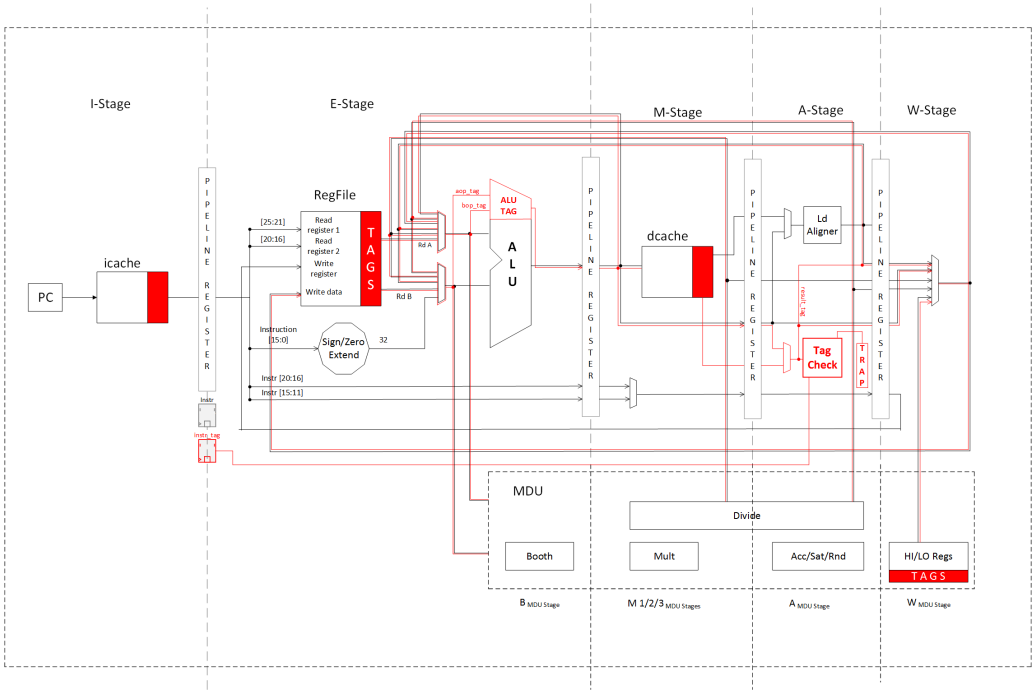
| **Όνομα** | **Πλάτος** | **Περιγραφή** |
|---|---|---|
| tag_rd_data | 50 bits | Read Tag |
| ws_rd_data | 14 bits | Read WS |
| rd_data | 64 bits | Read data |
| ***rd_data_tag*** | ***4 bits*** | ***Rd Data tag*** |
| num_sets | 3 bits | Way size |
| set_size | 2 bits | Associativity |
| hci | 1 bits | cache init? |

Πίνακας 5.8: Η τροποποιημένη Data Cache με τις επιπρόσθετες εισόδους/εξόδους, που σχετίζονται με τα tag bits, να επισημαίνονται με έντονα γράμματα.

Αναλυτικότερη περιγραφή των τροποποιημένων δομικών μονάδων του πυρήνα του επεξεργαστή γίνεται στο παράρτημα Α που ακολουθεί μετά το τέλος του 6ου κεφαλαίου.

Αφού παρουσιάσαμε, μέσω block diagram και πινάκων, τις βασικές αλλαγές που συντελέστηκαν σε κάποιες δομικές μονάδες του επεξεργαστή, σε αυτό το σημείο θα εμβαθύνουμε στο πώς τροποποιήθηκε το κεντρικότερο τμήμα του, η Integer Unit, παραθέτοντας το νέο datapath που προκύπτει μετά τις αλλαγές. Αναλυτικότερα, η Integer Unit είναι το κύριο τμήμα του πυρήνα του MIPS MicroAptiv επεξεργαστή στο οποίο εκτελούνται όλες οι εντολές που προέρχονται από την I-Cache και αποθηκεύουν δεδομένα στην D-Cache. Οι παρεμβάσεις-αλλαγές στον επεξεργαστή με σκοπό την επέκτασή του, ξεκίνησαν από αυτή τη μονάδα προκειμένου να υλοποιήσουμε ένα σύστημα DIFT που θα αντιμετωπίζει τις low level επιθέσεις. Πιο συγκεκριμένα, οι αλλαγές ξεκίνησαν από το αρχείο *m14k_edp.v* το οποίο είναι από τα μεγαλύτερα αρχεία με περίπου 3000 γραμμές κώδικα Verilog.

Στο σχήμα που ακολουθεί παρουσιάζουμε το 5-stage pipeline datapath του τροποποιημένου επεξεργαστή έχοντας επισημάνει με κόκκινο χρώμα τα σήματα, καταχωρητές/μνήμες που προστέθηκαν και επεκτάθηκαν αντίστοιχα προκειμένου να υλοποιηθεί ο μηχανισμός του *Dynamic Information Flow Tracking* στο επίπεδο του υλικού.




Σχήμα 5.4: Το pipeline datapath του τροποποιημένου επεξεργαστή. Με κόκκινο χρώμα τα επιπρόσθετα σήματα, καταχωρητές/μνήμη για την υποστήριξη της λειτουργικότητας DIFT.

Επιπλέον, στο σχήμα παρουσιάζουμε και τις αλλαγές που συντελέστηκαν στο ξεχωριστό pipeline που σχετίζεται με την μονάδα πολ/σμου και διαίρεσης. Θυμίζουμε σε αυτό το σημείο, πως παράλληλα με το pipeline της Integer Unit εκτελείται και το ξεχωριστό pipeline της MDUnit, το οποίο αφορά τις πράξεις του πολ/σμού-διαίρεσης ακεραίων. Τα δύο pipelines είναι ανεξάρτητα επομένως δεν υπάρχει καθυστέρηση όταν το pipeline της IU καθυστερεί. Το τελευταίο επιτρέπει στις πράξεις πολ/σμου-διαίρεσης (που απαιτούν περισσότερους κύκλους για να ολοκληρωθούν) να 'καλύπτονται' εν μέρει από τις καθυστερήσεις του συστήματος και τις υπόλοιπες εντολές που αφορούν την IU.

## 5.5 Μεθοδολογία Υλοποίησης

Αρχικά, πριν γίνει οποιαδήποτε τροποποίηση, επιβεβαιώσαμε τη σωστή λειτουργία του επεξεργαστή, χρησιμοποιώντας κάποια scripts που δημιουργήσαμε και ελέγχοντας στη συνέχεια τις τιμές των κυματομορφών προσομοίωσης για τα κατάλληλα σήματα. Ενδεικτικά, ένα τέτοιο αρχείο παρουσιάζουμε στο σχήμα που ακολουθεί.

```
Machine Code        Instruction Address     Assembly Code
24090001            // bfc00000:            addiu $9, $0, 1   # val = 1
3c08bf80            // bfc00004:            lui   $8, 0xbf80  # $8=0xbf800000
ad090000            // bfc00008:        L1: sw    $9, 0($8)   # mem[0xbf800000] = val
25290001            // bfc0000c:            addiu $9, $9, 1   # val = val+1
1000fffd            // bfc00010:            beqz  $0, L1      # branch to L1
00000000            // bfc00014:            nop               # branch delay slot
```

Σχήμα 5.5: Παράδειγμα αρχείου που χρησιμοποιήθηκε κατά την προσομοίωση του επεξεργαστή για την επιβεβαίωση σωστής λειτουργίας του.

Έπειτα, προχωρήσαμε με τις τροποποιήσεις των βασικών μονάδων και του pipeline, όπως περιγράψαμε παραπάνω. Για την επιβεβαίωση της ορθής υλοποίησης του μηχανισμού ακολουθήθηκε η παρακάτω λογική:

- χωρίσαμε τις εντολές ανά κατηγορία, ανάλογα με την λειτουργικότητά τους. Η διάκριση έγινε στις εξής κατηγορίες: αριθμητικές εντολές (όλες οι εντολές τύπου add, sub, εντολές πολ/σμού-διαίρεσης), λογικές εντολές, εντολές τύπου load, εντολές τύπου store και jump εντολές.

- υλοποιήσσμε τα αντίστοιχα scripts για κάθε κατηγορία εντολών, προκειμένου να γίνει πιο εύκολος ο έλεγχος για τη σωστή υλοποίηση του μηχανισμού των tags (ο έλεγχος αυτός αφορά κυρίως τη σωστή παραγωγή-διάδοση των tags).
- υλοποιήσαμε τα τελικά scripts -χωρίς να γίνει κατηγοριοποίηση των εντολών- προκειμένου να ελέγξουμε την εξαίρεση ασφαλείας που δημιουργείται και αν αυτή προκύπτει στις περιπτώσεις που είναι αναγκαία.

Σε αυτό το σημείο παρουσιάζουμε κάποια από τα αρχεία που χρησιμοποιήθηκαν ως simulation sources για την επιβεβαίωση της σωστής λειτουργίας του μηχανισμού των tags ως προς την παραγωγή-διάδοσή τους. Ενδεικτικά, για τις αριθμητικές εντολές χρησιμοποιήθηκε το αρχείο arithmetic_ops.txt το περιεχόμενο του οποίου φαίνεται παρακάτω:

```
1           //                                          P bit   T bit
2           //
3 20080005  //       addi $8, $0, 5      $8 = 5
4 20090003  //       addi $9, $0, 3                ---->    extra DIFT
5 79284000  //       wr_tag $8, $9       Tag($8) = $9     instructions Tag($8) = 3
6           //                                          (Valid pointer & Tainted)
7 108001a   //       div  $8, $8         LO = 1             0       1
8 3012      //       MFLO $6             $6 = LO = 1        0       1
9 21080006  //       addi $8, $8, 6      $8 = 11            1       1
10 1064022  //       subu $8, $8, $6     $8 = 10            1       1
11 1060018  //       mult $8, $6         LO = 10            0       1
12 3012     //       MFLO $6             $6 = 10            0       1
13 63043    //       sra  $6, $6, 1      $6 = 5             0       1
14
```

Σχήμα 5.6: Αρχείο που χρησιμοποιήθηκε ως simulation source στην περίπτωση των αριθμητικών εντολών. Ως σχόλια έχουν επισημανθεί οι εντολές σε γλώσσα Assembly καθώς και οι τιμές που παίρνουν τα tags ανάλογα με την εντολή που εκτελείται.

Στο σχήμα, αρχικά, παρατηρούμε τις extra DIFT εντολές που προσθέτονται για την ανάθεση τιμών στα tags. Όσον αφορά την παραγωγή του Taint bit παρατηρούμε πως, μετά τον ορισμό του που γίνεται στις γραμμές 4 και 5, το T bit είναι αληθές για όλες τις υπόλοιπες πράξεις που εκτελούνται, πράγμα λογικό αφού η παραγωγή του βασίζεται στον εξής κανόνα: αν τουλάχιστον ένας από τους τελεστέους που χρησιμοποιούνται στην πράξη είναι μολυσμένος τότε και το αποτέλεσμα που προκύπτει είναι επίσης μολυσμένο.

Στην περίπτωση του Pointer bit η διαδικασία είναι διαφορετική καθώς ο ορισμός του, συμβαίνει μόνο κατά την τέλεση έγκυρων πράξεων μεταξύ pointers. Έτσι, κατά τη διαίρεση (γραμμή 7,8) και πολ/σμό (γραμμή 11,12) η τιμή του P bit είναι μηδέν (δεν αποτελούν έγκυρες πράξεις μεταξύ δεικτών) ενώ στην περίπτωση της πρόσθεσης (γραμμή 9) και αφαίρεσης (γραμμή 10) το P bit ορίζεται ως αληθές αφού τουλάχιστον ένας τελεστέος που συμμετέχει στις πράξεις ($8 και στις δύο περιπτώσεις) έχει το Pointer bit του ορισμένο ως αληθές.

Όπως προαναφέρθηκε, μετά την διάκριση των εντολών ανά κατηγορία και την υλοποίηση των αντίστοιχων scripts για τον έλεγχο της σωστής λειτουργίας της διαδικασίας παραγωγής-διάδοσης των tags, δημιουργήθηκαν τα κατάλληλα αρχεία για τον έλεγχο της ορθής λειτουργίας ολόκληρου του μηχανισμού (παραγωγή και δημιουργία εξαίρεσης ασφαλείας). Στο παρακάτω σχήμα παραθέτουμε ένα από τα αρχεία που χρησιμοποιήθηκαν για να γίνει ο προαναφερόμενος έλεγχος.

```
        //                                                                                  CHECK RULE
        //
3C08BFC0 //    bfc00000:    lui $8, 1011111111000000       // $8 = bfc00000                 -
20090001 //    bfc00004:    addi $9, $0, 1                                                   -
79284000 //    bfc00008:    wr_tag       $8, $9            // Tag($8) = $9                  -
35080028 //    bfc0000c:    ori $8, $8, 0000000000101000   // $8 = bfc00028                 -
100F8809  //    bfc00010:    jalr $31, $8                   // $31 = bfc00014, PC = $8     TRAP
2002000A //    bfc00014:    addi $2, $0, 10                // $2 = 10                       -
                    .
                    .
                    .
200403E8 //    bfc00028:    addi $4, $0, 1000              // $4 = 1000                     -
3E00008  //    bfc0002c:    jr   $31                       // PC = bfc00014                 -
```

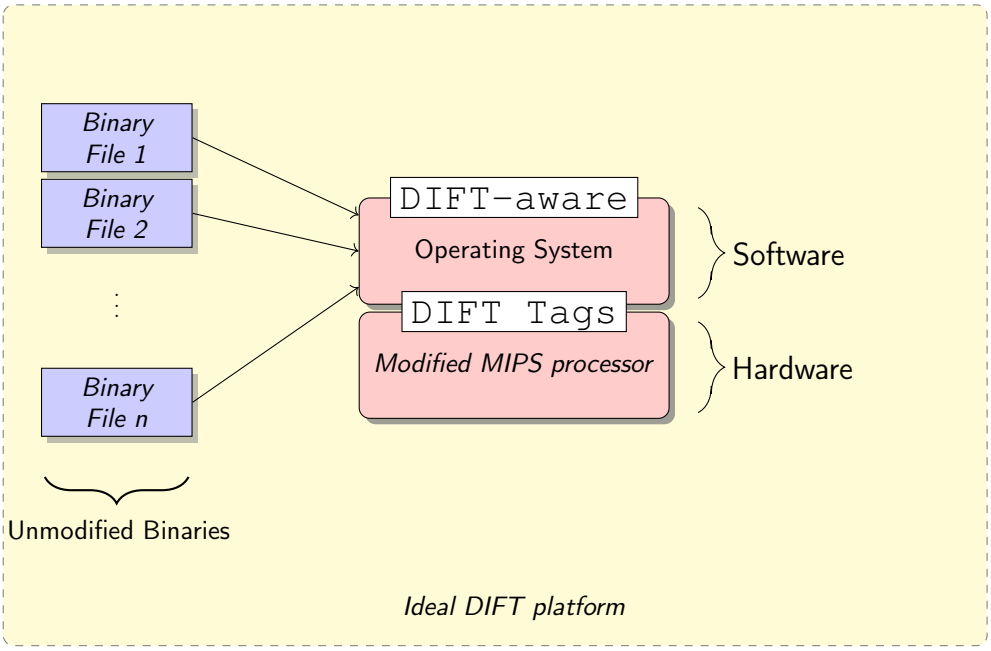
Σχήμα 5.7: Το αρχείο *final_sim_src.txt* που χρησιμοποιήθηκε ως simulation source για την επιβεβαίωση της σωστής λειτουργίας ολόκληρου του μηχανισμού. Ως σχόλια έχουν επισημανθεί οι αντίστοιχες εντολές Assembly και οι τιμές των tags για κάθε εντολή που εκτελείται.

Όπως φαίνεται από το παραπάνω σχήμα στις γραμμές 1 και 2 δημιουργείται και εκχωρείται στον καταχωρητή $8 η διεύθυνση *bfc00028* που παρακάτω θα χρησιμοποιηθεί στην εντολή άλματος. Στη συνέχεια, μέσω των extra DIFT εντολών που προστέθηκαν αναθέτουμε τιμή στα tags (υποθέτοντας πως σε αυτήν την περίπτωση έχουμε τον εξής συνδυασμό: non valid pointer - tainted). Κατόπιν το πρόγραμμα προχωρά στην εκτέλεση της εντολής jalr η οποία δεν εκτελείται (δεν γίνεται η υποχρεωτική διακλάδωση στη διεύθυνση *bfc00028*) καθώς η διεύθυνση που επιχειρείται να εκχωρηθεί στον PC είναι μη έγκυρη και tainted ταυτόχρονα.

## 5.6 Επισκόπηση Συστήματος

Για να θεωρείται ένα σύστημα DIFT πλήρως ολοκληρωμένο [29] θα πρέπει να περιλαμβάνει τα εξής:

1. το λειτουργικό σύστημα το οποίο θα αναγνωρίζει, μέσω ενός αλγορίθμου, τα δεδομένα που προέρχονται από μη έμπιστες πηγές (π.χ. network I/O) και στη συνέχεια θα τα μαρκάρει ως ύποπτα.
2. τον τροποποιημένο κατάλληλα, για την υποστήριξη της λειτουργικότητας DIFT, επεξεργαστή ο οποίος θα δημιουργεί μια εξαίρεση ασφαλείας στην περίπτωση που τα ύποπτα δεδομένα χρησιμοποιούνται επισφαλώς.

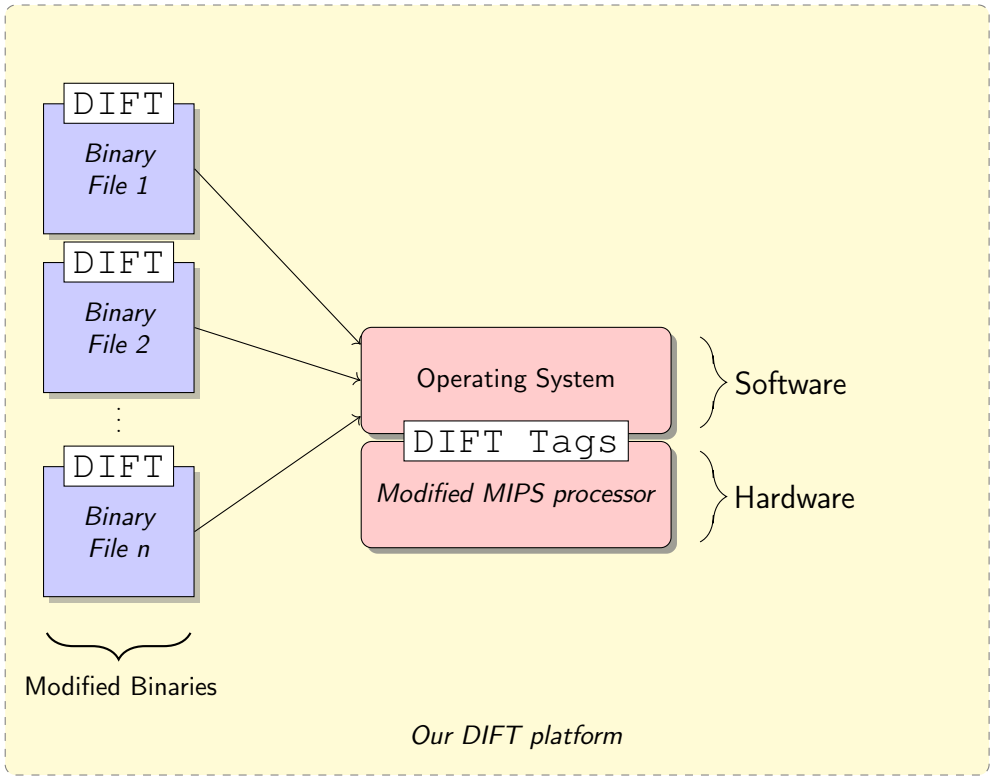



Σχήμα 5.8: Ολοκληρωμένη πλατφόρμα DIFT που πληροί τις παρακάτω προϋποθέσεις: λειτουργικό σύστημα το οποίο, μέσω κατάλληλου αλγορίθμου, μαρκάρει τα ύποπτα δεδομένα και τον τροποποιημένο MIPS επεξεργαστή που έχει επεκταθεί για να υποστηρίζει τη λειτουργία των tags.

Στην περίπτωση αυτής της υλοποίησης, δεν απαιτείται κάποια αλλαγή στον κώδικα των αρχείων που τρέχουν στο σύστημα καθώς το λειτουργικό σύστημα είναι εκείνο που αποφασίζει ποια δεδομένα προέρχονται από αναξιόπιστες πηγές. Στο *Σχήμα 5.8*, παραπάνω, απεικονίζεται διαγραμματικά αυτό που περιγράφτηκε προηγουμένως, δηλαδή μια ιδανική πλατφόρμα

DIFT. Παρατηρούμε τα μη τροποποιημένα binaries καθώς επίσης και την λειτουργικότητα DIFT τόσο στο επίπεδο του υλικού όσο και στο επίπεδο του λογισμικού.

Όσον αφορά την υλοποίησή μας, διαφέρει από την αντίστοιχη ιδανική ως προς το κομμάτι του λογισμικού. Με άλλα λόγια, πληρούται η προϋπόθεση του τροποποιημένου επεξεργαστή, ο οποίος επεκτείνεται ώστε να υποστηρίζει τη λειτουργικότητα του μηχανισμού (tags), αλλά δεν υλοποιείται, σε επίπεδο λογισμικού, ο αλγόριθμος εκείνος που θα όριζε ποια δεδομένα θεωρούνται ύποπτα και ποια όχι.




Σχήμα 5.9: Η δική μας προσέγγιση που περιλαμβάνει τον τροποποιημένο κατάλληλα MIPS επεξεργαστή. Λόγω της μη υλοποίησης της λειτουργικότητας DIFT στο επίπεδο του λογισμικού, τροποποιούνται τα binaries με τρόπο τέτοιο ώστε να ορίζεται (hardcoded) ποια δεδομένα προέρχονται από αναξιόπιστες πηγές και θεωρούνται ύποπτα.

Για την κάλυψη αυτής της έλλειψης οδηγούμαστε στην τροποποίηση των binaries έτσι ώστε να ορίζουμε *hardcoded*, στο επίπεδο της Assembly, τα αναξιόπιστα δεδομένα. Πιο αναλυτικά, τροποποιούμε τον κώδικα Assembly,

προσθέτοντας τις extra εντολές που περιγράφηκαν παραπάνω προκειμένου να ορίσουμε hardcoded το tag των δεδομένων που προέρχονται από έμπιστες ή μη πηγές, ελλείψει της υλοποίησης ενός αλγορίθμου που θα καθόριζε ποια δεδομένα είναι αξιόπιστα και ποια όχι. Στο *Σχήμα 5.9* παρουσιάζουμε τη δική μας προσέγγιση με την υλοποίηση του μηχανισμού σε επίπεδο υλικού και την αλλαγή των αρχείων (προσθήκη επιπρόσθετων εντολών Assembly) έτσι ώστε να καλύψουμε την λειτουργικότητα της τεχνικής στο επίπεδο του λειτουργικού συστήματος.

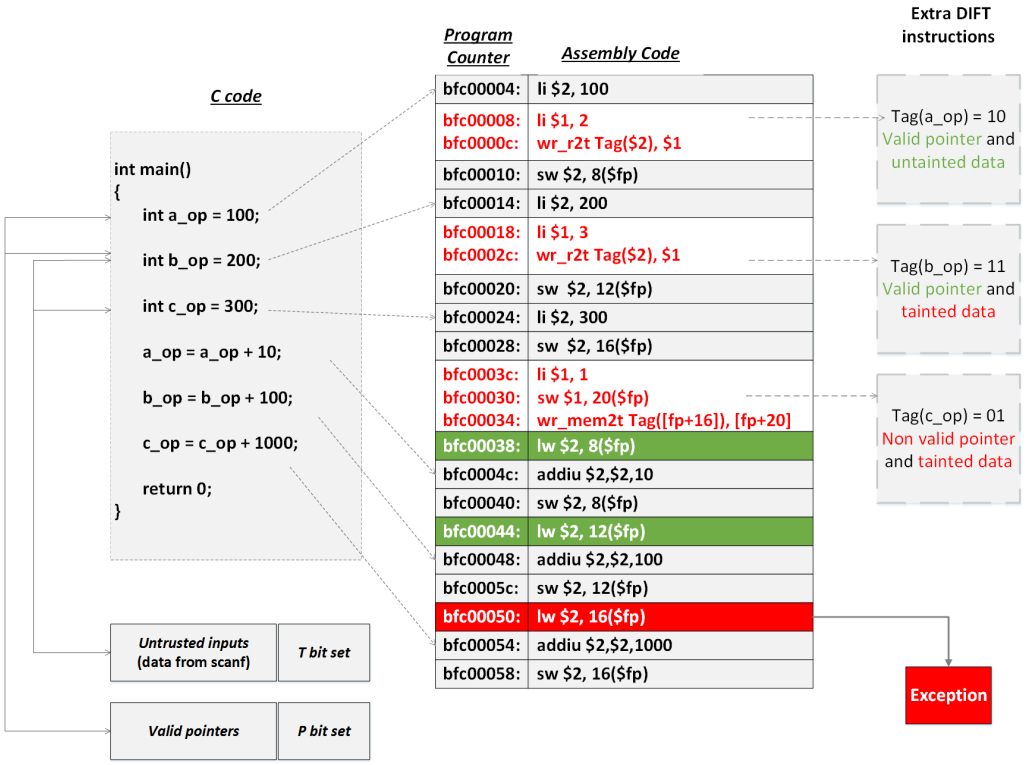
## 5.7 Προσομοίωση Συστήματος

### 5.7.1 Διαδικασία Προσομοίωσης

Κατά την προσομοίωση του συστήματος DIFT τροποποιούμε κατάλληλα τα binaries, προσθέτοντας extra εντολές για την ανάθεση τιμών στα tags (αναλυτική περιγραφή στην ενότητα 5.3). Αφού γίνει η παραπάνω διαδικασία - παρέμβαση, είμαστε έτοιμοι να 'τρέξουμε' τα αρχεία αυτά στον τροποποιημένο MIPS επεξεργαστή προκειμένου να εξετάσουμε την ορθότητα της σχεδίασης του μηχανισμού DIFT στο επίπεδο του υλικού.

Στα παραδείγματα που ακολουθούν παρουσιάζονται προγράμματα σε γλώσσα C μαζί με τις εντολές Assembly που τους αντιστοιχούν (αφού γίνει η διαδικασία του compile). Επισημαίνονται επίσης και οι επιπρόσθετες εντολές που προστέθηκαν για την υποστήριξη της λειτουργικότητας του μηχανισμού στο επίπεδο του λογισμικού. Τα παραδείγματα δείχνουν την ροή και διάδοση της πληροφορίας των tags από τη μια μεταβλητή στην άλλη, όπως αυτές ορίζονται από τους κανόνες που αναφέρθηκαν στην υπο-ενότητα 5.1.1.

Αναλυτικότερα, στο παράδειγμα που παρατίθεται στο *Σχήμα 5.8* υποθέσαμε πως οι τιμές των μεταβλητών b_op,c_op προέρχονται από μια μη έμπιστη εξωτερική πηγή (όπως για παράδειγμα από μια scanf εντολή), επομένως τα δεδομένα τους θεωρούνται αναξιόπιστα καθώς επίσης και ότι ο δείκτης c_op είναι μη έγκυρος δείκτης. Με βάση τα παραπάνω, εκχωρήσαμε τις κατάλληλες τιμές στα tags, μέσω των extra DIFT εντολών (κόκκινο χρώμα στο σχήμα). Στη συνέχεια, η διάδοση της πληροφορίας των tags και τελικά ο έλεγχός τους έχουν ως αποτέλεσμα την εξαίρεση ασφαλείας που δημιουργείται όταν η μη έγκυρη εντολή load με *Program Counter bfc00050* επιχειρείται να εκτελεσθεί από τον τροποποιημένο επεξεργαστή.




Σχήμα 5.10: Παράδειγμα προγράμματος C και οι αντίστοιχες Assembly εντολές μαζί με τις extra DIFT εντολές που προσθέσαμε (κόκκινο χρώμα). Στο παράδειγμα υποθέσαμε ότι οι μεταβλητές b_op, c_op είναι αναξιόπιστες είσοδοι (σε αντίθεση με τα δεδομένα της a_op που είναι καθαρά) και ότι η μεταβλητή c_op αντιστοιχεί σε μη έγκυρη θέση μνήμης.

### 5.7.2 Αξιολόγηση Απόδοσης

Σε αυτό το σημείο γίνεται η αξιολόγηση της απόδοσης του νέου συστήματος, που περιλαμβάνει τον τροποποιημένο, για την υποστήριξη της λειτουργικότητας DIFT, επεξεργαστή. Πιο αναλυτικά, υπολογίζεται το κόστος της νέας σχεδίασης συγκρίνοντας, μέσα από ένα συγκεντρωτικό πίνακα, συγκεκριμένα δομικά στοιχεία μιας FPGA (slices LUTs, Flip-Flops, Multiplexers, Registers, Block RAM). Αφού πρώτα παρουσιαστούν επιγραμματικά οι δομικές μονάδες, στη συνέχεια παρατίθενται οι πίνακες για τον υπολογισμό του κόστους ανάμεσα στις δύο υλοποιήσεις.

- *Flip-Flops, LUTs, and Slices*: Ένα slice περιέχει έναν αριθμό από LUTs, flip-flops and multiplexers. Ένα LUT είναι μια συλλογή από λογικές πύλες που αποθηκεύει μια προκαθορισμένη λίστα από εξόδους για κάθε

συνδυασμό εισόδων, παρέχοντας έτσι ένα γρήγορο τρόπο ανάκτησης της εξόδου για οποιαδήποτε λογική πράξη.

- *Block Ram*: μνήμη RAM ενσωματωμένη στην FPGA για την αποθήκευση δεδομένων.

| **Resources** | **Fpga Availability** | **MIPS** | **MIPS + DIFT** | **Util Diff (%)** |
|---|---|---|---|---|
| *Slice LUTs* | 63400 | 9354 | 9540 | 0.29 |
| *Slice Regs* | 126800 | 7423 | 7599 | 0.14 |
| *Block Ram* | 135 | 109.5 | 113.5 | 2.96 |

Table 5.9: Συγκεντρωτικός πίνακας που παρουσιάζει τη χρήση των Slice LUTs, Regs και της Memory στην περίπτωση των δύο υλοποιήσεων. Η τρίτη στήλη αναφέρεται στον μη τροποποιημένο επεξεργαστή MIPS, η τέταρτη στήλη αφορά τον επεξεργαστή που υποστηρίζει τη λειτουργικότητα DIFT, ενώ η τελευταία στήλη καταδεικνύει την % διαφορά χρήσης των δομικών στοιχείων της fpga ανάμεσα στις δύο υλοποιήσεις.

Από τον παραπάνω πίνακα, παρατηρούμε την αύξηση της χρήσης των slice LUTs και registers η οποία παρ' όλα αυτά δεν επιφέρει ιδιαίτερη επιβάρυνση σε σχέση με το μη τροποποιημένο σύστημα. Όπως φαίνεται στη στήλη Util Diff (%), η διαφορά της χρήσης των *slice LUTs* είναι 0.29% (έχουμε αύξηση από 14.75%→15.04%) ενώ στην περίπτωση των *Slice Registers* η αύξηση αυτή είναι 0.14% (από 5.85%→5.99%).

Τέλος, στην τελευταία σειρά του πίνακα, εστιάζουμε στη χρήση της *Block RAM*, παρατηρώντας μια επιβάρυνση σχεδόν 3% ανάμεσα στις δύο υλοποιήσεις (81.11%→84.07%).

Σημειώνεται πως οι αναλυτικοί πίνακες, όπως αυτοί εξήχθησαν από το εργαλείο Vivado, παρατίθενται στο παράρτημα Β ¨Αξιολόγηση Απόδοσης¨.

# Κεφάλαιο 6

# Επίλογος

Σε αυτό το κεφάλαιο παρουσιάζεται ο επίλογος της εργασίας κάνοντας μια ανασκόπηση του μηχανισμού DIFT και των παραμέτρων/στοιχείων εκείνων που ήταν αναγκαία για να οδηγηθούμε στην υλοποίηση του τεχνικής DIFT στον επεξεργαστή MIPS.

Οι υπάρχοντες μηχανισμοί άμυνας για την αντιμετώπιση των κακόβουλων ενεργειών υστερούν σε τομείς όπως η ευελιξία, πρακτικότητα και η ταχύτητα καθώς επίσης και στην αποτελεσματικότητά τους αφού πολλές φορές μπορούν να παρακαμφθούν από τους επιτιθέμενους, μη παρέχοντας ουσιαστική προστασία από τις κακόβουλες ενέργειες. Για τον λόγο αυτό, αναπτύχθηκε η τεχνική του *Dynamic Information Flow Tracking* που αποτελεί ένα πολλά υποσχόμενο και ισχυρό εργαλείο απέναντι στις επιθέσεις, καθώς είναι η πρώτη και ίσως μοναδική τεχνική που μπορεί να παρέχει πλήρη προστασία έναντι των κακόβουλων ενεργειών. Σε αυτήν την διπλωματική εργασία παρουσιάζουμε τον μηχανισμό του DIFT και την πολύπλευρη αποτελεσματικότητά του έναντι μεγάλου εύρους επιθέσεων ασφαλείας, ανάλογα με την πολιτική που κάθε φορά εφαρμόζεται.

Πιο αναλυτικά, παραθέτουμε την εφαρμογή του DIFT στον επεξεργαστή MIPS, αναφέροντας όλες τις τροποποιήσεις που απαιτείται να γίνουν προκειμένου να υλοποιηθεί επιτυχώς ο μηχανισμός. Οι περισσότερες από αυτές τις αλλαγές συντελέστηκαν στον πυρήνα του επεξεργαστή (αρχείο *m14k_core.v*) και όχι σε όλο τον *MIPS32 MicroAptiv*. Οι μετατροπές αφορούσαν κατά κύριο λόγο την επέκταση των καταχωρητών και μνημών για την υποστήριξη της επιπρόσθετης πληροφορίας, που σχετίζεται με την αξιοπιστία των δεδομένων. Επιπλέον, υλοποιήθηκε η μονάδα για την παραγωγή του *Taint - Pointer bit* καθώς και το κατάλληλο module για τον έλεγχο της αξιοπιστίας των δεδομένων ανάλογα με τις τιμές των tags. Στην εργασία δόθηκε έμφαση στην παρακολούθηση της ροής των δεδομένων στο επίπεδο

του υλικού (επεξεργαστής MIPS) και όχι στην υλοποίηση ενός ολοκληρωμένου συστήματος που δέχεται μη τροποποιημένα binaries με 'ευάλωτο' - ενδεχομένως κακόβουλο, κώδικα. Λόγω της έλλειψης της λειτουργικότητας DIFT στο επίπεδο του λογισμικού, επιλέξαμε την τροποποίηση των binaries, προσθέτοντας επιπλέον εντολές ανάμεσα στις οποίες και κάποιες *extra DIFT* εντολές, τις οποίες εντάξαμε ως έγκυρες στο σύνολο εντολών του επεξεργαστή. Για την ένταξη των εντολών στο σύνολο εντολών του MIPS ήταν απαραίτητη η τροποποίηση του *decode stage* (αρχείο *m14k_mpc_dec.v*) καθώς και του pipeline (αρχείο *m14k_mpc_ctl.v*). Αυτή η μέθοδος, μας επέτρεψε να προσομοιώνουμε πολλά διαφορετικά binary αρχεία χωρίς να απαιτείται κάθε φορά η τροποποίηση της σχεδίασης στο επίπεδο του υλικού.

Ως τελικό στάδιο, μετά την υλοποίηση, ακολούθησε ο έλεγχος και η αξιολόγηση της πλατφόρμας DIFT. Με τη χρήση binaries γραμμένων σε *C* ή *Assembly* εξετάσαμε το σύστημα DIFT, καταλήγοντας στο συμπέρασμα πως αποτρέπονται επιτυχώς οι low level επιθέσεις, όπως το buffer overflow, εγείροντας το κατάλληλο exception τη στιγμή που απαιτείται.

Τέλος, παρουσιάζουμε τις βελτιώσεις που επιδέχεται η πλατφόρμα που υλοποιήσαμε, οι οποίες αφορούν κάποια μελλοντική μελέτη πάνω στον μηχανισμό του *Dynamic Information Flow Tracking*. Πιο συγκεκριμένα, ένα ουσιαστικό και πολύ ενδιαφέρον ζήτημα είναι η εφαρμογή της λειτουργικότητας της τεχνικής και στο κομμάτι του λογισμικού, με σκοπό την υλοποίηση ενός πλήρους συστήματος που θα αποτρέπει επιτυχώς τις κακόβουλες ενέργειες. Με άλλα λόγια, μια νέα πρόκληση είναι η εύρεση ενός αλγορίθμου που θα ορίζει και θα επισημαίνει τα δεδομένα που προέρχονται από αναξιόπιστες πηγές έτσι ώστε να μην απαιτείται κάποια τροποποίηση στα binaries με την προσθήκη έξτρα εντολών και την hardcoded ανάθεση τιμών στα tag bits.

Επιπλέον, μια άλλη πρόκληση είναι οι πολιτικές DIFT και πώς αυτές μπορούν να χρησιμοποιηθούν συνδυαστικά, με την προσθήκη νέων bits, ώστε να παρέχουν προστασία έναντι μεγάλου εύρους low level και high level επιθέσεων.

# Παράρτημα Α΄

# I/O signals in Verilog

Στην ενότητα αυτή παραθέτουμε τις βασικές τροποποιήσεις που συντελέστηκαν στον κώδικα των αρχείων (Verilog) των κύριων τμημάτων του επεξεργαστή. Με σχόλια επισημαίνουμε κάθε φορά σε ποια μονάδα αντιστοιχεί ο κώδικας Verilog που παρουσιάζουμε, δείχνοντας τις εισόδους-εξόδους κάθε δομικής μονάδας καθώς και μια σύντομη περιγραφή της σημασίας του κάθε σήματος. Σημειώνεται πως δίπλα από κάθε είσοδο/έξοδο

```
1  /* ---------------------Register-File--(m14k_rf_reg.v) -------------- */
2  module m14k_rf_reg(
3          mpc_dest_w,
4          mpc_rega_cond_i,
5          mpc_rega_i,
6          mpc_regb_cond_i,
7          mpc_regb_i,
8          mpc_rfwrite_w,
9          edp_wrdata_w,
10         wrdata_w_tag,     // regfile write data tag
11         gclk,
12         greset,
13         rf_init_done,
14         rf_adt_e,
15         rfa_tag,         // read data 1 tag
16         rf_bdt_e,
17         rfb_tag);        // read data 2 tag
18
19         /* Inputs */
20         input [8:0]     mpc_dest_w;             // destination register
21         input           mpc_rega_cond_i;        // regfile source A register condition
22         input [8:0]     mpc_rega_i;             // regfile source A register
23         input           mpc_regb_cond_i;        // regfile source B register condition
24         input [8:0]     mpc_regb_i;             // regfile source B register
25         input           mpc_rfwrite_w;      // regfile write enable
26         input [31:0]    edp_wrdata_w;       // regfile write data
27         input [1:0]     wrdata_w_tag;       // regfile write data tag
```

```
28          input           gclk;               // Global clock
29          input           greset;
30
31          /* Outputs */
32          output          rf_init_done;
33          output [31:0]   rf_adt_e;               // regfile A read port
34          output [1:0]    rfa_tag;        // regfile A read port tag
35          output [31:0]   rf_bdt_e;               // regfile B read port
36          output [1:0]    rfb_tag;        // regfile B read port tag
```

```
1   /* ----------------------DCache-I/O---(m14k_dc.v) -----------------*/
2   module m14k_dc(
3           gclk,
4           greset,
5           tag_addr,
6           tag_wr_en,
7           tag_rd_str,
8           tag_wr_str,
9           tag_wr_data,
10          tag_rd_data,
11          ws_addr,
12          ws_wr_mask,
13          ws_rd_str,
14          ws_wr_str,
15          ws_wr_data,
16          ws_rd_data,
17          data_addr,
18          wr_mask,
19          data_rd_str,
20          data_wr_str,
21          wr_data,
22          wr_data_tag,            // Data in tag
23          rd_data,
24          rd_data_tag,            // Read_data tag
25          num_sets,
26          set_size,
27          hci,
28          cache_present);
29
30          input           gclk;
31          input           greset;
32          /* tag array port */
33          input [13:4]    tag_addr;                       // Index into tag array
34          input [(`M14K_MAX_DC_ASSOC-1):0]   tag_wr_en;
35          input           tag_rd_str;                     // Tag Read Strobe
36          input           tag_wr_str;                     // Tag Write Strobe
37          input [T_BITS-1:0]      tag_wr_data;            // Data for Tag write
38
39          output [(T_BITS*`M14K_MAX_DC_ASSOC-1):0]        tag_rd_data;    // read tag
40          input [13:4]    ws_addr;          // Index into WS array
41          input [13:0]    ws_wr_mask;       // Write mask to write dirty bit & LRU
```

```
42          input           ws_rd_str;          // WS Read Strobe
43          input           ws_wr_str;          // WS Write Strobe
44          input [13:0]    ws_wr_data;         // Data for WS write
45
46          output [13:0] ws_rd_data;
47
48          /* data array port */
49          input [13:2]    data_addr;
50          input [(4*`M14K_MAX_DC_ASSOC-1):0]      wr_mask;        // Byte Mask for writes
51          input           data_rd_str;        // Data Read Strobe
52          input           data_wr_str;        // Data Write Strobe
53          input [D_BITS-1:0]      wr_data;    // Data in
54          input [1:0] wr_data_tag;            // Data in tag
55
56          output [(D_BITS*`M14K_MAX_DC_ASSOC-1):0] rd_data;  //Read data for N ways of cache
57          output [(2*`M14K_MAX_DC_ASSOC-1):0] rd_data_tag; //Read Data tag for N ways of cache
58
59          // Static outputs to tell core what cache looks like;
60          output [2:0]    num_sets;       // Way Size:  0=1KB, 1=2KB, 2=4KB, 3=8KB, 4=16KB
61          output [1:0]    set_size;       // Associativity:
62                                          // 0=DM, 1=2WSA, 2=3WSA, 3=4WSA
63          output          hci;            // hardware cache init?
64          output          cache_present;  // Do we have a cache?
```

```
1  /* ---------------------------MDUnit--I/O--(m14k_mdl.v)------------------- */
2  input [31:0]    edp_abus_e;
3  input [1:0]     edp_abus_tag;     // MDU's src A tag
4  input [31:0]    edp_bbus_e;
5  input [1:0]     edp_bbus_tag;     // MDU's src A tag
6  input           gclk;
7  input           greset;
8  input           gscanenable;
9  input           mpc_ekillmd_m;
10 input           mpc_wrdsp_e;
11 input [31:0]    mpc_ir_e;
12 input [22:0]    mpc_predec_e;
13 input           mpc_umipspresent;
14 input           mpc_irval_e;
15 input           mpc_killmd_m;
16 input           mpc_run_ie;
17 input           mpc_run_m;
18 input           mpc_srcvld_e;
19
20 // AG stage I/O
21 input           MDU_run_ag;     // Run signal from ALU
22 input [31:0]    MDU_ir_ag;   // Instruction word for source/dest and RI decode
23 input           MDU_dec_ag;  // Instruction word on MDU_ir_ag decodes to a
24                               // MDU/UDI instruction. This is used by the MDU
25                               // to determine when to assert stallreq without
26                               // using MDU_opcode_issue_ag because
27                               // MDU_opcode_issue_ag is timing critical.
```

```
28 input           MDU_opcode_issue_ag; // issue Instruction on MDU_ir_ag; active
29                                       // the last cycle Instruction is in AG
30 // EX stage I/O
31 input           MDU_run_ex;     // Run signal from ALU
32 input             MDU_nullify_ex;  // Nullify inst already sent to MDU before
33                                     // it gets to MS stage
34 input           MDU_data_valid_ex;  // Data word(s) to udi/MDU valid.
35                                       // Asserted for the remain of EX
36 input [31:0]    MDU_rs_ex;      // Source operand
37 input [31:0]    MDU_rt_ex;      // Source operand
38
39 // MS stage I/O
40 input           MDU_run_ms;     // Run signal from ALU
41 input           MDU_nullify_ms;       // Nullify MDU instruction in MS
42 input           MDU_data_ack_ms; // Result returned to ALU when this signal
43                                     // is asserted asserted is accepted.
44                                     // Otherwise, MDU_rd_strobe_ms must be kept
45                                     // asserted until this signal is asserted.
46 // ER stage I/O
47 input           MDU_run_er;     // Run signal from ALU
48 input           MDU_nullify_er;      // Nullify MDU instruction in ER
49 input           MDU_kill_er;       // Kill signal due to an exception
50                                       // generated by an earlier instruction
51 input [4:0]     mpc_dest_e;
52 input           MDU_data_val_ex;
53 output          MDU_stallreq_ag;  // Stalls MDU instr moving from AG to EX
54                                      // next cycle.
55 // no particular stage I/O
56 output          mdu_type;    //MDU type: 0 normal, 1 iterative
57 output          MDU_rfwrite_ms;
58 output [4:0]    MDU_dest_ms;
59 output          mdu_alive_gpr_m1;
60 output          mdu_alive_gpr_m2;
61 output          mdu_alive_gpr_m3;
62 output          mdu_mf_m1;
63 output          mdu_mf_m2;
64 output          mdu_mf_m3;
65 output          mdu_mf_a;
66 output          mdu_nullify_m2;
67 output [4:0]    mdu_dest_m1;
68 output [4:0]    mdu_dest_m2;
69 output [4:0]    mdu_dest_m3;
70 output          mdu_alive_gpr_a;
71
72 output          mdu_busy;
73 output [31:0]   mdu_res_w;         // MDU result
74 output [1:0]    mdu_res_w_tag;     // result tag of MDUnit
75 output          mdu_stall;
76 output          mdu_result_done;

1
```

```
2  /* --------------------Execution-Datapath--(m14k_edp.v) -------------- */
3  input [31:0]     rf_adt_e;               // edp_abus_e data from register file
4  input [1:0]      rfa_tag;                // edp_abus_e data tag from regfile
5  input [31:0]     rf_bdt_e;               // edp_bbus_e data from register file
6  input [1:0]      rfb_tag;                // edp_bbus_e data tag from regfile
7
8  input [31:0]     dcc_ddata_m;            // Dcache Read data
9  input [1:0]      rd_data_tag;            // Dcache Read data tag
10 input [31:0]     mdu_res_w;              // MDU result (MUL, MFHI/LO)
11 input [1:0]      mdu_res_w_tag;          // MDU result tag
12 input [31:0]     mpc_ir_e;               // Full instn
13
14 // res_m mux controls
15 input            mpc_muldiv_w;        // MDU op in W
16 input            mpc_lnksel_m;        // Link instn (JAL/BAL)
17 input            mpc_lnksel_e;        // Link instn (JAL/BAL)
18 input            mpc_compact_e;       // Compact jump
19 input            mpc_alusel_m;        // Select ALU
20 input            mpc_udisel_m;        // Select UDI
21 input            mpc_clsel_e;         // Select Count Leading
22 input            mpc_clinvert_e;      // Invert Count Leading input (CL0)
23 input            mpc_clvl_e;          // Count extension
24 input            mpc_udislt_sel_m;    // Select UDI or SLT
25
26 // A/edp_bbus_e bypass controls
27 input            mpc_aselres_e;        // Bypass res_m as src A
28 input            mpc_aselwr_e;         // Bypass res_w as src A
29 input            mpc_bselres_e;        // Bypass res_m as src B
30 input            mpc_bselall_e;        // Bypass res_w as src B
31
32 input            mpc_imsgn_e;         // Sign Bit to extend immediate with
33 input            mpc_selimm_e;        // Select Immediate
34 input            mpc_br16_e;          // UMIPS Branch instn in E
35 input            mpc_br32_e;          // MIPS32 Branch instn in E
36 input            mpc_sequential_e;    // sequential instn stream
37 input            mpc_apcsel_e;        // Select PC as src A
38 input            icc_pcrel_e;         // Mask low 2b of PC for PC relative ops
39 input            mpc_pcrel_e;         // Mask low 2b of PC for PC relative ops
40 input            mpc_lxs_e;              // Load index scaled command
41 input            mpc_usesrca_e;
42 input            mpc_usesrcb_e;
43
44 // Rotate control
45 input            mpc_selrot_e;    // Perform rotate instead of shift
46
47 // ALU Controls
48 input [1:0]      mpc_alufunc_e;     // Logic Function 00-AND,01-OR,10-XOR,11-NOR
49 input            mpc_aluasrc_e;     // A src for ALU (0-edp_abus_e, 1-BBusOrImm)
50 input            mpc_alubsrc_e;     // B src for ALU (0-edp_abus_e, 1-BBusOrImm)
51 input            mpc_sellogic_m;    // Select Logic Function output
52 input            mpc_write_reg_to_tag_m;
```

```
53  input           mpc_read_tag_to_reg_m;
54
55  input           mpc_cmov_e;              // Conditional move instn
56  input           mpc_run_m;            // M-stage Run signal
57  input           mpc_run_ie;           // E-stage Run signal
58
59  // Shifter controls
60  input           mpc_addui_e;             // Add Upper Immediate
61  input           mpc_cnvt_e;              // Convert/Swap Operations: SEB/SEH/WSBH
62  input           mpc_cnvts_e;             // Convert signextension operations: SEB/SEH
63  input           mpc_cnvtsh_e;            // Convert signextension operations: SEH
64  input           mpc_swaph_e;             // Swap operations: WSBH
65  input           mpc_shright_e;           // Right shift (vs. Left)
66  input [4:0]     mpc_shamt_e;             // 1st stage shift amount
67  input           mpc_sharith_e;           // Arithmetic shift (vs. logical)
68  input           mpc_shvar_e;             // instn is shift variable
69  input           mpc_insext_e;            // INS/EXT instruction
70  input           mpc_ext_e;               // EXT instruction
71  input [4:0]     mpc_insext_size_e;       // INS/EXT filed size
72
73  input           mpc_shf_rot_cond_e;      // cregister condition for shf_rot_e->m
74  input           mpc_prealu_cond_e;       //  cregister condition for prealu_e->m
75  input           mpc_movci_e;
76
77  // Load Aligner controls
78  input           mpc_selcp_m;         // Select Cp0 read data or Cp2 To data
79  input           mpc_selcp2to_m;          // Select Cp2 To data
80  input           mpc_selcp2from_m;        // Select Cp2 To data
81  input           mpc_selcp2from_w;        // Select Cp2 From data
82  input           mpc_selcp1to_m;          // Select Cp1 To data
83  input           mpc_selcp1from_m;        // Select Cp1 To data
84  input           mpc_selcp1from_w;        // Select Cp1 From data
85  input           mpc_selcp0_m;        // Select Cp0 read data (MFC0 or SC)
86  input           mpc_updateldcp_m;    // Capture Load or CP0 data
87  input           mpc_dcba_w;              // Select all bytes from Dcache
88  input           mpc_signd_w;             // Use sign bit of byte D
89  input           mpc_signc_w;             // Use sign bit of byte C
90  input           mpc_signb_w;             // Use sign bit of byte B
91  input           mpc_signa_w;             // Use sign bit of byte A
92  input [1:0]     mpc_lda31_24sel_w;       // Mux select of ld_algn_w[31:24]
93  input [1:0]     mpc_lda23_16sel_w;       // Mux select of ld_algn_w[23:16]
94  input [1:0]     mpc_lda15_8sel_w;        // Mux select of ld_algn_w[15:8]
95  input [1:0]     mpc_lda7_0sel_w;         // Mux select of ld_algn_w[7:0]
96  input           mpc_cdsign_w;        // Sign extend to fill upper half
97  input           mpc_bsign_w;         // Sign extend to fill byte B
98
99  // AGEN Controls
100 input           mpc_subtract_e;      // Use adder to Subtract
101 input           mpc_signed_m;        // This is a signed operation
102
103
```

```
104 /* Outputs */
105 output [31:0]   edp_abus_e;          // Src A bus
106 output [1:0]    edp_abus_tag;       // Src A tag;
107 output [31:0]   edp_alu_m;          // ALU output for MTC0
108 output [31:0]   edp_bbus_e;         // Src B bus
109 output [1:0]    edp_bbus_tag;       // Src B tag;
110 output [31:0]   edp_dva_e;          // Data virtual address (E-stage)
111 output [31:29]  edp_dva_mapped_e;   // Data virtual address mapped (E-stage)
112 output [31:0]   edp_iva_p;          // Instn virtual address (preI-stage)
113 output [31:0]   edp_iva_i;          // Instn virtual address (I-stage)
114 output [31:0]   edp_epc_e;          // Exception Program Counter
115 output          edp_eisa_e;         // Exception ISA Mode
116 output [31:0]   edp_stdata_m;       // Store Data
117 output [1:0]    wr_data_tag;        // Store Data tag
118 output [31:0]   edp_wrdata_w;           // Register File write data
119 output [1:0]    edp_wrdata_w_tag;       // RegFile write data tag
120 output [31:0]   edp_ldcpdata_w;         // Load or MFC0 data
121 output [1:0]    rd_data_tag;            // Load Data tag
122 output [31:0]   edp_res_w;              // Store data for trace
123
124 output [31:0]   edp_cacheiva_p;   //replaces the edp_iva_p for memories
125 output [31:0]   edp_cacheiva_i;
126 output          edp_trapeq_m;        // Trap compare result
127 output          edp_cndeq_e;         // Branch compare result
```

# Παράρτημα Β΄

# Αξιολόγηση Απόδοσης

Σε αυτό το σημείο παραθέτουμε τους πίνακες που παρουσιάζουν τη χρήση των δομικών στοιχείων μιας fpga στις περιπτώσεις των δύο υλοποιήσεων, έτσι όπως αυτοί εξήχθησαν μετά τη διαδικασία του synthesis στο εργαλείο Vivado.

1. **Επεξεργαστής MIPS**

   Αρχικά παρουσιάζουμε κάποιους συγκεντρωτικούς πίνακες που αφορούν τον επεξεργαστή και τη χρήση των Slice LUTs, Registers, πολυπλεκτών και μνήμης, προτού γίνει οποιαδήποτε αλλαγή στον MIPS.

1. Slice Logic

| Site Type | Used | Fixed | Available | Util% |
|---|---|---|---|---|
| Slice LUTs | 9354 | 0 | 63400 | 14.75 |
| LUT as Logic | 9347 | 0 | 63400 | 14.74 |
| LUT as Memory | 7 | 0 | 19000 | 0.03 |
| LUT as Distributed RAM | 0 | 0 | | |
| LUT as Shift Register | 7 | 0 | | |
| Slice Registers | 7423 | 0 | 126800 | 5.85 |
| Register as Flip Flop | 7423 | 0 | 126800 | 5.85 |
| Register as Latch | 0 | 0 | 126800 | 0.00 |
| F7 Muxes | 320 | 0 | 31700 | 1.00 |
| F8 Muxes | 30 | 0 | 15850 | 0.18 |

1. Memory

| Site Type | Used | Fixed | Available | Util% |
|---|---|---|---|---|
| Block RAM Tile | 109.5 | 0 | 135 | 81.11 |
| RAMB36/FIFO* | 96 | 0 | 135 | 71.11 |
| RAMB36E1 only | 96 | | | |
| RAMB18 | 27 | 0 | 270 | 10.00 |
| RAMB18E1 only | 27 | | | |

2. **Επεξεργαστής MIPS + Μηχανισμός DIFT**

Έπειτα, παρουσιάζουμε τους αντίστοιχους πίνακες μετά την υλοποίηση της τεχνικής DIFT στον επεξεργαστή, παρατηρώντας την αυξημένη χρήση των LUTs, Registers, πολυπλεκτών και μνήμης σε σχέση με τον μη τροποποιημένο επεξεργαστή.

1. Slice Logic

| Site Type | Used | Fixed | Available | Util% |
|---|---|---|---|---|
| Slice LUTs | 9540 | 0 | 63400 | 15.04 |
| LUT as Logic | 9533 | 0 | 63400 | 15.03 |
| LUT as Memory | 7 | 0 | 19000 | 0.03 |
| LUT as Distributed RAM | 0 | 0 | | |
| LUT as Shift Register | 7 | 0 | | |
| Slice Registers | 7599 | 0 | 126800 | 5.99 |
| Register as Flip Flop | 7599 | 0 | 126800 | 5.99 |
| Register as Latch | 0 | 0 | 126800 | 0.00 |
| F7 Muxes | 408 | 0 | 31700 | 1.28 |
| F8 Muxes | 58 | 0 | 15850 | 0.36 |

Memory

| Site Type | Used | Fixed | Available | Util% |
|---|---|---|---|---|
| Block RAM Tile | 113.5 | 0 | 135 | 84.07 |
| RAMB36/FIFO* | 100 | 0 | 135 | 74.07 |
| RAMB36E1 only | 100 | | | |
| RAMB18 | 27 | 0 | 270 | 10.00 |
| RAMB18E1 only | 27 | | | |

# Βιβλιογραφία


[1] Hovav Shacham, Matthew Page, Ben Pfaff and Dan Boneh. *On the effectiveness of address-space randomization*. In Proceedings of the 11th ACM Conference on Computer and Communications Security, CCS 2004, Washington, DC, USA, 2004.

[2] The PaX project. http://pax.grsecurity.net.

[3] Ollie Whitehouse. *GS and ASLR in Windows Vista*. 2007.

[4] Crispin Cowan, Calton Pu, Dave Maier and Qian Zhang. *StackGuard: Automatic Adaptive Detection and Prevention of Buffer-overflow Attacks*. In Proceedings of the 7th USENIX Security Symposium, San Antonio, Texas, USA, 1998.

[5] Arash Baratloo and Navjot Singh. *Transparent Run-Time Defense Against Stack Smashing Attacks*. In Proceedings of the USENIX Annual Technical Conference, San Diego, California, USA, 2000.

[6] Crispin Cowan, Steve Beattie, Ryan Finnin Day, Calton Pu, Perry Wagle and Erik Walthinsen. *Protecting Systems from Stack Smashing Attacks with StackGuard*.

[7] Hiroaki Etoh. *Gcc Extension for Protecting Applications from Stack-smashing Attacks*. 2004.

[8] Crispin Cowan, Matt Barringer, Steve Beattie and Greg Kroah-Hartman. *FormatGuard: Automatic Protection From printf Format String Vulnerabilities*. In Proceedings of the 10th USENIX Security Symposium, Washington, DC, USA, 2001.

[9] Timothy Tsai and Navjot Singh. *Libsafe 2.0: Detection of Format String Vulnerability Exploits*. USA, 2001.

[10] OWASP - SQL Injection. https://www.owasp.org/index.php/SQL_Injection

[11] A. S. Yeole and B. B. Meshram. *Analysis of Different Technique for Detection of SQL Injection*. In Proceedings of the International Conference & Workshop on Emerging Trends in Technology, Mumbai, India, 2011.

[12] OWASP - Cross-site Scripting (XSS). https://www.owasp.org/index.php/Cross-site_Scripting_(XSS).

[13] OWASP - XSS Prevention Cheat Sheet. https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet.

[14] Security Enhanced Linux.
https://selinuxproject.org/page/Main_Page.

[15] Juan José Conti and Alejandro Russo. *A Taint Mode for Python via a Library*. In Proceedings of the 15th Nordic Conference on Information Security Technology for Applications, Finland, 2010.

[16] Anh Nguyen-Tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley and David Evans. *Automatically Hardening Web Applications Using Precise Tainting*. In Proceedings of the 20th IFIP International Conference on Information Security, Chiba, Japan, 2005

[17] Perl Programming Documentation.
http://perldoc.perl.org/perlsec.html.

[18] Andrew Hurst. *Analysis of Perl's Taint Mode*. 2004.

[19] Ruby Programming Documentation. http://ruby-doc.org.

[20] Package javax.mail. https://docs.oracle.com/javaee/7/api/.

[21] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler and Robert Morris. *Information Flow Control for Standard OS Abstractions*. In Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP), New York, NY, USA, 2007.

[22] G. Edward Suh, Jae W. Lee, David Zhang and Srinivas Devadas. *Secure Program Execution via Dynamic Information Flow Tracking*. In Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XI), Boston, MA, USA, 2004.

[23] Hari Kannan, Michael Dalton and Christos Koryzakis. *Decoupling Dynamic Information Flow Tracking with a Dedicated Coprocessor*. In Proceedings of the International Conference on Dependable Systems & Networks, Lisbon, Portugal, 2009.

[24] Shuo Chen, Jun Xu, Nithin Nakka and Ravishankar K. Iyer. *Defeating Memory Corruption Attacks via Pointer Taintedness Detection*. In Proceedings of the International Conference on Dependable Systems and Networks, Yokohama, Japan, 2005.

[25] Michael Dalton, Hari Kannan and Christos Kozyrakis. *Raksha: A Flexible Information Flow Architecture for Software Security*. In Proceedings of the 34th Annual International Symposium on Computer Architecture, San Diego, California, USA, 2007.

[26] Michael Dalton, Hari Kannan and Christos Kozyrakis. *Deconstructing Hardware Architectures for Security*. In Proceedings of the 5th Annual Workshop on Duplicating, Deconstructing, and Debunking (WDDD), Boston, MA, USA, 2006.

[27] S. Chen, B. Falsaf, Phillip B. Gibbons, Michael Kozuch, Todd C. Mowry, Radu Teodorescu, Anastassia Ailamaki, Limor Fix, Gregory R. Ganger and Steven W. Schlosser. *Logs and Lifeguards: Accelerating Dynamic Program Monitoring*. Technical Report IRP-TR-06-05, Intel Research, Pittsburgh, PA, 2006.

[28] Lap Chung Lam and Tzi-cker Chiueh. *A General Dynamic Information Flow Tracking Framework for Security Applications*. In Proceedings of the 22nd Annual Computer Security Applications Conference, Miami Beach, FL, USA, 2006.

[29] Jedidiah R. Crandall and Frederic T. Chong. *Minos: Control Data Attack Prevention Orthogonal to Memory Model*. In Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture, Portland, OR, USA, 2004.

[30] Daniel Y. Deng and G. Edward Suh. *High-performance Parallel Accelerator for Flexible and Efficient Run-time Monitoring*. In Proceedings of the 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Boston, MA, USA, 2012.

[31] Hari Kannan. *Ordering decoupled metadata accesses in multiprocessors*. In Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, New York, NY, USA, 2009.

[32] Vijay Nagarajan, Ho-Seop Kim, Youfeng Wu and Rajiv Gupta. *Dynamic Information Flow Tracking on Multicores*. In Proceedings of the 12th INTERACT, Salt Lake City, UT, 2008.

[33] microAptiv Processor Core. https://www.imgtec.com/mips/aptiv/microaptiv/.

[34] Loading a 32 bit Immediate. https://https://www.cs.umd.edu/class/sum2003/cmsc311/Notes/Mips/load32.html.

[35] Michael Dalton, Hari Kannan and Christos Kozyrakis *Real-World Buffer Overflow Protection for Userspace & Kernelspace*. In Proceedings of the 17th conference on Security symposium, San Jose, CA, USA, 2008.

[36] Michael Dalton, Hari Kannan and Christos Kozyrakis *Tainting is not pointless*. In ACM SIGOPS Operating Systems Review Volume 44 Issue 2, New York, NY, USA, 2010.