ΠΟΛΥΤΕΧΝΕΙΟ ΚΡΗΤΗΣ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ & ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Αδρομερής Δυναμική Ανάλυση Βιβλιοθηκών Λογισμικού

Συγγραφέας:
Γρηγόριος Ντουσάκης

Εξεταστική επιτροπή:
Αναπληρωτής Καθ. Μιχαήλ Γ. Λαγουδάκης
Αναπληρωτής Καθ. Σωτήριος Ιωαννίδης
Δρ. Νίκος Βασιλάκης

Οκτώβριος 2020

TECHNICAL UNIVERSITY OF CRETE
SCHOOL OF ELECTRICAL & COMPUTER ENGINEERING

DIPLOMA THESIS

Coarse-Grained Dynamic Analysis Of Software Libraries

Author:
Grigorios Ntousakis

Committee:
Associate Prof. Michael G. Lagoudakis
Associate Prof. Sotirios Ioannidis
Dr. Nikos Vasilakis

October 2020

Περίληψη

Η δυναμική ανάλυση προγραμμάτων είναι μία ευρέως διαδεδομένη τεχνική που χρησιμοποιείται για την εύρεση πληροφοριών σχετικά με την εκτέλεση του προγράμματος. Σε αυτή την διπλωματική εργασία παρουσιάζεται μία νέα μορφή δυναμικής ανάλυσης που στοχεύει σε σύγχρονες δυναμικές γλώσσες προγραμματισμού, όπως η Python, η Lua και η Javascript. Αυτή η μορφή αξιοποιεί τις δυνατότητες του συστήματος εισαγωγής βιβλιοθηκών και, πιο συγκεκριμένα, το γεγονός ότι οι βιβλιοθήκες εισάγονται με μορφή κειμένου. Στον πυρήνα της, πραγματοποιεί αποσύνθεση, μετασχηματισμό και ανασύνθεση των βιβλιοθηκών, διατηρώντας την αρχική τους λειτουργικότητα. Κατά τον μετασχηματισμό εισάγει κώδικα κατασκευασμένο από τον χρήστη, ειδικό για την εκάστοτε ανάλυση, στον πηγαίο κώδικα της βιβλιοθήκης. Αυτός ο κώδικας σε συνδυασμό με διάφορα περιτυλίγματα, και τον μηχανισμό παρεμβολής των γλωσσών δίνει την δυνατότητα να αναλύονται οι αλληλεπιδράσεις του προγράμματος σε επίπεδο βιβλιοθήκης χωρίς να απαιτείται μορφοποίηση του περιβάλλοντος εκτέλεσης. Αυτές οι δυνατότητες επιδεικνύονται με την χρήση της Lya, ενός συστήματος γραμμένου στην Javascript, που στοχεύει να πραγματοποιήσει δυναμική ανάλυση για την ίδια την Javascript. Τα αποτελέσματα δείχνουν ότι η Lya προσφέρει 2–3 τάξεις μεγέθους ταχύτερες αναλύσεις, σε σύγκριση με σύγχρονα συστήματα δυναμικής ανάλυσης. Δίνεται η δυνατότητα στις αναλύσεις να τρέχουν κατά την διάρκεια της παραγωγής, ανιχνεύοντας προβλήματα και συμπεριφορές μοναδικές κάτω από αυτές τις συνθήκες. Οι αναλύσεις προσφέρουν πληθώρα διαφορετικών περιπτώσεων χρήσης και γράφονται σε 100 γραμμές κώδικα κατά μέσο όρο.

Abstract

Dynamic program analysis is a technique for obtaining information about a program and its execution. In this thesis, we present a new type of dynamic analysis that targets modern dynamic languages such as Python, Lua and Javascript. This kind of dynamic analysis uses the capabilities of the module-import mechanism, and more specifically the fact that libraries are imported as text. At its core, it performs disassembly, transformation and recomposition of the imported libraries, maintaining their original functionality. During the transformation phase, it injects user-generated code, specific to each analysis, into the source code of the library. This imported code, combined with various context wraps and the use of proxies, make it possible to detect interactions at the library level without altering the runtime environment. We implement this approach in Lya, a system targeting the JavaScript ecosystem, written in Javascript. The results show that Lya offers 2–3 orders of magnitude faster analyses compared to conventional dynamic analysis systems. Analyses can be enabled to run during production, detecting problems and behaviors unique to these conditions. Different analyses offer a variety of different use cases and are written in 100 lines of code on average.

Ευχαριστίες

Θα ήθελα καταρχήν να ευχαριστήσω τον Δρ. Νίκο Βασιλάκη, για την διαρκή υποστήριξη, ενθάρρυνση και βοήθεια σε όλους τους τομείς αυτής της διπλωματικής εργασίας. Επιλέγοντας να παραστώ στην ομιλία του στο Πολυτεχνείο Κρήτης ήταν από τις κομβικότερες αποφάσεις που έχω λάβει στην ζωή μου. Έπειτα, θα ήθελα να ευχαριστήσω τον Δρ. Μιχαήλ Γ. Λαγουδάκη, για τις πολύτιμες συμβουλές που μου παρείχε, και που συμφώνησε να είναι επιβλέποντας καθηγητής σε αυτήν την εργασία. Επιπλέον, θα ήθελα να ευχαριστήσω τον Δρ. Σωτήρη Ιωαννίδη, για την συμμετοχή του στην εξεταστική επιτροπή και για την αξιολόγηση της εργασίας μου. Θα ήθελα να ευχαριστήσω όσους στάθηκαν δίπλα μου σε όλη την διάρκεια των φοιτητικών χρόνων. Ιδιαίτερα τους γονείς μου, Στέφανο και Άρτεμη, και τον μικρό μου αδελφό, Γιώργο. Δεν θα μπορούσα να πραγματοποιήσω τίποτα χωρίς την βοήθειά σας και την υποστήριξή σας. Σας ευχαριστώ.

Περιεχόμενα

Κατάλογος Σχημάτων

Κατάλογος Κώδικα

Κεφάλαιο 1

Εισαγωγή

Τα τελευταία χρόνια, το λογισμικό εισέρχεται όλο και περισσότερο σε όλες τις πτυχές της ζωής μας. Μικροσυσκευές μέχρι και πολύπλοκες αποστολές στο διάστημα, βασίζονται σε διαφορετικές υλοποιήσεις εφαρμογών. Οι προγραμματιστές προσπαθούν να βελτιστοποιήσουν όσο το δυνατόν περισσότερο την αποδοτικότητα και την ποιότητα του κώδικα που παράγουν, προσπαθώντας να αποδώσουν χαρακτηριστικά ασφάλειας και ταχύτητας. Αυτό το φαινόμενο οδηγεί σε όλο και μεγαλύτερη επαναχρησιμοποίηση του κώδικα. Πιο συγκεκριμένα, οι προγραμματιστές επιλέγουν να χρησιμοποιούν, μεταξύ άλλων, κώδικα που βρίσκουν έτοιμο σε διάφορες διαδικτυακές σελίδες ή βιβλιοθήκες τρίτων. Έτσι καταφέρνουν να προγραμματίζουν *γρηγορότερα*, *ευκολότερα* και *αποδοτικότερα*.

1.1 Ανάλυση προβλήματος

Η χρήση διαφόρων βιβλιοθηκών στις γλώσσες προγραμματισμού έχει πολλαπλά οφέλη. Πιο συγκεκριμένα, οι προγραμματιστές μπορούν να επαναχρησιμοποιούν τον ίδιο κώδικα κερδίζοντας έτσι χρόνο, τον οποίο μπορούν να αξιοποιήσουν σε άλλες πτυχές της εκάστοτε εργασίας. Παραδείγματα τέτοιων πτυχών είναι η δημιουργία δοκιμασιών (*tests*) για να ελέγξουν την ποιότητα του κώδικα ή να κάνουν διάφορες ανακατασκευές (*refactoring*) του κώδικα για να τον βελτιώσουν.

Είναι λογικό ότι η χρήση κώδικα που έχει γραφτεί από τρίτους άγνωστους προγραμματιστές δημιουργεί πολλαπλά ρίσκα [48]. Μπορούμε να θεωρήσουμε τον κώδικα αυτό ως ένα "μαύρο" κουτί στο οποίο δίνουμε κάποια στοιχεία και πραγματοποιεί κάποιες λειτουργίες. Είναι σχετικά εύκολο να παρατηρήσουμε τα δεδομένα που προσφέρονται και τα αποτελέσματα που λαμβάνονται, χάνοντας όμως όλη την ενδιάμεση διαδικασία που συμβαίνει στο εσωτερικό αυτών των κουτιών. Αυτό έχει ως αποτέλεσμα, να μην έχουμε γνώση πως λειτουργούν εσωτερικά αυτές οι βιβλιοθήκες τρίτων (*third-party libraries*), ποια στοιχεία του λειτουργικού συστήματος επηρεάζουν ή ακόμα και γιατί ξαφνικά μια βιβλιοθήκη λογισμικού που λειτουργούσε ταχύτατα ξεκίνησε να προσδίδει σημαντική καθυστέρηση στο πρόγραμμά μας.

Σε αυτή την διπλωματική εργασία προσπαθούμε να επιτρέψουμε την εύρεση και ανάλυση προβλημάτων που σχετίζονται με την *ορθότητα/αξιοπιστία* των προγραμμάτων, την *ασφάλεια/προστασία* των δεδομένων μας και την *απόδοση* του εκάστοτε προγράμματος [38].

1.2 Προηγούμενες λύσεις

Οι πιο διαδεδομένες λύσεις για το πρόβλημα που αναφέρθηκε στην προηγούμενη ενότητα είναι η χρήση στατικών και δυναμικών αναλύσεων [4]. Στατική ανάλυση ονομάζουμε τον τρόπο ανάλυσης του πηγαίου κώδικα που βγάζει συμπεράσματα, για διαφορετικά μέρη του προγράμματος, χωρίς να χρειάζεται να τρέξει το πρόγραμμα. Αυτού του είδους η ανάλυση δεν επαρκεί, καθώς οι γλώσσες που στοχεύουμε να αναλύσουμε σε αυτή την διπλωματική εργασία είναι δυναμικές.

Δυναμική ανάλυση Στις δυναμικές γλώσσες προγραμματισμού, όπως `Javascript`, `Ruby` ή και `Python`, είναι ευρέως διαδεδομένη η χρήση των δυναμικών αναλύσεων. Δυναμική ανάλυση ονομάζουμε την τεχνική κατά την οποία ελέγχουμε, καταλαβαίνουμε και πιθανώς επεμβαίνουμε στην συμπεριφορά του προγράμματος *κατά την εκτέλεσή του* [4]. Δεν πραγματοποιείται ανάλυση του πηγαίου κώδικα ως κείμενο, αλλά εκτελείται με δεδομένα στην είσοδό του, χρησιμοποιώντας διάφορες τεχνικές που προσφέρουν ενδιαφέροντα αποτελέσματα. Η δυναμική ανάλυση προσφέρει πιο λεπτομερή ανάλυση από την στατική ανάλυση και μπορεί να ανιχνεύσει προβλήματα που εμφανίζονται μόνο κατά την εκτέλεση του κώδικα, με σημαντικό κόστος όμως σε ταχύτητα. Συνήθως γράφεται σε διαφορετική γλώσσα προγραμματισμού από την αρχική γλώσσα του προγράμματος.

Προβλήματα δυναμικής ανάλυσης Οι τρέχοντες τρόποι δυναμικής ανάλυσης παρουσιάζουν διάφορα προβλήματα. Καθώς γράφονται σε διαφορετική γλώσσα προγραμματισμού από αυτήν που αναλύουν, αυτό έχει ως αποτέλεσμα ο χρήστης να απαιτείται να κατανοεί και να χρησιμοποιεί παραπάνω γλώσσες, αυξάνοντας την πολυπλοκότητα. Ταυτόχρονα, για να μπορέσουν να τρέξουν τις αναλύσεις καθίσταται αναγκαίο να εικονικοποιείται ολόκληρη η εκτέλεση. Αυτό οδηγεί την εκάστοτε ανάλυση σε σημαντική καθυστέρηση, που σημαίνει ότι οι αναλύσεις μπορούν και τρέχουν κυρίως εκτός παραγωγής αλλά πιθανώς κατά την διαδικασία της ανάπτυξης. Πολλά προβλήματα όμως παρουσιάζονται μόνο κατά την διάρκεια της παραγωγής, κάτι που καθιστά την κλασσική δυναμική ανάλυση αδύναμη να τα ανιχνεύσει.

1.3 Αδρομερής προσέγγιση

Αυτή η διπλωματική εργασία προτείνει ένα νέο τρόπο δυναμικής ανάλυσης. Αυτός ο τρόπος ονομάζεται αδρομερής ανάλυση και λειτουργεί στα όρια των βιβλιοθηκών, αναλύοντας τις αλληλεπιδράσεις τους [44, 43]. Ο αδρομερής τρόπος ανάλυσης στοχεύει στις σύγχρονες γλώσσες προγραμματισμού δυναμικής μορφής, όπως η `Lua`, η `Python` και η `Javascipt`. Χαμηλώνοντας την λεπτομέρεια της ανάλυσης έχει ως αποτέλεσμα το κόστος να γίνεται δύο–τρεις τάξεις μεγέθους χαμηλότερο συγκριτικά με τεχνολογίες αιχμής. Ο στόχος είναι να μπορούν να τρέχουν οι αναλύσεις παράλληλα με την εκτέλεση του προγράμματος στην παραγωγή, χωρίς να χρειάζεται να απενεργοποιούνται, προσφέροντας έτσι πληροφορίες με συνεχή ρυθμό. Το εργαλείο αδρομερούς ανάλυσης που δημιουργήθηκε για να επιδείξουμε τα οφέλη αυτού του είδους ανάλυσης, το ονομάσαμε Lya και υλοποιήθηκε στην γλώσσα προγραμματισμού `Javacript`, την ίδια γλώσσα που επιθυμούμε να αναλύσουμε.

Υλοποιήθηκαν τρεις αναλύσεις —μία ανάλυση ασφαλείας, μία ανάλυση απόδοσης και τέλος μία ανάλυση εξαγωγής τύπων. Η κάθε ανάλυση απαιτεί περίπου 100 γραμμές κώδικα για να υλοποιηθεί. Παρατηρούμε ότι παρουσιάζει χαμηλή καθυστέρηση στις μικρές βιβλιοθήκες που την εφαρμόζουμε, αποδεικνύοντας την χρησιμότητά της ακόμα και στην παραγωγή. Συγκριτικά με το εργαλείο ανάλυσης `Jalangi` η Lya παρουσιάζει 2–3 φορές βελτίωση απόδοσης, όταν τρέχει στην σουίτα δοκιμών του `Jalangi`.

1.4 Επισκόπηση διπλωματικής εργασίας

Σε όλη την διάρκεια του κείμενου οι όροι πακέτο και βιβλιοθήκη χρησιμοποιούνται εναλλακτικά και έχουν την ίδια έννοια. Τα περιεχόμενα των κεφαλαίων της διπλωματικής εργασίας έχουν ως εξής:

- **Κεφάλαιο 2**: Σε αυτό το κεφάλαιο θα παρουσιαστεί η σχετική δουλειά που έχει πραγματοποιηθεί τα τελευταία χρόνια πάνω στο αντικείμενο που πραγματεύεται αυτή η διπλωματική εργασία. Θα παρουσιαστούν οι μετασχηματισμοί, οι αναλύσεις και η έννοια του *θεματοστρεφή προγραμματισμού*.

- **Κεφάλαιο 3**: Σε αυτό το κεφάλαιο θα παρουσιαστεί ένα παράδειγμα χρήσης του εργαλείου αδρομερούς ανάλυσης που δημιουργήθηκε. Στην αρχή παρουσιάζεται το προγραμματιστικό πρόβλημα που τέθηκε προς επίλυση, αναλύεται, επιτελείται μία αναδρομή στα συστήματα βιβλιοθηκών ανά την ιστορία, και τέλος παρουσιάζεται η εφαρμογή της Lya στο παράδειγμα.

- **Κεφάλαιο 4**: Σε αυτό το κεφάλαιο θα παρουσιαστούν τα διαφορετικά στάδια σχεδίασης της αδρομερούς ανάλυσης. Πιο συγκεκριμένα, θα παρουσιαστούν τα στάδια της αποσύνθεσης, των μετασχηματισμών ανάλυσης και τέλος της ανασύνθεσής του, ώστε να διατηρηθεί η αρχική του λειτουργία.

- **Κεφάλαιο 5**: Σε αυτό το κεφάλαιο θα παρουσιαστούν μερικές από τις αναλύσεις που έχουν υλοποιηθεί για να αναδειχθούν οι λειτουργίες της αδρομερούς ανάλυσης. Θα πραγματοποιηθεί μία διερεύνηση σε μερικές κατηγορίες χρήσιμων αναλύσεων που έχουν δημιουργηθεί από το μηδέν με την χρήση του προσφερόμενου εργαλείου μας. Θα δοθούν διαφορετικά παραδείγματα χρήσης τους και θα παρουσιαστούν τα *άγκιστρα (hooks)* που μας επιτρέπουν υλοποίηση των αναλύσεων.

- **Κεφάλαιο 6**: Σε αυτό το κεφάλαιο θα παρουσιαστεί η χρήση της Lya τόσο προγραμματιστικά, όσο και με την χρήση του εργαλείου της γραμμής εντολών, και δίνονται παραδείγματα για κάθε μία από τις δύο περιπτώσεις.

- **Κεφάλαιο 7**: Σε αυτό το κεφάλαιο θα παρουσιαστούν τα αποτελέσματα από τις μετρήσεις που πραγματοποιήθηκαν και απαντώνται βασικά ερωτήματα που δείχνουν τα κέρδη της χρήσης αδρομερούς ανάλυσης.

- **Κεφάλαιο 8**: Σε αυτό το κεφάλαιο θα παρουσιαστούν γενικά συμπεράσματα της διπλωματικής εργασίας, πιθανή μεταφορά σε άλλες γλώσσες προγραμματισμού, η πιθανή μελλοντική δουλειά που μπορεί να πραγματοποιηθεί πάνω στο σύστημα και ο ευρύτερος αντίκτυπός της.

- **Παράρτημα 1**: Σε αυτό το παράρτημα θα παρουσιαστεί ένα μικρός οδηγός προκειμένου να τρέξει ο χρήστης το κλασσικό παράδειγμα `Hello world`, στην γλώσσα προγραμματισμού `Javascript`.

- **Παράρτημα 2**: Σε αυτό το παράρτημα θα παρουσιαστεί ένας οδηγός με όλα τα απαραίτητα βήματα εγκατάστασης και χρήσης της Lya.

Κεφάλαιο 2

Σχετική δουλειά

Σε γενικές γραμμές οι διαφορές με προηγούμενες υλοποιήσεις δυναμικών αναλύσεων είναι ότι η Lya είναι *ταχύτερη*, *διατηρεί* την αρχική λειτουργικότητα, και δίνει την δυνατότητα να *χρησιμοποιείται κατά την παραγωγή*. Είναι γραμμένη στην *ίδια* γλώσσα προγραμματισμού που στοχεύει να αναλύσει —την `Javascript`. Αξιοποιεί τα δυναμικά στοιχεία της *ίδιας* της `Javascript`, για να μπορέσει να αναλύσει τις βιβλιοθήκες που χρησιμοποιούνται, *χωρίς* να χρειάζεται να επέμβει στο περιβάλλον που τρέχει το πρόγραμμα.

2.1 Εργαλεία δυναμικής ανάλυσης

Τα τελευταία χρόνια δημιουργούνται όλο και περισσότερα εργαλεία δυναμικής ανάλυσης. Ιδιαίτερα για την γλώσσα προγραμματισμού `Javascript`, που τα τελευταία χρόνια έχει καθιερωθεί ως η πιο δημοφιλής γλώσσα προγραμματισμού, έχουν υλοποιηθεί πολλαπλά εργαλεία [7, 17, 39, 25, 34]. Αυτά τα εργαλεία συνήθως επιτρέπουν αρκετά πολύπλοκες και στοχευμένες αναλύσεις, διατηρώντας πληροφορία ακόμα και για στοιχεία της γλώσσας προγραμματισμού, όπως το `if, while, break`. Αυτό έρχεται με σημαντικό κόστος στην ταχύτητα και οδηγεί σε αύξηση της πολυπλοκότητας των αναλύσεων που γράφονται. Άρα ο στόχος αυτών των εργαλείων είναι διαφορετικός συγκριτικά με της Lya που στοχεύει σε γρήγορες αναλύσεις, χρήσιμες στην ώρα της παραγωγής.

Το NodeProf [39] για παράδειγμα προσφέρει δυναμική ανάλυση που βασίζεται στην χρήση μεταπτροπής αφηρημένων συντακτικών δέντρων (`Abstract Syntax Trees, AST`) υλοποίησης, για να εισάγει τον κώδικα της ανάλυσης. Τα `AST` χρησιμοποιούνται εσωτερικά από τους μεταγλωττιστές και δίνουν μία δομημένη, δενδρώδη αναπαράσταση του αναλυμένου πηγαίου κώδικα, που χρησιμοποιείται συνήθως για σημασιολογική ανάλυση και δημιουργία κώδικα [26, 14]. Παρόλο που προσφέρει πιο ισχυρές αναλύσεις σε σύγκριση με την Lya, δουλεύει με την χρήση των εργαλείων Graal [45] και Truffle [46]. Το `Truffle` είναι συμβατό αλλά και αρκετά διαφορετικό από το `Node.js`, με αποτέλεσμα να μην μπορεί να υποστηρίξει χωρίς τροποποιήσεις το περιβάλλον της γλώσσας.

Η γλώσσα προγραμματισμού Javascript είναι συγγενική με την `WebAssemply`. Η `WebAssemply` είναι ένα υποσύνολο της Javascript που χρησιμοποιείται ως πρότυπο για την ανάπτυξη εφαρμογών σε Javascript. Η πρώτη δυναμική ανάλυση που υλοποιήθηκε για την `WebAssemply`, το Wasabi [20] έχει στόχους που είναι κοντινοί με την Lya—δηλαδή στοχεύει σε εύκολη σε υλοποίηση της ανάλυσης και έλεγχο του κώδικα παρά την μορφοποίησή του. Αντίθετα από την Lya όμως, αντιμετωπίζει σημαντικό κόστος σε χρονικό επίπεδο κατά την χρήση.

Υπάρχουν περιβάλλοντα δυναμικής ανάλυσης [22, 27, 24, 13] που τυλίγουν βασικά στοιχεία σταδιακά και πριν την εκτέλεση, σε γενικές γραμμές κοντά στην φιλοσοφία αδρομερούς ανάλυσης.

Διαφοροποιούνται όμως από το γεγονός ότι λειτουργούν σε χαμηλότερο επίπεδο από αυτό που λειτουργεί η Lya, είναι πολύ πιο λεπτομερή και νοσογόνα σε χρονικό επίπεδο. Επιπλέον συνήθως δεν είναι διαθέσιμα για σύγχρονες γλώσσες προγραμματισμού που στοχεύουν στην αναγνώριση και ανάλυση των βιβλιοθηκών.

Μία άλλη βασική ιδέα που ενέπνευσε την υλοποίηση της Lya, είναι η σύνθεση και η αποσύνθεση προγραμμάτων [2, 35] ένας τομέας που τεχνικά στοχεύει περισσότερο στην σύνθεση προγράμματος και στην αυτόματη επίλυση προβλημάτων.Το `PFR` σπάει τα προγράμματα σε μικρότερα κομμάτια με σκοπό την εναλλαγή κομματιών από τρίτους δοτές. Αντίθετα, η Lya λειτουργεί σε ένα κύριο πρόγραμμα, διατηρώντας την λειτουργικότητα και την αποσύνθεση του κώδικα στο ελάχιστο, αξιοποιώντας τα ήδη υπάρχοντα πακέτα μέσα στις ίδιες τις βιβλιοθήκες και την γλώσσα προγραμματισμού.

2.2 Θεματοστρεφής προγραμματισμός

Ο θεματοστρεφής προγραμματισμός (`aspect-oriented programming, AOP`) [18], είναι μία κατηγορία/είδος των γλωσσών προγραμματισμού που έχει ως στόχο να αυξήσει την αρθρωτότητα των προγραμμάτων, δίνοντας την δυνατότητα να διαχωριστούν με βάση διαφορετικά στοιχεία από αυτά που αφορούν την απλή εκτέλεση του προγράμματος [18, 15]. Αυτό επιτυγχάνεται προσθέτοντας λογική πάνω στον ήδη υπάρχων κώδικα, χωρίς όμως να επεξεργάζονται τον κώδικα τον ίδιο, αλλά προσδιορίζοντας ποιος κώδικας έχει μορφοποιηθεί δίνοντας κάποιες συγκεκριμένες πληροφορίες. Ένα τέτοιο παράδειγμα θα ήταν, *σημείωσε όλες τις κλήσεις συναρτήσεων, όταν το όνομα της συνάρτησης ξεκινάει με* `set`. Με αυτό τον τρόπο μπορούμε να εισάγουμε λογική που είναι χρήσιμη για το πρόγραμμα, χωρίς όμως να προσθέτουμε περιττές γραμμές στον κύριο κώδικα και να αυξάνουμε την πολυπλοκότητα.

Η λογική αυτή συνήθως εισάγεται με την χρήση πακέτων που κάνουν επέκταση των γλωσσών προγραμματισμού (όπως `AspectJ` και `AspectC++`) και/ή εφαρμόζοντας μορφοποιήσεις στην αρχική γλώσσα προγραμματισμού ή στο περιβάλλον που εκτελείται.

Η Lya αντιθέτως χρησιμοποιεί το ήδη υπάρχον σύστημα για δυναμικό φόρτωμα και τις δυνατότητες μετά-προγραμματισμού των ίδιων των σύγχρονων δυναμικών γλωσσών προγραμματισμού και έτσι επιτυγχάνει να εκτελείται χωρίς να χρειάζεται κάποια μορφοποίηση του περιβάλλοντος ή της γλώσσας που χρησιμοποιείται.

Κεφάλαιο 3

Παράδειγμα και βασικές γνώσεις

Παρακάτω βλέπουμε ένα παράδειγμα εφαρμογής αδρομερούς ανάλυσης πάνω σε απλό πρόγραμμα που χρησιμοποιεί δύο βιβλιοθήκες τρίτων.

3.1 Βασικές γνώσεις

Ο αρθρωτός προγραμματισμός (*Modular Programming*) είναι μία τεχνική προγραμματισμού που δίνει έμφαση στον διαχωρισμό της λειτουργικότητας των προγραμμάτων σε βιβλιοθήκες. Οι βιβλιοθήκες είναι διαφορετικά, ανταλλάξιμα κομμάτια που το καθένα περιέχει κώδικα απαραίτητο για να εκτελεστεί μόνο μία πτυχή της επιθυμητής λειτουργικότητας. Ένας από τους στόχους του αρθρωτού προγραμματισμού είναι μέσω των διαφορετικών κομματιών κώδικα να επιτευχθεί επαναχρησιμοποίηση λειτουργικότητας. Αυτό σημαίνει ότι υπάρχει η επιθυμία να μπορεί να χρησιμοποιηθεί ξανά στο μέλλον ο εκάστοτε κώδικας που γράφεται. Αυτή η λειτουργικότητα συνήθως εντάσσεται σε δύο κατηγορίες:

- Είτε υπάρχει ενσωματωμένη με την εκάστοτε γλώσσα προγραμματισμού, πιθανόν τυλίγοντας στοιχεία του λειτουργικού συστήματος, όπως το σύστημα αρχείων, επιτυγχάνοντας έτσι λειτουργία ανεξαρτήτως λειτουργικού συστήματος και διατηρώντας τις συμβάσεις της εκάστοτε γλώσσας.
- Είτε παρέχονται ως επιπλέον λειτουργικότητα από άλλους προγραμματιστές που γράφουν κώδικα και που ενδέχεται να είναι χρήσιμος για το ευρύ κοινό.

Από την πλευρά του προγραμματιστή, κάνοντας εισαγωγή κάποιας βιβλιοθήκης δίνεται η δυνατότητα να αξιοποιηθεί ο κώδικάς της, δεσμεύοντας ένα όνομα που θα αναφέρεται στην εν λόγω βιβλιοθήκη. Έτσι, κάθε φορά που ο προγραμματιστής θέλει να καλέσει τον κώδικα αυτό θα μπορεί να κάνει αναφορά στο όνομα που δεσμεύτηκε.

Από την πλευρά του συντηρητή του πακέτου, για να μπορεί να επιτύχει αυτό το μοίρασμα κώδικα πρέπει να προγραμματίσει κάποια μορφή εξαγωγής του. Θα διαλέξει τις τιμές και τις συναρτήσεις που επιθυμεί να γίνουν διαθέσιμες και θα τις εξάγει. Ταυτόχρονα να μπορεί εισάγει και άλλες βιβλιοθήκες, γραμμένες ακόμα και σε άλλες γλώσσες προγραμματισμού, αυξάνοντας την λειτουργικότητα του πακέτου του, πιθανώς προκαλώντας όμως διάφορες παρενέργειες τόσο σε επίπεδο συστήματος όσο και σε επίπεδο προγράμματος.

Φόρτωση Για να γίνει εισαγωγή κάποιου πακέτου σε γλώσσα/περιβάλλον γλώσσας, όπως η `Javascript`, πρέπει να γίνουν διάφορα βήματα. Πρώτα το σύστημα πρέπει να βρει την βιβλιοθήκη

στο σύστημα του χρήστη. Αμέσως μετά πρέπει να την διαβάσει και να την συνδέσει με τα τοπικά ονόματα που εμφανίζονται στο πλαίσιο (`context`), όπως για παράδειγμα `print, eval, process`. Έπειτα, πρέπει να ερμηνεύσει τον κώδικα χρησιμοποιώντας τον διερμηνέα της γλώσσας προγραμματισμού. Αυτό μπορεί να έχει διάφορες παρενέργειες, καθώς και πιθανή έξοδο του αρχικού προγράμματος λόγω κάποιου προβλήματος. Τέλος, ο κώδικας γίνεται διαθέσιμος στο κύριο πρόγραμμα για να χρησιμοποιηθεί με βάση το όνομα που έχει επιλεγεί από τον προγραμματιστή.

Μερικές φορές για να μειωθεί ο χρόνος και να διατηρηθεί η συνέχεια, η γλώσσα προγραμματισμού επιλέγει να προσπαθήσει να διαβάσει το πακέτο της βιβλιοθήκης από ενδιάμεση δομή (*cache*) που διατηρείται στην μνήμη, αν έχει διαβαστεί στο παρελθόν το πακέτο, σε αυτή την εκτέλεση. Αυτό έχει ως αποτέλεσμα πολλές φορές να πραγματοποιείται μια κυκλική εξάρτηση σε επίπεδο εισαγωγής της βιβλιοθήκης. Μία άλλη επιπλοκή που μπορεί να παρουσιαστεί είναι, εφόσον πλέον επιτρέπεται αρκετά συχνά στις σύγχρονες γλώσσες προγραμματισμού, να υπάρχουν πολλές διαφορετικές εκδόσεις της βιβλιοθήκης στο ίδιο πρόγραμμα, για να μην παρουσιαστεί μία μοναδική επιλογή. Αυτό το φαινόμενο ονομάζεται *κόλαση εξαρτήσεων* [40]. Ως αποτέλεσμα μία εισαγωγή κάποιου πακέτου, μπορεί να μην αναφέρεται πάντα στην ίδια έκδοση ή δύο διαφορετικά πακέτα να επιστρέφουν το ίδιο στοιχείο από την προσωρινή μνήμη. Οι παραπάνω δυνατότητες, κάνουν την δυναμική ανάλυση πολύπλοκη. Σε επόμενα κεφάλαια θα εξηγηθεί πως το εργαλείο αδρομερούς ανάλυσης που δημιουργήθηκε διαχειρίζεται και αντιμετωπίζει αυτά τα θέματα.

3.2 Κώδικας παραδείγματος

Έστω ότι ένας χρήστης θέλει να δημιουργήσει ένα πρόγραμμα στη γλώσσα προγραμματισμού *Javascript*. Το πρόγραμμα αυτό λαμβάνει δύο πίνακες και εμφανίζει το πρώτο στοιχείο από τον πρώτο πίνακα και το τελευταίο στοιχείο από τον δεύτερο. Οι τρόποι που μπορεί να διαπεραιώσει αυτό το προγραμματιστικό πρόβλημα είναι πολλοί. Θα μπορούσε να γράψει τον κώδικα που απαιτείται από το μηδέν, με αρκετό χρονικό κόστος για τον ίδιο, ή και να χρησιμοποιήσει κάποιες έτοιμες βιβλιοθήκες που υπάρχουν στο οικοσύστημα της *Javascript*, μειώνοντας σημαντικά την πολυπλοκότητα.

Κώδικας 3.1: Κεντρικό Πρόγραμμα

```
const first = require('array-first');
const last = require('array-last');

const myArray = [1, 3, 5, 6];
const smallArray = [1 , 2];

console.log(first(smallArray));
console.log(last(myArray));
```

Έστω ότι διαλέγει να χρησιμοποιήσει βιβλιοθήκες. Με ελάχιστη προσπάθεια μπορεί να βρει ότι μερικές από τις βιβλιοθήκες που πραγματοποιούν τις λειτουργίες που επιθυμεί είναι η *array-first* [5] και η *array-last* [31] αντίστοιχα. Έτσι, απλά εισάγοντας αυτά τα πακέτα στο πρόγραμμα που δημιουργεί, καταφέρνει με μικρό αριθμό γραμμών κώδικα (*6 γραμμές*) να επιλύσει το πρόβλημα του.

Κώδικας 3.2: Βιβλιοθήκη Array-First

```
var isNumber = require('is-number');
var slice = require('array-slice');

module.exports = function arrayFirst(arr, num) {
    if (!Array.isArray(arr)) {
        throw new Error('Error');
    }

    if (arr.length === 0) {
        return null;
    }

    var first = slice(arr, 0, isNumber(num) ? +num : 1);
    if (+num === 1 || num == null) {
      return first[0];
    }
    return first;
};
```

Όμως στην πραγματικότητα ο αριθμός των γραμμών του προγράμματος είναι πολύ μεγαλύτερος από αυτόν που φαίνεται. Η κάθε μία από τις βιβλιοθήκες που εισήχθησαν εγκαθιστά και φορτώνει και άλλες βιβλιοθήκες. Πέρα από τον κώδικά τους συμπεριλαμβάνουν και τα πρόγραμμα ελέγχου τους, τα αρχεία κειμένου τους και οτιδήποτε άλλο έχει συμπεριλάβει ο συντηρητής του εκάστοτε πακέτου. Πιο συγκεκριμένα, για αυτά τα δύο πακέτα κατέβηκαν συνολικά *299 γραμμές* `Javascript` κώδικα (πηγαίος κώδικας, προγράμματα ελέγχου, κτλ) μέσα στο δέντρο εξάρτησης. Οπότε καταλαβαίνουμε ότι είναι χρήσιμο να βρεθεί ένας αποδοτικός τρόπος ο προγραμματιστής να λαμβάνει χρήσιμες αναλύσεις πάνω σε αυτές τις βιβλιοθήκες και τις εξαρτήσεις τους.

Κώδικας 3.3: Βιβλιοθήκη Array-Last

```
var isNumber = require('is-number');

module.exports = function last(arr, n) {
    if (!Array.isArray(arr)) {
        throw new Error('Error');
    }

    var len = arr.length;
    if (len === 0) {
        return null;
    }

    n = isNumber(n) ? +n : 1;
    if (n === 1) {
        return arr[len - 1];
    }

    var res = new Array(n);
    while (n--) {
```

```
        res[n] = arr[--len];
    }

    return res;
};
```

3.3 Εφαρμογή αδρομερούς ανάλυσης

Για να μπορέσουμε να αναλύσουμε την λειτουργία των παραπάνω βιβλιοθηκών θα χρησιμοποιήσουμε αδρομερή ανάλυση μέσω της Lya. Πιο συγκεκριμένα θα εφαρμόσουμε μία μορφή *επίτρεψε/απόρριψε (allow/deny)* αδρομερούς ανάλυσης ασφαλείας. Ο στόχος είναι να δούμε ποιες βιβλιοθήκες εισήχθησαν, χρησιμοποιήθηκαν και με πιο τρόπο. Ο κώδικας που απαιτείται να γραφτεί για να εφαρμοστεί η ανάλυση που αναφέραμε είναι ο παρακάτω:

Κώδικας 3.4: Κώδικας χρήσης ανάλυσης ασφαλείας

```
let lya = require("@andromeda/lya");
let conf = {
  save: require("path").join(__dirname, "dynamic.json"),
  analysis: lya.preset.ON_OFF,
};
lya.configRequire(require, conf);
require("./main.js");
```

Βλέπουμε ότι ο παραπάνω κώδικας πραγματοποιεί τις εξής λειτουργίες:

- Αναλαμβάνει την εισαγωγή του εργαλείου (`require('lya.js')`).
- Αναλαμβάνει τον προσδιορισμό της τοποθεσίας στο σύστημα του χρήστη που θα αποθηκευτεί το αποτέλεσμα της δυναμικής ανάλυσης (`require("path").join(dirname, "dynamic.json")`).
- Αναλαμβάνει τον προσδιορισμό του ονόματος του αρχείου αποθήκευσης (`dynamic.json`).
- Αναλαμβάνει τον προσδιορισμό του είδους της ανάλυσης που θα εφαρμοστεί (`On-Off`).

Τρέχοντας το πρόγραμμα του παραδείγματος σε συνδυασμό με την αδρομερή ανάλυση, επιστρέφεται το εξής αποτέλεσμα:

Κώδικας 3.5: Αποτέλεσμα Αδρομερούς Ανάλυσης Ασφαλείας

```
{
  "Path/to/main.js": {
    "require": true,
    "require('array-first')": true,
    "require('array-last')": true,
    "console": true,
    "console.log": true,
    "require('array-last').last": true
  },
  "Path/to/array-first/index.js": {
    "require": true,
    "require('is-number')": true,
```

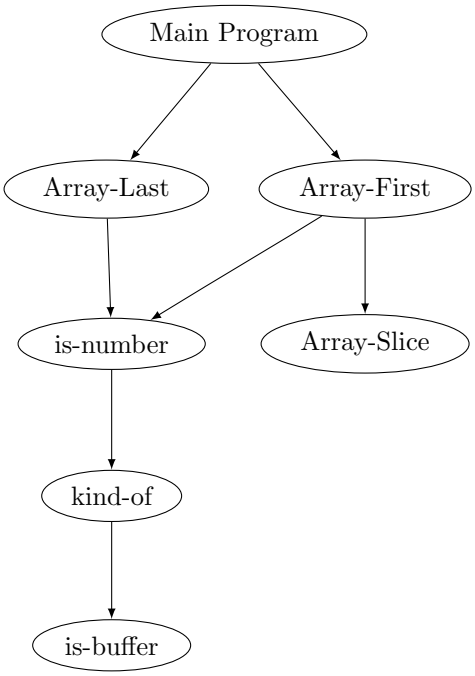
```
    "require('is-number').isNumber": true,
    "require('array-slice')": true,
    "require('array-slice').slice": true,
    "module": true,
    "module.exports": true,
    "Array": true
  },
  "Path/to/is-number/index.js": {
    "require": true,
    "require('kind-of')": true,
    "require('kind-of').kindOf": true,
    "module": true,
    "module.exports": true
  },
  "Path/to/kind-of/index.js": {
    "require": true,
    "require('is-buffer')": true,
    "Object": true,
    "global": true,
    "global.toString": true,
    "module": true,
    "module.exports": true
  },
  "Path/to/is-buffer/index.js": {
    "module": true,
    "module.exports": true
  },
  "Path/to/array-slice/index.js": {
    "module": true,
    "module.exports": true,
    "Math": true,
    "Math.min": true
  },
  "Path/to/array-last/index.js": {
    "require": true,
    "require('is-number')": true,
    "module": true,
    "module.exports": true,
    "Array": true
  },
  "Path/to/is-number/index.js": {
    "module": true,
    "module.exports": true,
    "Array": true,
    "Array.isArray": true,
    "String": true,
    "Number": true
  }
}
```

Παρατηρώντας το αποτέλεσμα της ανάλυσης μπορούμε να δούμε ότι επιστρέφεται ένα αρχείο `json` χωρισμένο σε διαφορετικά επίπεδα. Το κάθε επίπεδο αντιστοιχεί σε μία από τις βιβλιοθήκες που έχουν εισαχθεί στο κεντρικό πρόγραμμα. Χρησιμοποιούμε την απόλυτη διαδρομή της κάθε βιβλιοθήκης σαν κλειδί (`Path/to/main.js`) και ως στοιχεία τις διαφορετικές προσβάσεις που συμβαίνουν στους πόρους του υπολογιστή/γλώσσας προγραμματισμού (`console.log, require`).

Η ανάλυση απαντάει στα αρχικά ερωτήματα με τον εξής τρόπο:

- Βλέπουμε ότι εισάγονται οι βιβλιοθήκες *array-first, is-number, kind-of, is-buffer, array-slice, array-last, is-number* [5, 32, 33, 1, 30, 31].
- Βλέπουμε ξεκάθαρα τα στοιχεία όπου έχει πρόσβαση η κάθε βιβλιοθήκη.
- Βλέπουμε τις συναρτήσεις που εκτελεί και τον τρόπο τον οποίο αλληλεπιδρά με τα στοιχεία του λειτουργικού συστήματος στα οποία έχει πρόσβαση.

Άρα από το αποτέλεσμα της ανάλυσης μπορούμε να δούμε εύκολα το δέντρο εξάρτησης των βιβλιοθηκών που χρησιμοποιούμε και τα στοιχεία του λειτουργικού συστήματος που προσπελάζονται. Επιτυγχάνουμε με μόλις 7 γραμμές κώδικα να έχουμε 67 γραμμές αποτελεσμάτων ανάλυσης ασφαλείας χρήσιμες στον χρήστη —και όπως θα δούμε στο Κεφάλαιο 7 χωρίς ουσιαστικό κόστος σε ταχύτητα.



Γράφημα 3.1: Δέντρο εξάρτησης κύριου προγράμματος.

Κεφάλαιο 4

Αρχιτεκτονική αδρομερούς ανάλυσης

Το εργαλείο βασίζει την λειτουργία του στον τρόπο εισαγωγής και ανάγνωσης των βιβλιοθηκών μίας εφαρμογής. Αναλαμβάνει τον έλεγχο της εισαγωγής της βιβλιοθήκης από τις βασικές διεργασίες της γλώσσας προγραμματισμού και εφαρμόζει μία μορφή αποσύνθεσης, μετατροπής και ανασύνθεσης των εισαγόμενων βιβλιοθηκών. Η αποσύνθεση πραγματοποιείται στα όρια των εισαγόμενων βιβλιοθηκών, η μετατροπή εισάγει κώδικα ειδικά κατασκευασμένο από την εκάστοτε ανάλυση, τέλος η ανασύνθεση επανενώνει τα κομμάτια διατηρώντας την αρχική τους λειτουργικότητα. Παρακάτω, θα αναλυθούν και θα δοθούν παραδείγματα για το κάθε στάδιο ξεχωριστά.

4.1 Αποσύνθεση

Όταν ένα πρόγραμμα ξεκινάει, το πρώτο πράγμα που φορτώνει είναι η Lya μαζί με την ανάλυση που έχει επιλεγεί. Για να επιτευχθεί η φόρτωση του εργαλείου, επιβάλλεται να δηλωθεί σε διαφορετικό αρχείο και να καλέσει το κύριο πρόγραμμα που θέλουμε να αναλύσουμε ή να εισαχθεί στην πρώτη γραμμή του πηγαίου κώδικα που θέλουμε να αναλυθεί. Περισσότερα σχετικά με τον τρόπο λειτουργίας των ίδιων των αναλύσεων θα δούμε στο Κεφάλαιο 5.

Το εργαλείο αυτόματα δημιουργεί ένα γράφημα εξάρτησης (`dependency graph`), που περιέχει το μοναδικό αναγνωριστικό της κάθε βιβλιοθήκης, που είναι συνήθως η τοποθεσία της εσωτερικά του συστήματος αρχείων. Η επίγνωση αυτού του γραφήματος προσδίδει ιδιαίτερα οφέλη, όπως:

- Δίνει την δυνατότητα να παράγονται αναλυτικά και ωφέλιμα αποτελέσματα. Αυτό συμβαίνει καθώς προσφέρει ένα ιδιαίτερο κλειδί αποθήκευσης για την κάθε πρόσβαση και το κάθε στοιχείο της βιβλιοθήκης –καθώς και για την ίδια την βιβλιοθήκη.
- Δίνει την δυνατότητα αυτόματης αναγνώρισης του επίπεδου της ανάλυσης και το βάθος που έχει διασχίσει στο δέντρο εξάρτησης.

Αμέσως μετά αναλύει ποιες από τις βιβλιοθήκες που χρησιμοποιούνται από το πρόγραμμα υπάρχει επιθυμία από τον χρήστη να καταγραφούν και να ελεγχθούν. Τα στοιχεία αυτά περνιούνται με την χρήση ενός ιδανικού αντικειμένου από τον χρήστη της ανάλυσης και, αν δεν υπάρξει κάποιος προσδιορισμός, το εργαλείο θεωρεί ότι καταγράφονται όλες οι βιβλιοθήκες λογισμικού. Επιπλέον, μέσω των ορισμάτων που έχει δώσει ο χρήστης ελέγχονται το πλήθος και η ταυτότητα των στοιχείων που προσφέρει η γλώσσα προγραμματισμού και θέλουμε να καταγραφούν. Μερικά από αυτά τα στοιχεία είναι:

- **User Global Variables** (Καθολικές Μεταβλητές Χρηστών): Είναι οι μεταβλητές που δηλώνονται πάνω στο `global tree` με την χρήση του προθέματος `global`. Μερικά παραδείγματα είναι μία δήλωση όπως `global.x, global.y, global.x`.
- **With Global Variables** (Καθολικές Μεταβλητές): Είναι οι μεταβλητές που δηλώνονται πάνω στο `global tree`, αλλά χωρίς πρόθεμα `global`. Σε αυτή την περίπτωση η ανίχνευση του είναι δυσκολότερη και επιτυγχάνεται με συνδυασμό της λειτουργικότητας `with` και του αγκιστριού `onHas`.
- **Node Globals** (Μεταβλητές του περιβάλλοντος Node.js): Είναι όλες οι μεταβλητές που εισάγονται από το περιβάλλον `node.js` πάνω στην `Javascript`. Μερικά παραδείγματα από τέτοιους τελεστές είναι `console, process`.
- **Module Locals** (Μεταβλητές που περικλείονται σε επίπεδο βιβλιοθήκης): Είναι οι μεταβλητές που είναι ξεχωριστές για κάθε διαφορετική βιβλιοθήκη που εισάγεται. Μερικά παραδείγματα είναι `require, modules, _filename_, _dirname_`.
- **Module Returns** (Μεταβλητές που επιστρέφονται από την βιβλιοθήκη): Είναι το σύνολο των στοιχείων που επιστρέφονται από μία βιβλιοθήκη. Μερικά παραδείγματα είναι το `exports, module.exports`.

Μόλις ολοκληρωθεί η φάση της προσδιορισμού των παραπάνω επιμέρους στοιχείων της ανάλυσης, το εργαλείο αντικαθιστά δυναμικά την κλήση τελεστή εισαγωγής `(import)`. Πλέον κάθε κάλεσμα του `import` καλεί την Lya. Με αυτή την δυνατότητα το μορφοποιημένο `import` κάνει τις εξής ενέργειες:

1. Ελέγχει την προσωρινή μνήμη *cache*, αν η βιβλιοθήκη έχει κληθεί ήδη.
2. Αν έχει κληθεί ήδη, ελέγχει αν έχει το ίδιο άγκιστρο *(hook)*, που αναλογεί στην εκάστοτε ανάλυση που θέλουμε να τρέξουμε.

Αν και στις δύο περιπτώσεις η απάντηση είναι θετική, τότε επιστρέφεται το ίδιο στοιχείο από την μνήμη με την προσαρμοσμένη τιμή. Αν έχει κληθεί με κάποια διαφορετική ανάλυση, άρα και έχει τυλιχτεί με διαφορετικά άγκιστρα, τότε καλεί από την μνήμη ένα καθαρό αντικείμενο χωρίς αλλαγές και του εφαρμόζει τις απαραίτητες μορφοποιήσεις. Ο λόγος που μπορεί να έχει τυλιχτεί με διαφορετικά άγκιστρα από το επιθυμητό, είναι η περίπτωση χρήσης που για διαφορετικά κομμάτια του δέντρου εξάρτησης έχουν εφαρμοστεί διαφορετικές αναλύσεις. Τέλος, στην περίπτωση που το αντικείμενο δεν υπάρχει στην προσωρινή μνήμη, τότε καλείται η ενσωματωμένη βιβλιοθήκη της γλώσσας που αναλαμβάνει την φόρτωση του στοιχείου. Για να φορτωθεί ένα νέο στοιχείο από την μνήμη πρέπει:

1. Να διαβαστούν τα απαραίτητα στοιχεία πηγαίου κώδικα από την μνήμη.
2. Να ερμηνευτεί, να αναλυθεί και να εφαρμοστούν οι απαραίτητες μορφοποιήσεις.

Τα δεδομένα που επιστρέφονται είναι σε μορφή κείμενου *(string)* και για να ερμηνευτούν χρησιμοποιείται ο μεταγλωττιστής της γλώσσας προγραμματισμού για να μετατρέψει τον κώδικα κειμένου σε στοιχεία μέσα στην μνήμη.

4.2 Μετασχηματισμοί ανάλυσης

Για κάθε βιβλιοθήκη που γίνεται ανάλυση, το εργαλείο πρέπει να θέσει άγκιστρα γύρω από τα όρια της, δηλαδή όχι μόνο στην διεπαφή. Αυτό επιτυγχάνεται με τρία λογικά βήματα:

1. Μετασχηματίζοντας εξωτερικά το πλαίσιο της βιβλιοθήκης, δημιουργώντας πίνακα αντιστοίχησης (`map`) από ονόματα μεταβλητών σε τιμές που είναι διαθέσιμες εξωτερικά από το πακέτο.

2. Μετασχηματίζοντας το πλαίσιο της ίδιας της βιβλιοθήκης, έτσι ώστε κάθε φορά που γίνεται κάποιο κάλεσμα μεταβλητής μέσα από αυτό το πλαίσιο, να επιστρέφονται οι τιμές που έχουν μορφοποιηθεί από την Lya.

3. Μετασχηματίζοντας τις τιμές της διεπαφής της βιβλιοθήκης (*export*), δηλαδή τις τιμές που επιστρέφονται στο αρχικό πρόγραμμα που κάλεσε την βιβλιοθήκη.

Πριν συζητήσουμε το κάθε σημείο στο οποίο εφαρμόζεται ο κάθε μετασχηματισμός, θα παρουσιαστεί ο τρόπος με τον οποίο θα εφαρμοστεί αυτός ο μετασχηματισμός.

Ο μετασχηματισμός του εργαλείου αδρομερούς ανάλυσης που δημιουργήθηκε, επί της ουσίας είναι ένας βασικός μετασχηματισμός, ή αλλιώς περιτύλιγμα, όλων των τιμών ενός αντικειμένου και των παιδιών του αντίστοιχα. Σε υψηλό επίπεδο αυτό σημαίνει ότι έχουμε μία συνάρτηση `wrap`, η οποία δέχεται ως είσοδο μία τιμή x, και εφαρμόζει πάνω της συγκεκριμένα κομμάτια της ανάλυσης που έχει διαλέξει ο προγραμματιστής, επιστρέφοντας μία νέα τιμή x' που έχει όλα τα πεδία του περιτυλιγμένα. Αυτό σημαίνει ότι αν έχει κάποια παιδιά f, αντικαθιστώνται με τα f' και κάθε φορά που χρησιμοποιούνται καλούν κάποιο από τα κομμάτια της ανάλυσης που περάστηκαν στο x, αμέσως μετά καλείται η αρχική f που μετά καλεί ένα άλλο κομμάτι της ανάλυσης και επιστρέφεται το αποτέλεσμα στην f.

Κώδικας 4.1: Παράδειγμα μετασχηματισμού

```
x'.f' = (...arguments) => {
  analysis 1;
  r <- x.f(...arguments);
  analysis 2;
  return r
}
```

Πιο συγκεκριμένα η συνάρτηση `wrap`, μπορεί να εφαρμοστεί σε οποιαδήποτε στοιχείο της γλώσσας, όπως μία βασική τιμή, μία συνάρτηση ή ακόμα και μία σύνθετη τιμή —όπως για παράδειγμα, μία λίστα από τιμές ή ένα αντικείμενο που περιέχει ένα σύνολο από κλειδιά. Ο μετασχηματισμός των σύνθετων τιμών ξεκινάει από την ρίζα τους και συνεχίζει να επεξεργάζεται τις τιμές τους ανάλογα με τον τύπο τους, μέχρι τα φύλλα τους. Αυτό σημαίνει ότι:

- Στις *συναρτήσεις*, γίνεται περιτύλιγμα από άγκιστρα που περιλαμβάνουν ειδικό κώδικα για την εκάστοτε ανάλυση που εφαρμόζεται.

- Στα *αντικείμενα* και στις *λίστες*, μετασχηματίζονται αναδρομικά οι μέθοδοί τους, λάβε (*getter*) και θέσε (*setter*), με τρόπο παρόμοιο με αυτόν των συναρτήσεων.

- Στα βασικά στοιχεία, είτε μορφοποιούνται κατευθείαν, είτε αντιγράφονται χωρίς μορφοποίηση, εφαρμόζεται πάνω τους μέθοδος που καλείται κάθε φορά που πραγματοποιείται κάποια πρόσβαση.

Για να εξασφαλιστεί ότι δεν γίνονται ατέρμονες επαναλήψεις, καθώς πραγματοποιούνται οι παραπάνω λειτουργίες, διατηρούμε ένα χάρτη (*map*) που ελέγχεται στην αρχή κάθε αναδρομικής κλήσης. Αν παρατηρήσει ότι έχουν περιτυλιχτεί τα στοιχεία που δίνονται, επιστρέφει και τελειώνει την μορφοποίηση σε αυτό το αντικείμενο.

Κάθε φορά που υπάρχει κάποια άμεση πρόσβαση σε κάποιο πεδίο, όπως για παράδειγμα ανάθεση κάποιας τιμής, απαιτείται να υπάρξει ανίχνευση κατά την πρόσβαση. Για να επιτευχθεί αυτό, η Lya τυλίγει, με την χρήση ενός μηχανισμού παρεμβολής (`proxy`), κάθε πρόσβαση στο πεδίο ως κλήση κάποιας συνάρτησης. Σε περίπτωση που επιθυμούμε να υπάρξει κάποια αντικατάσταση του μετασχηματισμένου στοιχείου, θα πρέπει να ξεκινήσει η μορφοποίηση της τιμής αντικατάστασης πριν αποθηκευτεί. Για παράδειγμα, εάν σε ένα πεδίο ενός μετασχηματισμένου αντικειμένου γίνει ανάθεση μίας νέας τιμής, τότε αυτή η τιμή πρέπει να μετασχηματιστεί πριν ενωθεί με το αντικείμενο.

Η Lya επιτρέπει να ενεργοποιούνται/απενεργοποιούνται αναλύσεις, να εναλλάσσονται αναλύσεις ή ακόμα και να υπάρχει ένωση των αναλύσεων κατά την εκτέλεση του προγράμματος. Για να μπορέσει να επιτευχθεί αυτό διατηρούνται οι τιμές των στοιχείων όπου δεν έχει εφαρμοστεί μετασχηματισμός, αλλά και των μετασχηματισμένων τιμών. Οι τιμές όπου έχουν εφαρμοστεί αλλαγές μπορούν να χρησιμοποιηθούν για να καλέσουμε ή να ενώσουμε αναλύσεις και στις τιμές που δεν έχουν εφαρμοστεί αλλαγές μπορούν να χρησιμοποιηθούν για να τρέξουν διαφορετικές αναλύσεις.

Μετασχηματισμός πλαισίου Για να μπορέσουμε να εφαρμόσουμε αδρομερή ανάλυση στα όρια των βιβλιοθηκών, πρέπει να υπάρχει η δυνατότητα να παρέχουμε στην κάθε βιβλιοθήκη στοιχεία που έχουν τυλιχθεί με την χρήση της συνάρτησης `wrap` —σε κάθε ένα από τα ονόματα όπου η βιβλιοθήκη έχει πρόσβαση. Αυτό περιλαμβάνει τις καθολικές μεταβλητές, αλλά και τις ψευδο-καθολικές μεταβλητές που παρέχονται από την γλώσσα προγραμματισμού, καθώς και το ευρύτερο περιβάλλον του λειτουργικού συστήματος.

Για να μπορέσει να επιτευχθεί αυτό, η Lya πρέπει να δημιουργήσει ένα μετασχηματισμένο αντίγραφο του πλαισίου της βιβλιοθήκης —δηλαδή τον πίνακα αντιστοιχίας από τα ονόματα των στοιχείων που προβλέπεται να είναι εσωτερικά του πεδίου εφαρμογής του πακέτου. Για να επιτευχθεί αυτό το εργαλείο, δημιουργεί έναν πίνακα που έχει ως κλειδιά τα ονόματα που περιέχονται στο πλαισίου που αναφέρθηκε και έχει ως τιμές τα μετασχηματισμένα στοιχεία που αντιστοιχούν. Σαν κλειδί μπορεί να είναι οποιοδήποτε στοιχείο που είναι διαθέσιμο στην βιβλιοθήκη και επιστρέφει κάποια τιμή αν κληθεί. Μερικά παραδείγματα τέτοιων κλειδιών είναι στοιχεία που περιλαμβάνονται στα `globals, built-ins, module-locals`. Οι μετασχηματισμένες τιμές δημιουργούνται εφαρμόζοντας την συνάρτηση `wrap` στο πλαίσιο, αξιοποιώντας όλα τα απαραίτητα άγκιστρα που προσφέρονται.

Ιδιαίτερη προσοχή χρειάζονται τα στοιχεία που είναι ξεχωριστά για την κάθε βιβλιοθήκη. Τέτοια στοιχεία είναι προσβάσιμα από όλα τα σημεία που είναι μέσα στα όρια της βιβλιοθήκης, παρόμοια με τις καθολικές μεταβλητές, αλλά επιστρέφουν διαφορετικές τιμές για το κάθε πακέτο. Μερικά παραδείγματα τέτοιων στοιχείων είναι το `__name__` που επιστρέφει το όνομα του αρχείου, το στοιχείο `export` που μας δίνει τις τιμές που επιστρέφονται από την βιβλιοθήκη στο κύριο πρόγραμμα ή ακόμα και το `require.main` το οποίο δείχνει αν η βιβλιοθήκη έχει κληθεί ως κύρια βιβλιοθήκη. Αν υπάρξει πρόσβαση σε αυτά τα στοιχεία κατευθείαν μέσα από τα όρια του εργαλείου, η λειτουργία θα αποτύχει, καθώς αντί να επιστραφούν τα στοιχεία που είναι ξεχωριστά για την βιβλιοθήκη θα επιστραφούν τα στοιχεία του κύριου προγράμματος όπου εφαρμόζεται η ανάλυση. Αυτό το πρόβλημα λύνεται, καθώς η Lya αφήνει τα στοιχεία αυτά άδεια μέσα στον πίνακα και τα συμπληρώνει την στιγμή που γίνεται πρόσβαση στην εκάστοτε βιβλιοθήκη.

Δέσμευση πλαισίου Για να μπορέσουμε να πετύχουμε την σύνδεση του μορφοποιημένου πλαισίου με το απλό της βιβλιοθήκης, η Lya επεμβαίνει στον κώδικα του πακέτου, στο σημείο πριν

ερμηνευτεί από την γλώσσα προγραμματισμού και περιτυλιχθεί με τον πρόλογο. Ο πρόλογος είναι της μορφής:

```
local print = ctx.print
local error = ctx.error
...
```

Έτσι επιτυγχάνεται η αντικατάσταση των στοιχείων των καθολικών μεταβλητών με τοπικά για την κάθε βιβλιοθήκη. Όπως βλέπουμε παραπάνω, η συνάρτηση `print` γίνεται τοπική και αντικαθιστάται από το μορφοποιημένο στοιχείο που έχουμε μέσα στον χάρτη των στοιχείων (χάρτης με το όνομα `ctx`). Ο πρόλογος εκτελείται κάθε φορά πριν από οποιαδήποτε άλλη λειτουργία στην βιβλιοθήκη, επιτυγχάνοντας, έτσι αυτές οι μορφοποιήσεις να είναι σε πρωταρχικό στάδιο πριν μπορέσουν να χρησιμοποιηθούν τα ονόματα του πλαισίου. Τα στοιχεία του πλαισίου που περιέχουν μεταβλητές που είναι ξεχωριστές για την εκάστοτε βιβλιοθήκη, περιτυλίγονται την στιγμή της εισαγωγής της βιβλιοθήκης.

Μετασχηματισμός διεπαφής βιβλιοθήκης Το τελευταίο στοιχείο που μετασχηματίζεται είναι τα δεδομένα που επιστρέφονται προγραμματιστικά στο κύριο πρόγραμμα από την βιβλιοθήκη. Εφαρμόζεται ένα περιτύλιγμα στο εξωτερικό αντικείμενο που τα περιέχει και κάθε φορά που καλείται για να χρησιμοποιηθεί κάποιο από τα στοιχεία της διεπαφής, εφαρμόζεται σε αυτό το δικό του περιτύλιγμα. Ο στόχος είναι να μπορέσει να επιτευχθεί διαχωρισμός μεταξύ των ορίων του πακέτου που χρησιμοποιείται και των υπόλοιπων βιβλιοθηκών.

4.3 Ανασύνθεση

Για να μπορέσουμε να ανασυνθέσουμε επιτυχώς την εφαρμογή, η Lya πρέπει να σιγουρευτεί ότι όλες οι αναφορές μεταξύ των βιβλιοθηκών επιλύονται σωστά. Ο κεντρικός μηχανισμός για να επιτευχθεί αυτό είναι η χρήση προσωρινής μνήμης βιβλιοθηκών.

Για να μπορέσουν να επιτευχθούν πολλαπλά περιτυλίγματα σε μία βιβλιοθήκη που έχει μόνο ένα επίπεδο προσωρινής μνήμης, καθίσταται αναγκαίο να επεκταθεί κατά δύο επίπεδα (συνολικά τρία επίπεδα). Ο λόγος που προσθέτουμε αυτά τα δύο επίπεδα είναι ότι οι βιβλιοθήκες έχουν συνήθως μόνο μία ανάλυση σε επίπεδο πλαισίου, αλλά πολλαπλές αναλύσεις σε επίπεδο διεπαφής —συνήθως μία για κάθε χρήστη της βιβλιοθήκης. Ο μετασχηματισμός πλαισίου εφαρμόζεται ελάχιστες φορές (συνήθως μονάχα μία), ενώ οι μετασχηματισμοί στο αντικείμενο που επιστρέφεται από την βιβλιοθήκη στο κύριο πρόγραμμα πραγματοποιείται σε κάθε εισαγωγή (`import`) που συμβαίνει. Σε κάθε βιβλιοθήκη το πρώτο επίπεδο χρησιμοποιείται για να κρατηθούν τα κλειδιά από τα χαρακτηριστικά ονόματα των βιβλιοθηκών, το δεύτερο επίπεδο για στοιχεία της ανάλυσης πλαισίου και το τρίτο για την ανάλυση της διεπαφής που εξάγεται. Οι περισσότερες βιβλιοθήκες περνάνε όλα τα στάδια κατά την μορφοποίηση εκτός από το τελευταίο. Δηλαδή έχουν ερμηνευτεί και έχουν το πλαίσιο τους μετασχηματισμένο και συνδεόμενο, αλλά δεν έχουν ανατεθεί ακόμα άγκιστρα στα στοιχεία εσωτερικά της διεπαφής τους. Αυτό συμβαίνει καθώς αναμένουν να χρησιμοποιηθούν από τον κύριο χρήστη της ανάλυσης και να υπάρξει ο ξεκάθαρος διαχωρισμός των επιπέδων και τον βιβλιοθηκών.

Ένα από τα στοιχεία που κρατούνται στην μνήμη είναι η αρχική μορφή της εκάστοτε βιβλιοθήκης που εισαγάγουμε, σε μορφή κειμένου. Αυτό συμβαίνει, καθώς δίνεται η δυνατότητα στον χρήστη της ανάλυσης, είτε να αποκλείσει συγκεκριμένα πακέτα, είτε να την εφαρμόσει μόνο σε συγκεκριμένα πακέτα.

Συνοψίζοντας, όταν μία νέα ανάλυση εφαρμόζεται στην βιβλιοθήκη, η Lya δημιουργεί πίνακα αντιστοίχησης, με κλειδί το απόλυτο όνομα της βιβλιοθήκης, και εφαρμόζει στο πλαίσιο της συγκεκριμένα περιτυλίγματα ανάλογα την ανάλυση. Αυτόν τον μετασχηματισμό τον αποθηκεύει στο

επόμενο επίπεδο προσωρινής μνήμης. Όταν μία βιβλιοθήκη έχει ήδη εισαχθεί, η Lya κάνει τις απαραίτητες ενέργειες, έτσι ώστε να μπορέσει να ανακτήσει την μορφοποιημένη βιβλιοθήκη που είναι αποθηκευμένη. Αμέσως μετά εφαρμόζει ένα *περιτύλιγμα* στο εξωτερικό αντικείμενο που περιέχει την διεπαφή και εισάγει την ολοκληρωμένη βιβλιοθήκη στο τρίτο επίπεδο της προσωρινής μνήμης.

4.4 Σύνοψη της λειτουργίας της Lya

Πρώτα όμως ανακεφαλαιώνουμε την λειτουργία της Lya, και τα βήματα που απαιτούνται για να επιτευχθεί μια αδρομερής ανάλυση:

1. Φορτώνουμε το εργαλείο με την χρήση του τελεστή `require` της `Javascript`.
2. Περνάμε με ένα αντικείμενο ρυθμίσεων όλα τα στοιχεία που θα καθορίσουν το τρόπο με τον οποίο θα εκτελεστεί η τρέχουσα συνεδρία. Μερικά από αυτά τα στοιχεία είναι το επίπεδο, το όνομα της ανάλυσης, αν επιθυμούμε να αποθηκεύσουμε τα αποτελέσματα της ανάλυσης σε κάποιο αρχείο και άλλες επιλογές.
3. Ξεκινάει η εκτέλεση της Lya με την χρήση της συνάρτησης `lyaStartUp`. Από αυτήν την στιγμή και έπειτα το εργαλείο καταγράφει όλα τα στοιχεία που ενθυμούνται από τον χρήστη και τα αποθηκεύει στα ανάλογα σημεία.
4. Σε κάθε εισαγωγή βιβλιοθήκης με την χρήση του τελεστή `require`, καλείται το επεξεργασμένο `require`, από την βιβλιοθήκη `Module`. Έτσι αντί να κληθεί το `require` του συστήματος καλείται το επεξεργασμένο της Lya.
5. Αμέσως μετά καλείται μία διαδοχική σειρά από συναρτήσεις του συστήματος που έχουν αλλαχθεί από το εργαλείο, έτσι ώστε να διατηρούν διάφορες χρήσιμες πληροφορίες για την εκτέλεση του προγράμματος. Μερικές από αυτές είναι η απόλυτη διαδρομή του πακέτου που εισάγεται, ο κώδικας του σε μορφή κειμένου και άλλα.
6. Πραγματοποιούνται όλα τα απαραίτητα περιτυλίγματα στοιχείων, ώστε να καταγράφονται οι ενέργειες των πακέτων που επιθυμεί ο χρήστης να χρησιμοποιηθούν.
7. Τέλος, επιστρέφεται η διεπαφή της βιβλιοθήκης περιτυλιγμένη, μόνο εξωτερικά, από το ανάλογο άγκιστρο. Κάθε φορά που γίνεται πρόσβαση σε κάποιο στοιχείο της διεπαφής, πριν επιστραφεί στον χρήστη, περιτυλίγεται με το απαραίτητο άγκιστρο και μετά επιστρέφεται. Αυτό συμβαίνει καθώς έτσι εξασφαλίζονται τα όρια των βιβλιοθηκών.

4.5 Υπόλοιπες λεπτομέρειες υλοποίησης

Σε αυτό το σημείο θα δοθούν μερικές ακόμα τεχνικές λεπτομέρειες της υλοποίησης, θα αναλυθούν μερικά από τα προβλήματα που παρουσιάστηκαν και οι τρόποι με τους οποίους αντιμετωπίστηκαν.

Ένα από τα προβλήματα που παρουσιάστηκαν ήταν η δυνατότητα καταγραφής των προσβάσεων στις καθολικές μεταβλητές, όταν ο προγραμματιστής δεν χρησιμοποιεί τον τελεστή `global`. Καθώς, όταν πάει να προσπελάσει κάποιο στοιχείο στο καθολικό πλαίσιο, χωρίς αυτό τον τελεστή, η ανάγνωση είναι αρκετά πιο πολύπλοκη να ανιχνευτεί. Ο τρόπος που επιλύθηκε αυτό το πρόβλημα είναι η χρήση τόσο του μηχανισμού παρεμβολής `has`, όσο και του τελεστή `with`.

Ένα άλλο πρόβλημα που αντιμετωπίστηκε ήταν η διαφοροποίηση των στοιχείων του πλαισίου μεταξύ των βιβλιοθηκών. Στις πρώτες εκδόσεις της Lya, δεν ήταν εύκολο να επιτευχθεί διαχωρισμός μεταξύ των στοιχείων του πλαισίου, καθώς επί της ουσίας μόλις περνούσε ο μηχανισμός παρεμβολής επέστρεφαν τα ίδια στοιχεία σε όλες τις περιπτώσεις. Για να επιλυθεί αυτό το πρόβλημα,

πριν την εκτέλεση της εισαγωγής, πραγματοποιείται κλωνοποίηση όλων των στοιχείων του πλαισίου. Με αυτόν τον τρόπο ο κλώνος του κάθε στοιχείου είναι μοναδικός για την κάθε βιβλιοθήκη και μπορεί να χρησιμοποιηθεί σαν κλειδί σε κάποιο νέο πίνακα αντιστοίχησης, επιστρέφοντας τα ανάλογα δεδομένα για κάθε βιβλιοθήκη.

Κεφάλαιο 5

Αναλύσεις

Παρακάτω θα παρουσιαστούν και θα αναλυθούν διαφορετικές αναλύσεις που μπορούν να εφαρμοστούν με την χρήση της LYA, και παραδείγματα για την κάθε μία.

5.1 Ανάλυση ασφάλειας

Πρόβλημα Τα τελευταία χρόνια, με την επικράτηση των βιβλιοθηκών τρίτων, παρατηρείται μία πρωτοφανής αύξηση σε επιθέσεις κατά την αλυσίδα εφοδιασμού (supply-chain) [23, 36, 19]. Μικρά προβλήματα, εσωτερικά των βιβλιοθηκών, ή ακόμα και κακόβουλος κώδικας που μπορεί να εισαχθεί, δημιουργούν πρωτοφανή εμπόδια. Τα περισσότερα σύγχρονα προγράμματα χρησιμοποιούν χιλιάδες βιβλιοθήκες με εκατοντάδες γραμμές κώδικα αθροιστικά. Αντίστοιχα, μικρές, σε αριθμό γραμμών κώδικα, βιβλιοθήκες χρησιμοποιούνται από εκατοντάδες εφαρμογές [47, 49]. Ως αποτέλεσμα, τέτοιου είδους επιθέσεις είναι δύσκολο να αντιμετωπιστούν και επηρεάζουν ένα μεγάλο εύρος του οικοσυστήματος εφαρμογών.

Στην πράξη Ένα παράδειγμα τέτοιας επίθεσης είναι το πρόσφατο ατύχημα στο *event stream* [37, 28] όπου ένας συντηρητής ενός ιδιαίτερα δημοφιλούς πακέτου, εισήγαγε κώδικα που είχε ως στόχο να αποσπάσει τους κωδικούς ασφαλείας πορτοφολιών ενός δημοφιλούς ηλεκτρονικού νομίσματος *(Bitcoin)*. Η χρήση της στατικής ανάλυσης δεν θα βοηθούσε στην περίπτωση αυτή, καθώς αυτός ο κακόβουλος κώδικας ήταν κρυπτογραφημένος. Αντίστοιχα, η χρήση των κλασσικών δυναμικών αναλύσεων [34] δεν θα βοηθούσε, καθώς αυτά τα προβλήματα θα πυροδοτούνται στην παραγωγή.

Ανάλυση Στην περίπτωση αυτή είναι ιδανική λύση η χρήση μίας *επίτρεψε/απόρριψε (allow/deny)* αδρομερούς ανάλυσης. Η επιλογή αυτού του είδους ανάλυσης είναι η βέλτιστη, καθώς προσφέρει χαμηλό κόστος χρήσης, δίνοντας την δυνατότητα να τρέχει συνεχώς στην παραγωγή ανιχνεύοντας έτσι τις ασυνήθιστες προσβάσεις στους πόρους από το *event-Stream*. Καθώς το *event-Stream* θα έκανε εισαγωγή των βιβλιοθηκών *fs, http*, θα ανιχνεύονταν η ασυνήθιστη συμπεριφορά και θα γινόταν αναφορά στον χρήστη. Αυτού του είδους η ανάλυση μπορεί να ανιχνεύσει προσβάσεις στις καθολικές μεταβλητές, εισαγωγές βιβλιοθηκών και στην προσωρινή μνήμη της βιβλιοθήκης.

Παράδειγμα Θα παρουσιαστεί ένα παράδειγμα χρήσης μίας *επίτρεψε/απόρριψε* αδρομερούς ανάλυσης σε ένα κακόβουλο κώδικα που τυπώνει τον κωδικό του χρήστη που είναι αποθηκευμένος σε ένα αρχείο *json*.

Κώδικας 5.1: Αρχείο json

```
{
    "username": "user",
    "password": "secret"
}
```

Κώδικας 5.2: Βιβλιοθήκη Serial

```
module.exports = {
    dec: (str) => {
        let obj;
        obj = eval('(' + str + ')');
        return obj
    },
}
```

Βλέπουμε ότι αυτή η βιβλιοθήκη χρησιμοποιεί την συνάρτηση `eval`, έτσι ώστε να πραγματοποιήσει σειριοποίηση κάποιας εντολής. Όμως, λόγω του τρόπου λειτουργίας της `eval`, θα εκτελέσει *οποιαδήποτε* εντολή περαστεί ως κείμενο —χωρίς να μπορεί να ελεγχθεί.

Κώδικας 5.3: Ενδεικτικό Πρόγραμμα

```
global.dbc = require("./dbc.json");

var dispatch = (obj, res) => {
    console.log(obj)
}

var srv = (req, res) => {
    let srl, obj;
    srl = require("./serial.js");
    obj = srl.dec(req.body);
    dispatch(obj, srl)
}

srv({body: 'console.log(dbc.password)'});
```

Εκμεταλλευόμενος το γεγονός ότι η `eval` επιτρέπει την εκτέλεση οποιασδήποτε εντολής δοθεί σε αυτήν με μορφή κειμένου, ο κακόβουλος χρήστης περνάει την εντολή `console.log(dbc.password)`. Με αυτόν τον τρόπο, μπορεί και τυπώνει τον κωδικό που έχει αποθηκεύσει ο χρήστης, στο αρχείο `json`. Για να μπορέσουμε να ανιχνεύσουμε την εκτύπωση αυτών των προσωπικών στοιχείων θα εφαρμόσουμε, την αδρομερή ανάλυση επίτρεψε/απόρριψε που βλέπουμε στον παρακάτω κώδικα:

Κώδικας 5.4: Ανάλυση επίτρεψε/απόρριψε

```
let env;
const fs = require('fs');

const save = (file, name) => {
  if (Object.prototype.hasOwnProperty.
```

```
      call(file, name) === false) {
    file[name] = true;
  }
};

const onRead = (info) => {
  if (info.nameToStore !== 'global') {
    save(env.analysisResult[info.currentModule],
        info.nameToStore.split('.')[0]);
    }
    save(env.analysisResult[info.currentModule],
        info.nameToStore);
  }
};

const onWrite = (info) => {
  if (info.parentName) {
    save(env.analysisResult[info.currentModule], info.parentName);
  }
  save(env.analysisResult[info.currentModule], info.nameToStore);
};

const onCallPre = (info) => {
  if (info.typeClass === 'module-locals') {
    save(env.analysisResult[info.currentModule],
        'require');
    save(env.analysisResult[info.currentModule],
        info.nameToStore);
  } else {
    if (info.typeClass === 'node-globals') {
      save(env.analysisResult[info.declareModule],
          info.nameToStore.split('.')[0]);
    }
    save(env.analysisResult[info.declareModule],
        info.nameToStore);
  }
};

const onConstruct = (info) => {
  save(env.analysisResult[info.currentName],
      info.nameToStore);
};

const onExit = () => {
  if (env.conf.SAVE_RESULTS) {
    fs.writeFileSync(env.conf.SAVE_RESULTS,
        JSON.stringify(env.analysisResult, null, 2), 'utf-8');
  }
  if (env.conf.print) {
```

```
    console.log(JSON.stringify(env.analysisResult, null, 2));
  }
};

module.exports = (e) => {
  env = e;
  return {
    onRead: onRead,
    onCallPre: onCallPre,
    onWrite: onWrite,
    onConstruct: onConstruct,
    onExit: onExit,
  };
};
```

Βλέπουμε ότι η παραπάνω ανάλυση χρησιμοποιεί τα διαφορετικά άγκιστρα που προσφέρει η Lya, και δημιουργεί την ανάλυση με 61 γραμμές κώδικα. Συνοπτικά, ο κώδικας της ανάλυσης πραγματοποιεί τις εξής λειτουργίες:

- Καλεί κάποιο από τα άγκιστρα που χρησιμοποιούνται (`onRead, onCallPre, onWrite, onConstruct`) ανάλογα με τον τύπο της πρόσβασης.
- Καλεί τη συνάρτηση `save` για να αποθηκεύσει τις προσβάσεις σε ένα πεδίο μόλις ολοκληρωθούν οι απαραίτητες λειτουργίες από το εκάστοτε άγκιστρο.
- Καλεί το άγκιστρο `onExit` στο τέλος του προγράμματος, τυπώνοντας στην οθόνη ή/και αποθηκεύοντας σε κάποιο αρχείο το αποτέλεσμα της ανάλυσης.

Εκτελώντας την παραπάνω αδρομερή ανάλυση στον κακόβουλο κώδικα ανιχνεύουμε αυτόματα την πρόσβαση στο πεδίο του κωδικού πρόσβασης. Αυτό φαίνεται ξεκάθαρα από το παρακάτω αποτέλεσμα και πιο συγκεκριμένα από το πεδίο `require('./ dbc.json').password`.

Κώδικας 5.5: Αποτέλεσμα Ανάλυσης

```
"Path/To/main.js": {
    "require": "true",
    "require('./dbc.json')": "true",
    "global": "true",
    "global.dbc": "true",
    "console": "true",
    "console.log": "true"
    "require('./serial.js')": "true",
    "require('./serial.js').dec": "true"
  },
"Path/To/serial.js": {
    "module": "true",
    "module.exports": "true",
    "eval": "true"
    "require('./dbc.json')": "true",
    "require('./dbc.json').password": "true"
}
```

5.2 Ανάλυση απόδοσης

Πρόβλημα Η ανίχνευση και η καταμέτρηση της απόδοσης του προγράμματος που εκτελείται είναι ένας ιδιαίτερα δύσκολος στόχος, καθώς όπως έχουμε σημειώσει, πλέον οι περισσότερες εφαρμογές βασίζονται σε μεγάλο αριθμό βιβλιοθηκών. Μέσα σε εκατοντάδες γραμμές κώδικα, είναι δύσκολο να ανιχνευτεί ο λόγος της ξαφνικής μείωσης της ταχύτητας της εφαρμογής [42]. Βαθιά μέσα στο δέντρο εξάρτησης (dependency tree) είναι πιθανόν μία βιβλιοθήκη να ενημερωθεί και λόγω κάποιου κώδικα κακής συμπεριφοράς που θα εισχωρήσει, θα δημιουργηθούν τεράστιες καθυστερήσεις.

Στην πράξη Για παράδειγμα, η βιβλιοθήκη *minimatch* για να μπορέσει να πραγματοποιήσει αντιστοίχηση αρχείων χρησιμοποιεί κανονικές εκφράσεις (regular expressions) [21]. Οι κανονικές εκφράσεις μπορούν να δημιουργήσουν καθυστέρηση στο πρόγραμμα που τις χρησιμοποιεί. Έτσι, μειώθηκε η ταχύτητα του πακέτου *redus* [9], λόγω της χρήσης της βιβλιοθήκης *minimatch*, που ήταν βαθιά στο δέντρο εξάρτησης. Για να μπορέσει να αντιμετωπιστεί και να ανιχνευτεί το πρόβλημα αυτό με τα υπάρχοντα μέσα, ο προγραμματιστής θα επέλεγε να συλλέξει και να αναπαράξει ίχνη ενάντια σε προηγούμενες εκδόσεις του συστήματος ή θα δημιουργούσε στατιστικά προφίλ για να ανιχνεύσει αυτές τις διαδρομές κώδικα. Αυτές οι τεχνικές θα ήταν χρονοβόρες.

Ανάλυση Όμως με την χρήση της αδρομερούς ανάλυσης απόδοσης είναι εύκολο να εξαγάγει τα ποσοστά καθυστέρησης και να γνωστοποιηθεί αυτόματα η βιβλιοθήκη που είναι υπεύθυνη για την μειωμένη απόδοση.

Παράδειγμα Παρακάτω παρατίθεται ένα παράδειγμα χρήσης της αδρομερούς ανάλυσης απόδοσης σε μία μαθηματική βιβλιοθήκη. Στην πράξη του πολλαπλασιασμού, όπως φαίνεται και από τον κώδικα, πραγματοποιείται ένα επαναληπτικό άθροισμα που έχει ως αποτέλεσμα να προσδίδει καθυστέρηση στην εκτέλεση του προγράμματος.

Κώδικας 5.6: Μαθηματική Βιβλιοθήκη

```
module.exports= {
    add: (a, b) => a + b,
    sub: (a, b) => a - b,
    mult: (a, b) => {
        let number = 1;
        for (counter=0; counter< 1000000000; counter++) {
            number = number + counter
        }
        return a * b;
    }
}
```

Κώδικας 5.7: Βιβλιοθήκη Απλών Πράξεων

```
const math = require('./math.js');
math.add(1, 3);
math.sub(3, 1);
math.mult(1, 3);
```

Για να μπορέσουμε να αναλύσουμε τους χρόνους που απαιτούνται για την εκτέλεση των συναρτήσεων και να ανιχνεύσουμε την συνάρτηση που προσδίδει καθυστέρηση, θα εφαρμόσουμε την αδρομερή ανάλυση απόδοσης που έχει τον παρακάτω κώδικα:

Κώδικας 5.8: Ανάλυση Απόδοσης

```
let env;
const fs = require('fs');
const storeTime = new Map();
const timeCapsule = {};

const toMillis = (a, b) => (a * 1e9 + b) * 1e-6;

const onCallPre = (info) => {
  storeTime.set(info.target, process.hrtime());
};

const onCallPost = (info) => {
  const level = env.requireLevel;
  const time = storeTime.get(info.target);
  const diff = process.hrtime(time);
  const thisTime = toMillis(diff[0], diff[1]);
  if (timeCapsule[level]) {
    timeCapsule[level] += toMillis(diff[0], diff[1]);
  } else {
    timeCapsule[level] = toMillis(diff[0], diff[1]);
  }

  if (timeCapsule[level+1]) {
    env.analysisResult[info.currentModule][info.nameToStore] =
      thisTime - timeCapsule[level+1];
    timeCapsule[level+1] = 0;
  } else {
    env.analysisResult[info.currentModule][info.nameToStore] =
      thisTime;
  }
};

const onExit = (intersection, candidateModule) => {
  if (env.conf.SAVE_RESULTS) {
    fs.writeFileSync(env.conf.SAVE_RESULTS,
        JSON.stringify(env.analysisResult, null, 2), 'utf-8');
  }
  if (env.conf.print) {
    console.log(JSON.stringify(env.analysisResult, null, 2),
        'utf-8');
  }
};
```

```
module.exports = (e) => {
  env = e;
  return {
    onCallPre: onCallPre,
    onCallPost: onCallPost,
    onExit: onExit,
  };
};
```

Βλέπουμε ότι η παραπάνω ανάλυση χρησιμοποιεί τα άγκιστρα (`onCallPre, onCallPost, onExit`) που προσφέρει η Lya, και δημιουργεί την ανάλυση με 43 γραμμές κώδικα. Ο κώδικας της ανάλυσης πραγματοποιεί τις εξής λειτουργίες:

- Καλεί το άγκιστρο `onCallPre` πριν την εκτέλεση κάποιας συνάρτησης και αποθηκεύει στην μνήμη το χρονικό στιγμιότυπο που πραγματοποιήθηκε η κλήση.
- Καλεί το άγκιστρο `onCallPost` μετά την εκτέλεση κάποιας συνάρτησης και αποθηκεύει στην μνήμη το χρονικό διάστημα που απαιτήθηκε για να εκτελεστεί η συνάρτηση. Ο χρόνος αποθηκεύεται σε *ms*.
- Καλεί το άγκιστρο `onExit` στο τέλος του προγράμματος, τυπώνοντας στην οθόνη ή/και αποθηκεύοντας σε κάποιο αρχείο το αποτέλεσμα της ανάλυσης.

Τρέχοντας το πρόγραμμα απλών πράξεων με την παραπάνω αδρομερή ανάλυση απόδοσης λαμβάνουμε το παρακάτω αποτέλεσμα και καταλήγουμε στο ότι η συνάρτηση που καθυστερεί το πρόγραμμα είναι η *mult* από την βιβλιοθήκη *math.js*.

Κώδικας 5.9: Αποτέλεσμα Αδρομερούς Ανάλυσης Απόδοσης

```
"Path/to/program.js": {
    "require('./math.js')": 2.351034,
    "require('./math.js').add": 0.032805,
    "require('./math.js').sub": 0.023874,
    "require('./math.js').mult": 1195.997027
  }
```

5.3 Εξαγωγή τύπων

Πρόβλημα Η εξαγωγή των τύπων μίας βιβλιοθήκης είναι μία ιδιαίτερα χρήσιμη λειτουργία που προσφέρεται από τις υπάρχουσες αναλύσεις του εργαλείου που δημιουργήθηκε. Είναι χρήσιμο καθώς, λόγου χάριν, βοηθάει στην ανίχνευση των κομματιών κώδικα που θέλουμε να διατηρηθούν κατά την εξέλιξη του προγράμματος [12] ή να οδηγήσουμε την μάθηση του προγράμματος και την αναδημιουργία του [6].

Στην πράξη Εστω ότι ο προγραμματιστής χρησιμοποιεί την βιβλιοθήκη *gRPC* για να πραγματοποιεί σειριοποίηση (serializing) και από-σειριοποίηση (de-serializing) σε αντικείμενα [41]. Για να μπορέσει να χρησιμοποιηθεί αυτή η βιβλιοθήκη, είναι απαραίτητη η χρήση πίνακα προδιαγραφών (protocol-buffer) ο οποίος θα περιγράφει τους τύπους των μεταβλητών που θα χρησιμοποιηθούν. Άρα, στην περίπτωση που ο χρήστης έχει κάποια βιβλιοθήκη όπως, *bignum* ή *crypto*, πρέπει πρώτα να την καλέσει χειροκίνητα και να σημειώσει τους τύπους των εισόδων και τους τύπους των στοιχείων που επιστρέφονται, για να συμπληρωθεί ο πίνακας των προδιαγραφών.

Ανάλυση Όλα αυτά μπορούν να γίνουν αυτόματα με την χρήση της αδρομερούς ανάλυσης τύπων.

Παράδειγμα Για να γίνει κατανοητή η χρήση των τύπων, παραθέτω ένα παράδειγμα μιας βιβλιοθήκης που καλείται με διαφορετικές εισόδους και εξόδους και τους τύπους που επιστρέφει.

Κώδικας 5.10: Εργαλειοθήκη

```
module.exports= {
    add: (a, b) => a + b,
    sub: (a, b) => a - b,
    toUpperCase: (a) => a.toUpperCase(),
}
```

Κώδικας 5.11: Κύριο Πρόγραμμα

```
const toolbox = require('./toolbox.js');

toolbox.add(1, 3);
toolbox.sub(3, 1);
toolbox.toUpperCase('Hello');
```

Για να μπορέσουμε να αναλύσουμε τους τύπους των συναρτήσεων, θα πρέπει να εφαρμόσουμε την αδρομερή ανάλυση τύπων που έχει τον παρακάτω κώδικα:

Κώδικας 5.12: Ανάλυση τύπων

```
let env;
const fs = require('fs');
const types = [];

const save = (file, name, types) => {
  const saveData = {core: types};
  if (Object.prototype.hasOwnProperty.
      call(file, name) === false) {
    file[name] = saveData;
  }
};

const onCallPre = (info) => {
  const inputType = [];
  if (!info.argumentsList.length) {
    inputType.push('no-input');
  } else {
    for (let i = 0; i < info.argumentsList.length; i++) {
      inputType.push(typeof info.argumentsList[i]);
    }
  }
  types[info.nameToStore] = inputType;
};
```

```
const onCallPost = (info) => {
  types[info.nameToStore].push(info.result ?
      typeof info.result : 'no output');
  save(env.analysisResult[info.currentModule],
      info.nameToStore, types[info.nameToStore]);
};

const onExit = (intersection, candidateModule) => {
  if (env.conf.SAVE_RESULTS) {
    fs.writeFileSync(env.conf.SAVE_RESULTS,
        JSON.stringify(env.analysisResult, null, 2), 'utf-8');
  }
  if (env.conf.print) {
    console.log(JSON.stringify(env.analysisResult, null, 2));
  }
};

module.exports = (e) => {
  env = e;
  env.conf.context.include = ['module-returns'];
  return {
    onCallPre: onCallPre,
    onCallPost: onCallPost,
    onExit: onExit,
  };
};
```

Βλέπουμε ότι η παραπάνω ανάλυση χρησιμοποιεί τα άγκιστρα (`onCallPre, onCallPost, onExit`) που προσφέρει η LYA, και δημιουργεί την ανάλυση με 45 γραμμές κώδικα. Ο κώδικας της ανάλυσης πραγματοποιεί τις εξής λειτουργίες:

- Καλεί το άγκιστρο `onCallPre` πριν την εκτέλεση κάποιας συνάρτησης και αποθηκεύει στην μνήμη τούς τύπους των ορισμάτων που δόθηκαν ως είσοδο στη συνάρτηση.
- Καλεί το άγκιστρο `onCallPost` μετά την εκτέλεση κάποιας συνάρτησης και αποθηκεύει στην μνήμη τούς τύπους των αποτελεσμάτων που δόθηκαν στην έξοδο από την συνάρτηση.
- Καλεί την συνάρτηση `save`, για να αποθηκευτούν στο αρχείο οι τύποι των ορισμάτων της εισόδου όσο και οι τύποι των αποτελεσμάτων της εξόδου.
- Καλεί το άγκιστρο `onExit` στο τέλος του προγράμματος, τυπώνοντας στην οθόνη ή/και αποθηκεύοντας σε κάποιο αρχείο το αποτέλεσμα της ανάλυσης.

Οπότε εφαρμόζοντας την ανάλυση τύπων στην παραπάνω περίπτωση καταλήγουμε στο εξής αποτέλεσμα:

Κώδικας 5.13: Αποτέλεσμα Ανάλυσης Τύπων

```
"Path/to/program": {
    "require('./toolbox.js').add": {
```

```
      "core": [
        "number",
        "number",
        "number"
      ]
    },
    "require('./toolbox.js').sub": {
      "core": [
        "number",
        "number",
        "number"
      ]
    },
    "require('./toolbox.js').toUpperCase": {
      "core": [
        "string",
        "string"
      ]
    }
}
```

5.4 Υπόλοιπες αναλύσεις

Μπορεί να γραφεί πληθώρα αναλύσεων με την λογική της αδρομερούς ανάλυσης και την χρήση του προαναφερθέντος εργαλείου. Μία ακόμα από τις δυνατότητες που προσφέρονται είναι η πρόσβαση και επεξεργασία του πηγαίου κώδικα των βιβλιοθηκών. Με την χρήση αυτής της δυνατότητας δημιουργήθηκε η ανάλυση αφαίρεσης σχολίων (*uncomment*) που δέχεται τον πηγαίο κώδικα σαν κείμενο και αφαιρεί όλα τα σχόλια που συναντά. Μία άλλη ανάλυση, δημιουργεί ευρετήριο (*term-index*) με βάση τον πηγαίο κώδικα της εκάστοτε βιβλιοθήκης.

Κώδικας 5.14: Ανάλυση Αφαίρεσης Σχολίων

```
const sourceTransform = (src) =>
        src.replace(/\/\*[\s\S]*?\*\/|\/\/.*/g, '');
```

Ένα άλλο ιδιαίτερα χρήσιμο εργαλείο είναι η ανίχνευση των βιβλιοθηκών που καλούνται. Μία τέτοια ανάλυση είναι η αδρομερής ανάλυση κλήσης βιβλιοθηκών. Αυτή η ανάλυση καταγράφει την κλήση όλων των διαφορετικών βιβλιοθηκών για την χρήση του προγραμματιστή.

Κώδικας 5.15: Ανάλυση Εισαγωγής Βιβλιοθηκών

```
const onImport = (caller, callee, name) => {
  console.log('lya:', caller, 'imports', callee, name);
};
```

5.5 Συγγραφή αναλύσεων

Πέρα από την δυνατότητα που προσφέρεται στον χρήστη να χρησιμοποιήσει έτοιμες αναλύσεις μπορεί να γράψει και καινούργιες αδρομερείς αναλύσεις που εξυπηρετούν τα δικά του ερευνητικά ή εμπορικά ενδιαφέροντα. Οι αναλύσεις γράφονται με συγκεκριμένο τρόπο, ώστε να αξιοποιούνται οι δυνατότητες του εργαλείου.

Ο τρόπος με τον οποίο προτείνεται να γίνεται η συγγραφή των αναλύσεων είναι με την χρήση του αρχείου που προσφέρεται ως βάση (*sample.js*). Αρχίζοντας από εκεί μπορούν να αξιοποιηθούν τα διαφορετικά άγκιστρα (*hooks*) που προσφέρονται. Το καθένα από αυτά αναλύεται παρακάτω:

1. **`sourceTransform`** (Μετατροπή Πηγαίου Κώδικα): Προσφέρει τον πηγαίο κώδικα των βιβλιοθηκών που καλούνται.

2. **`onImport`** (Στην εισαγωγή): Κάθε φορά που θα κληθεί κάποια βιβλιοθήκη ενεργοποιείται αυτό το άγκιστρο. Η λειτουργικότητα είναι διατηρητέα στην περίπτωση βιβλιοθήκης τρίτου αλλά και του συστήματος (όπως *fs*).

3. **`onRead`** (Στο διάβασμα): Κάθε φορά που γίνεται ανάγνωση κάποιου στοιχείου αυτό το άγκιστρο ενεργοποιείται.

4. **`onWrite`** (Στο γράψιμο): Κάθε φορά που πραγματοποιείται καταγραφή κάποιου στοιχείου αυτό το άγκιστρο ενεργοποιείται.

5. **`onCallPre`** (Πριν Την Εκτέλεση): Πριν την εκτέλεση κάποια συνάρτησης αυτό το άγκιστρο ενεργοποιείται.

6. **`onCallPost`** (Μετά την εκτέλεση): Αμέσως μετά την εκτέλεση κάποιας συνάρτησης αυτό το άγκιστρο ενεργοποιείται.

7. **`onConstruct`** (Στην δημιουργία): Κάθε φορά που χρησιμοποιείται ο τελεστής *new* αυτό το άγκιστρο ενεργοποιείται.

8. **`onHas`** (Στην ύπαρξη): Αυτό το άγκιστρο ενεργοποιείται κάθε φορά που το πρόγραμμα κάνει έλεγχο ύπαρξης κάποιας μεταβλητής. Είναι μία ιδιαίτερα προχωρημένη λειτουργικότητα που προσφέρει την δυνατότητα να καταγράφουμε τις δηλώσεις των καθολικών μεταβλητών από την πρώτη δήλωσή τους.

9. **`onExit`** (Στην έξοδο): Κάθε φορά που τελειώνει η εκτέλεση του προγράμματος που εκτελείται, αυτό το άγκιστρο ενεργοποιείται.

Μερικά παραδείγματα χρήσης των παραπάνω, για την δημιουργία αναλύσεων είναι το άγκιστρο *μετατροπής πηγαίου κώδικα* στην ανάλυση που προσφέρει την αφαίρεση σχολίων. Ένα άλλο παράδειγμα, είναι η χρήση των αγκίστρων *πριν/μετά την εκτέλεση* στην αδρομερή ανάλυση απόδοσης. Όταν καλείται μία συνάρτηση, αποθηκεύεται το χρονικό στιγμιότυπο όπου έγινε η εκτέλεση και το χρονικό στιγμιότυπο στο οποίο ολοκληρώθηκε η εκτέλεση. Με αυτές τις δύο πληροφορίες μπορούμε και συμπεραίνουμε το χρόνο εκτέλεσης μίας συνάρτησης.

Κεφάλαιο 6

Χρήση εργαλείου

Την Lya προβλέπεται να την χρησιμοποιήσουν δύο κατηγορίες χρηστών. Στην πρώτη κατηγορία ανήκουν οι χρήστες οι οποίοι επιθυμούν να την χρησιμοποιήσουν προγραμματιστικά ενώ στη δεύτερη κατηγορία ανήκουν οι χρήστες οι οποίοι επιθυμούν να την χρησιμοποιήσουν μέσω του εργαλείου της γραμμής εντολών.

6.1 Προγραμματιστικά

Η Lya μπορεί να χρησιμοποιηθεί προγραμματιστικά (δηλαδή σε ένα πρόγραμμα `Javascript`), είτε εισάγοντας την στην πρώτη γραμμή του αρχείου που θέλουμε να αναλύσουμε, είτε δημιουργώντας ένα νέο αρχείο και καλώντας το. Μπορούμε να περάσουμε ένα σύνολο ρυθμίσεων που θα προσδιορίζουν τις διαφορετικές συνθήκες και τις διαφορετικές επιλογές της ανάλυσης που θα εφαρμόσουμε. Θα εξηγήσουμε κάθε μία εντολή ξεχωριστά παρακάτω.

1. **`analysis`** (Ανάλυση): Προσδιορισμός της ανάλυσης που θέλουμε να φορτωθεί.
2. **`output path`** (Αρχείο Εξόδου): Προσδιορισμός της διαδρομής που θα αποθηκεύσουμε τα αποτελέσματα της ανάλυσης.
3. **`depth`** (Επίπεδο Ανάλυσης): Προσδιορισμός του επιπέδου ανάλυσης που διεξάγουμε. Η αύξηση του επιπέδου προσφέρει πιο λεπτομερείς αναλύσεις, αλλά ταυτόχρονα αυξάνει το κόστος εκτέλεσης.
4. **`print code`** (Εμφάνιση Κώδικα): Εμφάνιση του πηγαίου κώδικα της βιβλιοθήκης που κάνουμε εισαγωγή στο πρόγραμμά μας.
5. **`context`** (Γενικό Πλαίσιο):
   - **Ενεργοποίηση λειτουργίας** *`with`*: Προσδιορισμός της χρήσης `with` λειτουργίας.
   - **`include`** (Εισαγωγή): Σύνολο από στοιχεία πλαισίου που θέλουμε να καταγράψουμε στην ανάλυση, διαχωρισμένα με κόμμα.
   - **`exclude`** (Εξαγωγή): Σύνολο από στοιχεία πλαισίου που δεν θέλουμε να καταγράψουμε στην ανάλυση, διαχωρισμένα με κόμμα.
6. **`libraries`** (Βιβλιοθήκες):

- **include** (Εισαγωγή): Σύνολο από βιβλιοθήκες που θέλουμε να καταγράψουμε στην ανάλυση, διαχωρισμένες με κόμμα.
- **exclude** (Εξαγωγή): Σύνολο από βιβλιοθήκες που δεν θέλουμε να καταγράψουμε στην ανάλυση, διαχωρισμένες με κόμμα.

7. **fields** (Πεδία):
   - **include** (Εισαγωγή): Σύνολο από πεδία που θέλουμε να καταγράψουμε στην ανάλυση, διαχωρισμένα με κόμμα.
   - **exclude** (Εξαγωγή): Σύνολο από πεδία που δεν θέλουμε να καταγράψουμε στην ανάλυση, διαχωρισμένα με κόμμα.

Παράδειγμα Έστω ότι έχουμε την παρακάτω βιβλιοθήκη που χρησιμοποιείται για απλές μαθηματικές πράξεις. Την εισάγουμε σε ένα πρόγραμμα που έχει ως στόχο να υπολογίζει μία ακολουθία Fibonacci.

Κώδικας 6.1: Μαθηματική Βιβλιοθήκη

```
module.exports = {
    add: (a, b) => a + b,
    sub: (a, b) => a - b,
    mult: (a, b) => a * b,
}
```

Κώδικας 6.2: Πρόγραμμα Fibonacci

```
var math = require("math.js");
var fibonacci_series = function (n)
{
  if (n==1)
  {
    return [0, 1];
  }
  else
  {
    var s = fibonacci_series(math.sub(n, 1));
    s.push(math.add(s[s.length - 1], s[s.length - 2]));
    return s;
  }
};

console.log(fibonacci_series(8));
```

Παρακάτω θα δείξουμε τους δύο τρόπους με τους οποίους μπορεί να αναλυθεί ο παραπάνω κώδικας.

Κώδικας 6.3: Κώδικας Ανάλυσης σε νέο αρχείο

```
var lya = require("../../../src/core.js");
```

```
var conf = {
   analysis: lya.preset.ON_OFF,
   save_results: require("path").join(__dirname, "dynamic.json"),
};
lya.configRequire(require, conf);
require("/Path/To/fibonacci.js");
```

Ο παραπάνω κώδικας του παραδείγματος θα εφαρμόσει την ανάλυση `on-off` και θα αποθηκεύσει το αποτέλεσμα σε ένα αρχείο με το όνομα `dynamic.json`.

Κώδικας 6.4: Κώδικας ανάλυσης στο ήδη υπάρχον αρχείο

```
var lya = require("Path/To/core.js");
var conf = {
   analysis: lya.preset.ON_OFF,
   save_results: require("path").join(__dirname, "dynamic.json"),
};
lya.configRequire(require, conf);

var math = require("math.js");
var fibonacci_series = function (n)
{
  if (n==1)
  {
    return [0, 1];
  }
  else
  {
    var s = fibonacci_series(math.sub(n, 1));
    s.push(math.add(s[s.length - 1], s[s.length - 2]));
    return s;
  }
};

console.log(fibonacci_series(8));
```

Στην χρήση της ανάλυσης μέσω νέου αρχείου, αναλύεται και ο κώδικας του προγράμματος `fibonacci.js` ενώ όταν χρησιμοποιείτε η ανάλυση στο ίδιο αρχείο, αναλύεται μόνο το αρχείο `math.js`.

6.2 Χρήση εργαλείου της γραμμής εντολών

Με την χρήση του εργαλείου της γραμμής εντολών μπορούμε να παράγουμε παρόμοιες αναλύσεις με τον προγραμματιστικό τρόπο. Οι κανόνες που διέπουν αυτό το εργαλείο είναι παρόμοιες με το προγραμματιστικό. Δέχεται ένα πλήθος ορισμάτων που παρατίθενται αναλυτικά παρακάτω:

1. **`--help, -h`** (Βοήθεια): Προβάλει στην γραμμή εντολών την βοήθεια και το σύνολο τον εντολών που μπορούν χρησιμοποιηθούν.
2. **`--version, -v`** (Έκδοση): Προβάλει την έκδοση του εργαλείου.

3. `--depth, -d` (Επίπεδο Ανάλυσης): Προσδιορισμός του επιπέδου ανάλυσης που διεξάγουμε. Η αύξηση του επιπέδου προσφέρει πιο λεπτομερείς αναλύσεις, αλλά ταυτόχρονα αυξάνει το κόστος εκτέλεσης.

4. `--analysis, -a` (Ανάλυση): Προσδιορισμός της ανάλυσης που θέλουμε να φορτωθεί.

5. `--save, -s` (Αρχείο Εξόδου): Προσδιορισμός της διαδρομής που θα αποθηκεύσουμε τα αποτελέσματα της ανάλυσης.

6. `--rules, -r` (Κανόνες): Προσδιορισμός της διαδρομής προς το αρχείο που περιέχει τους κανόνες επιβολής για την εκάστοτε ανάλυση.

7. `--print, -p` (Εμφάνιση κώδικα): Εμφάνιση του πηγαίου κώδικα της βιβλιοθήκης που κάνουμε εισαγωγή στο πρόγραμμά μας.

8. `--print-prologue` (Εμφάνιση προλόγου): Εκτύπωση του προλόγου στο πρόγραμμά μας.

9. `context` (Γενικό πλαίσιο):

   - **Ενεργοποίηση λειτουργίας *with***: Προσδιορισμός της χρήσης *with* λειτουργίας.
   - `--context-include` (Εισαγωγή): Σύνολο από στοιχεία πλαισίου που θέλουμε να καταγράψουμε στην ανάλυση, διαχωρισμένα με κόμμα.
   - `--context-exclude` (Εξαγωγή): Σύνολο από στοιχεία πλαισίου που δεν θέλουμε να καταγράψουμε στην ανάλυση, διαχωρισμένα με κόμμα.

10. `libraries` (Βιβλιοθήκες):

    - `--libraries-include` (Εισαγωγή): Σύνολο από βιβλιοθήκες που θέλουμε να καταγράψουμε στην ανάλυση, διαχωρισμένες με κόμμα.
    - `--libraries-exclude` (Εξαγωγή): Σύνολο από βιβλιοθήκες που δεν θέλουμε να καταγράψουμε στην ανάλυση, διαχωρισμένες με κόμμα.

11. `fields` (Πεδία):

    - `--fields-include` (Εισαγωγή): Σύνολο από πεδία που θέλουμε να καταγράψουμε στην ανάλυση, διαχωρισμένα με κόμμα.
    - `--fields-exclude` (Εξαγωγή): Σύνολο από πεδία που δεν θέλουμε να καταγράψουμε στην ανάλυση, διαχωρισμένα με κόμμα.

Παράδειγμα Χρησιμοποιούμε το ίδιο πρόγραμμα από την προηγούμενη ενότητα, το οποίο υπολογίζει μία ακολουθία Fibonacci. Για να μπορέσουμε να αναλύσουμε τον κώδικα, αρκεί να τρέξουμε την παρακάτω εντολή από την γραμμή εντολών.

Κώδικας 6.5: Εντολή Εργαλείου Γραμμής Εντολών

```
lya fibonacci.js -a on-off -p
```

Όπως βλέπουμε παραπάνω το όνομα του προγράμματος που αναλύουμε είναι `fibonacci.js` και η εντολή του παραδείγματος θα εφαρμόσει την ανάλυση (-a) *on-off* και θα τυπώσει (-p) στην οθόνη μας το αποτέλεσμα της ανάλυσης.

Κεφάλαιο 7

Αξιολόγηση

Σε αυτό το κεφάλαιο θέλουμε να απαντήσουμε στις εξής ερωτήσεις:

1. Ποια είναι η απόδοση του εργαλείου αδρομερούς ανάλυσης όσον αφορά την δυνατότητα ανίχνευσης προβλημάτων, σε σύγκριση με προηγούμενες υλοποιήσεις δυναμικής ανάλυσης;
2. Ποιο είναι το χρονικό κόστος χρήσης και πώς συγκρίνεται με προηγούμενες υλοποιήσεις δυναμικής ανάλυσης;
3. Πόσες γραμμές κώδικα (LoC) απαιτούνται για την υλοποίηση των αναλύσεων και πως συγκρίνεται με προηγούμενες υλοποιήσεις δυναμικής ανάλυσης;

Για να μπορέσουν να απαντηθούν οι παραπάνω ερωτήσεις τρέχουμε πληθώρα διαφορετικών σημείων αναφοράς (*benchmarks*). Αναπτύξαμε τρεις διαφορετικές αναλύσεις και τις εφαρμόσαμε σε μικρές βιβλιοθήκες (*micro-packages*) αλλά και σε μεγαλύτερα προγράμματα. Οι γραμμές κώδικα που απαιτήθηκαν για να δημιουργηθούν οι αναλύσεις αυτές ήταν κατά μέσο όρο 94, στην γλώσσα προγραμματισμού `Javascript` και προκαλούν καθυστέρηση χαμηλότερη από 5%. Εφαρμόζουμε τις αναλύσεις σε προβλήματα που δηλώνονται μέσω της ιστοσελίδας `Github` και μέσω της βάσης δεδομένων Κοινών Ευπαθειών και Προβλημάτων (`Common Vulnerabilities and Exposures`), επιβεβαιώνοντας ότι ανιχνεύονται από το εργαλείο. Εφαρμόζοντας το εργαλείο που δημιουργήθηκε στο σύνολο των σημείων αναφοράς `Sunspider` του `Jalangi` [34] παρατηρείται ότι το κόστος χρήσης είναι κατά μέσο όρο 87 φορές χαμηλότερο από το `Jalangi`.

7.1 Μηχάνημα μετρήσεων

Το μηχάνημα που χρησιμοποιήθηκε για να πραγματοποιηθούν οι μετρήσεις ήταν ένας διακομιστής (`server`) με τα εξής τεχνικά χαρακτηριστικά:

- 2 Intel Core 2 Duo E8600 CPUs χρονισμένο στα 3.33GHz.
- 4GB μνήμης.

Παράλληλα τα προγράμματα που χρησιμοποιήθηκαν εσωτερικά του εξυπηρετητή είναι:

- Λειτουργικό σύστημα `Ubuntu Linux` με πυρήνα (`kernel`) v4.4.0-134.
- `Docker` έκδοσης 18.09.7.
- Εσωτερικά του `Docker` τρέχει η έκδοση `Ubuntu` 14.04.6.

- `Jalangi` έκδοση 1.
- Περιβάλλον `Node.js` έκδοσης 8.9.4 (σε συνδυασμό με `V8` έκδοσης 6.1.534.50, `LibUV` έκδοσης 1.15.0, και `npm` έκδοσης 6.4.1).

Όλα τα αποτελέσματα δηλώνονται σε μιλιδευτερόλεπτα (*ms*) και η κάθε δοκιμή τρέχει 1000 φορές. Το $A\Sigma$, ΔA και AT αναφέρονται αντίστοιχα σε ανάλυση ασφαλείας, διαγνωστικό απόδοσης και σε ανακάλυψη τύπων.

7.2 Αναλύσεις

Παρακάτω θα εξηγηθούν οι αναλύσεις που χρησιμοποιήθηκαν για τα πειράματα, και θα απαντηθεί το ερώτημα αν προσφέρουν ουσιαστικά οφέλη στην ανίχνευση προβλημάτων, που μπορεί να διαφύγουν από αναλύσεις που δεν μπορούν να τρέξουν στην παραγωγή. Κατά μέσο όρο οι παρακάτω αναλύσεις έχουν 88 γραμμές κώδικα με ένα σημαντικό μέρος τους (περίπου 20%) όμως να είναι ίδιο μεταξύ τους.

Ανάλυση ασφαλείας Για να μπορέσουμε να αντιμετωπίσουμε τα προβλήματα ασφαλείας που δημιουργεί η χρήση βιβλιοθηκών λογισμικού τρίτων δημιουργήσαμε μία μορφή ενεργοποίησης/απενεργοποίησης (*on/off*) αδρομερούς ανάλυσης (92 γραμμών κώδικα) που αναλύει προσβάσεις σε όλες τις βιβλιοθήκες που εισάγονται. Η ανάλυση αντιμετωπίζει τις ενσωματωμένες βιβλιοθήκες, τις καθολικές μεταβλητές και τις συναρτήσεις σαν στοιχεία που καταγράφονται σε ένα μοντέλο ενεργοποίησης (*on*), σε περίπτωση που ανιχνευτεί πρόσβαση, και απενεργοποίησης (*off*) σε περίπτωση που θέλουμε να αποτρέψουμε την πρόσβαση σε κάποιο στοιχείο. Με την εκτέλεση του προγράμματος δημιουργείται το μοντέλο ανάλογα τις προσβάσεις που πραγματοποιεί την ώρα της εκτέλεσης. Μερικά παραδείγματα προσβάσεων που θα καταγραφούν είναι:

1. Το διάβασμα κάποιας μεταβλητής, ακόμα και η ανάθεσή της σε κάποια άλλη μεταβλητή ή ανάθεσή της σε κάποιο άλλο πακέτο/`module` (επιφέρει τον χαρακτηρισμό *on*).
2. Η μορφοποίηση ή διαγραφή κάποιας μεταβλητής (επιφέρει τον χαρακτηρισμό *on*).
3. Η εκτέλεση κάποιας συνάρτησης ή το κάλεσμα του κατασκευαστή *constructor* (επιφέρει τον χαρακτηρισμό *on*).
4. Η απαγόρευση πρόσβαση σε κάποιο αρχείο που περιέχει ευαίσθητες πληροφορίες (επιφέρει τον χαρακτηρισμό *off*).

Τα αποτελέσματα της ανάλυσης αποδίδονται σε σύνολο από χάρτες (*maps*), ένα ανά βιβλιοθήκη, όπου αρχειοθετούνται με βάση την διαδρομή του αντικειμένου. Για παράδειγμα η βιβλιοθήκη `Math` που διαβάζεται και εκτελεί την συνάρτηση αθροίσματος (`Add`) θα έχει την εξής μορφή,
`require("Math").add: on`.

Ανάλυση απόδοσης Δημιουργούμε μία ανάλυση απόδοσης (87 γραμμών κώδικα) που λειτουργεί σε δύο επίπεδα:

1. Περιτυλίγουμε σε επίπεδο βιβλιοθηκών τις κλήσεις των συναρτήσεων και διατηρούμε στατιστικά για τις χρονικές στιγμές που πραγματοποιήθηκαν οι κλήσεις τους.
2. Αμέσως μετά το πέρας της εκτέλεσης των συναρτήσεων, εκτελείται μία μέθοδος που χρησιμοποιώντας τις χρονικές στιγμές που κλήθηκαν οι συναρτήσεις δημιουργεί ένα μοντέλο που απεικονίζει το φόρτο εκτέλεσης του προγράμματος.

Αυτή η ανάλυση διατηρεί πληροφορία για τις κλήσεις των συναρτήσεων, αγνοώντας όλα τα υπόλοιπα στοιχεία που υπάρχει η δυνατότητα να καταγραφούν. Οι συναρτήσεις περιτυλίγονται σε έναν πρόλογο και έναν επίλογο που διατηρεί χρόνους με την χρήση των χρόνων εκτέλεσης του `Node.js`.

Εξαγωγή τύπων Για να υπάρξει η δυνατότητα να μοντελοποιεί η ανάλυση εξαγωγής τύπων (86 γραμμών κώδικα) χρησιμοποιήθηκε σαν βάση το απλό μοντέλο λογισμού λάμδα (*lambda calculus*). Πιο συγκεκριμένα:

1. Ενώσεις μεταξύ στοιχείων, όπως *string* — *number*.
2. Τύποι που χρησιμοποιούνται στην γλώσσα προγραμματισμού της *Javascript*. Είναι όλοι οι τύποι που μπορούν να επικαλεστούν με την χρήση του τελεστή `typeOf`. Μερικά παραδείγματα είναι: `null, NaN,` ή `undefined`.
3. Οι πραγματικοί τύποι τιμών που δεν μπορούν να σειριοποιηθούν, όπως το `console.log`.

Οι ενώσεις μεταξύ των στοιχείων είναι χρήσιμες όταν κάποια βιβλιοθήκη ή κάποια συνάρτησή της καλείται με διαφορετικούς τύπους ορισμάτων. Όμως στην πράξη παρατηρείται ότι αυτό είναι σπάνιο και συνήθως έχουν ένα τύπο ορισμάτων εισόδου σε όλη την διάρκεια χρήσης τους [12].

7.3 Αναλύοντας εφαρμογές

Παρακάτω θα δείξουμε: ένα παράδειγμα από προβλήματα ασφαλείας που μπορεί να διερευνηθεί, ένα παράδειγμα ελέγχου απόδοσης σε μία δημοφιλή βιβλιοθήκη και τέλος πως μπορούμε να αξιοποιήσουμε τους τύπους των εφαρμογών που εξάγουμε από την ανάλυση.

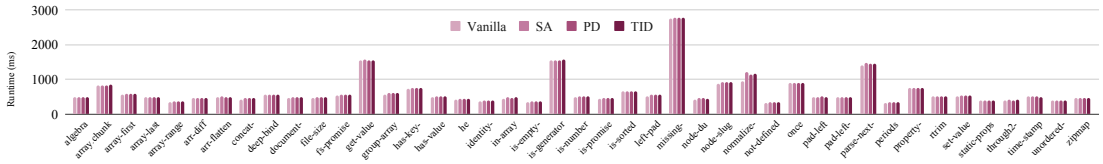
Παράδειγμα ανάλυσης ασφαλείας Θα εφαρμόσουμε την ανάλυση ΔΓΕ στο δημοφιλές πακέτο `safe-eval` (έκδοση 0.3), μία βιβλιοθήκη που έχει ως στόχο την απομόνωση του εκτελεστέου κώδικα, πριν την εκτέλεσή του, με την συνάρτηση `eval`. Μόλις τρέξουμε τον κώδικα μαζί με την ανάλυση, ελέγχουμε χειροκίνητα ότι ο κώδικας δεν έχει ξεφύγει από την απομόνωση που προσφέρεται.

Σε αναφορές που έχουν γίνει στο `Snyk` [3], μία δημόσια βάση δεδομένων που έχει στόχο την δημοσίευση αδυναμιών ασφαλείας, βλέπουμε ότι υπάρχουν διάφορα προβλήματα ασφαλείας που αφορούν το παραπάνω πακέτο. Ελέγχουμε ένα παράδειγμα που υπάρχει μέσα σε μία από αυτές τις δημοσιεύσεις και βλέπουμε ότι παραβιάζεται αυτή η απομόνωση που προσφέρεται από το πακέτο με την χρήση του στοιχείου `process`, μέσω της πρωτότυπης αλυσίδας (`prototype chain`), αξιοποιώντας το `child_proceess` για να εκτελεστεί η εντολή `whoami`.

Αξιοποιώντας όμως την αδρομερή ανάλυση ενεργοποίησης/απενεργοποίησης μπορούμε να δούμε ότι αυτές οι παραβιάσεις του πρωτοκόλλου του πακέτου γίνονται αναφορά. Μερικές παραβιάσεις που θα αναγνωριστούν είναι το *διάβασμα* της πρωτότυπης αλυσίδας, η *εκτέλεση* της εισαγωγής `require` και το *διάβασμα* του `child_process`.

Παράδειγμα ανάλυσης απόδοσης Εφαρμόζουμε το εργαλείο αδρομερούς ανάλυσης σε συνδυασμό με την ανάλυση απόδοσης στην βιβλιοθήκη `uri-js` (έκδοσης 2.1.1) [8]. Το πακέτο αυτό αναλαμβάνει να κάνει ανάλυση και επικύρωση σε συνδέσμους ιστοτόπων και είναι αρκετά γρήγορο στη μέση περίπτωση. Τρέχοντας ένα φόρτο εργασίας με 5000 αιτήσεις ανά δευτερόλεπτο σε ένα πακέτο δεδομένων δοκιμής που προσφέρει η βιβλιοθήκη, βλέπουμε ότι οι απαντήσεις επιστρέφονται κατά μέσο όρο σε *34,3 ms*.

Αναζητώντας μεταξύ διαφόρων δημόσιων προβλημάτων που έχουν εμφανιστεί στον ιστοτόπο `GitHub`, παρατηρούμε ότι με συγκεκριμένα παθολογικά δεδομένα εισόδου [11], το πακέτο `uri-js`



Γράφημα 7.1: Εφαρμογή της Lya πάνω σε δημοφιλείς βιβλιοθήκες. Παρουσιάζονται οι διαφορές μεταξύ: της κανονικής εκτέλεσης, ανάλυσης ασφαλείας, ανάλυσης απόδοσης και ανάλυσης τύπων.

αντιμετωπίζει μέσο λανθάνοντα χρόνο της τάξεως των *400 ms* ανά σύνδεσμο. Έτσι σε περιπτώσεις που το 99% είναι καλοήθεις σύνδεσμοι ιστοτόπων και το 1% των συνδέσμων εμφανίζουν παθογένειες, η απόδοση του πακέτου μειώνεται σε *197 αιτήσεις* ανά δευτερόλεπτο.

Εφαρμόζοντας το εργαλείο που δημιουργήθηκε σε συνδυασμό με την ανάλογη ανάλυση (*ανάλυση απόδοσης*), όλες αυτές οι διαφοροποιήσεις στην απόδοση μπορούν να ανιχνευτούν.

Παράδειγμα εύρεσης τύπων Εφαρμόζουμε την αδρομερή ανάλυση εύρεσης τύπων στο ενσωματωμένο πακέτο της σειριοποίησης των `JSON` αρχείων. Επιλέγοντας μία τιμή από ένα συγκεκριμένο σύνολο διαφορετικών μεταβλητών, η Lya μπορεί να ανιχνεύσει κάποιο στοιχείο που χρησιμοποιείται παραπάνω από μία φορά, με διαφορετική τιμή, με αποτέλεσμα να δημιουργεί στην δομή του κύκλους. Παρότι τα στοιχεία των πειραμάτων που τρέχουμε δεν περιέχουν κύκλους, η Lya μπορεί να ανιχνεύσει αυτές τις περίεργες περιπτώσεις που περιέχονται κύκλοι και να ενημερώνει ότι πρέπει να χρησιμοποιηθούν διαφορετικά πακέτα σειριοποίησης και απο-σειριοποιήσης.

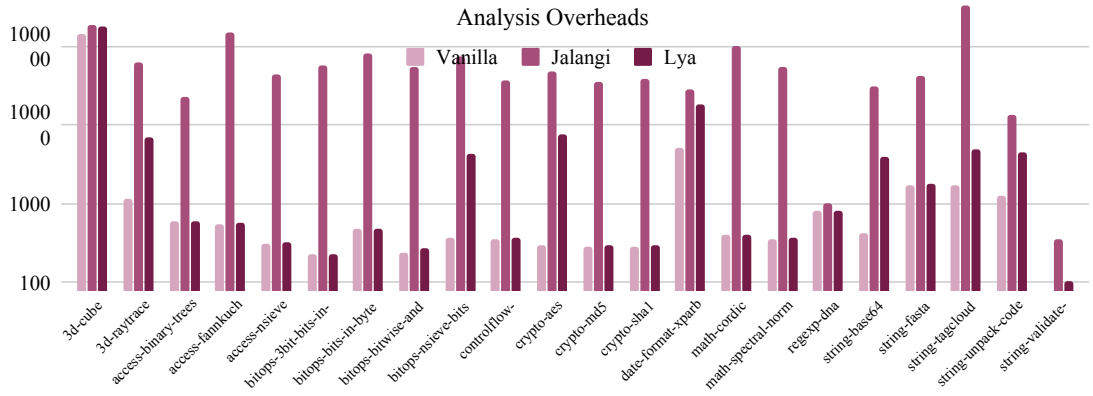
7.4 Απόδοση εργαλείου κατά την εκτέλεση

Για να μπορέσουμε να καταλάβουμε την απόδοση του εργαλείου που δημιουργήθηκε, πραγματοποιήσαμε δύο είδη πειραμάτων. Στο πρώτο πείραμα εφαρμόσαμε την Lya σε 50 δημοφιλείς βιβλιοθήκες από τον ιστοτόπο `npm`, και παρατηρήσαμε μέση καθυστέρηση 3.6–4.1%. Στο δεύτερο πείραμα, ελέγξαμε την απόδοση της Lya σε σύγκριση με το `Jalangi`, το κατεξοχήν σύστημα δυναμικών αναλύσεων. Τρέξαμε το σύνολο των ελέγχων πάνω στα πειράματα που συμπεριλαμβάνει το `Jalangi` και παρατηρήσαμε ότι η Lya τρέχει 2–3 τάξεις μεγέθους γρηγορότερα.

Εφαρμογή της Lya σε δημοφιλείς βιβλιοθήκες Στο πρώτο σύνολο από πειράματα που πραγματοποιήθηκαν, αξιολογήσαμε το σύστημά μας με την χρήση *50 δημοφιλών βιβλιοθηκών* της γλώσσας προγραμματισμού `Javascript`, από το οικοσύστημα του `npm`. Αυτά τα πακέτα προέρχονται από μία λίστα που παρέχει "μικρά, εξειδικευμένα `Node.js` πακέτα" [29], που προβάλλονται ταξινομημένα με βάση τα ποσοστά δημοφιλίας τους. Εμείς διαλέξαμε τα 50 πιο δημοφιλή, με μέσο όρο το καθένα 4.8 εκατομμύρια κατεβάσματα την βδομάδα (συνολικά 227 εκατομμύρια), και εξαρτώνται από αυτά 656 άλλες βιβλιοθήκες ή εφαρμογές κατά μέσο όρο (30000 συνολικά).

Για κάθε βιβλιοθήκη που δοκιμάστηκε, τρέξαμε τα πειράματα χρησιμοποιώντας την σουίτα δοκιμών που παρείχε το καθένα, μέσω της εντολής `npm test`. Λόγω του γεγονότος ότι αυτές οι βιβλιοθήκες είναι δημοφιλείς, οι συντηρητές τους έχουν δημιουργήσει αρκετές δοκιμές που στοχεύουν στην πιστοποίηση ελέγχου της λειτουργικότητας του εκάστοτε πακέτου. Αυτό προσδίδει δύο ενδιαφέροντα χαρακτηριστικά στην σουίτα των δοκιμών τους:

1. Εφόσον υπάρχει πληθώρα δοκιμών μέσα στην σουίτα, αυτό σημαίνει ότι ελέγχονται διαφορετικά κομμάτια της βιβλιοθήκης, με κάθε δοκιμή.



Γράφημα 7.2: Σύγκριση χρόνων στην δοκιμαστική σουίτα του `Jalangi`. Παρουσιάζονται οι διαφορές μεταξύ της απλής εκτέλεσης, της εκτέλεσης με την χρήση του `Jalangi` και με την χρήση της Lya

2. Ακόμα και αν κάθε πρόγραμμα που χρησιμοποιεί τις βιβλιοθήκες, χρησιμοποιεί μόνο ένα μικρό ποσοστό από την λειτουργικότητα που προσφέρει κάθε πακέτο, οι δοκιμές καλύπτουν το μεγαλύτερο μέρος των πιθανών τρόπων που μπορεί να χρησιμοποιηθεί. Παρατηρήθηκε ότι οι σουίτες δοκιμών έχουν μέχρι και 10 φορές μεγαλύτερο μέγεθος από το πακέτο που έχουν στόχο να ελέγξουν.

Η γραφική παράσταση **7.1** δείχνει το κόστος της εφαρμογής της Lya πάνω στα πακέτα που αναφέρονται παραπάνω. Ο παραπάνω χρόνος δίνεται με την μορφή του ποσοστού επί του συνολικού χρόνου. Ο χρόνος είναι ο συνολικός που απαιτείται για την εκτέλεση όλου του πακέτου, σε τρεις διαφορετικές αναλύσεις σε σύγκριση με τις βιβλιοθήκες χωρίς την χρήση αδρομερούς ανάλυσης. Παρατηρείται:

- **Αναλύσεις ασφάλειας**: Ο μέσος χρόνος καθυστέρησης είναι 4.14%, με μέγιστη τιμή 26.7% και ελάχιστη τιμή 0.13%.
- **Ανάλυση απόδοσης**: Ο μέσος χρόνος καθυστέρησης είναι 3.62%, με μέγιστη τιμή 19.86% και ελάχιστη τιμή 0.23%.
- **Ανάλυση τύπων**: Ο μέσος χρόνος καθυστέρησης είναι 3.86%, με μέγιστη τιμή 22.68% και ελάχιστη τιμή 1.21%.

Η γενική τιμή καθυστέρησης μεταξύ όλων τον αναλύσεων και όλων τον πειραμάτων είναι 3.8%. Στις εκτενέστερες βιβλιοθήκες, πραγματοποιούμε μια *ελαφρύτερη* καταμέτρηση των καθολικών προσβάσεων. Αυτό αναλύεται παρακάτω σε μεγαλύτερο βαθμό.

Σύγκριση της Lya με Jalangi Στο δεύτερο σύνολο από πειράματα ελέγχουμε την απόδοση της Lya σε σύγκριση με το `Jalangi` [34]. Το `Jalangi` είναι ένα δημοφιλές εργαλείο δυναμικής ανάλυσης προγραμμάτων για `Javascript`, που προσφέρει αναλύσεις ιδιαίτερα εξονυχιστικές, χρησιμοποιώντας κατά την εκτέλεση ένα προσαρμοσμένο διερμηνέα.

Για αυτά τα πειράματα χρησιμοποιήσαμε ένα διαφορετικό σύνολο από δοκιμές —την σουίτα δοκιμών του ίδιου του `Jalangi`. Αυτό το εφαρμόσαμε, καθώς το `Jalangi` δεν κατάφερε να τρέξει πάνω στα 50 πακέτα. Ο λόγος είναι ότι τα πακέτα χρησιμοποιούν τα νεότερα χαρακτηριστικά του `EcmaScript` —με μερικά παραδείγματα να είναι οι συναρτήσεις με βέλος, συμβολοσειρές προτύπων και άλλα. Ένα άλλο στοιχείο που δυσκολεύει την εκτέλεσή τους ήταν η χρήση του `npm test`,

καθώς εκτελείται από ένα πρόγραμμα εξωτερικά από το περιβάλλον του `Node.js`. Αντίθετα από το `Jalangi`, η Lya υποστηρίζει όλα αυτά τα χαρακτηριστικά και παρουσιάζει τα οφέλη να τρέχουμε σε ένα περιβάλλον εργασίας που δεν έχουμε επέμβει.

Οι κώδικες δοκιμών που προσφέρονται από το `Jalangi` συμπεριλαμβάνουν 26 προγράμματα από το πακέτο `Sun-Spider` [10], που εκτελούνται ως κομμάτια του περιβάλλοντος (`container`) [16] που παρέχεται από το ίδιο. Για να μπορέσουν τα πειράματα να είναι αντικειμενικά, και τα πειράματα της Lya έτρεξαν στο ίδιο περιβάλλον.

Σε επίπεδο ανάλυσης, και τα δύο συστήματα τρέχουν μία ανάλυση που καταγράφει τα όνομα των προσβάσεων σε καθολικές μεταβλητές. Η ανάλυση αυτή επιλέχθηκε, καθώς είναι ένα δείγμα χρήσιμης ανάλυσης που μπορεί να εκτελεστεί και από τα δύο περιβάλλοντα. Είναι χρήσιμη καθώς επιτρέπει να καταλάβουμε την λειτουργικότητα του προγράμματος και τις προσβάσεις στην καθολική κατάσταση του προγράμματος.

Τα αποτελέσματα δείχνουν ότι η Lya έχει σημαντικά ταχύτερη εκτέλεση από το `Jalangi`. Κατά μέσο όρο το εργαλείο που δημιουργήσαμε απαιτεί 87 φορές λιγότερο χρόνο για να εκτελεστεί (μέγιστη διαφορά είναι 266.1 φορές και η ελάχιστη διαφορά 3.1 φορές). Κατά μέσο όρο οι αναλύσεις στην Lya μπορούν να αποδοθούν με 4 γραμμές λιγότερο κώδικα.

7.5 Μικρο-αναλύσεις

Στην προηγούμενη ενότητα του κεφαλαίου παρουσιάσαμε τους λόγους που η αδρομερής ανάλυση προσφέρει σημαντικά οφέλη απόδοσης σε σύγκριση με τις τυπικές δυναμικές αναλύσεις. Στην τελευταία ενότητα παρουσιάζουμε μία σειρά μικρο-αναλύσεων που έχουν ως στόχο να εξηγήσουν τα βασικά αίτια των καθυστερήσεων στην Lya. Τα βασικά συμπεράσματα είναι:

1. Η μεγαλύτερη καθυστέρηση συμβαίνει καθώς σε ένα κομμάτι της υλοποίηση της Lya χρησιμοποιείται το χαρακτηριστικό `with` της `Javascript`. Αυτό το χαρακτηριστικό είναι χρήσιμο για την καταγραφή κάποιων αλλαγών που συμβαίνουν στις καθολικές μεταβλητές.

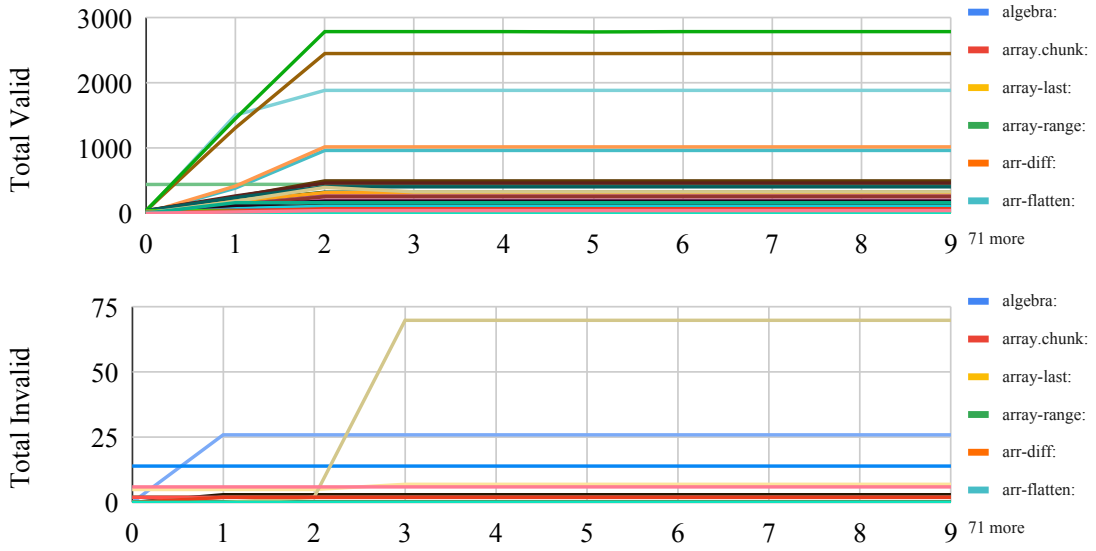
2. Οι γενικές παρεμβολές που συμβαίνουν έχουν αμελητέο κόστος.

3. Σε όσο περισσότερα επίπεδα εφαρμόζουμε ανάλυση τόσο περισσότερο αυξάνεται το χρονικό κόστος. Μάλιστα παρατηρούμε ότι αυτή η αύξηση συμβαίνει με εκθετικό ρυθμό.

4. Τα περισσότερα στοιχεία που τυλίγονται με τα άγκιστρα προέρχονται από το `Node.js` και από το `EcmaScript`, αντί για τα `imports` ή τις καθολικές μεταβλητές (`globals`).

Λόγοι καθυστέρησης Για να μπορέσουμε να κατανοήσουμε τους λόγους που παρουσιάζονται αυτές οι καθυστερήσεις, πραγματοποιούμε μία σειρά από μικρές δοκιμές, που καλούν διάφορες εσωτερικές βιβλιοθήκες χωρίς κάποια είσοδο. Ενεργοποιώντας διαφορετικά τμήματα της Lya, παρατηρούμε ότι ο βασικός λόγος των καθυστερήσεων συμβαίνει λόγω της χρήσης του `with` της `Javascript` —απενεργοποιώντας το, αποφεύγεται το 95% των καθυστερήσεων. Ο λόγος που το `with` δημιουργεί πληθώρα καθυστερήσεων είναι διπλός. Πρώτον, καταγράφει πληθώρα από διαφορετικές προσβάσεις —με μόνο ένα μικρό μέρος τους να είναι σχετικό με τις αναλύσεις της Lya. Δεύτερον, παραμένει αρκετά μη βελτιστοποιημένο και ακόμα και οι οδηγίες που προσφέρονται από τους δημιουργούς της `Javascript` προτείνουν την αποφυγή χρήσης του.

Επίπεδο ανάλυσης Για να καταλάβουμε την επίδραση του επιπέδου της ανάλυσης στην πράξη, στα γραφήματα 7.3 παρουσιάζεται το σύνολο από τα στοιχεία που έχουν περιτυλιχθεί, οι μοναδικές προσβάσεις σε αυτά και το νούμερο των συνολικών προσβάσεων σε όλες τις 50 βιβλιοθήκες. Το επίπεδο ανάλυσης παρουσιάζεται σε προηγούμενη ενότητα της εργασίας —περιληπτικά είναι κατά



Γράφημα 7.3: Σύνολο προσβάσεων με βάση το επίπεδο ανάλυσης. Από πάνω προς τα κάτω: Έγκυρες προσβάσεις, μη έγκυρες προσβάσεις σε συνάρτηση με το επίπεδο ανάλυσης.

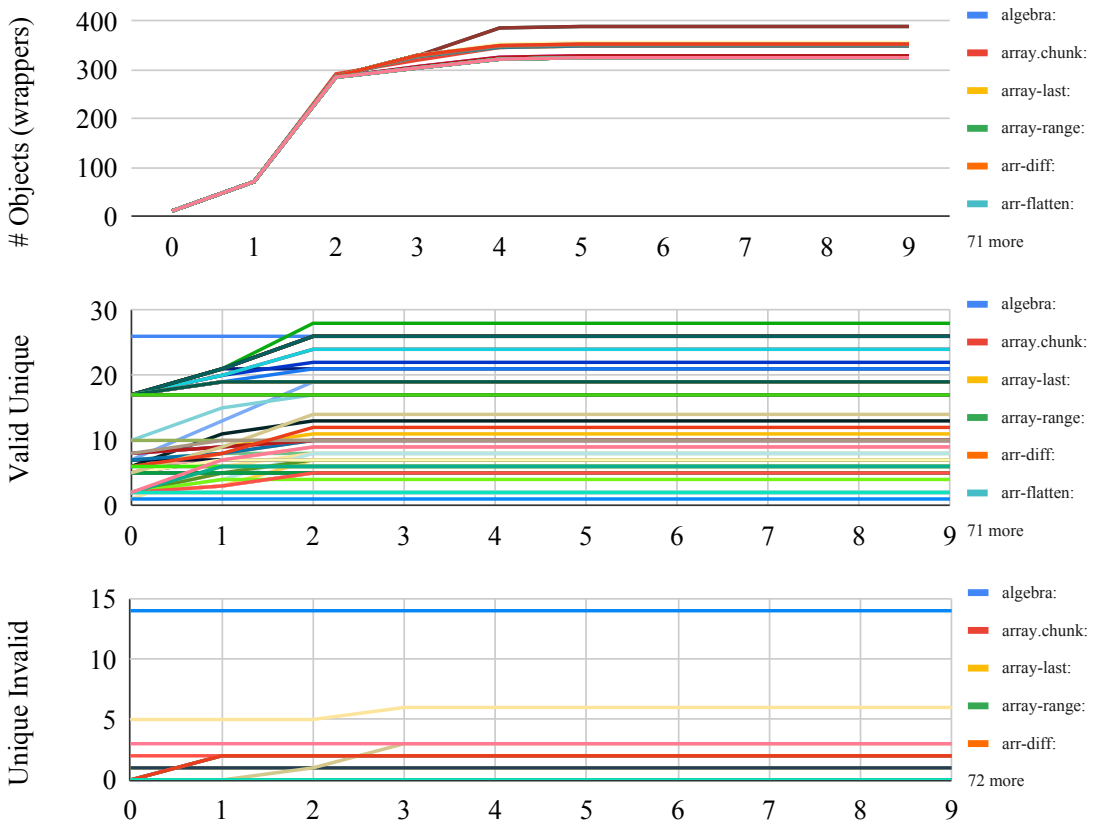
πόσο βαθιά μέσα στο δέντρο εξάρτησης η LYA θα περιτυλίξει αντικείμενα, ξεκινώντας από το όνομα που χρησιμοποιούμε για να δείξουμε αυτό το αντικείμενο. Για παράδειγμα, το `global.x.y` είναι στοιχείο 2 επιπέδων, ενώ το `console.log` είναι στοιχείο ενός επιπέδου.

Παρότι ο αριθμός των στοιχείων που περιτυλίγονται αυξάνεται αρκετά γρήγορα, αντίστοιχα γρήγορα φτάνει στο άνω όριο του, που είναι 400 στοιχεία. Αυξάνεται ταχύτατα στα πρώτα επίπεδα και μετά σταθεροποιείται περίπου στο επίπεδο 5. Αυτό συμβαίνει καθώς τα περισσότερα αντικείμενα έχουν προκαθορισμένο τρόπο με τον οποίο δηλώνονται τα στοιχεία και ακολουθείται σε γενικές γραμμές καθολικά το ίδιο πρότυπο.

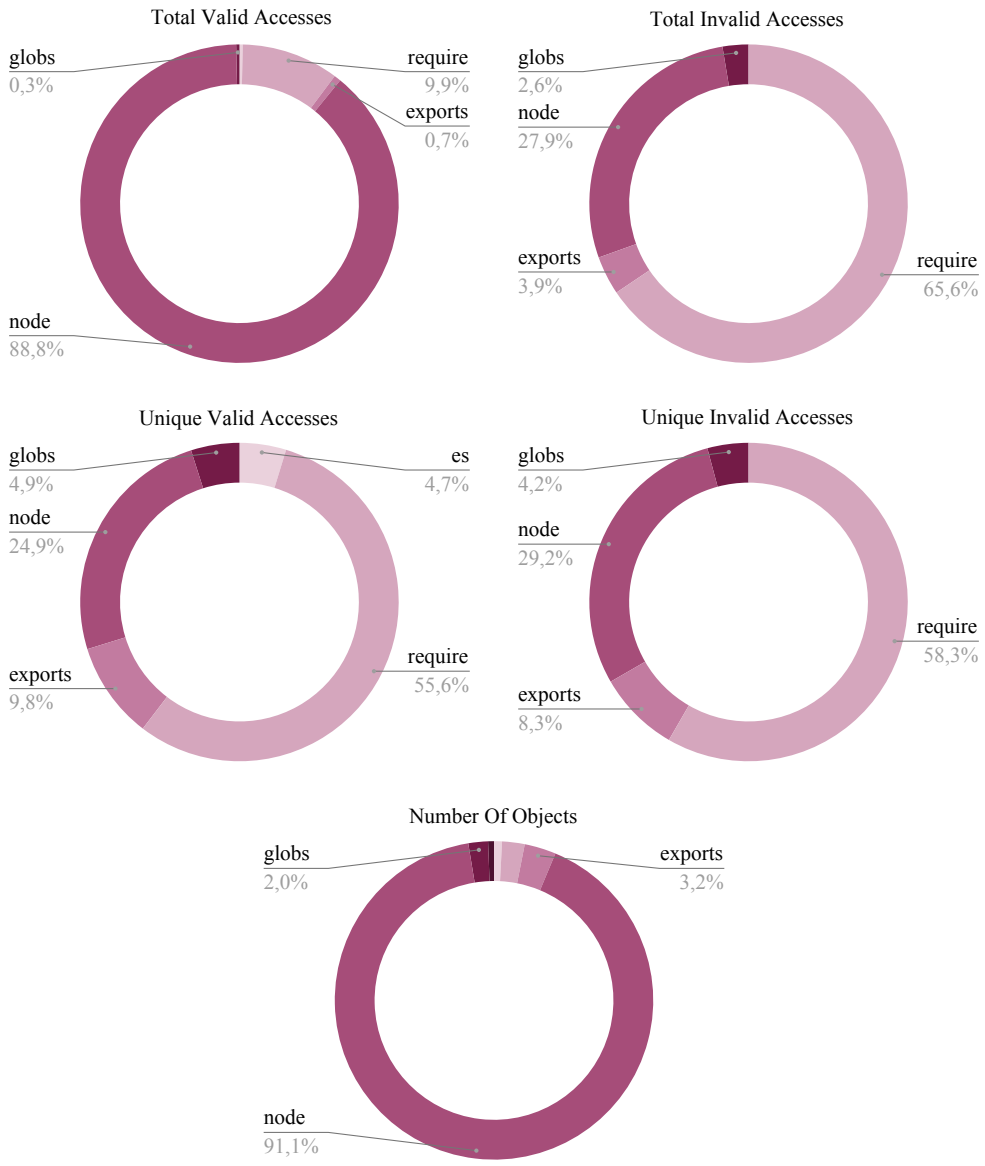
Πλαίσιο ανάλυσης Το πλαίσιο αναφέρεται στα ονόματα που χρησιμοποιούμε ανάλογα σε πιο κομμάτι της γλώσσας προγραμματισμού ανήκουν. Είναι τα εξής:

- Αυτά που δηλώνονται από το `EcmaScript` .
- Αυτά που δηλώνονται από με την εισαγωγή τους (`exports`).
- Αυτά που δηλώνονται από το `Node.js` περιβάλλον (`node`).
- Αυτά που είναι καθολικές μεταβλητές (`globs`).
- Αυτά που είναι καθολικές μεταβλητές, αλλά δηλώνονται από τον χρήστη χωρίς τη χρήση του `global` (`with-globals`).

Η εικόνα 7.4 δείχνει τα χαρακτηριστικά της LYA σε όλες τις βιβλιοθήκες. Σε αριθμό από στοιχεία που περιτυλίχθηκαν το 91.1% ήταν από το `node`. Σε αριθμό από μοναδικές εσφαλμένες και μη προσβάσεις οι περισσότερες έρχονται από το `require`, ενώ για μη μοναδικές προσβάσεις οι περισσότερες προέρχονται από τα `exports`.



Γράφημα 7.4: Σύνολο μοναδικών προσβάσεων και συνολικά στοιχεία που περιτυλίγονται με βάση το επίπεδο ανάλυσης. Με την έννοια μοναδικής πρόσβασης εννοούμε ότι η κάθε πρόσβαση προσμετράται μονάχα μία φορά. Από πάνω προς τα κάτω: Συνολικά περιτυλιγμένα στοιχεία, μοναδικές έγκυρες προσβάσεις, μοναδικές μη έγκυρες προσβάσεις.



Γράφημα 7.5: Σύνολο προσβάσεων και συνολικά τυλιγμένα στοιχεία ανά κατηγορία. Από την κορυφή με την φορά του ρολογιού: Σύνολο έγκυρων/μη έγκυρων προσβάσεων ανά κατηγορία, σύνολο μοναδικών έγκυρων/μη έγκυρων προσβάσεων ανά κατηγορία, αριθμός αντικειμένων ανά κατηγορία που περιτυλίχθηκαν. Με την έννοια μοναδικής πρόσβασης εννοούμε ότι η κάθε πρόσβαση προσμετράται μονάχα μία φορά.

Κεφάλαιο 8

Συμπεράσματα

Σε αυτή την διπλωματική εργασία παρουσιάσαμε μία νέα μορφή δυναμικής ανάλυσης, την αδρομερή, και το εργαλείο που δημιουργήθηκε για να την επιδείξουμε, την Lya. Το εργαλείο προσφέρει την δυνατότητα αποσύνθεσης, επεξεργασίας και επανασύνδεσης των βιβλιοθηκών που εισάγονται, διατηρώντας την αρχική λειτουργικότητά τους. Προσφέρεται βελτίωση 2–3 τάξεων σε επίπεδο ταχύτητας σε σύγκριση με τις καθιερωμένες τεχνικές δυναμικής ανάλυσης. Δίνεται η δυνατότητα να εφαρμοστούν πληθώρα αναλύσεων που υπάρχουν ήδη διαθέσιμες ή να γραφτούν καινούργιες σε λιγότερες από 100 γραμμές κώδικα. Η Lya μπορεί να χρησιμοποιηθεί είτε μέσω της γραμμής εργαλείων είτε με την ενσωμάτωσή της μέσα στο κώδικα του κύριου προγράμματος. Είναι γραμμένη στην ίδια γλώσσα που επιθυμεί να αναλύσει —την `Javascript`. Το εργαλείο υπάρχει διαθέσιμο για εγκατάσταση και πειραματισμό με άλλες εφαρμογές και αναλύσεις, είτε μέσω του `GitHub` είτε μέσω του `npm`.

8.1 Εφαρμογές σε άλλες γλώσσες προγραμματισμού

Οι τεχνικές που χρησιμοποιήθηκαν για να κατασκευαστεί το εργαλείο που εφαρμόζει την αδρομερή ανάλυση, είναι διαθέσιμες σε αρκετές σύγχρονες γλώσσες προγραμματισμού. Θα είχε ενδιαφέρον να γίνει κάποια διερεύνηση πάνω στα συστήματα βιβλιοθηκών και άλλων γλωσσών όπως `Python`, `Java` και `Go`.

8.2 Μελλοντικές εφαρμογές

Το σύστημα που δημιουργήθηκε στα πλαίσια αυτής της διπλωματικής εργασίας θα μπορούσε να βελτιωθεί και να επεκταθεί στα εξής σημεία:

- Να επιλυθούν διάφορα προβλήματα συμβατότητας με βιβλιοθήκες και να μπορέσει να βελτιωθεί η αξιοπιστία της Lya.
- Να δημιουργηθούν μερικές ακόμα αναλύσεις, έτσι ώστε να αναδειχθούν τα πλεονεκτήματα της αδρομερούς ανάλυσης.
- Να γίνει συνδυασμός στατικής και αδρομερούς ανάλυσης, αξιοποιώντας τα αποτελέσματα των αναλύσεων σε μία μορφή μετά-προγραμματισμού που θα έδινε ενδιαφέρονται συμπεράσματα για την αποδοτικότητα και την ακρίβεια, τόσο της στατικής ανάλυσης όσο και της δυναμικής.

8.3 Ευρύτερος αντίκτυπος

Τα τελευταία χρόνια με την επικράτηση της τεχνολογίας σε κάθε πτυχή της ζωής μας, είναι επακόλουθο ότι ένα εργαλείο που μπορεί να αναλύσει κώδικα εύκολα και γρήγορα θα ήταν ιδιαίτερα ωφέλιμο. Σε ανθρώπους που είναι σχετικοί με το αντικείμενο της επιστήμης των υπολογιστών, η Lya θα προσέφερε πλεονεκτήματα τόσο σε ακαδημαϊκό επίπεδο όσο και στην βιομηχανία.

Ακαδημαϊκά, η αδρομερής ανάλυση προσφέρει μία ιδανική πλατφόρμα για την παραγωγή και την εφαρμογή αναλύσεων, που με ελάχιστο κόστος μπορούν να διερευνηθούν ενδιαφέροντα ερωτήματα, τα οποία πιθανόν θα μπορούσαν να οδηγήσουν σε κάποια δημοσίευση. Στην βιομηχανία τα τελευταία χρόνια, η `Javascript` είναι πανταχού παρούσα. Οπότε όπως καταλαβαίνουμε τα προβλήματα που ενδέχεται να παρουσιαστούν, τόσο σε επίπεδο ασφαλείας ή ταχύτητας, μπορούν να επιφέρουν σημαντικές οικονομικές επιπτώσεις. Με την χρήση της Lya αυτά τα προβλήματα μπορούν να ανιχνευτούν και να αντιμετωπιστούν —ακόμα και στο στάδιο της παραγωγής.

Όμως πέρα από τους ανθρώπους που είναι σχετικοί με τον κλάδο τον υπολογιστών, η αδρομερής ανάλυση θα μπορούσε να ωφελήσει και τους υπόλοιπους κλάδους. Η αποδοτικότητα των προγραμμάτων και η ασφάλεια τους είναι στοιχεία που επηρεάζουν άμεσα τον μέσο χρήστη. Μία άλλη περίπτωση που θα μπορούσε να είναι χρήσιμη η αδρομερής ανάλυση θα ήταν για παράδειγμα η συνεχόμενη χρήση της σε κινητά έτσι ώστε να ανιχνεύονται παραβιάσεις ασφαλείας ή ακόμα και προβληματικές εφαρμογές. Αυτό μπορεί να επιτευχθεί λόγω του χαμηλού κόστους χρήσης της αδρομερούς ανάλυσης, καθώς αν είχε το κλασικό κόστος δυναμικών αναλύσεων δεν θα ήταν δυνατόν να τρέχει συνεχόμενα.

Βιβλιογραφία

[1] F. Aboukhadijeh. (2019) Determine if an object is a buffer (including the browserify buffer). https://www.npmjs.com/package/is-buffer. Accessed: 2020-09-27. [Online]. Available: https://www.npmjs.com/package/is-buffer

[2] P. Amidon, E. Davis, S. Sidiroglou-Douskos, and M. Rinard, "Program fracture and recombination for efficient automatic code reuse," in *2015 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2015, pp. 1–6.

[3] A. Anand. (2020) Sandbox escape: Safe eval. https://snyk.io/vuln/SNYK-JS-SAFEEVAL-608076. [Online]. Available: https://snyk.io/vuln/SNYK-JS-SAFEEVAL-608076

[4] E. Andreasen, L. Gong, A. Møller, M. Pradel, M. Selakovic, K. Sen, and C. . Staicu, "A survey of dynamic analysis and test generation for javascript," *ACM Computing Surveys*, vol. 50, no. 5, 2017, cited By :19. [Online]. Available: www.scopus.com

[5] J. S. Brian Woodward. (2015) Get the first element or first n elements of an array. https://www.npmjs.com/package/array-first. Accessed: 2020-09-23. [Online]. Available: https://www.npmjs.com/package/array-first

[6] J. P. Cambronero, T. H. Y. Dang, N. Vasilakis, J. Shen, J. Wu, and M. C. Rinard, "Active learning for software engineering," in *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, ser. Onward! 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 62–78. [Online]. Available: https://doi.org/10.1145/3359591.3359732

[7] L. Christophe, C. De Roover, and W. De Meuter, "Poster: Dynamic analysis using javascript proxies," in *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 813–814. [Online]. Available: http://dl.acm.org/citation.cfm?id=2819009.2819180

[8] G. Court. (2020) Uri.js is an rfc 3986 compliant, scheme extendable uri parsing/validating/resolving library for all javascript environments (browsers, node.js, etc). https://www.npmjs.com/package/uri-js. Accessed: 2020-09-28. [Online]. Available: https://www.npmjs.com/package/uri-js

[9] S. A. Crosby and D. S. Wallach, "Denial of service via algorithmic complexity attacks," in *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, ser. SSYM'03. USA: USENIX Association, 2003, p. 3.

[10] W. Developers. (2020) Sunspider 1.0.2 javascript benchmark. https://webkit.org/perf/sunspider/sunspider.html. [Online]. Available: https://webkit.org/perf/sunspider/sunspider.html

[11] P. Dotchev. (2016) Parse hangs on some long urls. https://github.com/garycourt/uri-js/issues/12. [Online]. Available: https://github.com/garycourt/uri-js/issues/12

[12] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The daikon system for dynamic detection of likely invariants," *Sci. Comput. Program.*, vol. 69, no. 1–3, p. 35–45, Dec. 2007. [Online]. Available: https://doi.org/10.1016/j.scico.2007.01.015

[13] C. Flanagan and S. N. Freund, "The roadrunner dynamic analysis framework for concurrent programs," in *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, ser. PASTE '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 1–8. [Online]. Available: https://doi.org/10.1145/1806672.1806674

[14] B. Fluri, M. Würsch, M. Pinzger, and H. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," *IEEE Transactions on Software Engineering*, vol. 33, no. 11, pp. 725–743, 2007.

[15] A. Holzinger, M. Brugger, and W. Slany, "Applying aspect oriented programming in usability engineering processes - on the example of tracking usage information for remote usability testing." 01 2011, pp. 53–56.

[16] Hrishikesh. (2018) Dockerhub: Jalangi docker container. https://hub.docker.com/r/hrishikeshrt/jalangi. [Online]. Available: https://hub.docker.com/r/hrishikeshrt/jalangi

[17] M. Keil and P. Thiemann, "Efficient dynamic access analysis using javascript proxies," in *Proceedings of the 9th Symposium on Dynamic Languages*, ser. DLS '13. New York, NY, USA: ACM, 2013, pp. 49–60. [Online]. Available: http://doi.acm.org/10.1145/2508168.2508176

[18] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *European conference on object-oriented programming*. Springer, 1997, pp. 220–242.

[19] T. Lauinger, A. Chaabane, and C. B. Wilson, "Thou shalt not depend on me," *Queue*, vol. 16, no. 1, pp. 62–82, 2018. [Online]. Available: www.scopus.com

[20] D. Lehmann and M. Pradel, "Wasabi: A framework for dynamically analyzing webassembly," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 1045–1058. [Online]. Available: https://doi.org/10.1145/3297858.3304068

[21] S. Ltd. (2018) minimatch@2.0.10. https://snyk.io/test/npm/minimatch/2.0.10. [Online]. Available: https://snyk.io/test/npm/minimatch/2.0.10

[22] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on*

Programming Language Design and Implementation, ser. PLDI '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 190–200. [Online]. Available: https://doi.org/10.1145/1065010.1065034

[23] M. Maass, "A theory and tools for applying sandboxes effectively," Ph.D. dissertation, Carnegie Mellon University, 2016.

[24] L. Marek, A. Villazón, Y. Zheng, D. Ansaloni, W. Binder, and Z. Qi, "Disl: A domain-specific language for bytecode instrumentation," in *Proceedings of the 11th Annual International Conference on Aspect-Oriented Software Development*, ser. AOSD '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 239–250. [Online]. Available: https://doi.org/10.1145/2162049.2162077

[25] J. Mickens, J. Elson, and J. Howell, "Mugshot: Deterministic capture and replay for javascript applications," in *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'10. USA: USENIX Association, 2010, p. 11.

[26] I. Neamtiu, J. S. Foster, and M. Hicks, "Understanding source code evolution using abstract syntax tree matching," vol. 30, no. 4, p. 1–5, May 2005. [Online]. Available: https://doi.org/10.1145/1082983.1083143

[27] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," *SIGPLAN Not.*, vol. 42, no. 6, p. 89–100, Jun. 2007. [Online]. Available: https://doi.org/10.1145/1273442.1250746

[28] npm, Inc. (2018) Details about the event-stream incident. https://blog.npmjs.org/post/180565383195/details-about-the-event-stream-incident. Accessed: 2018-12-18. [Online]. Available: https://blog.npmjs.org/post/180565383195/details-about-the-event-stream-incident

[29] A. Parodi. (2020) Awesome micro npm packages. https://github.com/parro-it/awesome-micro-npm-packages. [Online]. Available: https://github.com/parro-it/awesome-micro-npm-packages

[30] J. Schlinkert. (2017) Array-slice method. slices array from the start index up to, but not including, the end index. https://www.npmjs.com/package/array-slice. Accessed: 2020-09-27. [Online]. Available: https://www.npmjs.com/package/array-slice

[31] ——. (2017) Get the last or last n elements in an array. https://www.npmjs.com/package/array-last. Accessed: 2020-09-23. [Online]. Available: https://www.npmjs.com/package/array-last

[32] ——. (2018) Returns true if the value is a finite number. https://www.npmjs.com/package/is-number. Accessed: 2020-09-27. [Online]. Available: https://www.npmjs.com/package/is-number

[33] ——. (2020) Get the native type of a value. https://www.npmjs.com/package/kind-of. Accessed: 2020-09-27. [Online]. Available: https://www.npmjs.com/package/kind-of

[34] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs, "Jalangi: A selective record-replay and dynamic analysis framework for javascript," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: ACM, 2013, pp. 488–498. [Online]. Available: http://doi.acm.org/10.1145/2491411.2491447

[35] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, and M. Rinard, "Automatic error elimination by horizontal code transfer across multiple applications," *SIGPLAN Not.*, vol. 50, no. 6, p. 43–54, Jun. 2015. [Online]. Available: https://doi.org/10.1145/2813885.2737988

[36] Snyk. (2016) Find, fix and monitor for known vulnerabilities in node.js and ruby packages. https://snyk.io/. [Online]. Available: https://snyk.io/

[37] A. Sparling *et al.* (2018) Event-stream, github issue 116: I don't know what to say. https://github.com/dominictarr/event-stream/issues/116. Accessed: 2018-12-18. [Online]. Available: https://github.com/dominictarr/event-stream/issues/116

[38] C.-A. Staicu, D. Schoepe, M. Balliu, M. Pradel, and A. Sabelfeld, "An empirical study of information flows in real-world javascript," in *Proceedings of the 14th ACM SIGSAC Workshop on Programming Languages and Analysis for Security*, ser. PLAS'19. New York, NY, USA: Association for Computing Machinery, 2019, p. 45–59. [Online]. Available: https://doi.org/10.1145/3338504.3357339

[39] H. Sun, D. Bonetta, C. Humer, and W. Binder, "Efficient dynamic analysis for node.js," in *Proceedings of the 27th International Conference on Compiler Construction*, ser. CC 2018. New York, NY, USA: ACM, 2018, pp. 196–206. [Online]. Available: http://doi.acm.org/10.1145/3178372.3179527

[40] Y. Tanabe, T. Aotani, and H. Masuhara, "A context-oriented programming approach to dependency hell," 07 2018, pp. 8–14.

[41] The gRPC Authors. (2018) grpc. https://grpc.io/. Accessed: 2019-04-16. [Online]. Available: https://grpc.io/

[42] N. Vasilakis, B. Karel, Y. Palkhiwala, J. Sonchack, A. DeHon, and J. M. Smith, "Ignis: Scaling distribution-oblivious systems with light-touch distribution," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 1010–1026. [Online]. Available: https://doi.org/10.1145/3314221.3314586

[43] N. Vasilakis, B. Karel, N. Roessler, N. Dautenhahn, A. DeHon, and J. M. Smith, "Towards fine-grained, automated application compartmentalization," in *Proceedings of the 9th Workshop on Programming Languages and Operating Systems*, ser. PLOS'17. New York, NY, USA: ACM, 2017, pp. 43–50. [Online]. Available: http://doi.acm.org/10.1145/3144555.3144563

[44] ——, "Breakapp: Automated, flexible application compartmentalization," in *Networked and Distributed Systems Security*, ser. NDSS'18, 2018. [Online]. Available: http://dx.doi.org/10.14722/ndss.2018.23131

[45] T. Würthinger, C. Wimmer, C. Humer, A. Wöundefined, L. Stadler, C. Seaton, G. Duboscq, D. Simon, and M. Grimmer, "Practical partial evaluation for high-performance dynamic language runtimes," *SIGPLAN Not.*, vol. 52, no. 6, p. 662–676, Jun. 2017. [Online]. Available: https://doi.org/10.1145/3140587.3062381

[46] T. Würthinger, C. Wimmer, A. Wöundefined, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko, "One vm to rule them all," in *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, ser. Onward! 2013. New York, NY,

USA: Association for Computing Machinery, 2013, p. 187–204. [Online]. Available: https://doi.org/10.1145/2509578.2509581

[47] S. Yegulalp. (2016) How one yanked javascript package wreaked havoc. http://www.infoworld.com/article/3047177/javascript/how-one-yanked-javascript-package-wreaked-havoc.html. [Online]. Available: http://www.infoworld.com/article/3047177/javascript/how-one-yanked-javascript-package-wreaked-havoc.html

[48] M. Zimmermann, C.-A. Staicu, C. Tenny, and M. Pradel, "Small world with high risks: A study of security threats in the npm ecosystem," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 995–1010. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/zimmerman

[49] ——, "Smallworld with high risks: A study of security threats in the npm ecosystem," in *Proceedings of the 28th USENIX Conference on Security Symposium*, ser. SEC'19. USA: USENIX Association, 2019, p. 995–1010.

Παραρτήματα

Παράρτημα Α

Ένας μικρός οδηγός για την JavaScript

Α.1 Γενικές πληροφορίες για την γλώσσα

Η γλώσσα προγραμματισμού `Javascript` είναι μία γλώσσα προγραμματισμού που έχει ως βασικό σκοπό να προσφέρει αλληλεπίδραση στις σελίδες του διαδικτύου. Είναι μία πλήρως δυναμική γλώσσα προγραμματισμού που δημιουργήθηκε από τον *Brendan Eich* (συνδημιουργό του Mozilla project). Είναι γενικού σκοπού και μπορεί πλέον να τρέξει σε υπολογιστές, κινητά και στο διαδίκτυο.

Α.2 Ένα παράδειγμα Hello world

Για να μπορέσουμε να τρέξουμε το γνωστό πρώτο παράδειγμα των γλωσσών προγραμματισμού αρκεί να ακολουθήσω τα παρακάτω βήματα:

1. Δημιούργησε ένα νέο φάκελο που ονομάζεται `scripts`. Εσωτερικά του φακέλου δημιούργησε ένα νέο αρχείο που ονομάζεται `main.js` και αποθήκευσέ το.

2. Πρόσθεσε αυτόν το κώδικα στο `main.js` αρχείο:

```
console.log('Hello world')
```

3. Έλεγξε ότι έχει σωθεί το αρχείο της `Javascript`. Αμέσως μετά τρέξε το αρχείο με την εντολή:

```
node main.js
```

Παράρτημα Β

Χρησιμοποιώντας την Lya στην πράξη

Παρακάτω θα παρουσιαστούν μερικές βασικές οδηγίες για την εγκατάσταση και την χρήση της Lya. Θα θεωρήσουμε ότι το λειτουργικό σύστημα είναι κάποια έκδοση `Ubuntu Linux`.

Β.1 Διαθεσιμότητα και Όροι

Η Lya υπάρχει διαθέσιμη τόσο στο Github, όσο και στο npmjs υπό την MIT άδεια χρήσης.

Β.2 Εγκατάσταση

Υπάρχουν δύο τρόποι να εγκατασταθεί το εργαλείο που έχει δημιουργηθεί. Είτε μέσω του `npm` είτε από το `Github`. Θα παρουσιαστούν και οι δύο τρόποι. Σε περίπτωση που θέλουμε να γίνει η εγκατάσταση με την χρήση του `npm`, αρκεί ο χρήστης να τρέξει την παρακάτω εντολή:

```
npm install @andromeda/lya -g
```

Σε περίπτωση που ο χρήστης θέλει να χρησιμοποιήσει το `Github` αρκεί να κατεβάσει το πακέτο και να τρέξει την εγκατάσταση των βιβλιοθηκών. Πιο συγκεκριμένα:

```
git clone https://github.com/andromeda/lya
cd lya
npm install
```

Εφαρμόζοντας τον ένα ή τον άλλο τρόπο είστε έτοιμοι να χρησιμοποιήσετε και να δημιουργήσετε αδρομερείς αναλύσεις.

Β.3 Χρήση του εργαλείου

Μόλις έχουμε εγκαταστήσει την Lya, το επόμενο στάδιο είναι να τρέξουμε μία ανάλυση σε ένα μικρό πρόγραμμα. Παρακάτω θα γράψουμε μία βιβλιοθήκη που εξάγει μία συνάρτηση που κάνει πρόσθεση, και θα την διαβάσουμε σε ένα κύριο πρόγραμμα και θα την χρησιμοποιήσουμε. Θα θεωρήσουμε για ευκολία ότι έχει εγκατασταθεί το εργαλείο της γραμμής εργαλείων.

Οπότε σαν πρώτο βήμα θα δημιουργήσουμε έναν φάκελο και θα μπούμε σε αυτόν τον φάκελο.

```
mkdir test
cd test
```

Αμέσως μετά θα δημιουργήσουμε ένα αρχείο με όνομα `add.js` και περιεχόμενα τον παρακάτω κώδικα. Αυτός θα είναι ο κώδικας της βιβλιοθήκης που εξάγει μια συνάρτηση που πραγματοποιεί πρόσθεση.

```
module.exports = (a, b) => a + b;
```

Μετά θα δημιουργήσουμε ένα αρχείο που θα εισάγει την παραπάνω βιβλιοθήκη και θα τυπώνει το αποτέλεσμα της πρόσθεσης.

```
const add = require('./add.js');

const result = add(1, 3);
console.log(result);
```

Τέλος μπορούμε να τρέξουμε το αρχείο με την ανάλυση `on_off` έτσι ώστε να δούμε όλες τις προσβάσεις που συμβαίνουν σε αυτά τα δύο μικρά αρχεία κώδικα.

```
lya main.js -p
```

Τα αποτελέσματα της ανάλυσης θα φανούν στην οθόνη σας. Έχετε τρέξει την πρώτη σας ανάλυση!

Ενσωματωμένος Οδηγός Όλα τα στοιχεία που μπορεί να δεχθεί ως ορίσματα το εργαλείο είναι διαθέσιμα προγραμματιστικά μέσω της γραμμής εντολών. Αρκεί ο χρήστης να τρέξει την παρακάτω εντολή για να τυπωθούν στην οθόνη:

```
lya -h
```