ΤΕΧΝΟΛΟΓΙΚΟ ΕΚΠΑΙΔΕΥΤΙΚΟ ΙΔΡΥΜΑ ΚΕΝΤΡΙΚΗΣ ΜΑΚΕΔΟΝΙΑΣ

ΣΧΟΛΗ ΤΕΧΝΟΛΟΓΙΚΩΝ ΕΦΑΡΜΟΓΩΝ

ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ ΤΕ

ΥΛΟΠΟΙΗΣΗ ΕΡΓΑΛΕΙΟΥ ΘΟΛΩΣΗΣ ΠΡΟΓΡΑΜΜΑΤΩΝ JAVA

Πτυχιακή εργασία του

Αλέξανδρου Τσιλιγγίρη (2920)

Επιβλέπων: Ν. Πεταλίδης

ΣΕΡΡΕΣ, ΜΑΙΟΣ 2016

Υπεύθυνη δήλωση

Υπεύθυνη Δήλωση: Βεβαιώνω ότι είμαι συγγραφέας αυτής της πτυχιακής εργασίας και ότι κάθε βοήθεια την οποία είχα για την προετοιμασία της, είναι πλήρως αναγνωρισμένη και αναφέρεται στην πτυχιακή εργασία. Επίσης έχω αναφέρει τις όποιες πηγές από τις οποίες έκανα χρήση δεδομένων, ιδεών ή λέξεων, είτε αυτές αναφέρονται ακριβώς είτε παραφρασμένες. Επίσης βεβαιώνω ότι αυτή η πτυχιακή εργασία προετοιμάστηκε από εμένα προσωπικά ειδικά για τις απαιτήσεις του προγράμματος σπουδών του Τμήματος Μηχανικών Πληροφορικής ΤΕ του Τ.Ε.Ι. Κεντρικής Μακεδονίας.

Σύνοψη

Η γλώσσα Java έχει σχεδιαστεί ούτως ώστε να μεταφράζεται σε μια ακολουθία bytes η οποία είναι ανεξάρτητη πλατφόρμας. Αυτό δίνει το πλεονέκτημα σε προγράμματα που είναι γραμμένα στη γλώσσα να έχουν τη δυνατότητα να τρέξουν αυτούσια σε πολλές αρχιτεκτονικές και λειτουργικά συστήματα. Ταυτόχρονα όμως της δίνει το μειονέκτημα να είναι ευάλωτη σε επιθέσεις αποσυγκρότησης (reverse engineering) όπου κάποιος με πρόσβαση μόνο στο εκτελέσιμο μπορεί να εξάγει πληροφορίες για τον αρχικό κώδικα ή ακόμα και τον ίδιο τον κώδικα.

Μια τεχνική για να αποφευχθεί αυτό το πρόβλημα είναι αυτό της «θόλωσης» (obfuscation) του αρχικού κώδικα. Οι τεχνικές θόλωσης στηρίζονται στο άλλαγμα των ονομάτων των μεταβλητών, την ισοπέδωση των ιεραρχιών κτλ ούτως ώστε να μην είναι εύκολη η εξαγωγή συμπερασμάτων για τον πηγαίο κώδικα ενός εκτελέσιμου.

Σκοπός αυτής της πτυχιακής είναι η μελέτη και παρουσίαση των διαφορετικών τεχνικών θόλωσης καθώς και η υλοποίηση μιας εφαρμογής η οποία θα καθιστά δυνατή την θόλωση κώδικα Java.

Περιεχόμενα

Κατάλογος διαγραμμάτων

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τους γονείς μου, Αναστασία και Σπύρο, για τη στήριξη τους όλα αυτά τα χρόνια. Επίσης θα ήθελα να ευχαριστήσω θερμά τον επιβλέποντα καθηγητή μου, κύριο Νίκο Πεταλίδη, για τις γνώσεις και τον τρόπο σκέψης που μου μεταβίβασε, καθώς και για την καθοδήγηση που μου παρείχε στις διαλέξεις και κατά τη διάρκεια εκπόνησης της πτυχιακής εργασίας.

Ορισμοί

Potency Ο βαθμός στον οποίο ένας μετασχηματισμός καταφέρνει να μπερδέψει τον προγραμματιστή που προσπαθεί να τον αντιστρέψει.

Resilience Ο βαθμός στον οποίο ένας μετασχηματισμός καταφέρνει να αμυνθεί από αυτοματοποιημένες επιθέσεις εργαλείων ξεθόλωσης.

Cost Ο βαθμός στον οποίο ένας μετασχηματισμός επιβαρύνει τον χρόνο εκτέλεσης της εφαρμογής ή αυξάνει το μέγεθός της.

Stealth Ο βαθμός στον οποίο ένας μετασχηματισμός δεν μπορεί να διακριθεί ανάμεσα στον υπόλοιπο κώδικα της εφαρμογής.

Opaque Predicate Boolean έκφραση της οποίας το αποτέλεσμα δεν μπορεί να προσδιοριστεί εύκολα από τον επιτιθέμενο.

Layout Obfuscation Μετασχηματισμοί που αφαιρούν τις πληροφορίες που είναι χρήσιμες για τον προγραμματιστή, όπως σχόλια, ονόματα μεταβλητών, αριθμούς γραμμών και στοίχιση.

Data Obfuscation Μετασχηματισμοί που επιδρούν πάνω στις δομές δεδομένων της εφαρμογής, όπως πίνακες, κλάσεις και μεταβλητές

Control Flow Obfuscation Μετασχηματισμοί που αλλοιώνουν την πραγματική ροή εκτέλεσης της εφαρμογής.

Preventive Transformation Μετασχηματισμοί που αποτρέπουν τη σωστή λειτουργία των εργαλείων αποσυγκρότησης και εργαλείων ξεθόλωσης.

Decompiler Λογισμικό το οποίο δέχεται ως είσοδο ένα εκτελέσιμο και δημιουργεί υψηλού επιπέδου αρχεία πηγαίου κώδικα.

Κεφάλαιο 1

Εισαγωγή

Οι επιθέσεις αποσυγκρότησης προς μια εφαρμογή από χάκερ ή από ανταγωνιστές μπορεί να καταλήξει σε ανεπιθύμητη έκθεση των αλγορίθμων και ιδεών, των εσωτερικών δομών, των ενσωματωμένων συστημάτων αδειοδότησης (licensing) και συστημάτων ασφαλείας, και, σημαντικότερα, των προσωπικών δεδομένων των χρηστών μιας εφαρμογής.

Η γλώσσα Java μεταγλωττίζεται σε υψηλού επιπέδου, ανεξάρτητο πλατφόρμας bytecode, το οποίο περιέχει αντικείμενα, τύπους και πίνακες. Τα αρχεία αυτά έχουν κατάληξη .class και περιέχονται μέσα στο τελικό μη κρυπτογραφημένο αρχείο jar μιας εφαρμογής. Μια εφαρμογή Java αποτελείται συνήθως από πολλά μη κρυπτογραφημένα αρχεία jar. Σύμφωνα με τα παραπάνω, εφόσον ο κώδικας μιας εφαρμογής δεν θολωθεί, μια επίθεση αποσυγκρότησης θα δώσει πληροφορίες των ονομάτων των κλάσεων, μεθόδων και μεταβλητών, καθώς και όλων των import που χρησιμοποιεί το εκάστοτε αρχείο, τα οποία περιλαμβάνουν τις κλάσεις, τις μεθόδους και τις μεταβλητές που χρησιμοποιούνται.

Οι επιθέσεις αποσυγκρότησης σε προγράμματα Java είναι απλές σαν διαδικασία και επομένως μπορούν να αυτοματοποιηθούν πλήρως. Οποιοσδήποτε προγραμματιστής με βασικές γνώσεις μπορεί να χρησιμοποιήσει έναν decompiler, να εισάγει μια εφαρμογή και να διαβάσει τον κώδικα της, σχεδόν σα να ήταν εφαρμογή ανοιχτού κώδικα.

Δημιουργείται έτσι η ανάγκη προστασίας των εφαρμογών και των πνευματικών δικαιωμάτων μιας εταιρίας ή ενός ελεύθερου επαγγελματία προγραμματιστή. Τα εργαλεία θόλωσης (obfuscators) μετασχηματίζουν τον κώδικα με τέτοιο τρόπο ώστε να

χαθεί χρήσιμη (για ανθρώπους) πληροφορία, να μπερδευτεί η ροή εκτέλεσης και οι δομές του προγράμματος έτσι ώστε κάποιος επιτιθέμενος να πρέπει να σπαταλήσει το χρόνο του αντιστρέφοντας αυτούς τους μετασχηματισμούς. Αυτό δεν είναι εφικτό πλήρως διότι κατά τη διαδικασία της θόλωσης, μεγάλο μέρος της πληροφορίας, όπως τα ονόματα κλάσεων, μεθόδων και μεταβλητών καθώς και ο λογικός διαχωρισμός σε κλάσεις, χάνεται οριστικά.

Το εργαλείο θόλωσης που αναπτύχθηκε κατά τη διάρκεια εκπόνησης αυτής της πτυχιακής εργασίας ονομάζεται JCuttlefish και, ανόμοια από άλλα εργαλεία τα οποία θολώνουν ήδη μεταγλωττισμένο κώδικα, είναι ένα εργαλείο θόλωσης πηγαίου κώδικα Java.

1.1 Δομή της εργασίας

Κεφάλαιο 2 Παρουσίαση συχνών τεχνικών θόλωσης και γνωστών εργαλείων θόλωσης που διατίθενται στο εμπόριο ή δωρεάν.

Κεφάλαιο 3 Περιγραφή και ανάλυση του λογισμικού που χρησιμοποιήθηκε για την ανάπτυξη του εργαλείου θόλωσης καθώς και κάποια βασικά σενάρια χρήσης.

Κεφάλαιο 4 Ανάλυση της υλοποίησης των υποσυστημάτων και των σταδίων εκτέλεσης του εργαλείου θόλωσης.

Κεφάλαιο 5 Εγχειρίδιο, παραδείγματα χρήσης και παρουσίαση αποτελεσμάτων του εργαλείου θόλωσης.

Κεφάλαιο 6 Τελικά συμπεράσματα για τις μεθόδους θόλωσης και την ανάπτυξη του εργαλείου θόλωσης.

Κεφάλαιο 2

Τεχνικές θόλωσης και εργαλεία

2.1 Τεχνικές θόλωσης

Οι τεχνικές θόλωσης περιγράφουν αναλυτικά κάποιους μετασχηματισμούς που μπορούν να εφαρμοστούν σε ένα πρόγραμμα καθώς και τα σημεία-στόχους αυτών των μετασχηματισμών. Χωρίζονται σε κατηγορίες (Witkowska 2006) σύμφωνα με τους σημεία-στόχους των μετασχηματισμών. Οι μεγαλύτερες κατηγορίες, τις οποίες και επιλέξαμε να αναλύσουμε, είναι οι εξής :

- Layout obfuscation
- Data obfuscation
- Control flow obfuscation

Η ποιότητα των μετασχηματισμών μπορεί να περιγραφεί από τις παρακάτω παραμέτρους (Witkowska 2006) :

Potency Ο βαθμός στον οποίο ένας μετασχηματισμός καταφέρνει να μπερδέψει τον προγραμματιστή που προσπαθεί να τον αντιστρέψει.

Resilience Ο βαθμός στον οποίο ένας μετασχηματισμός καταφέρνει να αμυνθεί από αυτοματοποιημένες επιθέσεις εργαλείων ξεθόλωσης.

Cost Ο βαθμός στον οποίο ένας μετασχηματισμός επιβαρύνει τον χρόνο εκτέλεσης της εφαρμογής ή αυξάνει το μέγεθός της.

Stealth Ο βαθμός στον οποίο ένας μετασχηματισμός δεν μπορεί να διακριθεί ανάμεσα στον υπόλοιπο κώδικα της εφαρμογής.

2.1.1 Layout Obfuscation

Αναφέρεται στους μετασχηματισμούς οι οποίοι δεν αλλάζουν τη σειρά εκτέλεσης τους προγράμματος αλλά μόνο το λεξιλόγιο που χρησιμοποιεί, για παράδειγμα τα ονόματα των μεταβλητών. Είναι απλοί στην υλοποίηση και δεν προσθέτουν επιπλέον κόστος στο χρόνο εκτέλεσης και στο μέγεθος της εφαρμογής. Για πολλά εργαλεία θόλωσης, αποτελούν τον βασικό και πρωταρχικό μετασχηματισμό.

2.1.1.1 Αλλαγή ονομάτων μεταβλητών και υπογραφών συναρτήσεων

Τα ονόματα των μεταβλητών και οι υπογραφές των συναρτήσεων περιέχουν χρήσιμες πληροφορίες για τον προγραμματιστή οι οποίες μπορούν να αφαιρεθούν.
Για παράδειγμα, η μεταβλητή String email μπορεί να μετονομαστεί σε String a και η υπογραφή της συνάρτησης sendEmail(String email, String text) να μετασχηματιστεί σε a(String a, String b).

Ιδιαίτερο ενδιαφέρον παρουσιάζει η περίπτωση στην οποία δύο ή περισσότερες συναρτήσεις μετασχηματίζονται έτσι ώστε να έχουν ίδιο όνομα αλλά διαφορετική υπογραφή. Για παράδειγμα, οι συναρτήσεις sendEmail(String email, String text) και deleteEmail(String email) , μετασχηματίζονται σε a(String a, String b) και a(String a).

Potency Υψηλό, μη-αντιστρέψιμο. Η πληροφορία που δίνουν τα ονόματα χάνεται οριστικά.

Resilience Χαμηλό. Τα εργαλεία μπορούν εύκολα να αντικαταστήσουν τα ονόματα a,b,c με var_1, var_2, var_3 και function_1, function_2, function_3.

Cost Μηδενικό. Τα ονόματα μικραίνουν, άρα μικραίνει και το μέγεθος της εφαρμογής.

Stealth Χαμηλό.

2.1.2 Data Obfuscation

Αναφέρεται στους μετασχηματισμούς οι οποίοι τροποποιούν τις δομές δεδομένων της εφαρμογής. Συγκεκριμένα, οι στόχοι αυτών των μετασχηματισμών είναι η κωδικοποίηση των δεδομένων και ο τρόπος με τον οποίο είναι ομαδοποιημένα.

2.1.2.1 Θόλωση πινάκων

Ένας πίνακας μπορεί να μετασχηματιστεί σε δύο ή περισσότερους πίνακες, προσπαθώντας να μπερδέψει τον επιτιθέμενο για το περιεχόμενό του και για το αν τα δεδομένα των πινάκων σχετίζονται ή όχι. Από την άλλη πλευρά, δύο ή περισσότεροι πίνακες, άσχετοι μεταξύ τους, μπορούν να ενωθούν σε ένα πίνακα ώστε να μπερδέψουν τον επιτιθέμενο.

Οι διαστάσεις των πινάκων είναι ένας ακόμη στόχος των εργαλείων θόλωσης. Συγκεκριμένα, ένας μονοδιάστατος πίνακας μπορεί να μετασχηματιστεί σε πολυδιάστατο δημιουργώντας ερωτήματα για το αν τα δεδομένα σχετίζονται ή όχι. Αντίστοιχα, ένας πολυδιάστατος πίνακας μπορεί να μετασχηματιστεί σε μονοδιάστατο πετυχαίνοντας έτσι τα ίδια αποτελέσματα.

Potency Ανά περίπτωση. Σε μια εφαρμογή που οι πράξεις μεταξύ πινάκων είναι πολλές δημιουργείται τεράστια σύγχυση. Αντιθέτως, μια εφαρμογή που απλά κρατάει τις ηλικίες των χρηστών σε ένα πίνακα αντιστρέφεται σχεδόν άμεσα.

Resilience Χαμηλό. Ο επιτιθέμενος μπορεί εύκολα να αναλύσει τα περιεχόμενα των πινάκων, εισάγοντας δεδομένα στην εφαρμογή και παρακολουθώντας τη μνήμη.

Cost Μηδενικό.

Stealth Υψηλό.

2.1.2.2 Θόλωση Κλάσεων

Μια Κλάση μπορεί να χωριστεί σε πολλές, αυξάνοντας έτσι το βάθος της κληρονομικότητας και αφαιρώντας κάθε χρήσιμη πληροφορία ομαδοποίησης των δεδομένων. Επιπλέον, μπορούμε να παρεμβάλλουμε ψεύτικες κλάσεις στο δέντρο κληρονομικότητας προκαλώντας επιπλέον σύγχυση. Η παραπάνω διαδικασία μπορεί να εφαρμοστεί ανάποδα και τα επιφέρει τα ίδια αποτελέσματα, μόνο που αντί για ψεύτικες κλάσεις θα έχουμε ψεύτικες μεταβλητές και συναρτήσεις. Σε κάθε περίπτωση το scope των δεδομένων αλλάζει σε public.

Potency Μέτριο. Εφόσον οι ψεύτικες κλάσεις περιέχουν κώδικα, ο προγραμματιστής πρέπει να αναλύσει αυτόν τον κώδικα και να συμπεράνει αν πρόκειται για κώδικα της εφαρμογής ή όχι.

Ανάλογα με την πολυπλοκότητα της εφαρμογής, αυτό μπορεί να είναι εύκολο ή δύσκολο.

Resilience Χαμηλό. Τα εργαλεία μπορούν να αναλύσουν τον κώδικα και τις λειτουργίες που υλοποιεί και να καταλάβουν ότι πρόκειται για ψεύτικο κώδικα, εφόσον δεν χρησιμοποιηθεί και κάποια άλλη μέθοδος θόλωσης.

Cost Μηδενικό.

Stealth Υψηλό.

2.1.2.3 Θόλωση Μεταβλητών

Μια μεταβλητή μπορεί να σπάσει σε 2 μεταβλητές διαφορετικού τύπου και να δημιουργηθεί μια συνάρτηση αντιστοίχησης. Για παράδειγμα, αντί να χρησιμοποιήσουμε μια μεταβλητή boolean, μπορούμε να δημιουργήσουμε 2 μεταβλητές int, να ορίσουμε πότε ο συνδυασμός τους επιστρέφει true ή false και να χρησιμοποιήσουμε την συνάρτηση αντιστοίχησης στις εκφράσεις ελέγχου. Με αυτόν τον τρόπο, ο επιτιθέμενος θα πρέπει να ψάξει την συνάρτηση αντιστοίχησης και να την αποκωδικοποιήσει, καταναλώνοντας τον χρόνο του.

Potency Ανά περίπτωση. Εξαρτάται από την πολυπλοκότητα της συνάρτησης αντιστοίχησης.

Resilience Ανά περίπτωση. Εξαρτάται από το αν τα εργαλεία μπορούν να προσδιορίσουν το αποτέλεσμα της συνάρτησης αντιστοίχησης μέσω στατικής ανάλυσης.

Cost Ανά περίπτωση. Εξαρτάται από την πολυπλοκότητα των υπολογισμών της συνάρτησης αντιστοίχησης.

Stealth -

2.1.3 Control Flow Obfuscation

Αναφέρεται στους μετασχηματισμούς οι οποίοι τροποποιούν τη σειρά εκτέλεσης του προγράμματος με σκοπό να κρύψουν τα στάδια από τα οποία περνά η εφαρμογή. Συνήθως εισάγουν επιπλέον κόστος σε χρόνο εκτέλεσης και μέγεθος αρχείου.

2.1.3.1 Opaque Predicates – Aliasing And Threading

Opaque Predicates είναι συνθήκες των οποίων το αποτέλεσμα είναι γνωστό στο εργαλείο θόλωσης αλλά δεν μπορεί να προσδιοριστεί μέσω στατικής ανάλυσης από τον επιτιθέμενο. PT ονομάζουμε ένα predicate που επιστρέφει πάντα true, PF ονομάζουμε ένα predicate που επιστρέφει πάντα false και P? ονομάζουμε ένα predicate που κάποιες φορές επιστρέφει true και κάποιες άλλες false.

Ένα predicate μπορεί να γίνει opaque χρησιμοποιώντας τις τεχνικές aliasing και threading.

Aliasing Χτίζοντας μια πολύπλοκη δυναμική δομή η οποία αποτελείται από πολλά ίδια στοιχεία και διατηρώντας εσωτερικούς δείκτες μεταξύ των στοιχείων, μπορούμε να περιπλέξουμε τον τρόπο λειτουργίας της με αποτέλεσμα να χρειάζεται ανάλυση ολόκληρης της εφαρμογής για να προσδιοριστεί το αποτέλεσμα μιας έκφρασης που χρησιμοποιεί αυτή τη δομή για να "κρυφτεί".

Threading Σύμφωνα με αυτή την τεχνική, μια συνάρτηση χωρίζεται σε πολλά synchronous threads μεταξύ των οποίων δημιουργούνται συνθήκες συναγωνισμού ώστε να μην εκτελούνται παράλληλα.

Το τελικό αποτέλεσμα είναι πάντα το ίδιο αλλά κάθε φορά που ο επιτιθέμενος επιχειρεί να κάνει στατική ανάλυση η εφαρμογή συμπεριφέρεται διαφορετικά.

Potency Υψηλό.

Resilience Υψηλό.

Cost Ανά περίπτωση, υψηλό. Εξαρτάται από την πολυπλοκότητα των υπολογισμών και την ποσότητα και πολυπλοκότητα των threads.

Stealth Ανά περίπτωση. Εξαρτάται από την εφαρμογή και τις λειτουργίες που υλοποιεί.

2.1.3.2 Method Cloning – Smoke And Mirror

Οι τεχνικές Method Cloning και Smoke And Mirror αναφέρονται στο ίδιο πράγμα και προβλέπουν την δημιουργία πολλών εκδόσεων μιας συνάρτησης χρησιμοποιώντας διαφορετικές τεχνικές θόλωσης σε κάθε έκδοση. Με αυτόν τον τρόπο όλες οι εκδόσεις της συνάρτησης υλοποιούν τις ίδιες λειτουργίες αλλά διαφέρουν στον τρόπο με τον οποίο λειτουργούν. Ο επιτιθέμενος θα πρέπει να προβεί σε δυναμική ανάλυση όλων των συναρτήσεων της εφαρμογής και να τις ομαδοποιήσει,ώστε να καταλάβει πόσες και ποιες συναρτήσεις υπάρχουν στην εφαρμογή.

Potency Υψηλό.

Resilience Υψηλό, εφόσον γίνει χρήση ισχυρών opaque predicates.

Cost Ανά περίπτωση, υψηλό. Εξαρτάται από το κόστος των τεχνικών θόλωσης που θα χρησιμοποιηθούν.

Stealth Υψηλό.

2.1.3.3 Method In-lining and Out-lining

Το Method In-lining γίνεται από τους compilers ως μέθοδος βελτιστοποίησης αλλά μπορεί να χρησιμοποιηθεί και ως μέθοδος θόλωσης. Σύμφωνα με αυτή την τεχνική, η κλήση μιας συνάρτησης αντικαθίσταται με τον κώδικα της συνάρτησης στο σημείο κλήσης. Επιπλέον, άλλες εντολές ομαδοποιούνται σε συναρτήσεις ισοπεδώνοντας έτσι κάθε μορφή ομαδοποίησης του κώδικα. Ιδιαίτερη προσοχή πρέπει να δοθεί στις εξαρτήσεις, διότι μεταφέροντας κώδικα, υπάρχει περίπτωση το ένα κομμάτι να μη μπορεί να "δει" το άλλο.

Potency Υψηλό. Ο επιτιθέμενος δεν μπορεί να καταλάβει εάν πρόκειται για συνάρτηση ή όχι.

Resilience Υψηλό.

Cost Μηδενικό.

Stealth Υψηλό.

2.1.3.4 Dead/Cloned Code Injection

Το εργαλείο θόλωσης, κάνοντας χρήση των opaque predicates, παρεμβάλλει κώδικα ο οποίος δεν υλοποιεί κάποια λειτουργία, απλά και μόνο για να δυσκολέψει την στατική ανάλυση. Το εργαλείο ελέγχει για το αν ο κώδικας θα εκτελεστεί ή οχι. Για παράδειγμα, το εργαλείο παρεμβάλλει κώδικα που δεν κάνει τίποτα και φροντίζει να μην εκτελεστεί ποτέ, κάνοντας χρήση ενός PF. Επιπλέον, μπορεί να γίνει συνδυασμός με την μέθοδο “Smoke And Mirror” και να γίνει χρήση ενός P?. Το αποτέλεσμα του predicate είναι άγνωστο αλλά δεν επηρεάζει την εκτέλεση του προγράμματος, διότι πρόκειται για τον ίδιο κώδικα και στις δυο περιπτώσεις.

Potency Υψηλό.

Resilience Ανά περίπτωση. Τα opaque predicates εξασφαλίζουν άμυνα κατά της στατικής ανάλυσης αλλά υπάρχει περίπτωση ο κώδικας που παρεμβάλλουμε να μπορεί να αναλυθεί εύκολα, άρα να αφαιρεθεί.

Cost Ανά περίπτωση.

Stealth Ανά περίπτωση.

2.1.3.5 Function Parallelization

Οι λειτουργίες μιας συνάρτησης χωρίζονται σε διαφορετικά threads ώστε να δυσκολεύουν την ανάλυση της εφαρμογής. Για παράδειγμα, ο κώδικας μιας συνάρτησης μπορεί να σπάσει σε πολλά ασύγχρονα threads, εφόσον δεν υπάρχουν data dependencies. Αντίστοιχα, εφόσον υπάρχουν data dependencies, τα threads αυτά πρέπει να είναι σύγχρονα.

Η μέθοδος αυτή μπορεί να συνδυαστεί με την μέθοδο Dead Code Injection, δημιουργώντας επιπλέον threads που δεν κάνουν τίποτα, μη αφήνοντας άλλη επιλογή στον επιτιθέμενο παρά να αναλύσει ξεχωριστά τα threads, ένα προς ένα.

Potency Υψηλό.

Resilience Υψηλό

Cost Υψηλό. Τα threads και ο "νεκρός" κώδικας εισάγουν επιπλέον καθυστέρηση στην εκτέλεση της εφαρμογής.

Stealth Υψηλό.

2.1.3.6 Loop Reversing / Artificial Data Dependencies

Αυτή η μέθοδος, δημιουργεί ψεύτικες εξαρτήσεις στις δομές επανάληψης ώστε να αυξήσει την ποσότητα της πληροφορίας και να μπερδέψει τα εργαλεία ξεθόλωσης. Για παράδειγμα, ένας απλός μετασχηματισμός θα ήταν η προσθήκη ενός πίνακα και η επεξεργασία αυτού μέσα από την δομή επανάληψης.

Σε κάθε περίπτωση, η αντιστροφή του μετρητή είναι δυνατή, παρόλο που δε προκαλεί ιδιαίτερη σύγχυση.

Potency Ανά περίπτωση. Εξαρτάται από την πολυπλοκότητα και την μέθοδο θόλωσης που θα εφαρμοστεί στα ψεύτικα δεδομένα.

Resilience Ανά περίπτωση. Εξαρτάται από την πολυπλοκότητα και την μέθοδο θόλωσης που θα εφαρμοστεί στα ψεύτικα δεδομένα.

Cost Ανά περίπτωση. Εξαρτάται από την πολυπλοκότητα και την μέθοδο θόλωσης που θα εφαρμοστεί στα ψεύτικα δεδομένα

Stealth Υψηλό.

2.2 Εργαλεία

2.2.1 Proguard

Το ProGuard (Lafortune 2002-2015) είναι ένα δωρεάν εργαλείο συρρίκνωσης, βελτιστοποίησης, θόλωσης και επαλήθευσης που επιδρά επάνω σε αρχεία class. Ανιχνεύει και αφαιρεί μή χρησιμοποιούμενες κλάσεις, πεδία και μεθόδους. Βελτιστοποιεί την εφαρμογή σε επίπεδο bytecode και αφαιρεί περιττές εντολές. Μετονομάζει τις κλάσεις, τα πεδία και τις μεθόδους χρησιμοποιώντας μικρά ονόματα χωρίς νόημα. Τέλος, επαληθεύει το επεξεργασμένο κώδικα.

2.2.1.1 Δυνατότητες

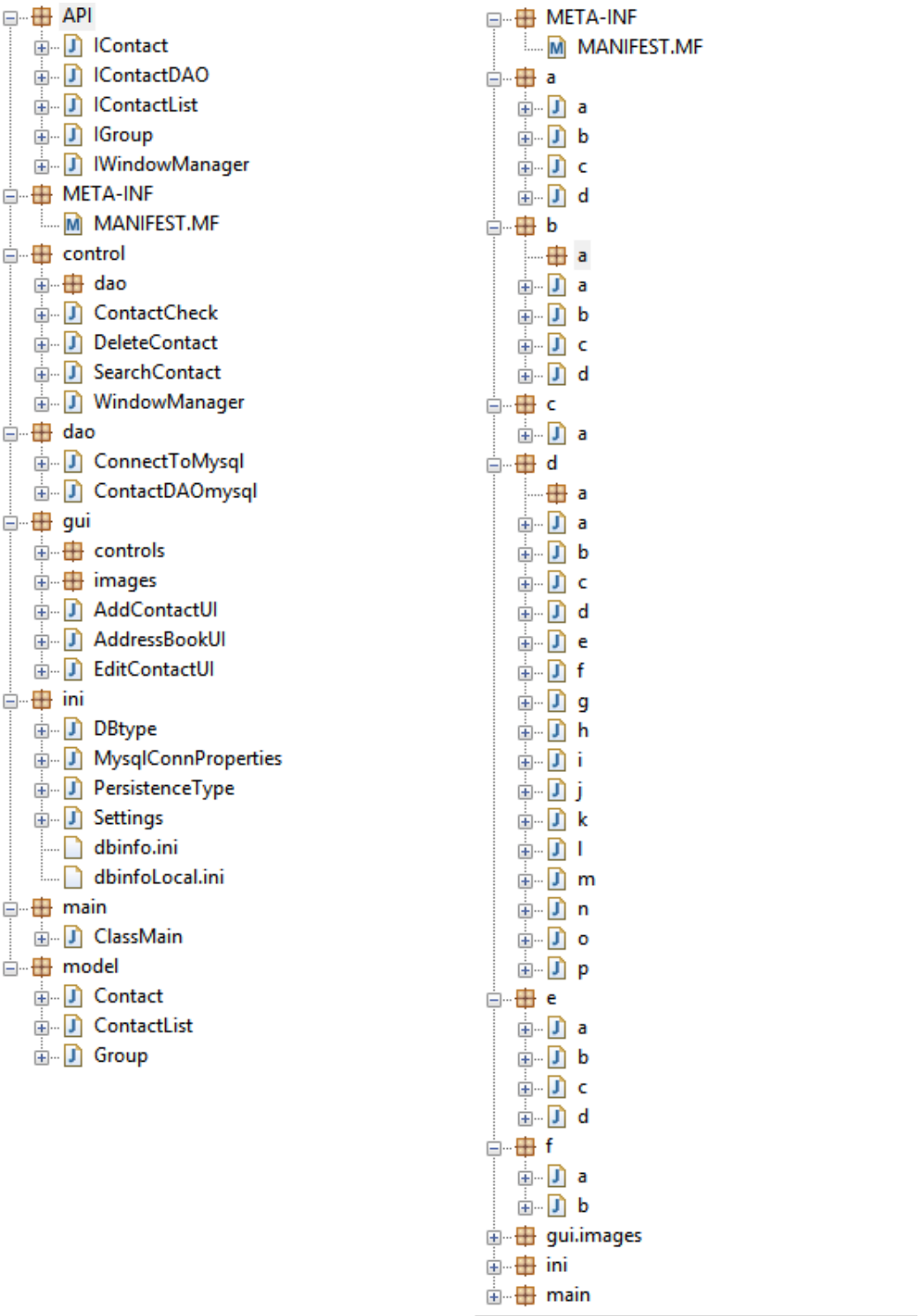
- Δημιουργία συμπαγούς κώδικα για γρηγορότερη φόρτωση, μεταφορά και εκτέλεση.
- Τροποποίηση κώδικα για να δυσκολέψει τον επιτιθέμενο που προσπαθεί να κατανοήσει τον κώδικα.
- Επανασχεδίαση των class αρχείων ώστε να κάνουν χρήση της γρηγορότερης τεχνικής φόρτωσης.
- Αυτόματο στην χρήση χωρίς να περιορίζει την ρύθμιση από τον χρήστη, εφόσον το επιθυμεί.

2.2.1.2 Σενάριο χρήσης

Το ProGuard δέχεται ως είσοδο ένα αρχείο jar και βγάζει ως έξοδο ένα άλλο αρχείο jar το οποίο έχει υποστεί τις τροποποιήσεις που παρουσιάστηκαν παραπάνω. Παρακάτω παρουσιάζονται ενδεικτικά τα αποτελέσματα της εκτέλεσης.

Αρχικό	Τροποποιημένο
96.9 KB	29.5 KB

Πίνακας 2.1: Μεγέθη ενδεικτικών αρχείων Jar πριν και μετά την τροποποίηση



Διάγραμμα 2.1: Θόλωση των πακέτων και ιεραρχιών

Contact.class

```
package model;

import API.IContact;

public class Contact
  implements IContact
{
  protected String id;
  protected String firstname;
  protected String lastname;
  protected String email;
  protected String telephone;
  protected String mtelephone;
  protected String address;
  protected String city;

  public Contact(String firstname, String telephone)
  {
    this.firstname = firstname;
    this.telephone = telephone;
    this.id = "";
    this.lastname = "";
    this.email = "";
    this.mtelephone = "";
    this.address = "";
    this.city = "";
  }

  public Contact()
  {
    this.id = "";
    this.firstname = "";
    this.lastname = "";
    this.email = "";
    this.telephone = "";
    this.mtelephone = "";
    this.address = "";
    this.city = "";
  }
```

a.class

```
package f;

public final class a
  implements a.a
{
  private String id;
  private String ag;
  private String ah;
  private String ai;
  private String aj;
  private String ak;
  private String address;
  private String al;

  public a(String paramString1, String paramString2)
  {
    this.ag = paramString1;
    this.aj = paramString2;
    this.id = "";
    this.ah = "";
    this.ai = "";
    this.ak = "";
    this.address = "";
    this.al = "";
  }

  public a()
  {
    this.id = "";
    this.ag = "";
    this.ah = "";
    this.ai = "";
    this.aj = "";
    this.ak = "";
    this.address = "";
    this.al = "";
  }
```

Διάγραμμα 2.2: Αρχική και τροποποιημένη κλάση

Στο διάγραμμα 2.1 βλέπουμε το αποτέλεσμα σε επίπεδο πακέτων. Τα πακέτα έχουν χωριστεί σε περισσότερα απ'ότι τα αρχικά και τα ονόματα αυτών έχουν αντικατασταθεί με ονόματα χωρίς νόημα.

Στο διάγραμμα 2.2 βλέπουμε το αποτέλεσμα σε επίπεδο πηγαίου κώδικα. Το όνομα της κλάσης, τα ονόματα των μεταβλητών της κλάσης, οι συναρτήσεις και οι παράμετροι αυτών έχουν μετονομαστεί σε αναγνωριστικά χωρίς νόημα.

2.2.1.3 Πώς λειτουργεί

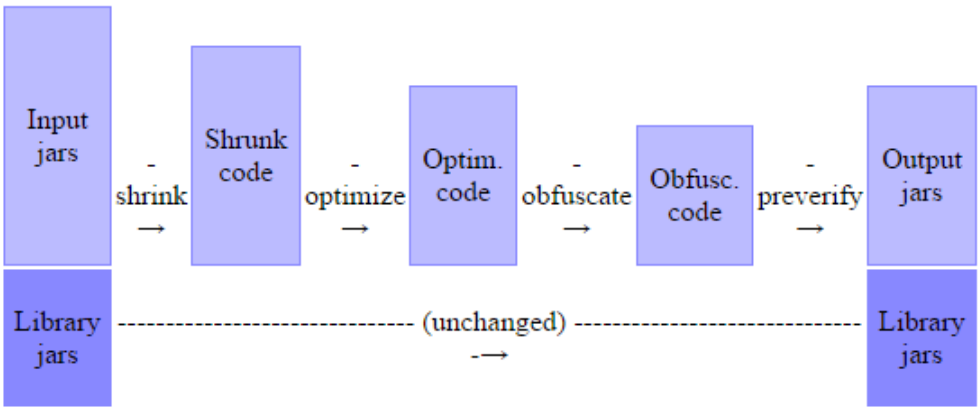
Ο χρήστης εισάγει τα αρχεία jar που επιθυμεί να επεξεργαστεί καθώς και τα αρχεία jar των βιβλιοθηκών που χρησιμοποιούνται από την εφαρμογή του. Στη συνέχεια το ProGuard επεξεργάζεται σταδιακά τον κώδικα της εφαρμογής. Τα στάδια επεξεργασίας παρουσιάζονται και αναλύονται παρακάτω.

Shrinking Αναλύει το bytecode και αφαιρεί κλάσεις, μεθόδους και πεδία που δε χρησιμοποιούνται πουθενά μέσα στην εφαρμογή

Optimisation Αναλύει τις ιεραρχίες, την ροή της εκτέλεσης και των δεδομένων και αφαιρεί αχρείαστους ελέγχους τύπων και συγκρίσεις, αφαιρεί μικρές μεθόδους και αντικαθιστά τις κλήσεις με τον κώδικα τους, αλλάζει τα επίπεδα πρόσβασης με σκοπό την ελάχιστη έκθεση και υλοποιεί διάφορες λοιπές βελτιώσεις.

Obfuscation Αντικαθιστά τα αναγνωριστικά των αρχείων, κλάσεων, μεθόδων και μεταβλητών με τυχαία αλφαριθμητικά. Επιπλέον, αφαιρεί τους αριθμούς γραμμών από τα αρχεία class.

Preverification Ελέγχει το τελικό jar για προβλήματα που ενδεχομένως να παρέμβουν στη σωστή λειτουργία του JVM και πιστοποιεί το jar με κάποιες ιδιότητες οι οποίες θα διευκολύνουν την εκτέλεση του.



Διάγραμμα 2.3: Τρόπος λειτουργίας του ProGuard

2.2.2 DashO

Το DashO (PreEmptiveSolutions 2016) είναι ένα εμπορικό προϊόν της εταιρίας PreEmptive Solutions το οποίο, εκτός από τη θόλωση των αναγνωριστικών, εστιάζει σε άλλες τεχνικές συρρίκνωσης, βελτιστοποίησης, και θόλωσης.

2.2.2.1 Δυνατότητες

- Το DashO επιδρά επάνω στη ροή εκτέλεσης του προγράμματος προσθέτοντας εντολές και αφαιρώντας τα πρότυπα που χρησιμοποιούν οι decompilers κατά μια επίθεση. Τα αποτελέσματα αυτού του μετασχηματισμού φαίνονται στο διάγραμμα 2.4.

Original Source Code Before Obfuscation	Reverse-Engineered Source Code After Control Flow Obfuscation

public int CompareTo(Object o) {
int n = occurrences –
((WordOccurrence)o).occurrences;
if (n == 0) {
n = String.Compare(word,
((WordOccurrence)o).word);
}
return(n);
}
	public virtual int _a(Object A_0) {
	int local0;
	int local1;
	local 10 = this.a – (c) A_0.a;
	if (local0 != 0) goto i0;
	while (true) {
	return local1;
	i0: local1 = local10;
	}
	i1: local10 = System.String.Compare(this.b, (c)
	A_0.b);
	goto i0;
	}

Διάγραμμα 2.4: DashO control flow obfuscation

- Τα αλφαριθμητικά κρυπτογραφούνται ώστε ο επιτιθέμενος να μη μπορεί να βρει τα κρίσιμα σημεία του προγράμματος, όπως ελέγχους για τη νόμιμη χρήση του λογισμικού.

- Η δυνατότητα του watermarking προσθέτει μοναδικά αναγνωριστικά μέσα στο εκτελέσιμο ώστε να είναι δυνατός ο εντοπισμός του χρήστη που κάνει παράνομη χρήση του λογισμικού.

- Η δυνατότητα tamper detection ελέγχει το εκτελέσιμο της εφαρμογής και την τερματίζει εφόσον εντοπιστούν τροποποιήσεις. Επίσης, εφόσον ο δημιουργός της εφαρμογής αγοράσει επιπλέον πακέτο από την εταιρία, το DashO τον ειδοποιεί για την πηγή και τη φύση των τροποποιήσεων.
- Τέλος, το DashO προσφέρει στους δημιουργούς τη δυνατότητα να φτιάξουν δοκιμαστικές εκδόσεις της εφαρμογής τους οι οποίες σταματούν να δουλεύουν μετά το πέρας κάποιου χρονικού διαστήματος.

Κεφάλαιο 3

Ανάλυση εργαλείων που χρησιμοποιήθηκαν

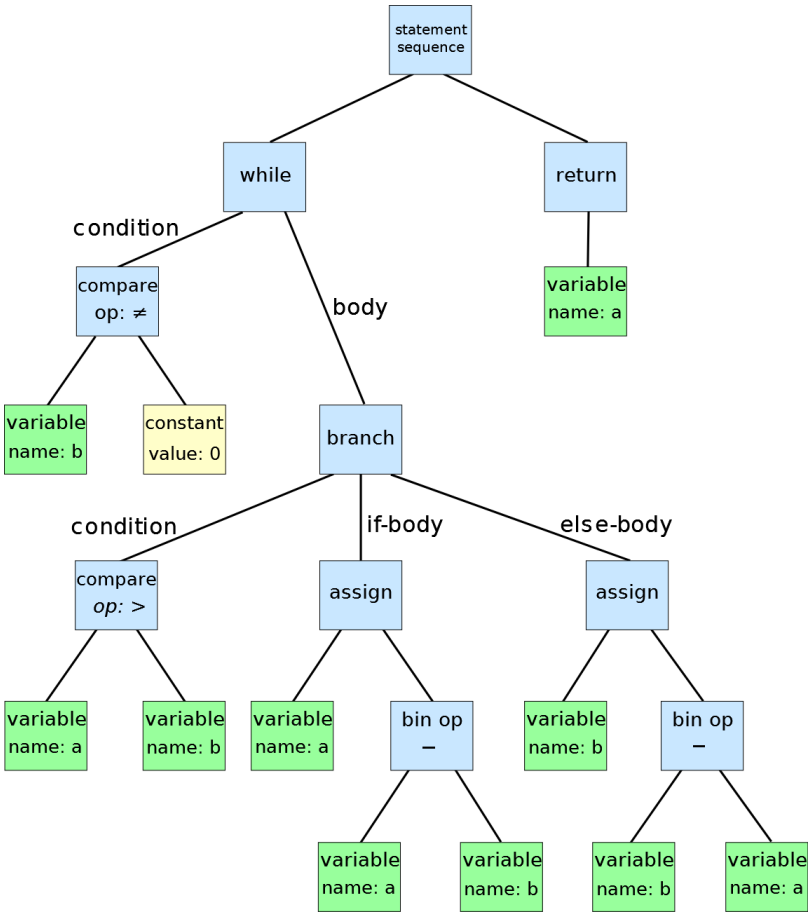
3.1 Eclipse JDT

3.1.1 Περιγραφή

Είναι ένα framework ανοιχτού κώδικα το οποίο αναλύει αρχεία Java και προσφέρει APIs για την διάσχιση του συντακτικού δέντρου τους, καθώς και για την τροποποίηση αυτού.

3.1.1.1 AST

Ένα αφηρημένο συντακτικό δένδρο (Abstract Syntax Tree) είναι μια δενδρική απεικόνιση της συντακτικής δομής του πηγαίου κώδικα. Κάθε κόμβος αυτού του δένδρου υποδηλώνει μια δομή στον πηγαίο κώδικα. Χαρακτηρίζεται ως αφηρημένο διότι δε περιλαμβάνει όλες τις πληροφορίες του πηγαίου κώδικα. Για παράδειγμα, οι παρενθέσεις μιας δομής if-else παραλείπονται στην αναπαράσταση της ως AST. Στο διάγραμμα 3.1 παρουσιάζεται το AST του αλγορίθμου του Ευκλείδη. Στο διάγραμμα 3.2 παρουσιάζεται ο πηγαίος κώδικας από τον οποίο προκύπτει το παραπάνω AST.



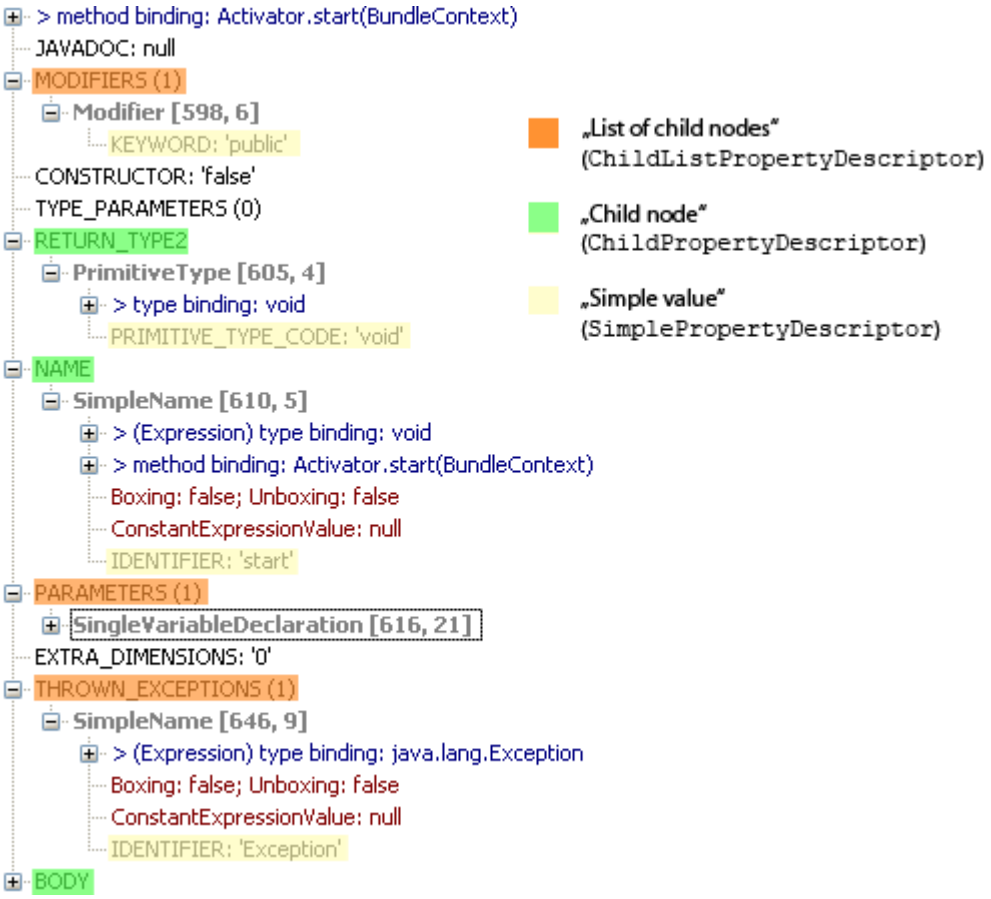
Διάγραμμα 3.1: AST of the Euclidean algorithm

```
while b ≠ 0
    if a > b
        a := a - b
    else
        b := b - a
return a
```

Διάγραμμα 3.2: Euclidean algorithm in pseudocode

Το Eclipse JDT χρησιμοποιεί ASTs εσωτερικά για να αναλύσει τον πηγαίο κώδικα java. Όλα τα αρχεία πηγαίου κώδικα αναπαριστάνονται ως κόμβοι ενός δένδρου. Όλοι αυτοί οι κόμβοι είναι υποκλάσεις της αφηρημένης κλάσης ASTNode. κάθε υποκλάση αναπαριστά κάποιο συγκεκριμένο στοιχείο της γλώσσας Java. Για παράδειγμα, υπάρχουν κόμβοι για δηλώσεις μεθόδων (MethodDeclaration), δηλώσεις μεταβλητών (VariableDeclarationFragment), εκχωρήσεις(Assignment) κ.ο.κ.. Ένας κόμβος που χρησιμοποιείται συχνά είναι ο SimpleName. Ένα SimpleName είναι οποιοδήποτε αλφαριθμητικό το οποίο δεν είναι λέξη κλειδί, τιμή boolean ή null. Για παράδειγμα, στην έκφραση ”i = 6 + j;”, το i και το j αναπαριστάνονται από κόμβους SimpleName.

Όλες οι κλάσεις που εμπλέκονται με το AST περιέχονται στο πακέτο org.eclipse.jdt.core.dom του Eclipse JDT framework.



Διάγραμμα 3.3: Graphical representation of a MethodDeclaration AST

3.1.1.2 ASTParser

Το AST ενός αρχείου πηγαίου κώδικα δημιουργείται μετά από ανάλυση του κώδικα Java. Αυτό επιτυγχάνεται χρησιμοποιώντας την κλάση ASTParser, η οποία δίνει τη δυνατότητα επεξεργασίας ολόκληρων αρχείων πηγαίου κώδικα ή μεμονωμένων μλοκ κώδικα. Ο ASTParser πρέπει να ρυθμιστεί για να εκτελέσει τις λειτουργίες που παρουσιάστηκαν παραπάνω. Η ρύθμιση γίνεται μέσω της μεθόδου setKind(..) η οποία δέχεται τις παρακάτω παραμέτρους :

- K_COMPILATION_UNIT
- K_EXPRESSION
- K_STATEMENTS
- K_CLASS_BODY_DECLARATIONS

3.1.2 Δυνατότητες

- Compiler ο οποίος μπορεί να τρέξει και να αποσφαλματώσει κώδικα που περιέχει συντακτικά λαθη.
- API για την πλοήγηση στο συντακτικό δέντρο ενός αρχείου Java.
- API για την τροποποίηση του συντακτικού δέντρου ενός αρχείου Java.
- Μηχανή αναζήτησης η οποία προσφέρει αναζήτηση οποιουδήποτε κόμβου, υπολογισμό ιεραρχιών και δυνατότητα επανασχεδιασμού από κάποιο αρχείο Java ή δυαδικό αρχείο.
- Μηχανή εκτίμησης εκφράσεων.
- Μηχανή μορφοποίησης κώδικα.

3.1.3 Σενάρια χρήσης

3.1.3.1 Θόλωση παραμέτρων μεθόδου

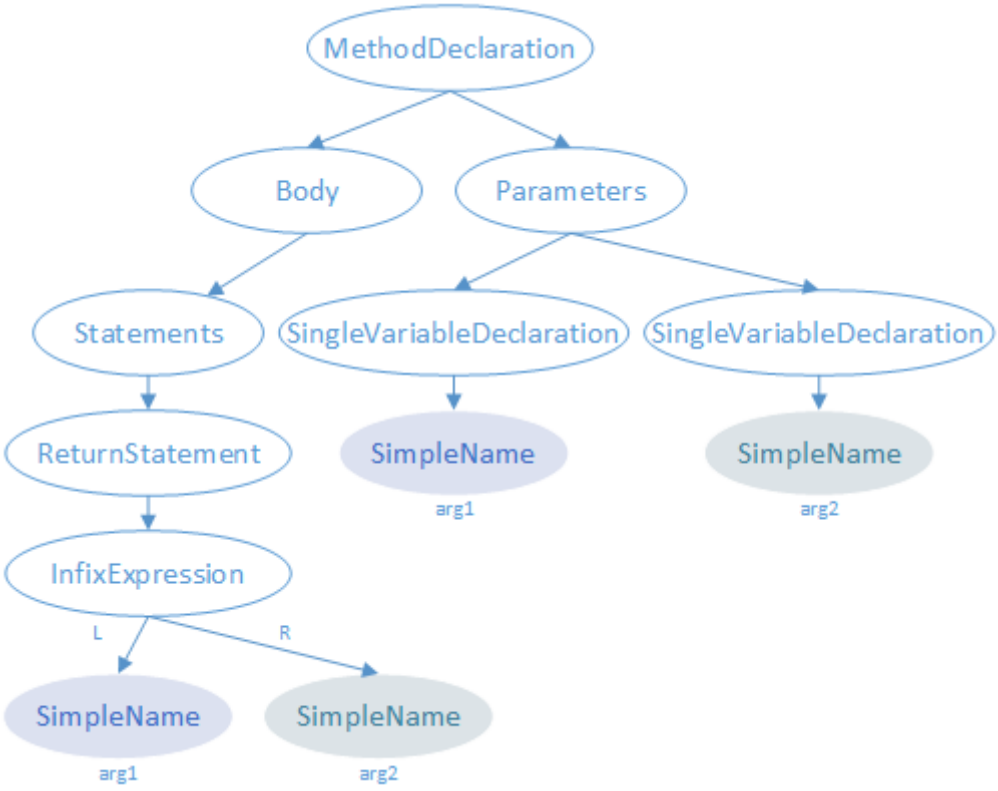
Έστω ότι έχουμε ένα αρχείο κώδικα γραμμένο σε Java. Θέλουμε να τροποποιήσουμε τα ονόματα των παραμέτρων της μεθόδου, καθώς και τις αναφορές αυτών μέσα

στην μέθοδο.

```
1
2  public class Example {
3
4      public int sumInts(int arg1, int arg2) {
5          return arg1 + arg2;
6      }
7  }
```

Διάγραμμα 3.4: Η μέθοδος πριν την θόλωση παραμέτρων

Στο διάγραμμα 3.4 έχουμε μια κλάση με μια απλή μέθοδο με δύο παραμέτρους τις οποίες θέλουμε να μετονομάσουμε με βάση κάποια στρατηγική. Αυτή μπορεί να είναι λατινικοί χαρακτήρες με οποιαδήποτε σειρά, τυχαία παραγόμενα ονόματα ή οτιδήποτε άλλο που συμφωνεί με τους κανόνες ονοματολογίας αναγνωριστικών στην Java.



Διάγραμμα 3.5: Το AST της μεθόδου

Στο διάγραμμα 3.5 έχουμε το AST της μεθόδου του διαγράμματος 3.4 όπως προκύπτει (εσωτερικά) απο το Eclipse JDT. Στόχος μας είναι να τροποποιήσουμε τα φύλλα ”SimpleName”, τα οποία αντιπροσωπεύουν αναγνωριστικά. Μέσω του API του Eclipse

JDT μπορούμε να συγκεντρώσουμε τις παραμέτρους της μεθόδου, καθώς και τις αναφορές αυτών μέσα στην μέθοδο, και να τις μετονομάσουμε με βάση την στρατηγική της επιλογής μας. Επίσης μπορούμε να κατασκευάσουμε προγραμματιστικά οποιαδήποτε δομή υποστηρίζει η Java και να επεξεργαστούμε το πρόγραμμα με οποιοδήποτε τρόπο.

Στο διάγραμμα 3.6 βλέπουμε ότι οι παράμετροι της μεθόδου, καθώς και οι αναφορές τους, έχουν αντικατασταθεί.

```
5
6  public class Example {
7
8      public int sumInts(int a, int b) {
9          return a + b;
10     }
11 }
```

Διάγραμμα 3.6: Η μέθοδος μετά την θόλωση παραμέτρων

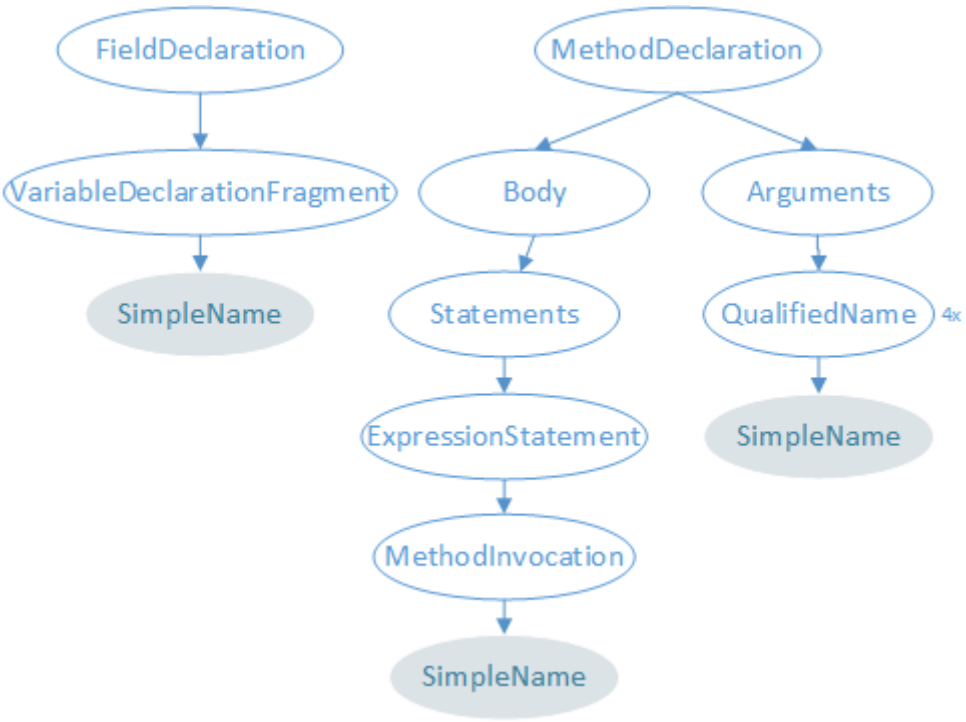
3.1.3.2 Θόλωση μεταβλητών κλάσης

Έστω ότι έχουμε ένα αρχείο κώδικα γραμμένο σε Java. Θέλουμε να τροποποιήσουμε τα ονόματα των μεταβλητών της κλάσης, καθώς και τις αναφορές αυτών μέσα στις μεθόδους της κλάσης.

```
4
5  public class Example {
6      private Rectangle rectangle;
7
8      public Example()
9      {
10         rectangle.set(
11                 rectangle.left,
12                 rectangle.top,
13                 rectangle.right,
14                 rectangle.bottom
15         );
16     }
```

Διάγραμμα 3.7: Η κλάση πριν την θόλωση παραμέτρων

Στο διάγραμμα 3.7 έχουμε μια κλάση με μια απλή μέθοδο με δύο παραμέτρους τις οποίες θέλουμε να μετονομάσουμε με βάση κάποια στρατηγική.



Διάγραμμα 3.8: Το AST της κλάσης

Στο διάγραμμα 3.8 έχουμε το AST της μεθόδου του διαγράμματος 3.7 όπως προ-

κύπτει (εσωτερικά) απο το Eclipse JDT. Στόχος μας είναι να τροποποιήσουμε τα φύλλα ”SimpleName”, τα οποία αντιπροσωπεύουν αναγνωριστικά.

```
4
5  public class Example {
6      private Rectangle a;
7
8      public Example()
9      {
10         a.set(
11                 a.left,
12                 a.top,
13                 a.right,
14                 a.bottom
15         );
16     }
```

Διάγραμμα 3.9: Η κλάση μετά την θόλωση μεταβλητών

Στο διάγραμμα 3.9 βλέπουμε ότι η μεταβλητή της κλάσης, καθώς και οι αναφορές αυτής, έχουν αντικατασταθεί.

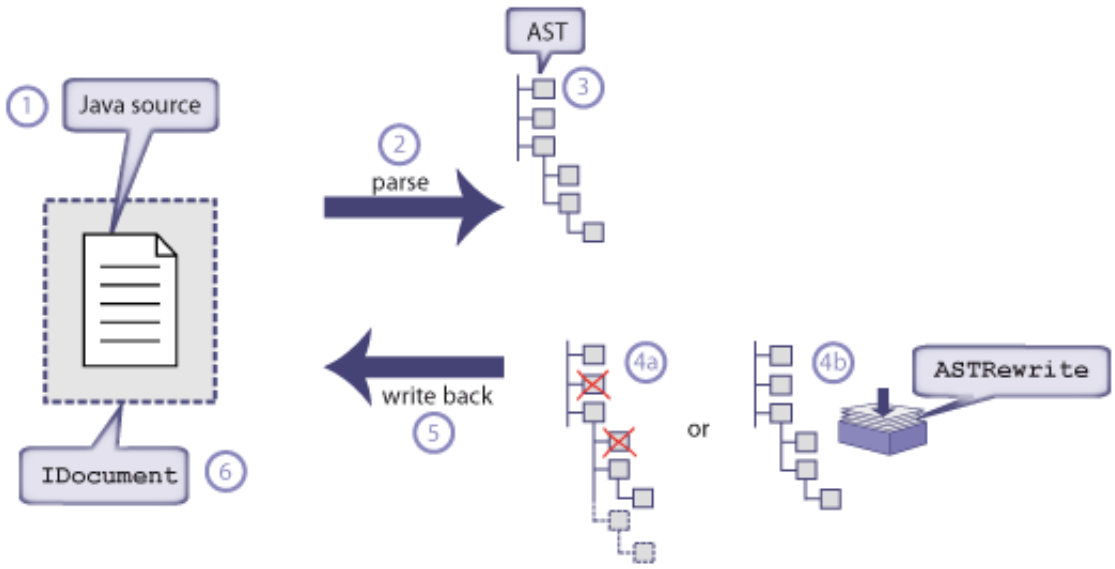
3.1.4 Πώς λειτουργεί

Εσωτερικά, κάθε κόμβος του AST περιέχει επιπλέον πληροφορίες που σχετίζονται με αυτόν (binding). Ένας κόμβος δήλωσης μεταβλητής περιέχει, εκτός του ονόματος του και τη γραμμή που γίνεται η δήλωση, αντίγραφο της μεθόδου στην οποία γίνεται η δήλωση (ή της κλάσης εάν είναι μεταβλητή κλάσης) και πληροφορίες για το αν πρόκειται για παράμετρο μεθόδου, μεταβλητή κλάσης, σταθερά ή τελική μεταβλητή. Επίσης, περιέχεται πληροφορία για τυχόν αρχική τιμή και το scope.

Οι τροποποιήσεις που γίνονται πάνω στο AST δεν γράφονται αμέσως στο αρχείο Java αλλά καταγράφονται εσωτερικά στο Eclipse JDT, μέχρις ότου ο χρήστης να ζητήσει την εφαρμογή τους. Παράλληλα δημιουργείται και ένα αρχείο καταγραφής, το οποίο μπορεί να χρησιμέψει για αντιστροφή της διαδικασίας. Στο διάγραμμα 3.11 αναπαριστάται γραφικά η διαδικασία τροποποίησης του AST.

```
SingleVariableDeclaration [58+8]
  > variable binding: arg2
      NAME: "arg2"
      KEY: "LExample;.sumInts(II)I#arg2"
      IS RECOVERED: false
      IS FIELD: false
      IS ENUM CONSTANT: false
      IS PARAMETER: true
      VARIABLE ID: 1
      MODIFIERS: 0x0 ()
    > TYPE: int
      DECLARING CLASS: null
    > DECLARING METHOD: Example.sumInts(int, int)
    > VARIABLE DECLARATION: arg2
      IS SYNTHETIC: false
      IS DEPRECATED: false
      CONSTANT VALUE: null
      IS EFFECTIVELY FINAL: true
      ANNOTATIONS (0)
      > LocalVariable: Example.sumInts(int, int).arg2 : int
    MODIFIERS (0)
  > TYPE
    VARARGS_ANNOTATIONS (0)
    VARARGS: 'false'
  NAME
    > SimpleName [62+4]
    EXTRA_DIMENSIONS2 (0)
    INITIALIZER: null
EXTRA_DIMENSIONS2 (0)
THROWN_EXCEPTION_TYPES (0)
```

Διάγραμμα 3.10: Το Binding ενός κόμβου SingleVariableDeclaration



Διάγραμμα 3.11: Διαδικασία τροποποίησης του AST

Κεφάλαιο 4

Ανάλυση και υλοποίηση του συστήματος

Το σύστημα που υλοποιήθηκε είναι ένα εργαλείο θόλωσης το οποίο επιδρά πάνω σε αρχεία πηγαίου κώδικα Java.

Οι λειτουργίες που υλοποιήθηκαν είναι οι εξής :

- Αλλαγή ονομάτων μεταβλητών και υπογραφών συναρτήσεων όπως περιγράφηκε στο κεφάλαιο 2.1.1.1, καθώς αποτελεί τη βάση πολλών άλλων εργαλείων θόλωσης (Witkowska 2006).
- Αλλαγή ονομάτων αρχείων πηγαίου κώδικα, μια λειτουργία που υλοποιεί το ProGuard (Lafortune 2002-2015) το οποίο αναλύθηκε στο κεφάλαιο 2.2.1.

Το σύστημα σχεδιάστηκε έτσι ώστε να περνά από στάδια τα οποία υλοποιούν αυτόνομες λειτουργίες. Με αυτή τη προσέγγιση γίνεται δυνατή η ανάλυση, βελτίωση, επέκταση ή και η εξολοκλήρου αντικατάσταση ενός σταδίου του συστήματος. Παρακάτω αναφέρουμε τα στάδια από τα οποία περνά το σύστημα καθώς και τα κεφάλαια στα οποία τα αναλύουμε περαιτέρω :

- Στάδιο εξαγωγής μονοπατιών στο κεφάλαιο 4.1
- Στάδιο συλλογής κόμβων στο κεφάλαιο 4.2
- Στάδιο ομαδοποίησης κόμβων στο κεφάλαιο 4.3.2.1
- Στάδιο φιλτραρίσματος και επεξεργασίας κόμβων στο κεφάλαιο 4.4

- Τελικό στάδιο αποθήκευσης τροποποιήσεων στο κεφάλαιο 4.5

4.1 Εξαγωγή μονοπατιών

Ένα σύστημα λογισμικού συνήθως περιέχει, εκτός από τα αρχεία κώδικα (π.χ. .java, .cpp), επιπλέον αρχεία διαφόρων τύπων. Μερικά παραδείγματα είναι τα αρχεία εικόνων, αρχεία ρυθμίσεων και αρχεία σήμανσης.

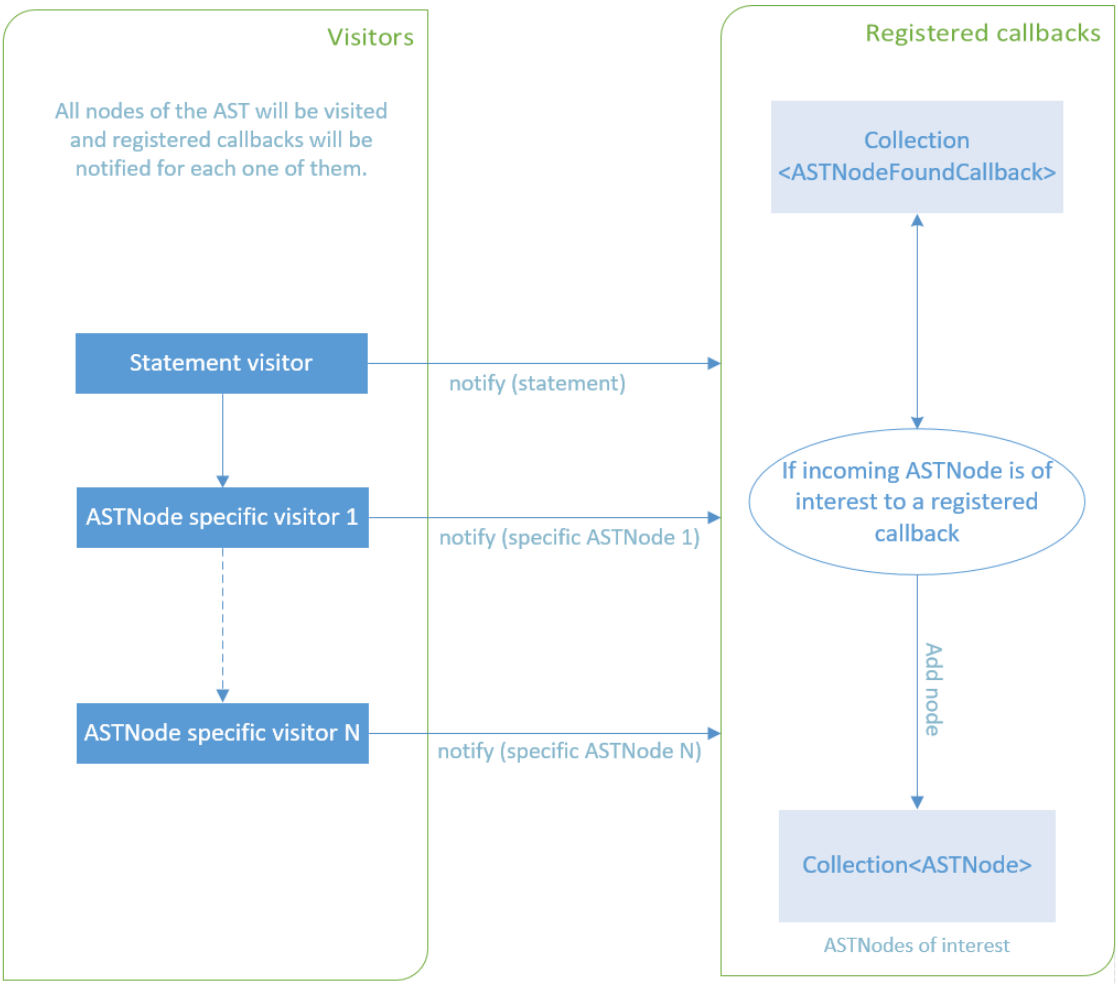
Το σύστημα προσφέρει την δυνατότητα εύρεσης και εξαγωγής μονοπατιών ενός συγκεκριμένου τύπου αρχείων το οποίο επιλέγει ο προγραμματιστής, ανάλογα με τις ανάγκες της λειτουργίας που θέλει να υλοποιήσει.

Οι είσοδοι του συστήματος είναι ένα μονοπάτι που αντιπροσωπεύει ένα φάκελο και ο επιθυμητός τύπος αρχείων. Όλοι οι υποφάκελοι επισκέπτονται αναδρομικά ανεξάρτητα από το επίπεδό τους. Η έξοδος του συστήματος είναι μια συλλογή από απόλυτα μονοπάτια.

4.2 Συλλογή κόμβων

Για την συλλογή των κόμβων, το σύστημα κάνει χρήση του προτύπου σχεδίασης ”επισκέπτης” (”GangOfFour” 1994). Το πρότυπο ”επισκέπτης” έχει στόχο τον διαχωρισμό ενός αλγορίθμου από τη δομή στην οποία επιδρά. Το αποτέλεσμα αυτού του διαχωρισμού είναι η δυνατότητα προσθήκης νέων λειτουργιών σε ήδη υπάρχουσες δομές, χωρίς να απαιτείται τροποποίηση αυτών των δομών.

Το Eclipse JDT παρέχει αφηρημένες κλάσεις επισκέπτη για κάθε κόμβο του AST τις οποίες ο προγραμματιστής υλοποιεί όπως επιθυμεί. Στο σύστημα έχουν υλοποιηθεί κλάσεις επισκέπτη για κάθε κόμβο οι οποίες, πριν επισκεφθούν τον εκάστοτε κόμβο, δέχονται μια συλλογή από callbacks. Κατά την επίσκεψη ενός κόμβου ειδοποιούνται όλα τα callbacks για την ύπαρξη του κόμβου και τους παρέχεται η αναφορά αυτού. Μια αφηρημένη κλάση, την οποία πρέπει να υλοποιούν όλα τα callbacks, διατηρεί μια συλλογή κόμβων στην οποία ο επισκέπτης προσθέτει τις αναφορές των κόμβων. Η παραπάνω υλοποίηση φαίνεται στο διάγραμμα 4.1.



Διάγραμμα 4.1: Συλλογή κόμβων

Στο διάγραμμα 4.2 βλέπουμε την υλοποίηση του προτύπου ”επισκέπτης” για τις δομές If. Οι ενδιαφερόμενοι (callbacks) ειδοποιούνται για την ύπαρξη της επισκεπτόμενης δομής If και τους παρέχεται η αναφορά αυτής. Στην συνέχεια, ο ”επισκέπτης” της δομής If συνεχίζει την προσπέλαση της δημιουργώντας και καλώντας επισκέπτες για τα περιεχόμενά της. Με την παραπάνω διαδικασία έχουμε πετύχει το διαχωρισμό των αλγορίθμων που θα εκτελεστούν (περιεχόμενο των callbacks) από την ίδια τη δομή If (IfStatement), χωρίς να κάνουμε κάποια τροποποίηση σε αυτή.

```java
public class IfStatementVisitor extends ASTVisitor
{

    private Collection<AstNodeFoundCallback> callbacks;

    public IfStatementVisitor ( Collection<AstNodeFoundCallback> callbacks )
    {
        this.callbacks = callbacks;
    }

    @Override
    public boolean visit ( IfStatement ifStatement )
    {
        this.callbacks.stream().forEach( c -> c.notify( ifStatement ) );

        new ExpressionVisitor( this.callbacks ).visit( ifStatement.getExpression() );
        new StatementVisitor( this.callbacks ).visit( ifStatement.getThenStatement() );

        Statement elseStatement = ifStatement.getElseStatement();
        if ( elseStatement != null )
        {
            new StatementVisitor( this.callbacks ).visit( elseStatement );
        }
        return false;
    }
}
```

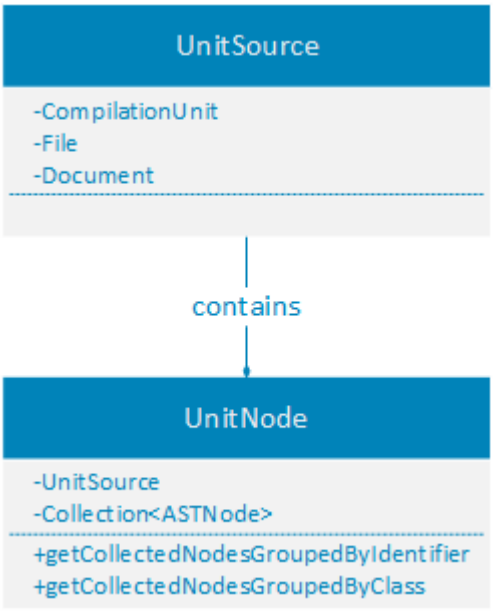
Διάγραμμα 4.2: Υλοποίηση προτύπου ”επισκέπτη” για τη δομή If

4.3 Εσωτερικές δομές

Στο σύστημα έχουν υλοποιηθεί δύο δομές για τον λογικό διαχωρισμό των δεδομένων και χρησιμοποιούνται καθ’όλη τη διάρκεια της εκτέλεσης του. Η υλοποίηση παρουσιάζεται στο διάγραμμα 4.3

4.3.1 UnitSource

Η κλάση UnitSource έχει αντιστοιχία ένα προς ένα με τα αρχεία του πηγαίου κώδικα της εφαρμογής που θέλουμε να θολώσουμε. Το πεδίο File περιέχει το απόλυτο μονοπάτι του αρχείου. Τα πεδία Document και CompilationUnit είναι εσωτερικές δομές του Eclipse JDT και περιέχουν τον πηγαίο κώδικα ως κείμενο και την ανάλυση αυτού ως AST αντίστοιχα. Το CompilationUnit μας δίνει την δυνατότητα ανάλυσης και τροποποίησης του αρχικού αρχείου πηγαίου κώδικα και παρέχει επιπλέον πληροφορίες μέσω του Binding του κάθε κόμβου του AST.



Διάγραμμα 4.3: UML class diagram εσωτερικών δομών

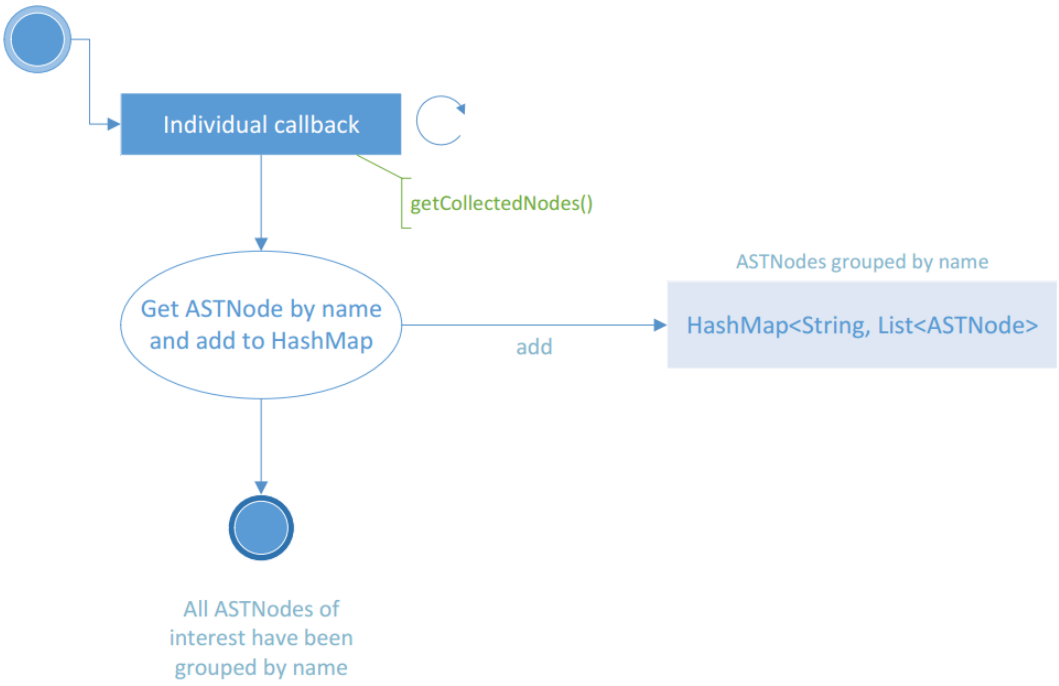
4.3.2 UnitNode

Η κλάση UnitNode έχει αντιστοιχία ένα προς ένα με την κλάση UnitSource καθώς περιέχει ένα αντικείμενο αυτής. Ο σκοπός της είναι η συσχέτιση ενός UnitSource με τους κόμβους του πηγαίου αρχείου, όπως αυτοί προέκυψαν από τη διαδικασία συλλογής κόμβων, και η ομαδοποίηση αυτών ανάλογα με τις απαιτήσεις του πελάτη.

4.3.2.1 Ομαδοποίηση κόμβων

Για τις ανάγκες της εφαρμογής έχουν δημιουργηθεί δύο μέθοδοι ομαδοποίησης, η getCollectedNodesGroupedByIdentifier και η getCollectedNodesGroupedByClass, όπως φαίνονται στο διάγραμμα 4.3. Η πρώτη μέθοδος ομαδοποιεί τους κόμβους σύμφωνα με το αναγνωριστικό τους και επιστρέφει μια λίστα με αυτούς. Η δεύτερη μέθοδος ομαδοποιεί τους κόμβους σύμφωνα με τον τύπο τους και επιστρέφει μια λίστα με αυτούς. Για παράδειγμα, εάν ο προγραμματιστής θέλει μια λίστα με τη μεταβλητή ”name” καθώς και όλες τις αναφορές της, μπορεί να κάνει get επάνω στο χάρτη και αποκτήσει τη λίστα με όλους τους κόμβους που τον ενδιαφέρουν. Το ίδιο ισχύει και στην περίπτωση που ο προγραμματιστής θέλει μια λίστα από κάποιο τύπο, όπως είναι το MethodDeclaration. Η ανάγκη αυτή εξυπηρετείται από τη δεύτερη μέθοδο. Η διαδικασία που ακολουθείται είναι παρόμοια και στις δύο παραπάνω μεθόδους. Ενδεικτικά

παρουσιάζεται η ομαδοποίηση ανά αναγνωριστικό στο διάγραμμα 4.4



Διάγραμμα 4.4: Ομαδοποίηση κόμβων ανά αναγνωριστικό

Η ομαδοποίηση κόμβων είναι χρήσιμη διότι επιτρέπει στον προγραμματιστή να επεξεργαστεί κόμβους μαζικά χωρίς να τον απασχολεί η εύρεσή αυτών. Ένα παράδειγμα είναι η μετονομασία μιας μεταβλητής κλάσης καθώς και όλων των αναφορών της. Το μόνο που έχει να κάνει ο προγραμματιστής είναι να πάρει από τον χάρτη τη λίστα με τους κόμβους και να τους μετονομάσει με κάποια δομή επανάληψης.

4.4 Επεξεργασία κόμβων

Το Eclipse JDT προσφέρει μεθόδους για απευθείας τροποποίηση των κόμβων. Σε ένα σενάριο θόλωσης των μεθόδων μίας κλάσης, ο προγραμματιστής ζητά τη λίστα των μεθόδων αυτής και τις μετονομάζει με βάση κάποια λίστα με νέα (θολωμένα) ονόματα. Στη συνέχεια, το όνομα της μεθόδου αναζητείται μέσα από τον χάρτη με τους ομαδοποιημένους κόμβους και μετονομάζονται όλες οι αναφορές της μεθόδου σε ολόκληρη την εφαρμογή.

4.4.1 Φιλτράρισμα

Σε πολλά στάδια επεξεργασίας κόμβων υπάρχουν μέθοδοι φιλτραρίσματος με βάση κάποιους κανόνες. Οι πιο βασικοί από αυτούς είναι οι εξής :

- Η μέθοδος main, καθώς και ολόκληρη η ιεραρχία πακέτων του αρχείου που την περιέχει, απαγορεύεται να θολωθούν διότι το τελικό jar της εφαρμογής περιέχει αναφορά αυτών στο αρχείο manifest. Σε περίπτωση που παρέμβουμε στα παραπάνω, το πρόγραμμα δε θα είναι δυνατό να εκτελεστεί.

- Οι μέθοδοι που έχουν annotation @Override φιλτράρονται κατά το στάδιο της θόλωσης μεθόδων διότι απαιτούν ειδική μεταχείριση σε περιπτώσεις υλοποίησης μιας διεπαφής ή περιπτώσεις κληρονομικότητας.

- Οι κλάσεις που υλοποιούν τη διεπαφή Serializable απαγορεύεται να θολωθούν διότι η διαδικασία του serialization χρησιμοποιεί τις υπογραφές των μεθόδων της κλάσης για να δημιουργήσει ένα μοναδικό αναγνωριστικό που απαιτεί η διαδικασία. Τα Enumerations είναι μια περίπτωση η οποία απαιτεί προσοχή, διότι υλοποιούν τη διεπαφή Serializable, άρα εντάσσονται στον παραπάνω κανόνα και απαγορεύεται να θολωθούν.

4.5 Αποθήκευση τροποποιήσεων

Στο τελικό στάδιο εκτέλεσης του εργαλείου χρησιμοποιείται η μέθοδος saveUnitSourcesToFiles η οποία δέχεται ως παράμετρο ένα Collection<UnitSource>. Το Collection είναι το ίδιο που προέκυψε στο αρχικό στάδιο της εκτέλεσης του εργαλείου και περιέχει τα τροποποιημένα αρχεία Java της εφαρμογής που θέλουμε να θολώσουμε. Ο επεξεργασμένος κώδικας Java που προέκυψε από το εργαλείο αντικαθιστά τον αρχικό κώδικα και το αρχείο αποθηκεύεται.

4.6 Έλεγχος ποιότητας

Η εφαρμογή περιέχει δοκιμές με μεμονωμένα αρχεία κώδικα και μικρές εφαρμογές για να εξασφαλιστεί η ορθή λειτουργία της. Εισάγουμε ένα αρχικό αρχείο κώδικα και ένα χειροκίνητα θολωμένο αρχείο κώδικα, εκτελούμε την εφαρμογή και συγκρίνουμε τα αποτελέσματα της εφαρμογής με το χειροκίνητα θολωμένο αρχείο που δημιουργήσαμε. Η δοκιμή θεωρείται επιτυχής όταν τα 2 αυτά αρχεία είναι ίδια.

Οι διαφορετικές λειτουργίες της εφαρμογής δοκιμάζονται με ξεχωριστές δοκιμές. Οι υλοποίηση των δοκιμών κάνει χρήση του JUnit framework. Συγκεκριμένα, έχουν υλοποιηθεί δοκιμές για :

- τη θόλωση ονομάτων μεταβλητών κλάσεων, μεταβλητών μέσα σε λογικές εκφράσεις, τοπικών μεταβλητών συναρτήσεων και μεταβλητών static
- τη θόλωση ονομάτων διεπαφών, κλάσεων και κλάσεων που κάνουν χρήση του μηχανισμού κληρονομικότητας
- τη θόλωση ονομάτων αρχείων Java του προγράμματος

Λόγω της φύσης της εφαρμογής και του framework που χρησιμοποιήθηκε, ακολουθήθηκε η διαδικασία Black-box testing κατά την οποία ελέγχεται η λειτουργικότητα της εφαρμογής χωρίς να περιεργάζεται η εσωτερική της δομή.

Διάγραμμα 4.5: Black-box testing

Κεφάλαιο 5

Εγχειρίδιο και παραδείγματα χρήσης

5.1 Λεπτομέρειες υποστήριξης

Η εφαρμογή βρίσκεται στην έκδοση 0.3, λειτουργεί σε windows και υποστηρίζει Java version 8. Συγκεκριμένα υποστηρίζεται η θόλωση των παρακάτω :

- Πίνακες
- Δομές If/Switch
- Δομές Try/Catch/Finally (μέχρι σύνταξη Java 6)
- Δομές For/While/Dowhile
- Παραμετροποιημένους τύπους (List<YourClass> σε List<a>)
- Διεπαφές (1 επίπεδο ιεραρχίας)
- Διεπαφές Mixin (Comparable<YourClass> σε Comparable<a>)
- Ονόματα κλάσεων και constructors
- Μεταβλητές κλάσης
- Ονόματα μεθόδων
- Παράμετροι μεθόδων
- Τοπικές μεταβλητές μεθόδων

- Imports
- Ονόματα αρχείων
- Κληρονομικότητα (1 επίπεδο ιεραρχίας)

5.2 Οδηγίες εκτέλεσης

Για την εκτέλεση της εφαρμογής ο χρήστης πρέπει να έχει εγκατεστημένη Java έκδοση 8 η μεγαλύτερη. Η εφαρμογή δέχεται 2 απόλυτα μονοπάτια μέσω του command line. Η πρώτη παράμετρος είναι το απόλυτο μονοπάτι του κώδικα προς θόλωση και το δεύτερο απόλυτο μονοπάτι είναι αυτό στο οποίο θα δημιουργηθεί ένας φάκελος με όνομα ”JCuttleFishBackup” στον οποίο θα αντιγραφεί ο κώδικας πριν επεξεργαστεί. Το εκτελέσιμο μπορεί να βρεθεί στον παρακάτω σύνδεσμο : https://github.com/alextsil/JCuttleFish/releases/tag/V.0.3

5.3 Παραδείγματα χρήσης

Παρακάτω παρουσιάζονται μερικά παραδείγματα χρήσης του εργαλείου θόλωσης. Για κάθε παράδειγμα βλέπουμε την είσοδο και την έξοδο που προκύπτει.

Παραδείγματα :

1. Θόλωση μιας απλής κλάσης Student
2. Θόλωση μιας απλής κλάσης Ticket

```
public class Student {

    private String rollNumber;
    private String name;
    private String standard;
    private int totalMarks;

    public Student() {
    }

    public Student(String rollNumber, String name, String standard,
            int totalMarks) {
        this.rollNumber = rollNumber;
        this.name = name;
        this.standard = standard;
        this.totalMarks = totalMarks;
    }

    public String getRollNumber() {
        return rollNumber;
    }

    public void setRollNumber(String rollNumber) {
        this.rollNumber = rollNumber;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getStandard() {
        return standard;
    }

    public void setStandard(String standard) {
        this.standard = standard;
    }

    public int getTotalMarks() {
        return totalMarks;
    }

    public void setTotalMarks(int totalMarks) {
        this.totalMarks = totalMarks;
    }

}
```

Διάγραμμα 5.1: Αρχική κλάση Student - Παράδειγμα 1

```
public class a {

    private String a;
    private String b;
    private String c;
    private int d;


    public a(){
    }

    public a(String aa, String bb, String cc, int dd){
        this.a = aa;
        this.b = bb;
        this.c = cc;
        this.d = dd;
    }

    public String a() {
        return a;
    }

    public void b(String aa) {
        this.a = aa;
    }

    public String c() {
        return b;
    }

    public void d(String aa) {
        this.b = aa;
    }

    public String e() {
        return c;
    }

    public void f(String aa) {
        this.c = aa;
    }

    public int g() {
        return d;
    }

    public void h(int aa) {
        this.d = aa;
    }

}
```

Διάγραμμα 5.2: Θολωμένη κλάση Student - Παράδειγμα 1

```
public class Ticket {
    private long id;
    private String message;
    private double latitude;
    private double longitude;
    private Date dateTime = new Date();
    private String imageName;

    public Ticket (){}
    public String getImageName () {
        return imageName;
    }

    public void setImageName ( String imageName ) {
        this.imageName = imageName;
    }

    public long getId () {
        return id;
    }

    public String getMessage () {
        return message;
    }

    public void setMessage ( String message ) {
        this.message = message;
    }

    public double getLatitude () {
        return latitude;
    }

    public void setLatitude ( double latitude ) {
        this.latitude = latitude;
    }

    public double getLongitude () {
        return longitude;
    }

    public void setLongitude ( double longitude ) {
        this.longitude = longitude;
    }

    public Date getDateTime () {
        return dateTime;
    }

    public void setDateTime ( Date dateTime ) {
        this.dateTime = dateTime;
    }
}
```

Διάγραμμα 5.3: Αρχική κλάση Ticket - Παράδειγμα 2

```
public class a {
    private long a;
    private String b;
    private double c;
    private double d;
    private Date e = new Date();
    private String f;

    public a() {}
    public String a() {
        return f;
    }

    public void b(String aa) {
        this.f = aa;
    }

    public long c() {
        return a;
    }

    public String d() {
        return b;
    }

    public void e(String aa) {
        this.b = aa;
    }

    public double f() {
        return c;
    }

    public void g(double aa) {
        this.c = aa;
    }

    public double h() {
        return d;
    }

    public void i(double aa) {
        this.d = aa;
    }

    public Date j() {
        return e;
    }

    public void k(Date aa) {
        this.e = aa;
    }
}
```

Διάγραμμα 5.4: Θολωμένη κλάση Ticket - Παράδειγμα 2

Κεφάλαιο 6

Συμπεράσματα

6.1 Θόλωση - Μετασχηματισμοί

Οι τεχνικές θόλωσης θα πρέπει να χρησιμοποιούνται πάντα σε συνδυασμό ώστε να είναι αποτελεσματικές.

Το κόστος των μετασχηματισμών συνδέεται άμεσα με την αποτελεσματικότητά τους. Όσο πιο αποτελεσματικός είναι ένας μετασχηματισμός, τόσο περισσότερο κόστος έχει. Μια εξαίρεση είναι η μέθοδος Αλλαγή ονομάτων μεταβλητών και υπογραφών συναρτήσεων, η οποία θολώνει την εφαρμογή και ταυτόχρονα μικραίνει το μέγεθός της.

Σύμφωνα με τα παραπάνω, θα πρέπει να δίνονται επιλογές στον χρήση του εργαλείου θόλωσης για το αν προτιμά μεγάλο βαθμό θόλωσης σε αντάλλαγμα με μειωμένη ταχύτητα εκτέλεσης, ή για το αν η ταχύτητα εκτέλεσης έχει μεγαλύτερη προτεραιότητα από την αποτελεσματικότητα της θόλωσης.

Υπάρχει ακόμη μια κατηγορία μετασχηματισμών που έχουν σκοπό να αποτρέψουν την σωστή λειτουργία των decompilers. Για παράδειγμα, αν επέμβουμε στο bytecode και εισάγουμε εντολές μετά το return μιας συνάρτησης, το virtual machine θα εκτελέσει κανονικά τον κώδικα, αλλά αν γίνει decompile, θα πρέπει αυτές οι εντολές να αφαιρεθούν χειροκίνητα από τον επιτιθέμενο.

6.2 Χρονοδιάγραμμα

Ο χρόνος από την ανάληψη της πτυχιακής εργασίας μέχρι την ολοκλήρωση της είναι προσεγγιστικά 24μήνες. Ο χρόνος αυτός δεν αντικατοπτρίζει τον πραγματικό χρόνο που δαπανήθηκε για την υλοποίηση της πτυχιακής εργασίας, καθώς η ανάπτυξη σταμάτησε λόγω υποχρεώσεων και πρακτικής άσκησης.

Ο πραγματικός χρόνος που δαπανήθηκε είναι 5 μήνες. Το χρονοδιάγραμμα που προκύπτει είναι το εξής :

- 1 μήνας για τη μελέτη της βιβλιογραφίας, των τεχνικών θόλωσης και των εργαλείων θόλωσης που είναι διαθέσιμα στο Διαδίκτυο.
- 3 μήνες για τη σχεδίαση του εργαλείου και τη συγγραφή του κώδικα.
- 1 μήνας για τη συγγραφή του παρόντος βιβλίου.

6.3 Δυσκολίες

Κατά την ανάπτυξη του εργαλείου θόλωσης παρουσιάστηκαν δυσκολίες τεχνικής φύσεως. Συγκεκριμένα, η ρύθμιση του Eclipse JDT framework χρειάστηκε αρκετές δοκιμές για την επίτευξη των επιθυμητών αποτελεσμάτων, καθώς προορίζεται κυρίως για χρήση εντός του Eclipse IDE, και όχι ως αυτόνομο framework. Επίσης, η τεκμηρίωση του Eclipse JDT είναι εξαιρετικά παλαιά και ελλιπής, με αποτέλεσμα να χρειάζονται πολλές δοκιμές για την επίτευξη των επιθυμητών αποτελεσμάτων.

6.4 Προτάσεις εξέλιξης

Το αποθετήριο του κώδικα καθώς και όλες οι πληροφορίες για το εργαλείο μπορούν να βρεθούν στη διεύθυνση https://github.com/alextsil/JCuttleFish
Παρακάτω παρουσιάζονται μερικές προτάσεις εξέλιξης:

- Η αφαίρεση όλων των σχολίων από όλα τα αρχεία πηγαίου κώδικα Java.
- Inlining/Outlining των μεθόδων, όπως περιγράφηκε στο 2.1.3.3.

- Σύστημα κρυπτογράφησης αλφαριθμητικών. Συγκεκριμένα, όλα τα αλφαριθμητικά μιας εφαρμογής μπορούν να κρυπτογραφηθούν για να αποφευχθούν περιπτώσεις στις οποίες το μήνυμα που προβάλλεται στο χρήστη είναι αρκετά περιγραφικό έτσι ώστε να ”προδώσει” τη λειτουργία του κώδικα σε εκείνο το σημείο.
- Η δυνατότητα επιλογής των μεθόδων θόλωσης που θα εφαρμοστούν.
- Υποστήριξη όλων των στοιχείων της γλώσσας Java, όπως δομές Functional και FunctionalInterfaces.

Γλωσσάρι

JDT Java Development Tools

AST Abstract Syntax Tree

API Application Programming Interface

IDE Integrated Development Environment

Βιβλιογραφία

Abstract Syntax Tree (2014). URL: http://www.eclipse.org/articles/Article-JavaCodeManipulation_AST/index.html (επίσκεψη 20/04/2014).

Chan, Jien-Tsai και Wuu Yang (2004). *Advanced obfuscation techniques for Java bytecode*.

Eclipse Java Development Tools. URL: https://github.com/eclipse/eclipse.jdt.core.

"GangOfFour" (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Gupta, Sonali (2014a). *Code obfuscation - part 2: Hiding control flows*. URL: http://palizine.plynt.com/issues/2005Oct/hiding-control-flows (επίσκεψη 14/05/2014).

— (2014b). *Code obfuscation - part 2: Obfuscating data structures*. URL: http://palizine.plynt.com/issues/2005Sep/code-obfuscation-continued (επίσκεψη 14/05/2014).

Hamilton, James και Sebastian Danicic (2009). *An evaluation of current java bytecode decompilers*.

Kalinovsky, Alex (2004). *Covert Java: Techniques for Decompiling, Patching, and Reverse Engineering*. Sams.

Lafortune, Eric (2002-2015). *ProGuard*. URL: http://proguard.sourceforge.net (επίσκεψη 21/03/2016).

Leskov, Dmitry (2014). *Protect Your Java Code. Through Obfuscators And Beyond*. URL: http://www.excelsior-usa.com/articles/java-obfuscators.html (επίσκεψη 20/04/2014).

Low, Douglas (1998). «Java Control Flow Obfuscation». Διδακτορική διατρ. University of Auckland, σσ. 21–23.

Macbride Jeffrey, .. (2005). *A comparative study of Java obfuscators*.

PreEmptiveSolutions (2016). *DashO*. URL: https://www.preemptive.com/products/dasho (επίσκεψη 21/03/2016).

Witkowska, Joanna (2006). *The quality of Obfuscation and Obfuscation Techniques*.

Η εργασία αυτή στοιχειοθετήθηκε με το πρόγραμμα XeLaTeX. Για τη στοιχειοθέτηση της βιβλιογραφίας χρησιμοποιήθηκε το πρόγραμμα `biber` και `biblatex`. Οι γραμματοσειρές που χρησιμοποιήθηκαν είναι οι Times New Roman και `Courier New`.