Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Πρακτική Στατική Ανάλυση για Προγράμματα Python

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΒΙΤΑΛΙΟΣ ΣΑΛΗΣ

Επιβλέπων : Νικόλαος Σ. Παπασπύρου
Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούνιος 2021

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Πρακτική Στατική Ανάλυση για Προγράμματα Python

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΒΙΤΑΛΙΟΣ ΣΑΛΗΣ

Επιβλέπων : Νικόλαος Σ. Παπασπύρου
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 28η Ιουνίου 2021.

Νικόλαος Σ. Παπασπύρου Καθηγητής Ε.Μ.Π.	Κωνσταντίνος Σαγώνας Αν. Καθηγητής Ε.Μ.Π.	Γεώργιος Γκούμας Αν. Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούνιος 2021

Βιτάλιος Σαλής

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Η στατική ανάλυση προγραμμάτων ενέχει προκλήσεις, ειδικά στο πλαίσιο γλωσσών υψηλού επιπέδου, όπως η Python, η οποία είναι modular και ενσωματώνει δυναμικά χαρακτηριστικά και συναρτήσεις υψηλού επιπέδου. Προτείνουμε ένα επεκτάσιμο πλαίσιο για την στατική ανάλυση Python προγραμμάτων και έχουμε υλοποιήσει ένα αντίστοιχο πρωτότυπο το οποίο ονομάζουμε *PyCG*. Το πρωτότυπο μας λειτουργεί μέσω του υπολογισμού όλων των σχέσεων ανάθεσης μεταξύ των αναγνωριστικών συναρτήσεων, μεταβλητών, κλάσεων, και modules μέσω μιας διασυναρτησιακής ανάλυσης χωρίς ευαισθησία περιβάλλοντος. Με βάση αυτές τις σχέσεις αναθέσεων, υλοποιούμε δύο επεκτάσεις της ανάλυσης μας: (1) παραγωγή γράφων κλήσεων συναρτήσεων και (2) ανίχνευση σφαλμάτων κλειδιού.

Οι γράφοι κλήσεων συναρτήσεων είναι κατευθυνόμενοι γράφοι που επισημαίνουν σχέσεις κλήσεων μεταξύ υπορουτινών. Παίζουν σημαντικό ρόλο σε διάφορα πλαίσια, όπως την δημιουργία προφίλ κώδικα και την ανάλυση διάδοσης ευπαθειών. Έχουμε επεκτείνει την ανάλυση που χρησιμοποιεί το *PyCG* ώστε να παράγουμε γράφους κλήσεων συναρτήσεων και έχουμε αξιολογήσει την μέθοδο μας σε δύο σημεία αναφοράς: μια βιβλιοθήκη από Python προγράμματα μικρής εμβέλειας και μια βιβλιοθήκη από πραγματικά Python πακέτα μεγάλης εμβέλειας. Τα αποτελέσματα της αξιολόγησης δείχνουν πως το *PyCG* ξεπερνάει τις υπάρχουσες κορυφαίες μεθόδους για την Python σε ακρίβεια ($\sim 99.2\%$) καθώς και σε ανάκληση ($\sim 69.9\%$).

Οι μη έγκυρες προσβάσεις σε πίνακες κατακερματισμού συμβαίνουν όταν ένας πίνακας διασχίζεται με ένα στοιχείο το οποίο δεν αντιστοιχεί σε κάποιο από τα κλειδιά του. Σε αυτή τη περίπτωση, η Python παράγει ένα *σφάλμα κλειδιού* και τερματίζει. Η ανίχνευση πιθανών σφαλμάτων κλειδιού μπορεί να είναι κρίσιμης σημασίας, ειδικά για εφαρμογές στην παραγωγή. Για να αντιμετωπίσουμε αυτά τα ρίσκα, έχουμε επεκτείνει την ανάλυση μας ώστε να μπορεί να ανιχνεύει μη έγκυρες προσβάσεις σε πίνακες κατακερματισμού, και την αξιολογούμε σε μια βιβλιοθήκη από μικρά Python προγράμματα και σε ένα σύνολο από υποβολές φοιτητών για ένα μεταπτυχιακό μάθημα που σχετίζεται με την ανάλυση δεδομένων. Τα αποτελέσματα μας υποδεικνύουν πως το *PyCG* μπορεί να ανιχνεύσει σφάλματα κλειδιού σε πολλές περιπτώσεις στην Python, ενώ μπορεί να βοηθήσει προγραμματιστές να βρουν πιθανά σφάλματα κλειδιού κατά την διάρκεια της συγγραφής κώδικα.

Λέξεις κλειδιά

Γλώσσες προγραμματισμού, Python, Γράφοι κλήσεων συναρτήσεων, Ανίχνευση σφαλμάτων κλειδιού

Abstract

Statically analyzing programs can be a challenging task, especially in the context of high-level languages such as Python, which is modular and incorporates dynamic features and higher-order functions. We propose an extendable static framework for the analysis of Python programs and have implemented a corresponding prototype which we call *PyCG*. Our prototype works by computing all assignment relations among the program identifiers of functions, variables, classes, and modules through an inter-procedural and context-insensitive analysis. Based on these assignment relations, we implement two extensions of our analysis: (1) call graph generation and (2) key error identification.

Call graphs are directed graphs that highlight calling relations between subroutines. They play an important role in different contexts, such as profiling and vulnerability propagation analysis. We have extended the analysis employed by *PyCG* to generate call graphs and have evaluated our approach using two benchmarks: a micro-benchmark suite that contains minimal Python programs and a set of macro-benchmarks that includes real-world Python packages. The evaluation results show that *PyCG* outperforms the state-of-the-art for Python in both precision ($\sim 99.2\%$) and recall ($\sim 69.9\%$).

Invalid dictionary accesses happen when a dictionary is traversed with an accessor that does not correspond to one of its keys. In that case, Python generates a *key error* and exits. Identifying potential key errors can be of critical importance, especially for production applications. To counteract these risks, we have extended our analysis to identify invalid dictionary accesses, and have evaluated our approach on a suite of minimal Python programs and a set of student submissions for a postgraduate course related to data analysis. Our results indicate that *PyCG* can identify key errors in many Python settings, while it can aid practitioners identify potential key errors during the development process.

Key words

Programming languages, Python, Call graphs, Detection of key errors.

Ευχαριστίες

Η παρούσα εργασία δεν θα υπήρχε αν δεν υπήρχαν οι άνθρωποι που με υπομονή με στήριξαν στο μονοπάτι για να μάθω την τέχνη μου – οι δάσκαλοι μου. Οφείλω την αγάπη μου για τη συγγραφή κώδικα και τη σχεδίαση κομψών συστημάτων στον Διονύση Ζήνδρο. Απο το πρώτο μου ”Hello World” πρόγραμμα σε C εως ιστοσελίδες δεκάδων χιλιάδων γραμμών, ο Διονύσης με δίδαξε το τρόπο σκέψης ενός προγραμματιστή, να δίνω έμφαση στη ποιότητα, και πως να διδάσκομαι μόνος μου. Ο Δημήτρης Μητρόπουλος με εμπιστεύτηκε να είμαι δημιουργικός, μου έδωσε ευθύνες, και με έμαθε πως να παράγω ερευνητικά αποτελέσματα και να τα αποτυπώνω με επιστημονικά τεκμηριωμένους όρους. Παρατηρώντας τις πράξεις του και με τις συμβουλές του ωρίμασα και έμαθα να έχω στόχους, πλάνο ολοκλήρωσης τους, καθώς και την εργατικότητα που απαιτείται ώστε να τους καταφέρω. Ο Γιώργος Τσουκαλάς μου έδωσε την ώθηση προς το δρόμο της αυτοκατανόησης και μαζί με τον Πάνο Λουρίδα με εμπιστεύτηκαν με μια θέση στην ομάδα των προγραμματιστών του GRNET παρά το νεαρό της ηλικίας μου. Ο Διομήδης Σπινέλλης με δέχτηκε στο εργαστήριο του και με καθοδήγησε ώστε να φτιάξω το πρώτο μου εργαλείο με γνώμονα την έρευνα.

Θα ήθελα να ευχαριστήσω τους ανθρώπους που συνεισέφεραν ώστε απο τον κώδικα και τα πειράματα να δημιουργηθεί μια ολοκληρωμένη ερευνητική εργασία. Ο επιβλέποντας μου, Νικόλαος Παπασπύρου, με καθοδήγησε κατά τη διάρκεια της εκπόνησης και μου παρείχε πολύτιμες συμβουλές. Ο Δημήτρης Μητρόπουλος μου έδωσε την ιδέα για την δημιουργία μιας πρακτικής στατικής ανάλυσης για την Python, και μαζί με τον Θοδωρή Σωτηρόπουλο, τον Πάνο Λουρίδα και τον Διομήδη Σπινέλλη, συνέβαλαν σημαντικά στην αφήγηση, τον τόνο, και την ορθή αποτύπωση της ανάλυσης, πράγμα που οδήγησε στη δημοσιεύση μέρους αυτής της εργασίας στο ICSE. Ο Χρήστος Λαμπράκος έδωσε σημαντική βοήθεια κατα τη διάρκεια της μετάφρασης στο Ελληνικό κείμενο.

Δεν μπορώ να μην αναφερθώ σε όλους μου τους φίλους που με την παρέα τους με στήριξαν στη σταδιοδρομία μου και υπήρξαν πηγή έμπνευσης και γνώσης για μένα. Ο Νίκος άκουγε πάντα υπομονετικά τους προβληματισμούς μου χωρίς κρίση. Ο Γιάννης με έβγαζε από το σπίτι όταν το είχα παρακάνει με τη δουλειά. Μαζί με αυτούς, ο Δημήτρης, ο Ραφαήλ, ο Άλεξ, ο Παντελής, και ο Αλέξης είναι καλοί μου φίλοι για το μεγαλύτερο κομμάτι της ζωής μου. Ο Χρήστος υπήρξε πηγή ατελείωτων φιλοσοφικών συζητήσεων στις οποίες μαζί αποκτήσαμε στόχους για τη ζωή. Ο Φίλιππος και ο Διονύσης ήταν μαζί μου σε κάθε εξεταστική-μάχη και μέσα από τις δυσκολίες μας χτίσαμε μια όμορφη φιλία.

Θα ήθελα να κλείσω με ένα ευχαριστώ προς την οικογένεια μου. Οι αδελφές μου, Γιάννα και Ιφιγένεια, υπήρξαν το στήριγμα μου και ήταν δίπλα μου σε κάθε επιλογή. Οι γονείς μου, Ηλίας και Ελένη, δούλεψαν σκληρά ώστε εγώ και οι αδελφές μου να έχουμε τις ευκαιρίες που δεν είχαν εκείνοι και μου έδωσαν με αγάπη όλα τα απαραίτητα εφόδια ώστε να ακολουθήσω το μονοπάτι μου. Η παρούσα εργασία είναι αφιερωμένη σε αυτούς.

Βιτάλιος Σαλής,

Αθήνα, 28η Ιουνίου 2021

Η εργασία αυτή είναι επίσης διαθέσιμη ως Τεχνική Αναφορά CSD-SW-TR-4-21, Εθνικό Μετσόβιο Πολυτεχνείο, Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών, Εργαστήριο Τεχνολογίας Λογισμικού, Ιούνιος 2021.

URL: `http://www.softlab.ntua.gr/techrep/`
FTP: `ftp://ftp.softlab.ntua.gr/pub/techrep/`

Περιεχόμενα

Κατάλογος πινάκων

Πίνακες στο αγγλικό κείμενο

Κατάλογος σχημάτων

Σχήματα στο αγγλικό κείμενο

Κατάλογος αλγορίθμων

Αλγόριθμοι στο αγγλικό κείμενο

Κεφάλαιο 1

Εισαγωγή

Η Python είναι μία γλώσσα προγραμματισμού υψηλού επιπέδου με δυναμικό σύστημα τύπων. Θεωρείται μία από τις πιο δημοφιλείς γλώσσες [1, 2] και υποστηρίζει πολλά διαφορετικά σχήματα προγραμματισμού, συμπεριλαμβανομένων εκείνων του αντικειμενοστραφή και του συναρτησιακού προγραμματισμού. Χρησιμοποιείται για μια πληθώρα εφαρμογών, από απλά scripts έως υπηρεσίες συστημάτων παραγωγής μεγάλης κλίμακας. Η δημοφιλία της έχει οδηγήσει στην εμφάνιση εργαλείων στατικής ανάλυσης που στοχεύουν να βοηθήσουν τόσο προγραμματιστές όσο και ερευνητές να αναλύσουν προγράμματα γραμμένα σε αυτήν. Αυτά τα εργαλεία συνήθως χρησιμοποιούνται κατά τη διάρκεια της ανάπτυξης κώδικα ή αφότου το λογισμικό έχει δημοσιευτεί.

Η στατική ανάλυση προγραμμάτων που έχουν γραφτεί σε υψηλού επιπέδου, δυναμικές γλώσσες μπορεί να καταλήξει σύνθετο εγχείρημα. Συγκεκριμένα, προκειμένου να εφαρμόσει κανείς στατική ανάλυση σε προγράμματα Python και Javascript, πρέπει να αντιμετωπίσει διάφορες προκλήσεις όπως συναρτήσεις υψηλότερης τάξης, δυναμικά στοιχεία καθώς και στοιχεία μεταπρογραμματισμού (π.χ. `eval`), και την ύπαρξη modules. Η επίλυση αυτών των προκλήσεων μπορεί να παίξει πολύ σημαντικό ρόλο στην βελτίωση αναλύσεων σχετικών με την επίδραση εξαρτήσεων [3, 4, 5], ειδικά στο πλαίσιων διαχειριστών πακέτων όπως το *npm* [6] και το *pip* [7].

Πολλές ερευνητικές εργασίες προτείνουν μεθόδους στατικής ανάλυσης για δυναμικές γλώσσες, με πρωταρχικό στόχο την πληρότητα, δηλαδή το να είναι τα συμπεράσματα που παράγονται από το σύστημα πράγματι ορθά [8, 9, 10]. Ωστόσο, στην περίπτωση των δυναμικών γλωσσών, η πληρότητα έρχεται με ένα τίμημα στην απόδοση. Συνεπώς, τέτοιες προσεγγίσεις εφαρμόζονται σπάνια στην πράξη λόγω ενδοιασμών κλιμακωσιμότητας [11]. Για αντιμετωπίσουν τέτοια ζητήματα, οι ερευνητές έχουν στραφεί σε *πρακτικές* μεθόδους που εστιάζουν σε μη-πλήρη στατική ανάλυση ώστε να επιτευχθεί καλύτερη επίδοση [12, 13]. Αυτή η ενίσχυση της επίδοσης είναι ο καταλυτικός παράγοντας για την υιοθέτηση τέτοιων μεθόδων σε εφαρμογές που αλληλεπιδρούν με σύνθετες βιβλιοθήκες [12], ή με Ολοκληρωμένα Περιβάλλοντα Ανάπτυξης (IDEs) [13]. Υπάρχουσες εργασίες εστιάζουν κυρίως σε κώδικα JavaScript και επιχειρούν να επιλύσουν προκλήσεις σχετικές με events και την ασύγχρονη φύση της συγκεκριμένης γλώσσας [14, 15].

Προτείνουμε μια πρακτική μέθοδο για τη στατική ανάλυση Python προγραμμάτων και υλοποιούμε ένα κατ' αντιστοιχία πρωτότυπο που ονομάζουμε *PyCG*. Η μέθοδός μας υπολογίζει τον *γράφο αναθέσεων*, μια δομή που απεικονίζει τις σχέσεις ανάθεσης μεταξύ των αναγνωριστικών του προγράμματος. Σχεδιάζουμε μια context-insensitive διασυναρτησιακή ανάλυση που λειτουργεί με βάση μια απλή ενδιάμεση αναπαράσταση στοχευμένη στην Python. Σε αντίθεση με υπάρχοντες στατικούς αναλυτές, η ανάλυσή μας μπορεί να χειριστεί σύνθετα χαρακτηριστικά της Python, όπως συναρτήσεις υψηλού επιπέδου, function closures, πολλαπλή κληρονομικότητα, και modules.

Η ανάλυσή μας ακολουθεί μια συντηρητική προσέγγιση και δεν χειρίζεται μη-συναρτησιακές τιμές. Αυτό σημαίνει ότι η ανάλυσή μας αγνοεί τους ελέγχους των βρόγχων και των τμημάτων κώδικα υπό συνθήκη. Επιπλέον, διακρίνουμε προσβάσεις σε attributes (π.χ. $e.x$) με βάση το namespace όπου το εκάστοτε attribute (x) έχει οριστεί. Αυτή η προσέγγιση ενισχύει την ακρίβεια, ειδικά όταν αντιμετωπίζουμε γνωρίσματα όπως modules, κληρονομικότητα και μοτίβα προγραμματισμού όπως το duck typing [16]. Προηγούμενες εργασίες ακολουθούν μια *field-based* μεθοδολογία που συσχετίζει attributes του ίδιου ονόματος με μια μοναδική καθολική τοποθεσία χωρίς να λαμβάνει υπόψη το namespace [13], πράγμα που οδηγεί σε ψευδώς θετικά συμπεράσματα. Οι επιλογές σχεδίασης μας

επιτρέπουν στη μέθοδό μας να επιτύχει υψηλά ποσοστά ακρίβειας, ενώ παραμένει αποδοτική και εφαρμόσιμη σε προγράμματα Python μεγάλης κλίμακας.

Η ανάλυσή μας έχει σχεδιαστεί έτσι ώστε να είναι κλιμακώσιμη και εύκολα επεκτάσιμη. Συγκεκριμένα, οι εσωτερικοί τομείς της ανάλυσης μπορούν να χρησιμεύσουν ως βάση για μια ευρεία ποικιλία εφαρμογών που στοχεύουν στην εκτέλεση μιας πρακτικής στατικής ανάλυσης. Με αυτό τον στόχο, έχουμε επεκτείνει το *PyCG* ώστε να υποστηρίζει δύο πρακτικές εφαρμογές: (1) την παραγωγή γράφων κλήσεων συναρτήσεων και την αναγνώριση εσφαλμένων προσβάσεων σε πίνακες κατακερματισμού (σφάλματα κλειδιού).

Ένας γράφος κλήσεων συναρτήσεων απεικονίζει τις σχέσης κλήσης μεταξύ των υπορουτινών ενός προγράμματος. Τέτοιοι γράφοι μπορούν να αξιοποιηθούν σε μια ποικιλία εργασιών, όπως το profiling [17], την ανίχνευση της διάδοσης μιας ευπάθειας [18], και την ανακατασκευή προγραμμάτων μέσω εργαλείων [19]. Η ανάλυσή μας χτίζει τον γράφο κλήσεων συναρτήσεων του αρχικού προγράμματος Python κάνοντας χρήση του γράφου αναθέσεων. Συγκεκριμένα, αξιοποιούμε τον γράφο για να ανιχνεύσουμε όλες τις συναρτήσεις που δύνανται να έχουν ανατεθεί στις καλούμενες μεταβλητές. Αυτό το μοτίβο είναι ιδιαίτερα συχνό στον προγραμματισμό υψηλού επιπέδου.

Τα σφάλματα κλειδιού προκύπτουν όταν επιχειρείται η πρόσβαση σε ένα ανύπαρκτο κλειδί ενός dictionary. Όταν αυτό συμβαίνει, η Python προκαλεί ένα σφάλμα και τερματίζει την εκτέλεση. Σφάλματα κλειδιού που παραμένουν κρυφά μπορούν να έχουν καταστροφικές επιπτώσεις για εταιρείες που χρησιμοποιούν λογισμικό γραμμένο σε Python. Αυτό δημιουργεί την ανάγκη για μεθόδους που μπορούν να ανιχνεύσουν πιθανά σφάλματα κλειδιού κατά τη διάρκεια της ανάπτυξης κώδικα. Υπάρχει ήδη μια πληθώρα εργαλείων που στοχεύουν να βοηθήσουν τους προγραμματιστές να εντοπίσουν πιθανά σφάλματα κατά την ανάπτυξη κώδικα [20, 21, 22, 23, 24]. Τα εργαλεία αυτά λειτουργούν απευθείας από τη γραμμή εντολών, και μπορούν να ενσωματωθούν σε IDEs και συστήματα continuous delivery. Ωστόσο, καμία από αυτές τις μεθόδους δεν μπορεί να ανιχνεύσει μη έγκυρες προσβάσεις σε dictionaries. Για να συμπληρώσουμε αυτό το κενό, χρησιμοποιούμε τον γράφο αναθέσεων και μία δομή που αποθηκεύει τα literals που έχουν ανατεθεί σε μια μεταβλητή. Στη συνέχεια ελέγχουμε αν το literal που χρησιμοποιείται για την πρόσβαση στο dictionary αντιστοιχεί σε ένα από τα κλειδιά του. Στην περίπτωση που αυτό δεν ισχύει, η ανάλυσή μας παράγει ένα σφάλμα κλειδιού.

Ελέγχουμε την αποτελεσματικότητα της μεθόδους μας μέσω συλλογών benchmarks μικρής και μεγάλης κλίμακας. Στην περίπτωση των γράφων κλήσεων συναρτήσεων, συγκρίνουμε τη μέθοδό μας με δύο άλλες διαθέσιμες γεννήτριες τέτοιων γράφων για τη γλώσσα Python, του *Pyan* και του *Depends*. Τα αποτελέσματα δείχνουν πως η μέθοδός μας επιτυγχάνει υψηλά επίπεδα precision ($\sim$ 99.2%) και επαρκές recall ($\sim$ 69.9%) κατά μέσο όρο, ενώ οι άλλοι αναλυτές αποδίδουν χειρότερα και για τις δύο μετρικές. Αναφορικά με τα σφάλματα κλειδιού, δείχνουμε ότι η μέθοδός μας μπορεί να τα ανιχνεύσει αποτελεσματικά σε πολλαπλά σενάρια που προκύπτουν από τη λειτουργικότητα της Python. Επιπροσθέτως, χρησιμοποιούμε τον μηχανισμό εντοπισμού σφαλμάτων κλειδιού σε υπάρχοντα προγράμματα Python. Τα αποτελέσματά μας δείχνουν ότι η μέθοδός μας μπορεί να εντοπίσει με αποτελεσματικότητα τέτοια σφάλματα κατά τη διάρκεια ανάπτυξης λογισμικού, ενώ παράλληλα διατηρεί χαμηλό αριθμό ψευδώς θετικών συμπερασμάτων.

1.1 Δομή της εργασίας

Το υπόλοιπο της εργασίας μας έχει την ακόλουθη δομή. Στο Κεφάλαιο 2 συνοψίζουμε τις προκλήσεις που παρουσιάζονται στη στατική ανάλυση Python κώδικα και τους περιορισμούς στους υπάρχοντες γεννήτορες γράφων κλήσεων συναρτήσεων. Το Κεφάλαιο 3 παρουσιάζει τη μεθοδολογία της ανάλυσής μας. Συγκεκριμένα, εισάγουμε μια ενδιάμεση αναπαράσταση στοχευμένη σε προγράμματα Python και τους τομείς της ανάλυσής μας. Στο Κεφάλαιο 4 αναφέρουμε τις σχεδιαστικές αποφάσεις της υλοποίησής μας και τη διεπαφή του εργαλείου μας. Η αξιολόγηση της μεθόδου μας παρουσιάζεται στο Κεφάλειο 5, το οποίο περιέχει λεπτομέρειες σχετικά με τα πειράματα μας, τα benchmarks που χρησιμοποιήσουμε, και τα αντίστοιχα αποτελέσματα. Στο Κεφάλαιο 6 παραθέτουμε τη σχετική βιβλιογραφία. Η εργασία μας ολοκληρώνεται με μια σύνοψη όλων των παραπάνω στο Κεφάλαιο 7.

Κεφάλαιο 2

Υπόβαθρο

2.1 Προκλήσεις στην Python

Η ανάλυση κώδικα Python ενέχει τις ακόλουθες προκλήσεις.

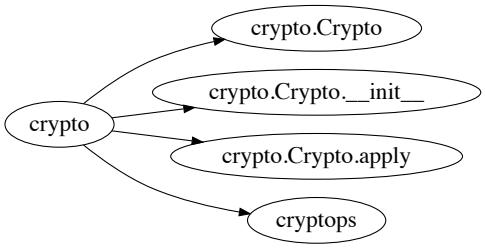
- *Συναρτήσεις Υψηλότερης Τάξης*: Σε μια γλώσσα υψηλού επιπέδου όπως η Python, μια συνάρτηση μπορεί να ανατεθεί σε μια μεταβλητή, να περαστεί ως όρισμα σε μια άλλη συνάρτηση, ή ακόμα και να λειτουργήσει ως τιμή επιστροφής.
- *Εμφωλευμένοι Ορισμοί*: Στην Python, οι ορισμοί συναρτήσεων και κλάσεων μπορούν να είναι εμφωλευμένοι σε άλλους ορισμούς. Αυτό σημαίνει ότι οι συναρτήσεις και οι κλάσεις μπορούν να οριστούν και να καλεστούν στο πλαίσιο άλλων συναρτήσεων ή κλάσεων.
- *Κλάσεις*: Η Python είναι μια αντικειμενοστραφής γλώσσα και επιτρέπει την λειτουργία σύνθετων σχημάτων κληρονομικότητας και σύνθεσης. Η αναγνώριση μεθόδων που έχουν κληρονομηθεί απο βασικές κλάσεις απαιτεί τον υπολογισμό της Σειράς Επίλυσης Μεθόδων (Method Resolution Order, MRO) κάθε κλάσης.
- *Modules*: Σε μια τυπική περίπτωση, ένα module Python μπορεί να εισάγει πολλά άλλα modules. Επιπλέον, μια τέτοια εισαγωγή μπορεί να επιλυθεί με διαφορετικό τρόπο σε διαφορετικά συστήματα. Η διαχείριση των διαφορετικών modules που έχουν εισαχθεί σε ένα module καθώς και η επίλυση τους είναι σημαντική πρόκληση.
- *Δυναμικό Χαρακτηριστικά*: Οι μεταβλητές μπορούν να πάρουν διάφορες τιμές κατά την εκτέλεση ενός προγράμματος και αυτές οι τιμές μπορούν να έχουν διαφορετικούς τύπους. Ακόμα, μια κλάση μπορεί να μεταβληθεί δυναμικά κατά την εκτέλεση.
- *Duck Typing*: Το duck typing είναι ένα μοτίβο προγραμματισμού που παρατηρείται συχνά σε δυναμικές γλώσσες όπως η Python [16]. Μέσω αυτού του μοτίβου, η παρουσία συγκεκριμένων μεθόδων και ιδιοτήτων καθορίζει την καταλληλότητα ενός αντικειμένου, αντί για το ίδιο το αντικείμενο. Το duck typing, καθιστά δύσκολη την ανίχνευση του namespace μιας κληθείσας μεθόδου που ορίζεται από δύο (ή παραπάνω) κλάσεις σε ένα συγκεκριμένο context.

2.2 Περιορισμοί Υπαρχόντων Γεννητριών Γράφων Κλήσεων Συναρτήσεων

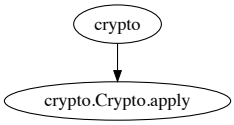
Παρά την δημοφιλία της Python [25], προκαλεί εντύπωση το ότι υπάρχουν τόσο λίγα εργαλεία με στόχο την παραγωγή γράφων κλήσεων συναρτήσεων για προγράμματα που έχουν γραφτεί στην γλώσσα. Το *Pyan* [26] διασχίζει το Αφηρημένο Συντακτικό Δέντρο (Abstract Syntax Tree, AST) του προγράμματος για να παράγει τον γράφο. Ώστοσο, αντιμετωπίζει προβλήματα στον τρόπο με τον οποία διαχειρίζεται εισαγωγές modules και την διασυνιαρτησιακή ροή των τιμών. Το *code2graph* [27, 28] έχει τους ίδιους περιορισμούς, καθώς χρησιμοποιεί γράφους που έχουν παραχθεί απο το *Pyan* για να τους οπτικοποιήσει. Το *Depends* [29] παράγει γράφους συμπαιρένοντας συντακτικές σχέσεις μεταξύ οντοτήτων του κώδικα. Παρ' όλα αυτά έχει μειονεκτήματα που σχετίζονται με συναρτήσεις υψηλότερης τάξης.

```python
import cryptops

class Crypto:
    def __init__(self, key):
        self.key = key

    def apply(self, msg, func):
        return func(self.key, msg)

crp = Crypto("secretkey")
encrypted = crp.apply("hello world", cryptops.encrypt)
decrypted = crp.apply(encrypted, cryptops.decrypt)
```

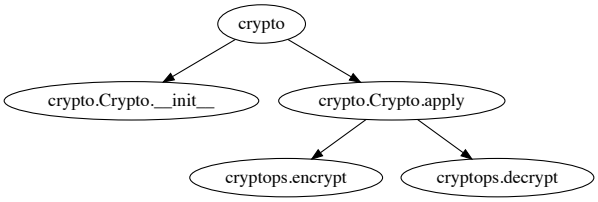
Σχήμα 2.1: Το `crypto` module. Τα υπάρχοντα εργαλεία αποτυγχάνουν στην αποτελεσματική παραγωγή του αντίστοιχου γράφου κλήσεων συναρτήσεων.



(a) Ο γράφος κλήσεων συναρτήσεων που έχει παραχθεί απο το `Pyan`.



(b) Ο γράφος κλήσεων συναρτήσεων που έχει παραχθεί απο το `Depends`.



(c) Ο ακριβής γράφος κλήσεων συναρτήσεων.

Σχήμα 2.2: Γράφοι κλήσεων συναρτήσεων για το `crypto` module.

Σε αυτή την ενότητα, μελετάμε τους περιορισμούς των δύο υπαρχόντων αναλυτών υπό όρους αποδοτικότητας και πρακτικότητας. Αυτό το κάνουμε εισάγοντας ένα Python module ονόματι `crypto` (Σχήμα 2.1), το οποίο υλοποιεί την κρυπτογράφηση και αποκρυπτογράφηση ενός μηνύματος "hello world". Αρχικά, εισάγει ένα εξωτερικό Python module με το όνομα `cryptops`, το οποίο ορίζει τις συναρτήσεις `encrypt(key, msg)` και `decrypt(key, msg)`. Κατόπιν, ορίζει την κλάση `Crypto`, που αρχικοποιείται με ένα κλειδί κρυπτογράφησης. Το στιγμιότυπο της κλάσης μπορεί ύστερα να χρησιμοποιηθεί για να κρυπτογραφήσει και να αποκρυπτογραφήσει μηνύματα καλώντας την μέθοδο `apply(self, msg, func)`, στην οποία η παράμετρος `func` είναι μια εκ των `encrypt(key, msg)`, `decrypt(key, msg)`. Το Σχήμα 2.2c δείχνει τον γράφο κλήσεων συναρτήσεων του module.

Το *Pyan* [26] παράγει τον ανακριβή γράφο που φαίνεται στο Σχήμα 2.2a. Το *Pyan* δεν καταγράφει την διασυναρτησιακή ροή των τιμών, και συνεπώς δεν μπορεί να συμπεράνει ποιες συναρτήσεις έχουν περαστεί ως ορίσματα στην `apply(self, msg, func)`, πράγμα που οδηγεί στην απουσία ακμών κλήσεων. Ακόμα, ο γράφος περιέχει ψευδή θετικά. Συγκεκριμένα, όποτε ένα αντικείμενο αρχικοποιείται, το *Pyan* προσθέτει ακμές κλήσεις και στο όνομα της κλάσης και στην μέθοδο `__init__()` της κλά-

σης. Επιπλέον, στην περίπτωση εισαγωγής modules, το *Pyan* παράγει ακμές κλήσεων με την αρχή να βρίσκεται στο namespace που εισάγει το εξωτερικό module και τον προορισμό να είναι το όνομα του τελευταίου.

Το *Depends* παράγει τον γράφο κλήσεων συναρτήσεων που φαίνεται στο Σχήμα 2.2b. Αρχικά, το *Depends* παρήγαγε έναν άδειο γράφο, καθώς δεν καταγράφει κλήσεις συναρτήσεων που προέρχονται απο το namespace του module (π.χ. `crypto.apply()`). Για να λάβουμε αποτέλεσμα, βάλαμε αυτές τις κλήσεις σε μια νέα συνάρτηση. Ο παραχθείς γράφος δεν περιέχει τις περισσότερες κλήσεις που παρουσιάζονται στο `crypto` module. Αυτό συμβαίνει επειδή το *Depends* δεν καταγράφει κλήσεις στην μέθοδο `__init__()` των κλάσεων. Επιπλέον, το *Depends* δεν καταγράφει τη διασυναρτησιακή ροή των συναρτήσεων και δεν συμπεραίνει από αυτές έχουν περαστεί ως ορίσματα στην `apply(self, msg, func)` πράγμα που οδηγεί σε απώλειες ακμών κλήσεις προς τις συναρτήσεις αυτές. Σε σύγκριση με το *Pyan*, το *Depends* ακολουθεί μια πιο συντηρητική προσέγγιση. Αυτό σημαίνει ότι συμπεριλαμβάνει μια ακμή κλήσης όποτε έχει όλη την απαραίτητη πληροφορία ώστε να προσδοκεί την πραγματοποίηση της κλήσης. Σε αντίθεση με το *Pyan*, αυτό μπορεί να οδηγήσει σε έναν γράφο που δεν περιέχει ψευδώς θετικές ακμές.

Κεφάλαιο 3

Ανάλυση

Η προσέγγισή μας χρησιμοποιεί μια context-insensitive διασυναρτησιακή ανάλυση που ενεργεί σε μια ενδιάμεση αναπαράσταση του δοθέντος Python προγράμματος. Η ανάλυση αξιοποιεί έναν επαναληπτικό αλγόριθμο σταθερού σημείου και σταδιακά χτίζει τον *γράφο αναθέσεων*, ο οποίος δείχνει τις σχέσεις ανάθεσης μεταξύ αναγνωριστικών του προγράμματος (Ενότητα 3.1). Ο γράφος αναθέσεων είναι ένα απαραίτητο στοιχείο που χρησιμοποιούμε για την επίλυση συναρτήσεων και των literals που μπορούν να ανατεθούν σε μια μεταβλητή. Αφότου η ανάλυση ολοκληρωθεί, χτίζουμε τον γράφο κλήσεων συναρτήσεων και ανιχνεύουμε σφάλματα κλειδιού εκμεταλλευόμενοι τον γράφο αναθέσεων που προέκυψε από το βήμα της ανάλυσης (Ενότητα 3.2,Ενότητα 3.3).

3.1 Η Κεντρική Ανάλυση

Η μέθοδός μας αρχικά υπολογίζει έναν γράφο αναθέσεων κάνοντας χρήση μιας διασυναρτησιακής ανάλυσης η οποία λειτουργεί πάνω σε μια ενδιάμεση αναπαράσταση στοχευμένη για Python προγράμματα.

Ένα στοιχείο κλειδί της ανάλυσής μας είναι η εξέταση των προσβάσεων σε attributes βασισμένοι στο namespace στο οποίο κάθε attribute ορίζεται. Για παράδειγμα, ας αναλογιστούμε το ακόλουθο κομμάτι κώδικα:

```
1  class A:
2    def func():
3      pass
4
5  class B:
6    def func():
7      pass
8
9  a = A()
10 b = B()
11 a.func()
12 b.func()
```

Η ανάλυσή μας μπορεί να διακρίνει τις δύο συναρτήσεις που ορίζονται στις γραμμές 2 και 6, καθώς αυτές ανήκουν σε δύο ξεχωριστές κλάσεις. Μέθοδοι βασισμένες σε πεδία για την JavaScript [13] θα αποτύχουν να διακρίνουν τις δύο κλήσεις, το οποίο προκαλεί ανακρίβεια. Αυτό συμβαίνει επειδή μέθοδοι βασισμένοι σε πεδία αντιστοιχούν όλες τις ομώνυμες προσβάσεις σε attributes στο ίδιο αντικείμενο.

3.1.1 Σύνταξη

Η ενδιάμεση αναπαράσταση πάνω στην οποία ενεργεί η ανάλυσή μας έχει τη σύνταξη μιας απλής προστακτικής, αντικειμενοστραφούς γλώσσας. Αυτή η σύνταξη φαίνεται στο Σχήμα 3.1. Τα πλαίσια αποτίμησης [30] για αυτή τη γλώσσα, τα οποία θα εξηγηθούν σύντομα, φαίνονται στον τελευταίο κανόνα του παραπάνω σχήματος.

$$
\begin{array}{rcl}
\langle e \in \textit{Expr} \rangle & ::= & o \mid x \mid f \mid x := e \mid \textbf{function}\ x\ (y\ldots)\ e \mid \textbf{return}\ e \mid \\
& & e(x{=}e\ldots) \mid \textbf{class}\ x\ (y\ldots)\ e \mid e.x \mid e.x := e \mid \\
& & e[x] \mid e[x] := e \mid \textbf{new}\ x\ (y = e\ldots) \mid \\
& & \textbf{import}\ x\ \textbf{from}\ m\ \textbf{as}\ y \mid \textbf{iter}\ x \mid e;e \\
\langle o \in \textit{Obj} \rangle & ::= & n, v \\
\langle v \in \textit{Definition} \rangle & ::= & x, \tau \\
\langle \tau \in \textit{IdentType} \rangle & ::= & \textbf{func} \mid \textbf{var} \mid \textbf{cls} \mid \textbf{mod} \\
\langle n \in \textit{Namespace} \rangle & ::= & (v)^* \\
\langle x, y \in \textit{Identifier} \rangle & ::= & \textit{is the set of program identifiers} \\
\langle m \in \textit{Modules} \rangle & ::= & \textit{is the set of modules} \\
\langle f \in \textit{Literal} \rangle & ::= & \textit{is the set of literals} \\
\langle E \rangle & ::= & [] \mid x := E \mid \textbf{return}\ E \mid E(x = e\ldots) \mid \\
& & o(x = E\ldots) \mid \textbf{new}\ x(y{=}E) \mid E.x \mid E.x := e \mid \\
& & o.x := E \mid o[x] \mid o[x] := E \mid \textbf{iter}\ o \mid E;e \mid o;E
\end{array}
$$

Σχήμα 3.1: Η σύνταξη που αντιπροσωπεύει τα δοθέντα προγράμματα Python και τα πλαίσια αποτίμησης.

Τα αναγνωριστικά είναι ένα σημαντικό στοιχείο αυτής της γλώσσας. Κάθε ένα τους μπορεί να ανήκει σε έναν από τους παρακάτω τύπους: (1) **func** που αντιστοιχεί στο όνομα μιας συνάρτησης (2) **var** που προσδιορίζει το όνομα μιας μεταβλητής, (3) **cls** για ονόματα κλάσεων, και (4) **mod** όταν το αναγνωριστικό είναι το όνομα ενός module. Κάθε ζεύγος $(x, \tau) \in \textit{Identifier} \times \textit{IdentType}$ σχηματίζει έναν ορισμό. Ένα αντικείμενο εκπροσωπείται από έναν ορισμό μαζί με το namespace του (βλ. κανόνα *Obj*). Ένα namespace συντίθεται ως μια ακολουθία ορισμών και είναι απαραίτητο για την διάκριση μεταξύ αντικειμένων τα οποία έχουν το ίδιο αναγνωριστικό. Για παράδειγμα, ας θεωρήσουμε το ακόλουθο κομμάτι Python κώδικα που βρίσκεται σε ένα module ονόματι `main`.

```
1  var = 10
2  class A:
3    var = 10
```

Η ανάλυσή μας μπορεί να διακρίνει τα αντικείμενα που ορίζονται στις γραμμές 1 και 3, καθώς το πρώτο από αυτά ορίζεται στο namespace [(`main`, **mod**)], ενώ το δεύτερο βρίσκεται στο namespace [(`main`, **mod**), (`A`, **cls**)].

Η μέθοδός μας χειρίζεται κάθε αντικείμενο ως την *τιμή* που προκύπτει από την αποτίμηση των εκφράσεων που ορίζονται από την γλώσσα. Συγκεκριμένα, η αναπαράστασή μας περιέχει εκφράσεις που περιγράφουν την διασυναρτησιακή ροή, τις εντολές ανάθεσης, τους ορισμούς κλάσεων και συναρτήσεων, τις εισαγωγές modules, και τους iterators / generators (βλ. κανόνα *Expr*). Η γλώσσα μπορεί να αναπαραστήσει αφαιρετικά διαφορά χαρακτηριστικά, συμπεριλαμβανομένων εκφράσεων lambda, ορισμάτων με λέξεις κλειδιά, κατασκευαστών, πολλαπλής κληρονομικότητας, και άλλων.

Κατ' αντιστοιχία με προηγούμενες εργασίες που εστιάζουν στην JavaScript [15, 31, 14], χρησιμοποιούμε πλαίσια αποτίμησης [30] τα οποία περιγράφουν σε ποια σειρά αποτιμώνται οι υποεκφράσεις. Για παράδειγμα, σε μια ανάθεση attribute $E.x := e$, το σύμβολο E υποδεικνύει ότι ο δέκτης του attribute x αποτιμάται αυτή τη στιγμή, και το $o.x := E$ σημαίνει ότι ο δέκτης έχει ήδη αποτιμηθεί σε ένα αντικείμενο $o \in \textit{Obj}$ (θυμίζουμε ότι η αποτίμηση εκφράσεων έχει ως αποτέλεσμά της αντικείμενα), και μετά η αποτίμηση θα συνεχίσει στο δεξί μέλος της ανάθεσης.

Σχόλια. Όταν καλείται μια συνάρτηση Python που επιστρέφει έναν generator (δηλ. όταν περιέχει μια εντολή `yield` αντί για `return`), οι κλήσεις λαμβάνουν χώρα μόνο όταν ο generator πράγματι χρησιμοποιείται. Όταν η ανάλυσή μας συναντά αυτές τις οκνηρές κλήσεις (π.χ. `gen = lazy_call(x)`), τις μοντελοποιεί δημιουργώντας ένα thunk (π.χ. `gen = lambda: lazy_call(x)`) το οποίο αποτιμάται μόνο όταν ο generator προσπελάζεται (μέσω του κατασκευάσματος **iter**).

$$\pi \in AssignG = Obj \hookrightarrow \mathcal{P}(Obj)$$
$$s \in Scope = Definition \hookrightarrow \mathcal{P}(Definition)$$
$$h \in ClassHier = Obj \hookrightarrow Obj^*$$
$$l \in ObjLit = Obj \hookrightarrow \mathcal{P}(Literal)$$
$$\sigma \in State = AssignG \times Scope \times Namespace \times ClassHier \times ObjLit$$

Σχήμα 3.2: Τομείς της ανάλυσης.

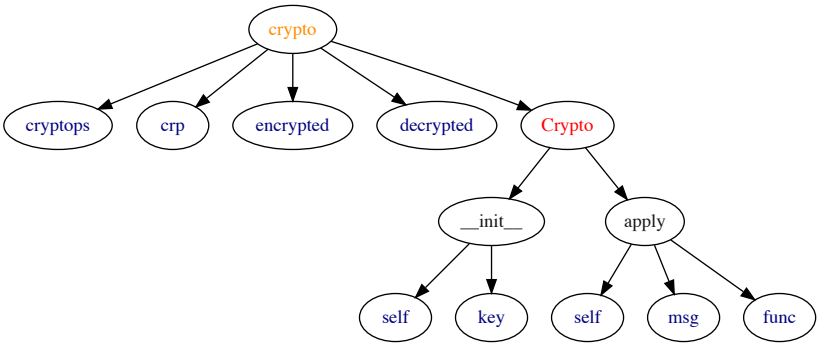
3.1.2 Κατάσταση

Έπειτα από την μετατροπή του αρχικού προγράμματος Python στην ενδιάμεση αναπαράστασή μας, η ανάλυσή μας ξεκινά την αποτίμηση της κάθε έκφρασης, και χτίζει σταδιακά τον γράφο αναθέσεων. Η ανάλυση το επιτυγχάνει αυτό διατηρώντας μια κατάσταση αποτελούμενη από τέσσερις τομείς, όπως φαίνεται στο Σχήμα 3.2, και συγκεκριμένα,*εμβέλεια*, *ιεραρχία κλάσεων*, *γράφος αναθέσεων*, *πίνακας αντιστοίχισης αντικειμένων σε literals*, και *τρέχον namespace*.

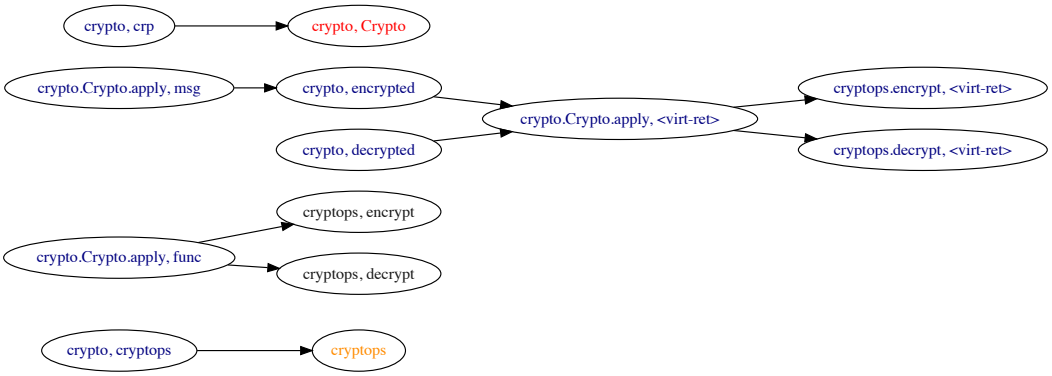
Η εμβέλεια είναι η αντιστοίχιση ορισμών σε ένα σύνολο από ορισμούς. Κατ' ουσίαν, μια εμβέλεια είναι ένα δέντρο στο οποίο κάθε κόμβος αντιστοιχείται σε έναν ορισμό (π.χ. μια συνάρτηση), και κάθε ακμή δείχνει τις σχέσεις γονιού / παιδιού μεταξύ ορισμών, δηλαδή ο κόμβος προορισμού ορίζεται μέσα στον ορισμό του κόμβου προέλευσης. Ο τομέας των εμβελειών βοηθά ώστε να επιλυθούν σωστά οι ορισμοί που είναι ορατοί μέσα σε ένα συγκεκριμένο namespace. Το Σχήμα 3.3a παρουσιάζει ένα δέντρο εμβέλειων του προγράμματος που φαίνεται στο Σχήμα 2.1, και απεικονίζει όλους τους ορισμούς του προγράμματος καθώς και τις μεταξύ τους σχέσεις. Οι κόκκινοι κόμβοι αντιστοιχούν σε ορισμούς κλάσεων, οι πορτοκαλί κόμβοι είναι ορισμοί modules, οι μαύροι κόμβοι εκπροσωπούν συναρτήσεις, ενώ οι μπλέ κόμβοι αντιστοιχούν σε μεταβλητές. Βάσει αυτού του δέντρου εμβελειών, συμπεραίνουμε πως η συνάρτηση `apply` ορίζεται μέσα στην κλάση μέσα στην κλάση `Crypto`, η οποία με την σειρά της ορίζεται μέσα στο module `module` (βλ. το μονοπάτι `crypto` → `Crypto` → `apply` → `func`). Αυτός ο τομέας μας βοηθά να χειριστούμε με ορθότητα χαρακτηριστικά της Python όπως function closures και εμφωλευμένους ορισμούς.

Η ιεραρχία κλάσεων είναι ένα δέντρο που εκπροσωπεί τις σχέσεις κληρονομικότητας μεταξύ κλάσεων. Μια ακμή από τον κόμβο u στον κόμβο v δείχνει ότι η κλάση v είναι γονιός της κλάσης u. Ο τομέας αυτός χρησιμοποιείται από την ανάλυση μας για την επίλυση attributes των κλάσεων (είτε μεθόδων είτε πεδίων) που ορίζονται σε βασικές κλάσεις του αντικειμένου-δέκτη. Μέσω αυτού του τομέα η ανάλυση μας δύναται να χειριστεί τα αντικειμενοστραφή χαρακτηριστικά της Python, και να διευθετήσει την πολλαπλή κληρονομικότητα και την σειρά επίλυσης των μεθόδων.

Ο γράφος αναθέσεων ορίζεται ως μια αντιστοίχιση από αντικείμενα σε ένα στοιχείο του υπερσυνόλου των αντικειμένων $\mathcal{P}(Obj)$. Ο γράφος αυτός περιέχει τις σχέσεις ανάθεσης μεταξύ αντικειμένων και περιγράφει τις αναθέσεις και την διασυναρτησιακή ροή του προγράμματος. Το Σχήμα 3.3b απεικονίζει τον γράφο αναθέσεων που αντιστοιχεί στο πρόγραμμα του Σχήματος 2.1. Κάθε κόμβος του γράφου (π.χ. `{crypto.Crypto.apply, func}`) αναπαριστά ένα αντικείμενο. Η ετικέτα κάθε κόμβου περιέχει δύο στοιχεία. Το πρώτο (π.χ., `crypto.Crypto.apply`) δείχνει ένα namespace όπου ορίζεται ένα αναγνωριστικό (π.χ. `func`). Τα χρώματα αντιστοιχούν στον τύπο ενός αναγνωριστικού όπως εξηγήθηκε παραπάνω (π.χ. το μαύρο χρώμα υπονοεί ορίσμους συναρτήσεων). Μια ακμή δείχνει τις πιθανές που μπορεί να έχει μια μεταβλητή. Για παράδειγμα, η μεταβλητή `func` που ορίζεται στο namespace `crypto.Crypto.apply` μπορεί να δείχνει στις συναρτήσεις `decrypt` και `encrypt`, οι οποίες έχουν αμφότερες οριστεί στο namespace `cryptops`. Ως ένα ακόμα παράδειγμα αναφέρουμε την ακμή που προέρχεται από τον κόμβο `{crypto.Crypto.apply, msg}` και οδηγεί στο `{crypto, encrypted}`). Η ακμή αυτή δείχνει ότι η παράμετρος `msg` της συνάρτησης `crypto.Crypto.apply` δείχνει στην μεταβλητή `encrypted` όταν η συνάρτηση καλείται στην γραμμή 12. Ο τομέας του γράφου ανάθεσης μας επιτρέπει να αντιμετωπίσουμε την πρόκληση αναφορικά με τον προγραμματισμό υψηλότερης τάξης στην Python.



(a) Το δέντρο εμβελειών για το `crypto` module.



(b) Ο γράφος αναθέσεων για το `crypto` module.

Σχήμα 3.3: Αναλύοντας το `crypto` module.

Ο πίνακας αντιστοίχισης αντικειμένων σε literals ορίζει μια σχέση μεταξύ αντικειμένων και μιας λίστας από literals. Συγκεκριμένα, για κάθε αντικείμενο διατηρούμε μια λίστα από literals στις οποίες το αντικείμενο μπορεί να δείχνει κατά τη διάρκεια της εκτέλεσης του προγράμματος. Ο τομέας αυτός είναι χρήσιμος για την αποδοτική επίλυση προσβάσεων σε dictionaries και λίστες. Ως παράδειγμα έχουμε την πρόσβαση `E[o]`. Μέσω του πίνακα αντιστοίχισης αντικειμένων σε literals μπορούμε να αναγνωρίσουμε τα πιθανά literals στα οποία δύναται να δείχνει το αντικείμενο o, και συνεπώς να επιλύσουμε την καταχώρηση για το dictionary στην οποία αυτή η πρόσβαση αποτιμάται.

Τέλος, το τρέχον namespace χρησιμοποιείται για τον εντοπισμό της θέσης όπου νέες μεταβλητές, κλάσεις, modules, και συναρτήσεις δηλώνονται. Ο τομέας αυτός μας επιτρέπει να εγκαθιδρύσουμε μια ακριβέστερη ανάλυση από αυτές που χρησιμοποιούνται σε προηγούμενες εργασίες οι οποίες είναι βασισμένες σε πεδία. Μέσω των namespaces, τα αντικείμενα και οι προσβάσεις σε attributes διαφοροποιούνται βάσει το namespace τους, διευθετώντας προκλήσεις όπως το duck typing.

3.1.3 Κανόνες Ανάλυσης

Η ανάλυση εξετάζει κάθε έκφραση που περιέχεται στην ενδιάμεση αναπαράσταση του αρχικού προγράμματος, και αλλάζει την κατάσταση της ανάλυσης με βάση τη σημασιολογία της κάθε έκφρασης. Ο αλγόριθμος επαναλαμβάνει αυτή τη διαδικασία μέχρι η κατάσταση να συγκλίνει, και ο γράφος αναθέσεων δίνεται από την τελική κατάσταση της ανάλυσης.

e-ctx

$$\frac{\langle \pi, s, n, h, l, e\rangle \hookrightarrow \langle \pi', s', n', h', l', e'\rangle}{\langle \pi, s, n, h, l, E[e]\rangle \to \langle \pi', s', n', h', l', E[e']\rangle}$$

compound

$$\frac{}{\langle \pi, s, n, h, l, E[o_1; o_2]\rangle \to \langle \pi, s, n, h, l, E[o_2]\rangle}$$

ident

$$\frac{o = \texttt{getObject}(s, n, x)}{\langle \pi, s, n, h, l, E[x]\rangle \to \langle \pi, s, n, h, l, E[o]\rangle}$$

assign

$$\frac{s' = \texttt{addScope}(s, n, x, \textbf{var}) \qquad o' = (n, (x, \textbf{var})) \qquad \pi' = \pi[o' \to \pi(o') \cup \{o\}]}{\langle \pi, s, n, h, l, E[x := o]\rangle \to \langle \pi', s', n, h, l, E[o']\rangle}$$

assign-lit

$$\frac{s' = \texttt{addScope}(s, n, x, \textbf{var}) \qquad o' = (n, (x, \textbf{var})) \qquad l' = l[o' \to l(o') \cup \{f\}]}{\langle \pi, s, n, h, l, E[x := f]\rangle \to \langle \pi, s', n, h, l', E[o']\rangle}$$

func

$$\frac{\begin{array}{c} s' = \texttt{addScope}(s, n, x, \textbf{func}) \\ n' = n \cdot (x, \textbf{func}) \qquad s'' = \texttt{addScope}(s', n', \texttt{ret}, \textbf{var}) \qquad s^{(3)} = \texttt{addScope}(s'', n', y, \textbf{var}) \end{array}}{\langle \pi, s, n, h, l, E[\textbf{function}\ x\ (y \dots)\ e]\rangle \to \langle \pi, s^{(3)}, n', h, l, E[e]\rangle}$$

return

$$\frac{o' = (n \cdot x, (\texttt{ret}, \textbf{var})) \qquad \pi' = \pi[o' \to \pi(o') \cup \{o\}]}{\langle \pi, s, n \cdot x, h, l, E[\textbf{return}\ o]\rangle \to \langle \pi', s, n, h, l, E[o']\rangle}$$

call

$$\frac{\begin{array}{c} o_1 = (n', (fn, \textbf{func})) \qquad o_2' = (n' \cdot fn, (y, \textbf{var})) \\ o_3 = (n' \cdot fn, (x, \textbf{var})) \qquad \pi' = \pi[o_2' \to \pi(o_2') \cup \{o_2\}] \qquad l' = l[o_3 \to l(o_3) \cup \{f\}] \end{array}}{\langle \pi, s, n, h, l, E[o_1(x = f, y = o_2, \dots)]\rangle \to \langle \pi', s, n, h, l, (n' \cdot fn, (\texttt{ret}, \textbf{var}))\rangle}$$

class

$$\frac{\begin{array}{c} s' = \texttt{addScope}(s, n, x, \textbf{cls}) \\ t = \langle \texttt{getObject(s, n, b)} \mid b \in (y \dots)\rangle \qquad h' = h[(n, (x, \textbf{cls})) \to t] \qquad n' = n \cdot (x, \textbf{cls}) \end{array}}{\langle \pi, s, n, h, l, E[\textbf{class}\ x\ \textbf{(}y \dots\textbf{)}\ e]\rangle \to \langle \pi, s', n', h', l, E[e]\rangle}$$

attr

$$\frac{o' = \texttt{getClassAttrObject}(o, x, h)}{\langle \pi, s, n, h, l, E[o.x]\rangle \to \langle \pi, s, c, h, l, E[o']\rangle}$$

new

$$\frac{o_3 = \texttt{getObject}(s, n, x) \qquad o_2 = \texttt{getClassAttrObject}(o_3, \texttt{\_\_init\_\_}, h)}{\langle \pi, s, n, h, l, E[\textbf{new}\ x(y = o_1 \dots)]\rangle \to \langle \pi, s, n, h, l, E[o_2(y = o_1 \dots)]; o_3]\rangle}$$

attr-assign

$$\frac{o_3 = \texttt{getClassAttrObject}(o_1, x, h) \qquad \pi' = \pi[o_3 \to \pi(o_3) \cup \{o_2\}]}{\langle \pi, s, n, h, l, E[o_1.x := o_2]\rangle \to \langle \pi', s, n, h, l, E[o_3]\rangle}$$

import

$$\frac{\begin{array}{c} o_2 = \texttt{getObject(s, m, x)} \\ s' = \texttt{addScope}(s, n, y, \textbf{var}) \qquad o_1 = (n, (y, \textbf{var})) \qquad \pi' = \pi[o_1 \to \pi(o_1) \cup \{o_2\}] \end{array}}{\langle \pi, s, n, h, l, E[\textbf{import}\ x\ \textbf{from}\ m\ \textbf{as}\ y]\rangle \to \langle \pi', s', n, h, l, E[o_1]\rangle}$$

iter-iterable

$$\frac{o' = \texttt{getClassAttrObject}(o, \texttt{\_\_next\_\_}, h)}{\langle \pi, s, n, h, l, E[\textbf{iter}\ o]\rangle \to \langle \pi, s, n, h, l, E[o'()]\rangle}$$

iter-generator

$$\frac{\texttt{getClassAttrObject}(o, \texttt{\_\_next\_\_}, h) = \text{undefined}}{\langle \pi, s, n, h, l, E[\textbf{iter}\ o]\rangle \to \langle \pi, s, n, h, l, E[o()]\rangle}$$

Σχήμα 3.4: Οι κανόνες της ανάλυσης.

Στο Σχήμα 3.4 δείχνουμε τους κανόνες μετάβασης καταστάσεων της ανάλυσης μας. Οι κανόνες ακολουθούν την μορφή:

$$\langle \pi, s, n, h, l, E[e] \rangle \rightarrow \langle \pi', s', n', h', l', E[e'] \rangle$$

Παρακάτω περιγράφουμε λεπτομερώς τον κάθε κανόνα.

Σύμφωνα με τον κανόνα [e-ctx], για μια έκφραση e στο πλαίσιο αποτίμησης E, έναν γράφο αναθέσεων π, ένα namespace n, μια ιεραρχία κλάσεων h, έναν πίνακα αντιστοίχισης αντικειμένων σε literals l, και μια εμβέλεια s, μπορεί να αποκτηθεί μια έκφραση e' στο πλαίσιο αποτίμησης E αν η αρχική έκφραση e αποτιμάται στην e'. Για τα ακόλουθα, η δυαδική πράξη $x \cdot y$ αναπαριστά την προσάρτηση του στοιχείου y στη λίστα x.

Ο κανόνας [compound] υποδεικνύει ότι στη περίπτωση μιας σύνθετης έκφρασης που αποτελείται από δύο αντικείμενα o_1, o_2, επιστρέφουμε το τελευταίο ως αποτέλεσμα της αποτίμησης. Σημειώνουμε ότι η αποτίμηση της σύνθετης έκφρασης προαπαιτεί να έχει αποτιμηθεί σε κάθε αντικείμενο κάθε υποόρος σύμφωνα με τα πλαίσια αποτίμησης που φαίνονται στο Σχήμα 3.1. Οι υπόλοιποι ακολουθούν την ίδια συμπεριφορά.

Ο κανόνας [ident] περιγράφει την περίπτωση όπου η αρχική έκφραση είναι ένα αναγνωριστικό x. Η ανάλυση ανακτά το αντικείμενο o που αντιστοιχεί στο αναγνωριστικό x, στο namespace n, βάσει του δέντρου εμβελειών s. Για να γίνει αυτό, η ανάλυση χρησιμοποιεί την συνάρτηση `getObject(`s`,` n, x`)`, η οποία διατρέχει κάθε στοιχείο y του namespace n σε αντίστροφη σειρά. Κατόπιν, εξετάζει το δέντρο εμβελειών s, για να ελέγξει αν ο κόμβος στοιχείου y έχει κάποιο παιδί που να ταιριάζει στο αναγνωριστικό x. Στην περίπτωση που δεν υπάρχει τέτοιο παιδί, η συνάρτηση `getObject` συνεχίζει με το επόμενο μέρος του namespace. Σημειώνουμε ότι αυτός ο κανόνας δεν επηρεάζει την κατάσταση της ανάλυσης.

Ο κανόνας [assign] αναθέτει το αντικείμενο o στο αναγνωριστικό x. Πρώτα, το αναγνωριστικό x προστίθεται στο namespace n του δέντρου εμβελειών s, κάνοντας χρήση της συνάρτησης `addScope(`s`,` n, x, τ`)`. Η συνάρτηση αυτή λειτουργεί προσθέτοντας μια ακμή με αρχή τον κόμβο που προσπελάζεται από το μονοπάτι n και καταλήγει στον κόμβο που δίνεται από τον ορισμό (x, τ). Έπειτα, ο γράφος αναθέσεων ανανεώνεται μέσω της πρόσθεσης μιας ακμής από το αντικείμενο που αντιστοιχεί στο αριστερό μέρος της ανάθεσης (o') προς το αυτό του δεξιού μέλους (). Σύμφωνα με αυτή την ανανέωση, η μεταβλητή x ορισμένη στο namespace n μπορεί να δείχνει στο αντικείμενο o.

Στην περίπτωση που ένα literal f βρίσκεται στο δεξί μέλος της ανάθεσης, χρησιμοποιούμε τον κανόνα [assign-lit], ο οποίος λειτουργεί παρόμοια με τον κανόνα [assign]. Η κύρια διαφορά είναι πως εδώ ανανεώνουμε τον πίνακα αντιστοίχισης αντικειμένων σε literals l αντί του γράφου αναθέσεων. Πιο συγκεκριμένα, ανανεώνουμε την καταχώρηση για το αναγνωριστικό x στον πίνακα έτσι ώστε να συμπεριλαμβάνει το f.

Ο κανόνας [func] ανανεώνει το δέντρο εμβελειών. Συγκεκριμένα, μια νέα εμβέλεια s' δημιουργείται μέσω της προσθήκης της συνάρτησης x στο τρέχον namespace n. Έπειτα, ένα νέο namespace n' δημιουργείται μέσω της προσθήκης του ορισμού συνάρτησης $(x, \textbf{func})$ στην κορυφή του τρέχοντος namespace. Ύστερα, ο κανόνας προσθέτει όλες τις παραμέτρους της συνάρτησης, και μια εικονική μεταβλητή ονόματι `ret`—η οποία αντιστοιχεί στην μεταβλητή που κρατά την τιμή επιστροφής της συνάρτησης— στο νέο namespace n'. Αυτό έχει ως αποτέλεσμα ένα νέο δέντρο εμβελειών $s^{(3)}$. Τέλος, η ανάλυση συνεχίζει με την αποτίμηση του σώματος της συνάρτησης x στο νέο namespace n' (σημειώνουμε ότι ο κανόνας αποτιμάται σε $E[e]$). Το νέο namespace n' ορθά αναγνωρίζει ότι όποια μεταβλητή έχει οριστεί μέσα στο e είναι πράγματι ορισμένη στο σώμα της συνάρτησης.

Ο κανόνας [return] αναθέτει το αντικείμενο o στην εικονική μεταβλητή `ret`, η οποία χρησιμοποιείται για την αποθήκευση της τιμής επιστροφής της συνάρτησης (υπενθυμίζουμε τον κανόνα [func]). Για την επίτευξη αυτού, η ανάλυση ανανεώνει τον γράφο αναθέσεων προσθέτοντας μια νέα ακμή από το αντικείμενο o' το οποίο αντιστοιχεί στην μεταβλητή επιστροφής `ret` προς το αντικείμενο o που είναι ο όρος του **return**. Στη συνέχεια, ο κανόνας αποτιμά το αντικείμενο o'. Μια παρόμοια ακολουθείται στην περίπτωση που επιστρέφεται ένα literal. Συγκεκριμένα, ανανεώνουμε τον πίνακα

αντιστοίχισης αντικειμένων σε literals αντί του γράφου αναθέσεων, με τρόπο παρόμοιο με αυτόν στον κανόνα [assign-lit]. Παραλείπουμε αυτή την περίπτωση, για λόγους συντομίας.

Η διασυναρτησιακή ροή ενθυλακώνεται από τον κανόνα [call]. Συγκεκριμένα, όταν η ανάλυση συναντά μια έκφραση κλήσης $o_1(x = f, y = o_2, \dots)$, εξετάζει το κληθέν αντικείμενο o_1 που σχετίζεται με την συνάρτηση fn η οποία ορίζεται στο namespace n'. Ύστερα, κάθε παράμετρος της fn συνδέεται με το κατάλληλο όρισμα που περάστηκε κατά την κλήση της συνάρτησης (π.χ. η παράμετρος y ανατίθεται στο o_2), οδηγώντας σε ένα νέο γράφο αναθέσεων π' και ένα νέο πίνακα αντιστοίχισης αντικειμένων σε literals. Για παράδειγμα, θεωρούμε ξανά τον γράφο του Σχήματος 3.3b. Οι ακμές που ξεκινούν στον κόμβο {`crypto.Crypto.apply, func`} κατασκευάζονται από αυτόν τον κανόνα. Αυτές οι ακμές υποδηλώνουν ότι οι συναρτήσεις `cryptops.encrypt` και `cryptops.decrypt`, οι οποίες περάστηκαν όταν κλήθηκε η συνάρτηση `crypto.Crypto.apply`, μπορεί να έχουν ανατεθεί στην παράμετρο `func` (Σχήμα 2.1).

Ο κανόνας [class] διαχειρίζεται ορισμούς κλάσεων. Ο κανόνας αυτός αρχικά προσθέτει την κλάση x στο δέντρο εμβελειών, μέσω της συνάρτησης `addScope`, και έπειτα ανακτά κάθε αντικείμενο που σχετίζεται με τις βασικές κλάσεις της x (δηλ. $y \dots$). Προκειμένου να γίνει αυτό, ο κανόνας συμβουλεύεται το δέντρο εμβελειών στο namespace n, και έπειτα από τον ορισμό μιας κλάσης ανακτά μια ακολουθία από αντικείμενα t που ακολουθεί τη σειρά με την οποία ορίστηκαν οι βασικές κλάσεις. Παρακάτω εξηγούμε γιατί είναι σημαντική η διατήρηση της σειράς με την οποία καταχωρήθηκαν οι βασικές κλάσεις. Ύστερα, αυτός ο κανόνας ανανεώνει την ιεραρχία κλάσεων έτσι ώστε μια προσφάτως ορισμένη κλάση x να είναι το παιδί των βασικών κλάσεων που δείχνονται από τα αναγνωριστικά $(y \dots)$. Στη συνέχεια, η ανάλυση προχωρά στο σώμα της κλάσης x σε ένα νέο namespace n'. Το νέο namespace περιέχει τον ορισμό της κλάσης στην κορυφή του τρέχοντος namespace (δηλ. $n \cdot (x, \textbf{cls})$). Τέλος, η ανάλυση ξεκινά να εξετάζει το σώμα της κλάσης κάνοντας χρήση ενός νέου namespace.

Ο κανόνας [attr] ακολουθεί προσέγγιση με τον κανόνα [ident]. Ωστόσο, σε αυτή τη περίπτωση, προκειμένου να αναγνωριστεί σωστά το αντικείμενο που αντιστοιχεί στο attribute x του αντικειμένου δέκτη o, η ανάλυση εξετάζει την ιεραρχία κλάσεων h χρησιμοποιώντας την συνάρτηση `getClassAttrObject`(o, x, h). Σε αυτό το σημείο η ανάλυση μας δύναται να διακρίνει attributes βάσει της τοποθεσίας (δηλ. o) στην οποία αυτά δηλώθηκαν.

Η συνάρτηση `getClassAttrObject` χειρίζεται την πολλαπλή κληρονομικότητα σεβόμενη την σειρά επίλυσης μεθόδων που υλοποιεί η Python. Για παράδειγμα, παραθέτουμε το ακόλουθο κομμάτι κώδικα.

```
1  class A:
2    def func():
3      pass
4
5  class B:
6    def func():
7      pass
8
9  class C(B, A):
10   pass
11
12 c = C()
13 c.func()
```

Στο παραπάνω παράδειγμα, η σειρά επίλυσης μεθόδων είναι $C \to B \to A$, καθώς η κλάση `B` είναι η πρώτη κλάση γονιός της `C`, ενώ η `A` είναι η δεύτερη κλάση γονιός. Σαν αποτέλεσμα, το `c.func()` οδηγεί στην κλήση της συνάρτησης `func` που ορίζεται στην κλάση `B`, αφού αυτή είναι η πρώτη μέθοδος που ταιριάζει στο όνομα `func` σύμφωνα με την σειρά επίλυσης μεθόδων. Η ορθή επίλυση των μελών της κλάσης εξηγεί γιατί ο τομέας της ιεραρχίας κλάσεων αντιστοιχεί κάθε αντικείμενο σε μια *ακολουθία* αντικειμένων αντί ενός συνόλου—οφείλουμε να καταγράφουμε την σειρά με την οποία οι γονείς μιας κλάσης καταχωρούνται.

Ο κανόνας [new] διαχειρίζεται την αρχικοποίηση αντικειμένων. Αυτός ο κανόνας ανακτά το αντικείμενο o_3 που σχετίζεται με τον ορισμό της κλάσης x. Μέσω της συνάρτησης `getClassAttrObject`

ο κανόνας ελέγχει την σειρά επίλυσης μεθόδων του αντικειμένου o_3 έτσι ώστε να αναγνωρίσει το πρώτο αντικείμενο o_2 που ταιριάζει με την συνάρτηση `__init__`. Υπενθυμίζουμε ότι αυτή η συνάρτηση καλείται κατά την δημιουργία ενός νέου αντικειμένου. Δίνουμε έμφαση στο πως αποτιμάται η έκφραση **new**: απλοποιείται στην $o_2(y = o_1); o_3$. Επι της ουσίας, πρώτα καλείται ο κατασκευαστής της κλάσης με τα ίδια ορίσματα που περάστηκαν μέσω της αρχικής έκφρασης (δηλ. $o_2(y = o_1)$), και έπειτα επιστρέφεται το αντικείμενο o_3 που αντιστοιχεί στον ορισμό της κλάσης, που είναι εν τέλει το αποτέλεσμα της έκφρασης **new**.

Ο κανόνας [attr-assign] χειρίζεται αναθέσεις attributes της μορφής $o_1.x := o_2$. Συγκεκριμένα, περιγράφει την περίπτωση όπου ένα attribute x ορίζεται σε κάποια τοποθεσία εντός της ιεραρχίας κλάσεων του αντικειμένου δέκτη o_1. Στη περίπτωση αυτή, η `getClassAttrObject` ανακτά το αντικείμενο o_3 που σχετίζεται με αυτό το attribute, και ο κανόνας ανανεώνει τον γράφο αναθέσεων έτσι ώστε το o_3 να δείχνει στο o_2 που προέρχεται από το δεξί μέλος της ανάθεσης. Στην περίπτωση που το attribute δεν έχει οριστεί εντός της ιεραρχίας κλάσεων, δηλαδή η `getClassAttrObject` επιστρέφει $\perp$, η ανάθεση attribute λειτουργεί παρόμοια με τον κανόνα [assign]. Αρχικά, προσθέτουμε το attribute x στην τρέχουσα εμβέλεια χρησιμοποιώντας την `addScope` και έπειτα ο γράφος ανανεώνεται. Αν ένα literal ανατίθεται αντί για ένα αναγνωριστικό η προσέγγιση μας συμπεριφέρεται με τον ίδιο τρόπο, με την μόνη διαφορά ότι ανανεώνει τον πίνακα αντιστοιχισής αντικειμένων σε literals αντί του γράφου ανάθεσεων.

Όταν η ανάλυση συναντά μια έκφραση **import** x **from** m **as** y, ανακτά το αντικείμενο o_2 που αντιστοιχεί στο εισαχθέν αναγνωριστικό x, το οποίο ορίζεται στο module m. Κατόπιν, δημιουργεί ένα ψευδώνυμο y για το x. Αυτό επιτυγχάνεται μέσω της προσθήκης του y στο δέντρο εμβελειών του τρέχοντος namespace, και μετέπειτα της ανανέωσης του γράφου αναθέσεων μέσω της προσθήκης μιας ακμής από το αντικείμενο y στο x. Μέσω αυτού του κανόνα αντιμετωπίζουμε το σύστημα των module της Python.

Η κατανάλωση iterables και generators επιτυγχάνεται μέσω της έκφρασης **iter** x. Όταν το αναγνωριστικό x δείχνει σε ένα iterable (δηλ. το αντικείμενο στο οποίο δείχνει το x έχει ένα attribute ονόματι `__next__`), ανακτούμε το αντικείμενο o' που αντιστοιχεί στο `__next__`. Έπειτα, η έκφραση **iter** αποτιμάται σε μια κλήση του $o'()$ (βλ. τον κανόνα [iter-iterable]). Διαφορετικά, το x αντιμετωπίζεται ως ένας generator ([iter-generator]), και η **iter** απλοποιείται σε μια κλήση $x()$. Θυμίζουμε από την Ενότητα 3.1.1 ότι οι generators μοντελοποιούνται ως thunks. Αυτό το σενάριο περιγράφει την αποτίμηση των thunks (generators) όταν αυτά πράγματι προσπελάζονται.

Τερματισμός της Ανάλυσης. Η ανάλυση διατρέχει τις εκφράσεις, αλλάζοντας την κατάσταση της σύμφωνα με τους κανόνες του Σχήματος 3.4, εώς ότου επιτευχθεί σύγκλιση. Η ανάλυση τερματίζει εγγυημένα επειδή οι τομείς της είναι πεπερασμένοι. Ακόμα και για τους τομείς της ιεραρχίας κλάσεων $h \in ClassHier$ και του πίνακα αντιστοίχισης αντικειμένων σε literals $l \in ObjLit$ (Σχήμα 3.2), οι οποίοι είναι θεωρητικά μη πεπερασμένοι, η ανάλυση εν τέλει τερματίζει. Αυτό οφείλεται στο γεγονός ότι ένα πρόγραμμα Python δεν μπορεί να έχει ένα μη φραγμένο αριθμό από κλάσεις. Επιπλέον, εισάγουμε καταχωρήσεις στον πίνακα αντιστοίχισης τιμών σε literals μόνο για εντολές που αναθέτουν (ή επιστρέφουν) ένα literal. Αποτελέσματα με την μορφή literal που οφείλονται σε κατασκευάσματα όπως οι generators, τα οποία θα μπορούσαν σε μη πεπερασμένο αριθμό από literals, αγνοούνται.

3.2 Κατασκευή Γράφου Κλήσεων Συναρτήσεων

Αφότου τερματίσει η ανάλυση, κατασκευάζουμε τον γράφο κλήσεων συναρτήσεων διατρέχοντας την ενδιάμεση αναπαράσταση του δοθέντος προγράμματος Python μια τελευταία φορά. Ο αλγόριθμος 3.1 περιγράφει τις λεπτομέρειες αυτού του περάσματος. Ο αλγόριθμος παίρνει δύο εισόδους: (1) ένα πρόγραμμα $p \in \mathit{Program}$ της γλώσσας μοντελοποίησης της οποίας η σύνταξη φαίνεται στο Σχήμα 3.1, και (2) την τελική κατάσταση $\sigma \in State$ που προκύπτει από το βήμα της ανάλυσης. Ο αλγόριθμος παράγει ένα γράφο κλήσεων συναρτήσεων

$$cg \in \mathit{CallGraph} = \mathit{Obj} \hookrightarrow \mathcal{P}(\mathit{Obj})$$

Αλγόριθμος 3.1: Κατασκευή Γράφου Κλήσεων Συναρτήσεων

```
   Input  : p ∈ Program
            σ ∈ State
   Output: cg ∈ CallGraph
1  foreach e in Program do
2      while e ∉ Obj do
3          ⟨σ, E[e]⟩ → ⟨σ′, E[e′]⟩
4          if e′ = o1(y = o2 ...) then // Call Expression
5              (π, s, n · f, h, l) ← σ′
6              c ← getReachableFuns(π, o1)
7              o3 ← getObject(s, n, f)
8              cg ← cg[f → cg(f) ∪ c] // Add Call Edges
9          end
10         e ← e′
11     end
12 end
13 return cg
```

Ο γράφος περιέχει μόνο αντικείμενα που συσχετίζονται με συναρτήσεις. Ένα στοιχείο $o \in Obj$, το οποίο αντιστοιχίζεται σε ένα σύνολο από αντικείμενα $t \in \mathcal{P}(Obj)$, σημαίνει ότι η συνάρτηση *ο μπορεί* να καλέσει οποιαδήποτε συνάρτηση συμπεριλαμβάνεται στο t.

Ο αλγόριθμος λειτουργεί εξετάζοντας κάθε έκφραση e που αναγνωρίστηκε στο πρόγραμμα (γραμμή 1), και κάθε έκφραση e αποτιμάται με βάση τους κανόνες μετάβασης καταστάσεων που περιγράφονται στο Σχήμα 3.4. Ο αλγόριθμος διατρέχει κατ'επανάληψιν τους κανόνες μετάβασης καταστάσεων, μέχρι η e να απλοποιήθει σε ένα αντικείμενο (γραμμές 2, 3). Οποτεδήποτε η e απλοποιείται σε μια έκφραση κλήσης της μορφής $o_1(y = o_2 \dots)$ (γραμμή 4), ο αλγόριθμος ανακτά το namespace όπου συνέβη η κλήση και αποσπά από αυτό το κορυφαίο στοιχείο (βλ. $n \cdot f$, γραμμή 5). Έπειτα, ο αλγόριθμος ανακτά όλες τις συναρτήσεις στις οποίες μπορεί να δείχνει το κληθέν αντικείμενο o_1. Για να το πετύχει αυτό, συμβουλεύεται τον γράφο αναθέσεων μέσω της συνάρτησης `getReachableFuns`(π, o_1), η οποία υλοποιεί έναν αλγόριθμο Αναζήτησης Κατά Βάθος (Depth-First Search, DFS) και ανακτά το σύνολο των συναρτήσεων c οι οποίες είναι προσβάσιμες από τον κόμβο o_1. Μετέπειτα, ο γράφος κλήσεων συναρτήσεων *cg* ανανεώνεται μέσω της προσθήκης όλων των ακμών από το κορυφαίο στοιχείο του τρέχοντος namespace προς το σύνολο των κληθέντων συναρτήσεων c (γραμμές 7, 8). Με άλλα λόγια, το αντικείμενο o_3 (γραμμή 7) που εκπροσωπεί το κορυφαίο στοιχείο του namespace στο οποίο πραγματοποιείται η κλήση είναι στην ουσία αυτό που καλεί τις συναρτήσεις στις οποίες δείχνει το αντικείμενο o_1.

3.3 Ανίχνευση Σφαλμάτων Κλειδιού

Η ανάλυση μας εντοπίζει πιθανά σφάλματα κλειδιού διατρέχοντας την ενδιάμεση αναπαράσταση μια ακόμα φορά αφότου η ανάλυση τερματίσει. Ο Αλγόριθμος 3.2 συνοψίζει τα βήματα αυτού του περάσματος. Ως είσοδο του, λαμβάνει ένα πρόγραμμα στη γλώσσα μοντελοποίησης μας (βλ. Σχήμα 3.1) και την τελική κατάσταση που προέκυψε από το βήμα της ανάλυσης. Ως έξοδο, προσφέρει μια λίστα με τις τοποθεσίες πιθανών σφαλμάτων κλειδιού

$$keyErr \in KeyErrorList = o \in Object$$

Αρχικά, ο αλγόριθμος εξετάζει όλες τις εκφράσεις που έχουν εντοπιστεί στο πρόγραμμα (γραμμή 1), εως ότου αυτές απλοποιηθούν σε μια πρόσβαση σε dictionary (γραμμές 2-4). Σε αυτή την περίπτωση, ο αλγόριθμος πρώτα ανακτά τα πιθανά dictionaries στα οποία μπορεί να ανατεθεί το εν λόγω dictionary (γραμμή 6). Αυτό επιτυγχάνεται μέσω της χρήσης της συνάρτησης `getReachableDicts`, η οποία διασχίζει τον γράφο αναθέσεων εκτελώντας τον αλγόριθμο Αναζήτησης Κατά Βάθος. Ύστερα,

Αλγόριθμος 3.2: Αναγνώριση Σφαλμάτων Κλειδιού

```
   Input  : p ∈ Program
            σ ∈ State
   Output: cg ∈ KeyErrorList
 1 foreach e in Program do
 2     while e ∉ Obj do
 3         ⟨σ, E[e]⟩ → ⟨σ′, E[e′]⟩
 4         if e′ = o1[o2] then // Dictionary Access
 5             (π, s, n, h, l) ← σ′
 6             Dicts ← getReachableDicts(π, o1)
 7             Vals ← getReachableLits(π, l, o2)
 8             foreach d in Dicts do
 9                 foreach v in Vals do
10                     if d · v ∉ π then // Dictionary entry does not exist
11                         KeyErrsList ← KeyErrsList ∪ [d · v]
12                     end
13                 end
14             end
15         end
16         e ← e′
17     end
18 end
19 return cg
```

γίνεται χρήση της μεθόδου `getReachableLits` για να ανακτηθούν τα πιθανά literals τα οποία μπορεί να λάβει ως τιμές το στοιχείο που προσπελαύνει το dictionary (γραμμή 7). Η μέθοδος αυτή, πρώτα διασχίζει τον γράφο αναθέσεων για να ανιχνεύσει τα αντικείμενα στα οποία έχει ανατεθεί αυτό το στοιχείο και μετέπειτα συμβουλεύεται τον πίνακα αντιστοίχισης αντικειμένων σε literals για να ανακτήσει τις πιθανές literal τιμές του. Τέλος, ο αλγόριθμος μας εξετάζει όλους τους συνδυασμούς dictionary-literal (γραμμές 8,9) και ελέγχει για την ύπαρξη ενός τέτοιου συνδυασμού ως κλειδί του dictionary στον γράφο αναθέσεων (γραμμή 10). Υπενθυμίζουμε ότι ο γράφος αναθέσεων περιέχει όλα τα αντικείμενα του προγράμματος Python ως κλειδιά, συμπεριλαμβανομένων των καταχωρήσεων των dictionaries. Αν ένας τέτοιος συνδυασμός δεν περιέχεται στον γράφο αναθέσεων, ο αλγόριθμος θεωρεί αυτή την πρόσβαση ως ένα σφάλμα κλειδιού και προσθέτει τη σχετική πληροφορία στη λίστα εξόδου (γραμμή 11).

3.4 Συζήτηση & Περιορισμοί

Μια βασική σχεδιαστική επιλογή είναι να αγνοήσουμε τους βρόγχους καθώς και τα αποτελέσματα υπολογισμού συνθηκών. Για παράδειγμα, όταν η ανάλυση μας συναντά μια εντολή `if`, λαμβάνει υπόψη και τους δύο κλάδους της εντολής. Αυτή η σχεδιαστική επιλογή ενισχύει την αποδοτικότητα χωρίς κάποιο υψηλό τίμημα για την ακρίβεια της ανάλυσης (όπως θα σχολιάσουμε στην Ενότητα 5). Άλλοι στατικοί αναλυτές [9, 10, 8] ακολουθούν μια πιο βαριά προσέγγιση και προσπαθήσουν να εκτιμήσουν τα αποτελέσματα των συνθηκών. Αυτοί οι στατικοί αναλυτές όμως, επιχειρούν να υπολογίσουν το σύνολο όλων των καταστάσεων που είναι προσβάσιμες βάση μιας αρχικής. Παρ' όλα αυτά, η εύρεση μιας τέτοιας αρχικής κατάστασης που εξασκεί όλα τα πιθανά μονοπάτια δεν είναι απλή, ειδικά όταν αναλύονται βιβλιοθήκες.

Η Python βασίζεται αρκετά σε αντικειμενοστραφή χαρακτηριστικά (π.χ. δυναμική αποστολή μεθόδων), duck typing [16], και modules. Συνεπώς, είναι σημαντικό να διαχωρίζονται οι προσβάσεις σε attributes βάσει του namespace στο οποίο το κάθε attribute ορίζεται. Αυτή η σχεδιαστική επιλογή ενισχύει—σε αντίθεση με προηγούμενες εργασίες [13]— την ακρίβεια της ανάλυσής μας χωρίς να θυσιάζει την κλιμακωσιμότητα της.

Επιπλέον, η ανάλυση μας δεν υποστηρίζει πλήρως όλα τα χαρακτηριστικά της Python. Δεν χειριζόμαστε σχήματα παραγωγής κώδικα, όπως κλήσεις στην built-in μέθοδο `eval`. Γενικά, αυτά τα δυναμικά κατασκευάσματα επιδεινώνουν την αποτελεσματικότητα οποιασδήποτε στατικής ανάλυσης, και συχνά χρησιμοποιούνται δυναμικές μέθοδοι ως αντίμετρο [32, 33]. Επιπροσθέτως, η μεθοδολογία μας δεν έχει μοντελοποιήσει τις built-in συναρτήσεις που αντιστοιχούν στους τύπους της Python. Συνεπώς, κλήσεις σε attributes που βασίζονται σε κάποιο συγκεκριμένο built-in τύπο (π.χ. `list.append()`) δεν επιλύονται. Τέλος, η ανάλυσή μας μπορεί να αναλύσει μόνο modules των οποίων ο κώδικας είναι διαθέσιμος. Όταν μια καλείται μια συνάρτηση—της οποίας ο ορισμός δεν είναι διαθέσιμος— η μέθοδός μας θα προσθέσει μια ακμή που καταλήγει σε αυτή την συνάρτηση, αλλά δεν θα προστεθούν ποτέ ακμές που να ξεκινούν από αυτήν, και η τιμή επιστροφής της θα αγνοηθεί.

Κεφάλαιο 4

Υλοποίηση

Έχουμε αναπτύξει το *PyCG*, ένα πρωτότυπο της μεθόδου μας γραμμένο σε Python 3. Το *PyCG* δεν εξαρτάται από πακέτα τρίτων και βασίζεται μόνο στα modules *symtable* και *ast* της standard βιβλιοθήκης της Python. Τα modules αυτά χρησιμοποιούνται αντίστοιχα για την δημιουργία του δέντρου εμβελειών και της ενδιάμεσης αναπαράστασης. Το πρωτότυπο μας λειτουργεί σαν ένα εργαλείο γραμμής εντολών (Command Line Interface, CLI), και έχει δοκιμαστεί σε περιβάλλοντα UNIX.

4.1 Χρήση

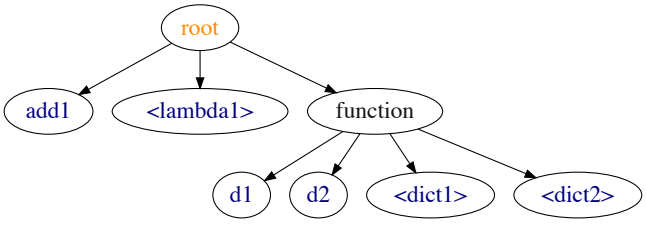
Το πρωτότυπο μας μπορεί να αναλύσει αποδοτικά πακέτα τα οποία εκτείνονται από απλά αρχεία σε μεγάλες βιβλιοθήκες. Στην περίπτωση πακέτων της Python, το πρωτότυπο μας δέχεται ένα όρισμα γραμμής εντολών το οποίο ορίζει τον κεντρικό φάκελο του πακέτου. Μέσω αυτού του ορίσματος καθίσταται δυνατή η χρήση των ακόλουθων λειτουργιών. Πρώτον, την παραγωγή των namespaces των εσωτερικών αντικειμένων σε σχέση με το namespace του ίδιου του πακέτου. Η ύπαρξη αυτών των σχετικών namespaces είναι σημαντική στο πλαίσιο εργαλείων που έχουν ως στόχο να συνδυάσουν τους γράφους κλήσεων συναρτήσεων αλληλοεξαρτώμενων πακέτων και κατόπιν επιτελέσουν διάφορα είδη ανάλυση σε αυτά (π.χ. dependency impact analysis). Δεύτερον, το πρωτότυπο μας υλοποιεί μια μεθοδολογία που μπορεί να ανακαλύψει αυτόματα τις τοποθεσίες των αρχείων του πακέτου στα οποία αναφέρεται μια εντολή *import*. Κατά συνέπεια, σε τέτοιες περιπτώσεις, το *PyCG* απαιτεί ως είσοδο μόνο εκείνα τα scripts που λειτουργούν ως σημεία εισόδου στο πακέτο, και έπειτα μπορεί να ανακαλύψει και να αναλύσει όλα τα εισαγόμενα modules. Περισσότερες λεπτομέρειες θα δοθούν στην Ενότητα 4.3

4.2 Εσωτερική Αναπαράσταση της Κατάστασης

Η εσωτερική αναπαράσταση του *PyCG* χρησιμοποιεί δομές δεδομένων για την αποδοτική ανάκτηση και ανανέωση των αντικειμένων που αποθηκεύονται από τους τομείς της ανάλυσης. Παρακάτω, περιγράφουμε λεπτομερώς την υλοποίηση του κάθε τομέα.

```
1  add1 = lambda x: x + 1
2  def function():
3      d1 = {
4          "key1": "val1",
5          "key2": "val2"
6      }
7      d2 = {"key3": "val3"}
```

(a) Ένα Python module που ορίζει ανώνυμα στοιχεία.



(b) Δέντρο εμβελειών του Python module.

Σχήμα 4.1: Δέντρο εμβελειών που περιέχει ανώνυμα στοιχεία ως thunks.

Namespace. Το τρέχον namespace υλοποιείται ως μια στοίβα από αναγνωριστικά. Στοιχεία προστίθενται και αφαιρούνται από την είσοδο κατά την είσοδο και αντίστοιχα την έξοδο από μια εμβέλεια. Αναπαριστούμε το namespace ως μια συμβολοσειρά μέσω της ένωσης των στοιχείων της στοίβας με το διαχωριστικό χαρακτήρα της τελείας (`.`).

Δέντρο Εμβελειών. Το δέντρο εμβελειών υλοποιείται ως ένας πίνακας κατακερματισμού που αντιστοιχεί namespaces σε καταχωρήσεις εμβέλειας. Κάθε καταχώρηση εμβέλειας αντιστοιχεί σε μια συγκεκριμένη εμβέλεια (π.χ. η εμβέλεια μιας συνάρτησης) και κράτα αποθηκευμένα δύο στοιχεία. Το πρώτο είναι μια αντιστοίχιση που χρησιμοποιούμε έτσι ώστε να ανακτήσουμε απευθείας τα αντικείμενα εκείνα στα οποία αναφέρονται τα αναγνωριστικά της εκάστοτε εμβέλειας. Το δεύτερο στοιχείο, είναι ένα σύνολο από μετρητές που αντιστοιχούν στα ανώνυμα στοιχεία που υποστηρίζει η ανάλυση μας, δηλαδή τα lambdas, τα dictionaries, και οι λίστες. Με αυτό τον τρόπο μπορούμε κάθε φορά να διακρίνουμε μεταξύ αυτών των ανώνυμων στοιχείων. Όταν η ανάλυση μας τα συναντά, δημιουργεί ένα νέο αναγνωριστικό το οποίο είναι ο συνδυασμός ενός thunk και του αντίστοιχου μετρητή, και στην συνέχεια προσθέτει αυτό το αναγνωριστικό ως παιδί της τρέχουσας εμβέλεια. Ως παράδειγμα, θεωρούμε το απόσπασμα κώδικα του Σχήματος 4.1a. Το απόσπασμα αυτό υλοποιεί ένα module που ορίζει ένα lambda και μια συνάρτηση που αρχικοποιεί δύο dictionaries. Το αντίστοιχο δέντρο εμβέλειων απεικονίζεται στο Σχήμα 4.1b. Σε αυτή τη περίπτωση, τα δύο λεξικά τα οποία ορίζονται στις γραμμές 2 και 4 μπορούν να διακριθούν βάσει του μετρητή που αντιστοιχεί στο ανώνυμο στοιχείο των dictionaries.

Χρησιμοποιούμε την built-in βιβλιοθήκη της Python *symtable* για να αποκτήσουμε πρόσβαση στους πίνακες συμβόλων του μεταγλωττιστή του κάθε module και να αρχικοποιήσουμε το δέντρο εμβελειών. Η διαδικασία που ακολουθούμε έχει ως εξής. Πρώτον, προσπελαύνουμε τις καταχωρήσεις συμβόλων της κύριας εμβέλειας του module και προσθέτουμε όλους τους ορισμούς συναρτήσεων και κλάσεων ως παιδιά της εμβέλειας αυτής στο δέντρο. Έπειτα, για κάθε παιδί διασχίζουμε τις καταχωρήσεις συμβόλων του αναδρομικά. Σημειώνουμε ότι αυτή η διαδικασία δεν προσθέτει ονόματα μεταβλητών και ανώνυμων στοιχείων στο δέντρο. Τα στοιχεία αυτά προσπελάζονται κατά τη διάρκεια του βήματος της ανάλυσης.

Ιεραρχία Κλάσεων. Ο τομέας της ιεραρχίας κλάσεων έχει υλοποιηθεί ως ένας πίνακας κατακερματισμού που χρησιμοποιείται για την αντιστοίχιση των namespaces στις ακολουθίες ιεραρχίας κλάσεων που ανήκουν σε αυτά. Οι ακολουθίες αυτές έχουν υλοποιηθεί ως λίστες. Κατασκευάζουμε την ιεραρχία κλάσεων για κάθε ορισμό κλάσεις συνδυάζοντας τις ιεραρχίες κλάσεων των γονικών κλάσεων βασιζόμενη στη Σειρά Επίλυσης Μεθόδων της Python[1].

Πίνακας Αντιστοίχισης Αντικειμένων σε Literals. Για μια ακόμη φορά, υλοποιούμε τον πίνακα αντιστοίχισης αντικειμένων σε literals ως μια αντιστοίχιση των namespaces των αντικειμένων σε σύνολα από literals. Στην περίπτωση της ανάθεσης ενός literal, ανακτούμε το σύνολο που διατηρείται για το αντικείμενο του αριστερού μέλους και ανανεώνουμε το σύνολο αυτό ώστε να συμπεριλαμβάνει το literal.

Γράφος Αναθέσεων. Διατηρούμε μια αντιστοίχιση από namespaces αντικειμένων σε καταχωρήσεις του γράφου αναθέσεων. Κάθε καταχώρηση περιέχει τον τύπο της και ένα σύνολο από namespaces στα οποία μπορεί να δείχνει η καταχώρηση αυτή. Υπενθυμίζουμε ότι η διαθέσιμοι τύποι είναι `function`, `module`, `variable`, και `class`.

4.3 Εντοπισμός της Τοποθεσίας Εισαγόμενων Modules

Κάνουμε χρήση της built-in βιβλιοθήκης της Python *importlib* για να ανακαλύψουμε τις τοποθεσίες των αρχείων των εισαγόμενων modules. Σημειώνουμε ότι η βιβλιοθήκη αυτή χρησιμοποιείται εσωτερικά από την Python για να επιλύσει εντολές εισαγωγής modules. Συγκεκριμένα, κάθε φορά που η Python ανιχνεύει μια εντολή εισαγωγής προχωρά με την ακόλουθη διαδικασία. Αρχικά, ανιχνεύεται η τοποθεσία του αρχείου του εισαγόμενου module, και στη συνέχεια ένας *loader* χρησιμοποιείται για

[1] Περισσότερες λεπτομέρειες για την Σειρά Επίλυσης Μεθόδων της Python μπορούν να βρεθούν στον σύνδεσμο `https://www.python.org/download/releases/2.3/mro/`.

```
{
  "cryptops.decrypt": [],
  "cryptops": [],
  "crypto.Crypto.apply": [
    "cryptops.encrypt",
    "cryptops.decrypt"
  ],
  "crypto.Crypto.__init__": [],
  "cryptops.encrypt": [],
  "crypto": [
    "crypto.Crypto.__init__",
    "crypto.Crypto.apply"
  ]
}
```

Σχήμα 4.2: Γράφος κλήσεων συναρτήσεων του `crypto` module σε μορφή JSON.

να εισάγει και να εκτελέσει τον κώδικα του module. Παρ' όλα αυτά η Python επιτρέπει την υλοποίηση loaders ειδικού σκοπού που μπορούν να αντικαταστήσουν τον προκαθορισμένο. Εκμεταλευτήκαμε αυτή τη δυνατότητα, και υλοποιήσαμε έναν loader που δεν εκτελεί τον κώδικα του εισαγόμενου module, αλλά αντ' αυτού αποθηκεύει την τοποθεσία του σε μια εσωτερική δομή που υλοποιήσαμε και κατόπιν τερματίζει. Η δομή αυτή, την οποία ονομάζουμε *γράφο εισαγωγών* υλοποιείται ως μια αντιστοίχιση από ονόματα modules σε σύνολα τοποθεσιών αρχείων. Έπειτα, το *PyCG* χρησιμοποιεί τον γράφο εισαγωγών για να αναλύσει τον κώδικα των εισαγόμενων modules. Μέσω της μεθοδολογίας αυτής, το μόνο που απαιτείται για την πλήρη ανάλυση ενός πακέτου Python είναι μια λίστα από σημεία εισόδου.

4.4 Δομή Εξόδου

Το πρωτότυπο μας υλοποιεί δομές εξόδου για κάθε μια από τις δύο από τις δύο λειτουργίες που περιγράψαμε, επιγραμματικά την παραγωγή γράφων κλήσεων συναρτήσεων και την ανίχνευση σφαλμάτων κλειδιού.

4.4.1 Δομή Γράφου Κλήσεων Συναρτήσεων

Το *PyCG* παράγει γράφους κλήσεων συναρτήσεων ως μια λίστα γειτνίασης σε δομή JSON. Το namespace για τον κάθε τύπο αντικειμένου που μπορεί να καλέσει συναρτήσεις (π.χ. module, συνάρτηση, κλάση) απεικονίζεται ως ένας κόμβος, και κάθε κόμβος διατηρεί μια λίστα από τα namespaces των συναρτήσεων που έχει καλέσει. Το Σχήμα 4.2 δείχνει την έξοδο για τον γράφο κλήσεων συναρτήσεων του module `crypto` (Σχήμα 2.1).

4.4.2 Δομή Σφαλμάτων Κλειδιού

Το *PyCG* δίνει στην έξοδο του τα πιθανά σφάλματα κλειδιού που ανιχνεύθηκαν στο πρόγραμμα Python που δόθηκε ως είσοδος ως μια λίστα από dictionaries. Κάθε dictionary περιέχει (1) την τοποθεσία του αρχείου στο οποίο ονομάστηκε το σφάλμα (2) τον αριθμό της γραμμής στον κώδικα, (3) το namespace του dictionary που προσπελάστηκε, και (4) το κλειδί με το οποίο επιχειρήθηκε η πρόσβαση. Για παράδειγμα, θεωρούμε το ενδεικτικό πρόγραμμα Python του Σχήματος 4.3a. Το πρόγραμμα αυτό ορίζει μια συνάρτηση που δέχεται δύο παραμέτρους: ένα dictionary και ένα κλειδί πρόσβασης. Έπειτα, ορίζει ένα dictionary με ένα και μοναδικό ζεύγος κλειδιού-τιμής. Στη γραμμή 6 το ενδεικτικό αυτό module επιχειρεί να προσπελάσει το dictionary με ένα μη έγκυρο κλειδί, πράγμα που οδηγεί σε ένα σφάλμα κλειδιού. Ακόμα, στις γραμμές 7 και 8 καλεί τη συνάρτηση που ορίζεται στη γραμμή 1 δίνοντας ως ορίσματα το dictionary και δύο διαφορετικά κλειδιά πρόσβασης– ένα μη

```python
def func(dct, name):
    dct[name]

dct = {"key1": "val1"}

dct["nokey"]
func(dct, "key2")
func(dct, "key1")
```

(a) Ένα ενδεικτικό Python πρόγραμμα με προσβάσεις σε dictionaries.

```
[{
    "filename": "mod.py",
    "lineno": 2,
    "namespace": "mod.<dict1>",
    "key": "key2"
},
{
    "filename": "mod.py",
    "lineno": 8,
    "namespace": "mod.<dict1>",
    "key": "nokey"
}]
```

(b) Σφάλματα κλειδιού την δομή που παράγει το *PyCG*.

Σχήμα 4.3: Ένα ενδεικτικό πρόγραμμα Python με σφάλματα κλειδιού και η έξοδος του *PyCG*.

έγκυρο και ένα έγκυρο αντίστοιχα. Το *PyCG* δίνει την έξοδο που συνοψίζεται στο Σχήμα 4.3b η οποία περιέχει τα δύο σφάλματα κλειδιού του προγράμματος εισόδου.

Κεφάλαιο 5

Αξιολόγηση

Σε αυτό το κεφάλαιο αξιολογούμε τις διαδικασίες παραγωγής γράφων κλήσεων συναρτήσεων και εντοπισμού μη έγκυρων προσπελάσεων dictionaries. Για να το κάνουμε αυτό, χρησιμοποιούμε benchmarks μικρής και μεγάλης κλίμακας. Για την παραγωγή γράφων κλήσεων συναρτήσεων, χρησιμοποιούμε ένα σύνολο από 112 Python benchmarks μικρής κλίμακας και πέντε benchmarks μεγάλης κλίμακας προερχόμενα από πραγματικές εφαρμογές. Αξιολογούμε τον μηχανισμού για τον εντοπισμό σφαλμάτων κλειδιού κάνοντας χρήση 25 Python benchmarks μικρής κλίμακας που οδηγούν σε σφάλματα κλειδιού και ένα σύνολο από 62 υποβολές εργασιών φοιτητών που υλοποιήθηκαν στα πλαίσια ενός μεταπτυχιακού μαθήματος ανάλυσης δεδομένων. Τα πειράματα μας πραγματοποιήθηκαν σε μια Debian 9 εικονική μηχανή με 16 επεξεργαστές και 16 GBs διαθέσιμης μνήμης RAM.

5.1 Γράφοι Κλήσεων Συναρτήσεων

5.1.1 Benchmarks Μικρής Κλίμακας

Προτείνουμε μια βιβλιοθήκη από benchmarks για την αξιολόγηση της παραγωγής γράφων κλήσεων συναρτήσεων σε Python. Άλλοι ερευνητές μπορούν να χρησιμοποιήσουν τη βιβλιοθήκη μας ώστε να αξιολογήσουν και να συγκρίνουν τις δικές τους μεθόδους με γνώμονα μια κοινή βάση. Οι Reif et al. [34] έχουν υλοποιήσει παρόμοια benchmarks για την γλώσσα Java, τα οποία περιέχουν μοναδικές περιπτώσεις γράφων κλήσεων συναρτήσεων οι οποίες έχουν οργανωθεί σε διάφορες κατηγορίες.

Η βιβλιοθήκη μας καλύπτει μια ευρεία γκάμα από λειτουργίες της Python και αποτελείται από 112 μοναδικά και απλά benchmarks μικρής κλίμακας. Τα benchmarks αυτά έχουν σχεδιαστεί ώστε να έχουν μικρή εμβέλεια και καλύπτουν συγκεκριμένες λειτουργίες (όπως decorators και lambdas). Έχουν οργανωθεί σε 16 διαφορετικές κατηγορίες, που κυμαίνονται από απλές κλήσεις συναρτήσεων μέχρι πιο σύνθετες λειτουργίες όπως συναρτήσεις υψηλότερης τάξης. Κάθε κατηγορία αποτελείται από ένα συγκεκριμένο αριθμό περιπτώσεων προς αξιολόγηση. Κάθε περίπτωση περιέχει (1) τον πηγαίο κώδικα, (2) τον γράφο κλήσεων συναρτήσεων (σε μορφή JSON, βλ. 4.4.1), και (3) μια περιγραφή (σε μορφή Markdown). Για κάθε περίπτωση, έχουμε υλοποίηση ένα μοναδικό μονοπάτι εκτέλεσης (π.χ. δεν υπάρχουν conditionals και βρόχοι) ώστε να υπάρχει μια σαφής αντιστοιχία στον γράφο κλήσεων συναρτήσεων. Ο Πίνακας 5.1 παραθέτει τις κατηγορίες μαζί με τον αριθμό των περιπτώσεων που υλοποιούν και μια αντίστοιχη περιγραφή.

Ο Πίνακας 5.2 παραθέτει τα αποτελέσματα της αξιολόγησης. Για κάθε benchmark που ανήκει σε μια συγκεκριμένη κατηγορία, ελέγχουμε αν το *PyCG* και το *Pyan* παρήγαγαν πλήρεις ή ορθούς γράφους κλήσεων συναρτήσεων. Σημειώνουμε ότι θεωρούμε έναν γράφο πλήρη όταν όλες του οι ακμές πράγματι υπάρχουν (μη ύπαρξη ψευδώς θετικών), και ορθό όταν περιέχει όλες τις ακμές οι οποίες αντιστοιχούν σε κλήσεις που πραγματοποιούνται κατά την εκτέλεση του προγράμματος (μη ύπαρξη ψευδώς αρνητικών).

Το *PyCG* παράγει έναν πλήρη γράφο σε σχεδόν όλες τις περιπτώσεις (111/112). Ακόμα, παράγει ορθούς γράφους για 103 από τα 112 benchmarks. Αυτό το έλλειμμα στην ορθότητα αποδίδεται σε μη πλήρη κάλυψη των λειτουργιών της Python, π.χ. τα starred assignments.

Πίνακας 5.1: Κατηγορίες benchmarks μικρής κλίμακας για γράφους κλήσεων συναρτήσεων.

Category	#tests	Description
`parameters`	6	Positional arguments that are functions
`assignments`	4	Assignment of functions to variables
`built-ins`	3	Calls to built in functions and data types
`classes`	22	Class construction, attributes, methods
`decorators`	7	Function decorators
`dicts`	12	Hashmap with values that are functions
`direct calls`	4	Direct call of a returned function (`func()()`)
`exceptions`	3	Exceptions
`functions`	4	Vanilla function calls
`generators`	6	Generators
`imports`	14	Imported modules, functions classes
`kwargs`	3	Keyword arguments that are functions
`lambdas`	5	Lambdas
`lists`	8	Lists with values that are functions
`mro`	7	Method Resolution Order (mro)
`returns`	4	Returns that are functions

Το *Pyan* σημειώνει χαμηλότερη επίδοση και για τις δύο μετρικές. Παρ' όλα αυτά, το *Pyan* υποστηρίζει καλύτερα την περίπτωση των αναθέσεων, πράγμα που οδηγεί σε ορθότερα αποτελέσματα. Για περαιτέρω κατανόηση των επιδόσεων του *Pyan*, πραγματοποιήσαμε μια ποιοτική ανάλυση των γράφων που παρήγαγε. Ανακαλύψαμε ότι το *Pyan* παράγει μη πλήρης γράφους επειδή δημιουργεί ακμές κλήσεων σε ονόματα κλάσεων καθώς και τις `__init__` μεθόδους τους (βλ. επίσης την Ενότητα 2.2). Ακόμα, παράγει ανακριβή αποτελέσματα επειδή δεν υποστηρίζει κάποιες από τις λειτουργίες της Python ($0/6$ generators και $0/3$ exceptions), δεν χειρίζεται την διασυναρτησιακή ροή ($0/6$ parameters και $0/4$ returns), δεν εντοπίζει κλήσεις σε εισαγόμενες συναρτήσεις ($4/14$), και έχει περιορισμένη υποστήριξη για κλάσεις ($10/22$).

Η αξιολόγηση του *Depends* δείχνει τόσο τους περιορισμούς του όσο και τα δυνατά του σημεία. Υπενθυμίζουμε ότι κάθε benchmark περιέχει πάντα μια κλήση που προέρχεται από το namespace του module. Τα αποτελέσματα μας δείχνουν ότι το *Depends* δεν ανιχνεύει αυτές τις κλήσεις, και συνεπώς η ορθότητα δεν επιτυγχάνεται ποτέ (0/112). Όσον αφορά την πληρότητα, το *Depends* επιτυγχάνει σχεδόν τέλεια αποτελέσματα (110/112) λόγω της συντηρητικής του φύσης—δηλαδή, προσθέτει μια ακμή όποτε πιστεύει με μεγάλη βεβαιότητα ότι η ακμή αυτή θα πραγματοποιηθεί.

5.1.2 Benchmarks Μεγάλης Κλίμακας

Κατασκευάσαμε με το χέρι τους γράφους κλήσεων συναρτήσεων πέντε δημοφιλών πακέτων αληθινών εφαρμογών. Τα πακέτα αυτά συλλέχθηκαν με την ακόλουθη διαδικασία. Πρώτον, ανακτήσαμε Python αποθετήρια μέσω του GitHub API και τα ταξινομήσαμε βάσει του αριθμού των αστεριών τους. Κατόπιν, κατεβάσαμε τον πηγαίο κώδικα του κάθε αποθετηρίου και μετρήσαμε τον συνολικό αριθμό των γραμμών του. Συμπεριλάβαμε στη βιβλιοθήκη μας τα πρώτα πέντε αποθετήρια που είχαν λιγότερες από 3.5χιλ. γραμμές κώδικα. Ο Πίνακας 5.3 παρουσιάζει τα αποθετήρια που επιλέξαμε μαζί με τις γραμμές κώδικα, τα αστέρια και τα forks στο GitHub, και μια σύντομη περιγραφή.

Επι του παρόντος, δεν υπάρχει κάποια διαθέσιμη υλοποίηση για την παραγωγή γράφων κλήσεων συναρτήσεων για προγράμματα Python με αποδοτικό τρόπο, συνεπώς εξετάσαμε με το χέρι τις εφαρμογές και κατασκευάσαμε τους γράφους τους σε μορφή JSON. Επιλέξαμε εφαρμογές μεσαίου μεγέθους (κατά μέσο όρο λιγότερες από 3.5χιλ. γραμμές κώδικα) προκειμένου να ελαχιστοποιήσουμε τον παράγοντα του ανθρώπινου σφάλματος.

Πίνακας 5.2: Αποτελέσματα της αξιολόγησης στα benchmarks μικρής κλίμακας για τα *PyCG* και *Pyan*. Το *Depends* είναι μη ορθό σε όλες τις περιπτώσεις και πλήρες σε 110/112 από αυτές και παραλείπεται.

Category	**PyCG**		**Pyan**	
	Complete	**Sound**	**Complete**	**Sound**
`assignments`	4/4	3/4	4/4	4/4
`built-ins`	3/3	1/3	2/3	0/3
`classes`	22/22	22/22	6/22	10/22
`decorators`	6/7	5/7	4/7	3/7
`dicts`	12/12	11/12	6/12	6/12
`direct calls`	4/4	4/4	0/4	0/4
`exceptions`	3/3	3/3	0/3	0/3
`functions`	4/4	4/4	4/4	3/4
`generators`	6/6	6/6	0/6	0/6
`imports`	14/14	14/14	10/14	4/14
`kwargs`	3/3	3/3	0/3	0/3
`lambdas`	5/5	5/5	4/5	0/5
`lists`	8/8	7/8	3/8	4/8
`mro`	7/7	5/7	0/7	2/7
`parameters`	6/6	6/6	0/6	0/6
`returns`	4/4	4/4	0/4	0/4
Total	**111/112**	**103/112**	**43/112**	**36/112**

Πίνακας 5.3: Λεπτομέρειες εφαρμογών της βιβλιοθήκης benchmarks μεγάλης κλίμακας για γράφους κλήσεων συναρτήσεων.

Project	**LoC**	**Stars**	**Forks**	**Description**
`fabric`	3,236	12.1k	1.8k	Remote execution & deployment
`autojump`	2,662	10.8k	530	Directory navigation tool
`asciinema`	1,409	7.9k	687	Terminal session recorder
`face_classification`	1,455	4.7k	1.4k	Face detection & classification
`Sublist3r`	1,269	4.4k	1.1k	Subdomains enumeration tool

Χρησιμοποιούμε την βιβλιοθήκη μας για να εξετάσουμε τα τρία εργαλεία ως προς τις μετρικές precision και recall. Η μετρική του precision μετρά το ποσοστό των έγκυρων παραγμένων κλήσεων δια του συνολικού αριθμού παραγμένων κλήσεων. Το recall, μετρά το ποσοστό των έγκυρων παραγμένων κλήσεων δια του συνόλου των κλήσεων.

Ο Πίνακας 5.4 παρουσιάζει τα αποτελέσματα. Σε δύο περιπτώσεις, το *Pyan* τερμάτισε απροσδόκητα κατά την παραγωγή του γράφου και για αυτό δεν συλλέξαμε αποτέλεσμα. Τα αποτελέσματα μας δείχνουν ότι το *PyCG* παράγει γράφους υψηλού precision. Σε όλες τις περιπτώσεις, πάνω από 98% των παραγόμενων ακμών κλήσεων είναι αληθώς θετικές, ενώ σε μια περίπτωση καμία από τις παραγμένες ακμές κλήσεων δεν είναι ψευδώς θετική. Τα αποτελέσματα όσον αφορά το recall, δείχνουν ότι κατά μέσο όρο το 69.9% του συνόλου των ακμών κλήσεων ανακτώνται επιτυχώς. Οι απούσες ακμές κλήσεων οφείλονται στους περιορισμούς της μεθόδους μας (βλ. Ενότητα 3.4), καθώς και σε ελλιπή υποστήριξη κάποιων από τις λειτουργίες της Python.

Το *Pyan* εκδηλώνει μέτριο precision και χαμηλό recall. Το μέτριο precision του *Pyan* οφείλεται στο γεγονός ότι το *Pyan* προσθέτει ακμές κλήσεων τόσο στα ονόματα κλάσεων όσο και στις μεθόδους `__init__()`. Επιπλέον, δεν χειρίζεται τη διασυναρτησιακή ροή των συναρτήσεων, οδηγώντας σε χαμηλό recall. Για παράδειγμα, ο πηγαίος κώδικας του πακέτου `face_classification` εξαρτάται

Πίνακας 5.4: Σύγκριση των αποτελεσμάτων για γράφους κλήσεων συναρτήσεων στην βιβλιοθήκη benchmarks μεγάλης κλίμακας.

Project	Precision (%)			Recall (%)		
	PyCG	Pyan	Depends	PyCG	Pyan	Depends
`autojump`	99.5	66.5	99.2	68.2	28.5	22.5
`fabric`	98.3	-	100	61.9	-	6.3
`asciinema`	100	-	98.1	68	-	15.5
`face_classification`	99.5	86.8	96.2	89.7	7.6	5.7
`Sublist3r`	98.8	69.8	100	61.6	25.6	21.9
Average	**99.2**	**74.4**	**98.7**	**69.9**	**20.6**	**14.4**

Πίνακας 5.5: Κατηγορίες benchmarks μικρής κλίμακας για σφάλματα κλειδιού.

Category	#tests	Description
`assignments`	3	Assignment of dictionaries to variables
`classes`	6	Class construction, attributes, methods
`dicts`	9	Dictionary operations
`kwargs`	2	Keyword arguments that are dictionaries
`lists`	2	Lists with values that are dictionaries
`parameters`	3	Positional arguments that are dictionaries

κατά κύριο λόγο από συναρτήσεις οι οποίες προέρχονται από εξωτερικά πακέτα. Το *Pyan* αγνοεί αυτές τις κλήσεις, πράγμα που οδηγεί σε recall 7.6%.

Τέλος, το *Depends* παράγει γράφους με υψηλό precision (98.7%) και χαμηλό recall. Τα αποτελέσματα για το precision μπορούν να αποδοθούν στην συντηρητική υλοποίηση που ακολουθεί το *Depends*. Επιπροσθέτως, το *Depends* δεν καταγράφει συναρτήσεις υψηλότερης τάξης και χάνει κλήσεις που προέρχονται από namespaces, πράγμα το οποίο οδηγεί στο χαμηλό του recall.

5.2 Σφάλματα Κλειδιού

5.2.1 Benchmarks Μικρής Κλίμακας

Υλοποιήσαμε μια βιβλιοθήκη από 25 μοναδικά και απλά προγράμματα Python τα οποία περιέχουν κώδικα που οδηγεί σε μη έγκυρες προσπελάσεις dictionary. Οι περιπτώσεις των δοκιμών μας χωρίζονται σε 6 διαφορετικές κατηγορίες, οι οποίες κυμαίνονται από διαφορετικές μεθόδους ανάθεσης ως την διασυναρτησιακή ροή των dictionaries και των στοιχείων που χρησιμοποιούνται για πρόσβαση σε αυτά. Χρησιμοποιούμε αυτή τη βιβλιοθήκη για να αξιολογήσουμε τον μηχανισμό ανίχνευσης σφαλμάτων κλειδιού σε διαφορετικές συνθήκες που μπορούν να προκύψουν σε ένα τυπικό Python πρόγραμμα.

Η βιβλιοθήκη μας είναι δομημένη με παρόμοιο τρόπο με αυτή της βιβλιοθήκης για τους γράφους κλήσεων συναρτήσεων. Συγκεκριμένα, κάθε κατηγορία υλοποιεί έναν αριθμό από περιπτώσεις. Κάθε περίπτωση περιέχει (1) τον πηγαίο κώδικα, (2) τα σφάλματα κλειδιού που θα προκύψουν από το πρόγραμμα στην ίδια δομή που χρησιμοποιούμε για την έξοδο του *PyCG* (βλ. Ενότητα 4.4.2)., και (3) μια σύντομη περιγραφή (σε μορφή Markdown). Ο Πίνακας 5.5 παραθέτει τις κατηγορίες, τον αριθμό των περιπτώσεων που περιέχουν, και μια αντίστοιχη περιγραφή.

Συνοψίζουμε τα αποτελέσματα της αξιολόγησης μας στον Πίνακα 5.6. Ελέγχουμε αν η υλοποίηση μας μπορεί να παράγει πλήρη και ορθά αποτελέσματα.

Πίνακας 5.6: Αποτελέσματα της αξιολόγησης στα benchmarks μικρής κλίμακας για σφάλματα κλειδιού.

	Complete	**Sound**
`assignments`	2/3	2/3
`classes`	6/6	6/6
`dicts`	9/9	8/9
`kwargs`	2/2	2/2
`lists`	2/2	2/2
`parameters`	3/3	3/3
Total	**24/25**	**23/25**

Το *PyCG* παράγει μια πλήρη λίστα από σφάλματα κλειδιού σε 24/25 περιπτώσεις και μια ορθή λίστα σε 23/25 περιπτώσεις. Τα ελλείμματα στην επίδοση για την κατηγορία των αναθέσεων οφείλονται στο ότι το *PyCG* δεν υποστηρίζει starred assignments. Ακόμα, το *PyCG* μια μη ορθή λίστα από σφάλματα σε μια περίπτωση της κατηγορίας των dictionaries, λόγω μιας κλήσης στην built-in μέθοδο `update` του τύπου του `dictionary`. Υπενθυμίζουμε (Ενότητα 3.4) ότι το *PyCG* αγνοεί κλήσεις σε built-in συναρτήσεις.

5.2.2 Benchmarks Μεγάλης Κλίμακας

Προκείμενου να αξιολογήσουμε τον μηχανισμού εντοπισμού σφαλμάτων κλειδιού σε αληθινές εφαρμογές Python, συλλέξαμε τις υλοποιημένες σε Python υποβολές των φοιτητών ενός μεταπτυχιακού μαθήματος πρακτικής ανάλυσης δεδομένων. Συγκεκριμένα, οι φοιτητές έπρεπε να υλοποιήσουν μια εφαρμογή αγοραπωλησίας μετοχών. Τους δόθηκε πρόσβαση σε αρχεία κειμένου που περιείχαν ιστορικά δεδομένα συναλλαγών του χρηματιστηρίου της Νέας Υόρκης και τους ζητήθηκε να παράξουν μια ακολουθία από συναλλαγές που θα οδηγούσε σε μέγιστο κέρδος.

Αναλύσαμε ένα σύνολο από 62 υποβολές και εντοπίσαμε 11 σφάλματα κλειδιού σε 3 από αυτές. Πραγματοποιήσαμε μια ποιοτική ανάλυση στις υποβολές κάνοντας την παραδοχή ότι οι υποβληθείσες λύσεις δεν θα οδηγούσαν σε μη έγκυρες προσβάσεις σε dictionaries, όταν αυτές εκτελούνταν στους υπολογιστές των ίδιων των φοιτητών. Η ανάλυση μας εστίασε στις τρεις υποβολές για τις οποίες ο μηχανισμός μας βρήκε σφάλματα κλειδιού, ενώ αναλύσαμε επίσης πολλές από τις υποβολές που έδωσαν καθαρό αποτέλεσμα, έτσι ώστε να αναγνωρίσουμε πιθανά ψευδώς αρνητικά αποτελέσματα.

Η ποιοτική μας ανάλυση έδειξε ότι ο μηχανισμός μας εντόπισε σφάλματα κλειδιού όταν τα προγράμματα εισόδου αρχικοποιούσαν τα dictionaries με βάση εξωτερικούς παράγοντες (π.χ. την ύπαρξη ενός αρχείου) και κατόπιν λειτουργούσαν κάνοντας την παραδοχή ότι αυτοί οι εξωτερικοί παράγοντες θα είχαν συγκεκριμένες ιδιότητες (π.χ. αρχεία με συγκεκριμένα ονόματα). Για παράδειγμα, θεωρούμε την παρακάτω απλοποιημένη προσέγγιση από μια από τις υποβολές.

```
1  files = os.listdir("stocks")
2  stock_contents = {}
3
4  for file in files:
5      with open(file) as f:
6          stock_contents[file] = f.read()
7
8  apple = stock_contents["AAPL.txt"]
```

Αρχικά, στην γραμμή 1 ανακτάται μια λίστα από ονόματα αρχείων και στη γραμμή 2 δημιουργείται ένα dictionary στο οποίο αργότερα θα αποθηκευτούν τα περιεχόμενα των αρχείων αυτών. Στη συνέχεια, στον βρόχο των γραμμών 4 εώς 6 διαβάζονται τα περιεχόμενα του κάθε αρχείου και αποθηκεύονται στο dictionary με το όνομα του αρχείου να λειτουργεί ως κλειδί. Τέλος, στην γραμμή 8

Πίνακας 5.7: Σύγκριση Χρόνου Εκτέλεσης και Μνήμης.

Project	**Time (sec)**			**Memory (MB)**		
	PyCG	**Pyan**	**Depends**	**PyCG**	**Pyan**	**Depends**
`autojump`	0.76	0.42	2.37	62.7	37.8	27.1
`fabric`	0.77	-	1.83	60.9	-	18.5
`asciinema`	0.87	-	2	61.6	-	19.4
`face_classification`	0.92	0.38	2.49	60.9	35.3	25.6
`Sublist3r`	0.51	0.33	2.01	60	35.8	19.4
Average	**0.77**	**0.38**	**2.14**	**61.2**	**36.3**	**22**

ανακτώνται τα περιεχόμενα του αρχείο `AAPL.txt`. Ο κώδικας αυτός, κάνει την παραδοχή ότι ο φάκελος `stocks` θα περιέχει ένα αρχείο `AAPL.txt` επειδή στην γραμμή 8 κάνει απευθείας πρόσβαση στην συγκεκριμένη καταχώρηση χωρίς να συμβουλευτεί την λίστα αρχείων που περιέχονται στο φάκελο. Σε τέτοιες περιπτώσεις, το *PyCG* αναγνωρίζει σφάλμα κλειδιού επειδή δεν έχει διαθέσιμη κάποια πληροφορία σχετικά με το περιβάλλον στο οποίο αναμένεται να εκτελεστεί ο κώδικας. Παρουσιάζει ενδιαφέρον το γεγονός ότι οι υποβολές για τις οποίες το *PyCG* δεν εντόπισε σφάλμα λειτουργούσαν με το ίδιο περιβάλλον αρχείων, αλλά φρόντισαν να αρχικοποιήσουν τα dictionaries τους με βάση την κατάσταση του περιβάλλοντος και χωρίς να κάνουν οποιαδήποτε παραδοχή— πράγμα που αποτελεί καλή πρακτική προγραμματισμού.

5.3 Επιδόσεις Χρόνου Εκτέλεσης και Μνήμης

Πραγματοποιούμε την αξιολόγησή μας ως προς τον χρόνο εκτέλεσης και την μνήμη χρησιμοποιώντας ως βάση την βιβλιοθήκη με benchmarks μεγάλης κλίμακας που χρησιμοποιήθηκε για την παραγωγή γράφων κλήσεων συναρτήσεων (Πίνακας 5.3). Όσον αφορά την ανίχνευση σφαλμάτων κλειδιού, το εργαλείο μας επιτελεί την ίδια ανάλυση, και συνεπώς παράγει παρόμοια αποτελέσματα σχετικά με τον χρόνο εκτέλεσης και την μνήμη. Ο Πίνακας 5.7 παρουσιάζει τα αποτελέσματα για το *PyCG*, *Pyan*, και *Depends*. Χρησιμοποιήσαμε την εντολή *time* του UNIX για να υπολογίσουμε τους χρόνους εκτέλεσης και την εντολή *pmap* για να μετρήσουμε την κατανάλωση μνήμης. Τα αποτελέσματα που παρουσιάζονται είναι ο μέσος όρος από 20 εκτελέσεις.

Σύμφωνα με τα αποτελέσματα μας, το *Pyan* είναι πιο αποδοτικό στον χρόνο εκτέλεσης και το *Depends* είναι πιο αποδοτικό στη μνήμη. Το *PyCG* και το *Pyan* χρειάζονται λιγότερο από ένα δευτερόλεπτο για να παράξουν τον γράφο κλήσεων συναρτήσεων του προγράμματος ($\leq$ 3.5χιλ. γραμμές κώδικα), ενώ το *Depends* χρειάζεται κατά μέσο όρο περισσότερο από δύο δευτερόλεπτα. Ακόμα, όλα τα εργαλεία απαιτούν ένα λογικό ποσό μνήμης, με το *PyCG*, το *Pyan*, και το *Depends* να χρειάζονται κατά μέσο όρο $\sim$61.4, $\sim$36.3, και $\sim$22 MBs αντίστοιχα. Το *Pyan* είναι κατά μέσο όρο δύο φορές πιο γρήγορο από το *PyCG*, αν και το *PyCG* χρησιμοποιεί 2.8 φορές περισσότερη μνήμη από το *Depends*. Αποδίδουμε αυτές τις διαφορές μεταξύ του *Pyan* και του *PyCG* στο γεγονός ότι το *Pyan* διασχίζει το AST του προγράμματος δύο φορές, ενώ το *PyCG* εκτελεί μια επανάληψη σταθερού σημείου (Κεφάλαιο 3). Το *Depends* είναι πιο αργό, εφόσον αφιερώνει το μεγαλύτερο μέρος του χρόνου εκτέλεσης στην συντακτική ανάλυση των αρχείων κώδικα. Σχετικά με την μνήμη, το *Pyan* και το *Depends* διατηρούν λιγότερη πληροφορία για την κατάσταση της ανάλυσης και αυτό οδηγεί σε καλύτερη επίδοση μνήμης.

Κεφάλαιο 6

Σχετικές Εργασίες

Στατική Ανάλυση για Δυναμικές Γλώσσες. Στην περίπτωση της JavaScript υπάρχουν πολλά frameworks που έχουν στόχο τη στατική ανάλυση προγραμμάτων. Το framework SAFE [9] υλοποιεί μια κλιμακώσιμη, ενσωματώσιμη, και ευέλικτη μεθοδολογία στατικής ανάλυσης. Το JSAI [10] προτείνει μια αποδεδειγμένα ορθή και τυπικά καθορισμένη ανάλυση η οποία χρησιμοποιεί abstract interpretation.

Άλλοι αναλυτές για τη JavaScript στοχεύουν σε διαφορετικές πτυχές της γλώσσας. Οι Madsen et al. προτείνουν το RADAR [35], ένα εργαλείο για την εύρεση λαθών σε event-driven προγράμματα. Οι Sotiropoulos et al. [14] υλοποιούν μια ανάλυση που στοχεύει σε ασύγχρονες συναρτήσεις. Το $\text{SAFE}_{\text{WAPI}}$ είναι ένα εργαλείο που μπορεί να εντοπίσει πιθανές λάθος χρήσεις API. Το $\text{SAFE}_{\text{WApp}}$, υλοποιεί μια στατική ανάλυση για κώδικα JavaScript που εκτελείται στην πλευρά του client.

Παραγωγή Γράφων Κλήσεων Συναρτήσεων. Οι μέθοδοι που παράγουν γράφους κλήσεων συναρτήσεων χωρίζονται σε δυναμικές [36] και στατικές [37]. Οι δυναμικές μέθοδοι συνήθως παράγουν λιγότερα ψευδώς θετικά, όμως υστερούν στο χρόνο εκτέλεσης. Επιπλέον, οι μέθόδοι αυτοί μπορούν να αναλύσουν ένα μοναδικό μονοπάτι εκτέλεσης και η αποτελεσματικότητα τους βασίζεται στην παρεχόμενη είσοδο. Αντιθέτως, οι στατικές μέθοδοι είναι πιο αποδοτικές ως προς τον χρόνο εκτέλεσης και συνήθως επιχειρούν να εντοπίσουν όλες τις πιθανές συμπεριφορές του προγράμματος, οδηγώντας σε μεγαλύτερη κάλυψη των μονοπατιών εκτέλεσης. Ο συνδυασμός των δύο προσεγγίσεων έχει δοκιμαστεί σε διάφορες περιπτώσεις [38, 39, 40].

Υπάρχουν πολλές μέθοδοι και εργαλεία που επιχειρούν την παραγωγή γράφων κλήσεων συναρτήσεων για γλώσσες προγραμματισμού με στατικό σύστημα τύπων. Το DOOP [41] και το WALA [42] ακολουθούν μια context-sensitive points-to ανάλυση για τη γλώσσα Java. Το PADDLE [43] υιοθετεί μια παρόμοια προσέγγιση και χρησιμοποιεί Binary Decision Diagrams (BDDs) [44]. Τέλος, το OPAL [45] είναι μια lattice based μέθοδος γραμμένη σε Scala. Οι Ali et al. [46], υλοποιούν το CGC, ένα μερικό γεννήτορα γράφων κλήσεων συναρτήσεων για Java που εστιάζει κατά κύριο λόγο στην αποδοτικότητα. Το εργαλείο δεν εντοπίζει κλήσεις που προέρχονται από εξωτερικές βιβλιοθήκες και εστιάζει στην ανάλυση του κώδικα του εκάστοτε πακέτου εισόδου. Επι του παρόντος ακολουθούμε μια παρόμοια μέθοδο. Παρ' όλα αυτά στοχεύουμε να αναλύουμε αποδοτικά κλήσεις συναρτήσεων από εξωτερικές βιβλιοθήκες στο μέλλον.

Σχετικά με δυναμικές γλώσσες, οι Ali et al. [47] υλοποιούν μια μέθοδο που μετατρέπει κώδικα Python σε JVM bytecode, και κατόπιν παράγει τον γράφο μέσω υπαρχουσών υλοποιήσεων για Java [42, 48, 49]. Τα αποτελέσματα τους δείχνουν ότι η παραγωγή γράφων μέσω αυτής της μεθόδου δεν είναι πραγματοποιήσιμη καθώς η έξοδος μερικές φορές περιέχει πάνω από 96% ψευδώς θετικά. Το *pycallgraph* [50] είναι μια δυναμική μέθοδος που παράγει Python γράφους κλήσεων συναρτήσεων για ένα μοναδικό μονοπάτι εκτέλεσης. Κατά συνέπεια η ανάλυση απαιτείται να συμπληρωθεί από κάποια άλλη μέθοδο (π.χ. fuzzing) ώστε να σημειώσει χρήσιμα αποτελέσματα. Στο χώρο της JavaScript, οι Feldthaus et al. [13] προτείνουν μια ανάλυση βασισμένη στη ροή για την παραγωγή γράφων. Το benchmark που χρησιμοποιούν για την αξιολόγηση περιέχει γράφους που έχουν παραχθεί με δυναμικές μεθόδους σε συνδυασμό με instrumentation, και η ανάλυση τους επιτυγχάνει $\geq 66\%$ precision και $\geq 85\%$ recall. Άλλοι αναλυτές για JavaScript συμπεριλαμβάνουν τα NPM call graph [51], IBM WALA [42], Approximate Call Graph (ACG) [13], Google closure compiler [52], και Type Analyzer

for JavaScript (TAJS) [8]. Το TAJS υλοποιεί μια lattice-based βασισμένη στη ροή ανάλυση που χρησιμοποιεί abstract interpretation. Παρ' όλο που αυτή η προσέγγιση σημειώνει περισσότερο υποσχόμενα αποτελέσματα, έχει ένα τίμημα στον χρόνο εκτέλεσης.

Benchmarks για Γράφους Κλήσεων Συναρτήσεων. Οι Reif et al. προτείνουν το Judge [34], ένα σύνολο εργαλείων για την ανάλυση γεννητόρων γράφων κλήσεων συναρτήσεων για Java. Αυτό το σύνολο εργαλείων, υλοποιεί μια βιβλιοθήκη με benchmarks για μια ευρεία γκάμα λειτουργιών της Java. Οι συγγραφείς χρησιμοποιούν αυτή τη βιβλιοθήκη για να συγκρίνουν διάφορους γεννήτορες γράφων κλήσεων συναρτήσεων, και συγκεκριμένα των Soot [48, 49], WALA [42], DOOP [41], και OPAL [45]. Οι Sui et al. [53], παρουσιάζουν μια ακόμα βιβλιοθήκη από Java benchmarks και την χρησιμοποιούν για να αξιολογήσουν και να συγκρίνουν τα Soot [48, 49], WALA [42], και DOOP [41]. Η παραπάνω βιβλιοθήκες μοιάζουν πολύ μεταξύ τους, πράγμα που ώθησε το Judge στο να τις συμπτύξει σε μια μοναδική βιβλιοθήκη. Υπενθυμίζουμε ότι στα πλαίσια αυτής της εργασίας υλοποιήσαμε μια παρόμοια βιβλιοθήκη μικρής κλίμακας (Κεφάλαιο 5).

Έλεγχος Σφαλμάτων στην Python. Οι Fromherz et al. [54] προτείνουν μια υλοποίηση που μπορεί να εντοπίσει σφάλματα χρόνου εκτέλεσης με ορθό τρόπο αποτιμώντας τους τύπους των μεταβλητών μέσω abstract interpretation. Ωστόσο, η μέθοδος τους βρίσκεται ακόμα υπό ανάπτυξη και δεν υποστηρίζει πολλά χαρακτηριστικά της Python, συμπεριλαμβανομένης της αναδρομής. Συγκριτικά, έχουμε υλοποιήσει ένα πρωτότυπο που μπορεί να διαχειριστεί τα περισσότερα χαρακτηριστικά της Python και να υπολογίσει τις τιμές των literals με ένα τρόπο ανεξάρτητο της ροής. Αυτό μας επιτρέπει να αναγνωρίσουμε αποδοτικά μη έγκυρες προσβάσεις σε dictionaries.

Υπάρχουν διάφορα εργαλεία που έχουν ως στόχο τον έλεγχο σφαλμάτων σε υπό ανάπτυξη προγράμματα Python. Το *pyflakes* [20] λειτουργεί πάνω στο συντακτικό δέντρο του κάθε αρχείο για να εντοπίσει σφάλματα όπως μη ορισμένες μεταβλητές. Το *pylint* [21] είναι μια δημοφιλής μέθοδος που παρέχει ενσωμάτωση με διάφορα IDEs καθώς και continuous integration pipelines. Μπορεί να εντοπίσει πολλά πιθανά σφάλματα συμπεριλαμβανομένων μη ορισμένων μεταβλητών και μη έγκυρων τύπων επιστροφής. Το *PyChecker* [55] εστιάζει στην αναγνώριση σφαλμάτων που συνήθως εντοπίζονται από μεταγλωττιστές για γλώσσες όπως οι C και C++. Ωστόσο, κανένα από αυτά τα εργαλεία δεν εντοπίζει μη έγκυρες προσβάσεις σε dictionaries.

Άλλα εργαλεία εστιάζουν στον εντοπισμό συχνών προβλημάτων ασφαλείας [22, 23], στον έλεγχο τύπων [56, 57, 58], και σε βελτιώσεις κώδικα [59, 60, 24]. Σε σύγκριση με τα προαναφερθέντα εργαλεία, η μέθοδος μας αξιοποιεί μια διασυναρτησιακή ανάλυση και εστιάζει στον εντοπισμό σφαλμάτων κλειδιού.

Κεφάλαιο 7

Συμπεράσματα

Παρουσιάσαμε μια πρακτική μέθοδο στατικής ανάλυσης προγραμμάτων της γλώσσας Python και υλοποιήσαμε ένα πρωτότυπό της το οποίο ονομάζουμε *PyCG*. Η μέθοδος μας πραγματοποιεί μια context-insensitive διασυναρτησιακή ανάλυση η οποία αναγνωρίζει την ροή των τιμών μέσω της κατασκευής ενός γράφου ο οποίος αποθηκεύει όλες τις σχέσεις ανάθεσης μεταξύ των αναγνωριστικών του προγράμματος. Προτείνουμε, δύο εφαρμογές που αξιοποιούν την ανάλυση μας, συγκεκριμένα, την παραγωγή γράφων κλήσεων συναρτήσεων και τον εντοπισμό σφαλμάτων κλειδιού. Στα πλαίσια της πρώτης εφαρμογής, διαπιστώσαμε ότι το *PyCG* σημειώνει υψηλές επιδόσεις τόσο όσον αφορά το precision όσο και το recall. Στη δεύτερη εφαρμογή, δείξαμε ότι η προσέγγιση μας μπορεί να εντοπίσει αποδοτικά σφάλματα κλειδιού τα οποία εκδηλώνονται σε μια ποικιλία περιπτώσεων.

Στο μέλλον, έχουμε σκοπό να επεκτείνουμε την ανάλυση μας ώστε να υποστηρίζει περισσότερα χαρακτηριστικά της Python καθώς και να περιέχει μια μοντελοποίηση των built-in μεθόδων, τύπων, και των επιδράσεων τους. Επιπλέον, θα αναλύσουμε τον πηγαίο κώδικα πακέτων απο τρίτους μέσω της προσθήκης ευρετικών τεχνικών που θα ανιχνεύουν την τοποθεσία του πηγαίου κώδικα τους.

Κείμενο στα αγγλικά

Chapter 1

Introduction

Python is a high-level dynamically typed programming language. It is considered to be one of the most popular languages [1, 2] and supports many different types of programming paradigms including object-oriented and functional programming, being used for many applications ranging from simple scripts to large scale production services. Its popularity has led to the emergence of static analysis tools that aim to help practitioners and researchers to analyze programs written in it. These analyzers are typically used either during the development process or after the software has been published.

The static analysis of programs written in high-level, dynamic programming languages can be a complex endeavor. Specifically, to perform static analysis on programs written in languages such as Python and JavaScript, one must deal with several challenges including higher-order functions, dynamic and metaprogramming features (e.g., `eval`), and modules. Addressing such challenges can play a significant role in the improvement of dependency impact analysis [3, 4, 5], especially in the context of package managers such as *npm* [6] and *pip* [7].

Several research works propose static analysis methods for dynamic languages, with their primary aim being completeness, i.e., facts deduced by the system are indeed true [8, 9, 10]. However, for dynamic languages, completeness comes with a performance cost. Thus, these approaches are rarely employed in practice due to scalability concerns [11]. To counteract these issues, researchers have turned to *practical* approaches focusing on incomplete static analysis for achieving better performance [12, 13]. This performance boost is the key enabler for adopting these approaches in applications that interact with complex libraries [12], or Integrated Development Environments (IDEs) [13]. Existing work is primarily targeted to JavaScript programs and attempts to address challenges related to events and the language's asynchronous nature [14, 15].

We propose a practical approach for statically analyzing Python programs and implement a corresponding prototype that we call *PyCG*. We design a context-insensitive inter-procedural analysis that operates on a simple intermediate representation targeted for Python. Our approach computes the *assignment graph*, a structure that shows the assignment relations among program identifiers. In contrast with existing static analyzers, our analysis is able to handle complex Python features, such as higher-order functions, function closures, multiple inheritance, and modules.

Our analysis follows a conservative approach and does not deal with non-functional values. This means that our analysis does not reason about loops and conditionals. Furthermore, we distinguish attribute accesses (i.e, $e.x$) based on the namespace where the attribute (x) is defined. This approach boosts precision, especially when dealing with features like modules, inheritance, and programming patterns such as duck typing [16]. Prior work follows a *field-based* approach that correlates attributes of the same name with a single global location without taking into account their namespace [13], leading to false positives. Our design choices make our approach achieve high rates of precision, while remaining efficient and applicable to large-scale Python programs.

Our analysis has been designed to be scalable and easily extendable. Specifically, the internal domains of the analysis can serve as the basis for a wide range of applications that aim to perform a practical static analysis. To that end, we have extended *PyCG* to support two practical applications, namely call graph generation and the identification of invalid dictionary accesses (key errors).

A call graph depicts calling relationships between subroutines in a computer program. Call graphs can be employed to perform a variety of tasks, such as profiling [17], vulnerability propagation [18],

and tool-supported refactoring [19]. Our analysis builds the call graph of the original Python program using the assignment graph. Specifically, we utilize the graph to resolve all functions that can be potentially pointed to by callee variables. Such a programming pattern is particularly common in higher-order programming.

Key errors arise when there is an access to a dictionary key that does not exist. When this happens, Python raises an exception and exits. Unidentified key errors can have catastrophic results for companies that deploy software written in Python. This leads to a need for methods that can identify potential key errors during the development process. There is already a plethora of tools implementing such methods [20, 21, 22, 23, 24]. These tools work directly from the command line and can also be integrated into IDEs and continuous delivery pipelines. However, none of these approaches can identify invalid dictionary accesses. To fill this gap, we use the assignment graph and a structure that stores the literals that a variable has been assigned to. Then, we check whether the dictionary accessor's value corresponds to a key of the dictionary. If that is not the case our analysis identifies a key error.

We evaluate the effectiveness of our method through corresponding micro- and macro-benchmark suites. In the case of call graphs, we compare our approach against two available Python call graph generators, namely *Pyan* and *Depends*. The results indicate that our approach achieves high levels of precision ($\sim 99.2\%$) and adequate recall ($\sim 69.9\%$) on average, while the other analyzers demonstrate lower rates in both measures. Regarding key errors, we show that our method can efficiently identify key errors on multiple scenarios that arise from Python's functionality. Furthermore, we use our key error identification mechanism on real world Python programs. The results show that our method can efficiently find potential key errors during the development process, while maintaining a low number of false positives.

1.1 Outline

The rest of the work is structured as follows. In Chapter 2 we outline the challenges for statically analyzing Python source code and discuss the limitations of existing call graph generators. Chapter 3 presents the methodology of our analysis. Specifically, we introduce an intermediate representation targeted for Python programs and the domains of our analysis. Then, we go into detail about how we traverse this intermediate representation using simple rules that we have created, and finally present algorithms for call graph generation and key error identification. Next, in Chapter 4 we mention our implementation decisions and describe our tool's interface. Our evaluation is presented in Chapter 5, which contains details about our experiment setup, our micro- and macro-benchmarks, and the corresponding results. Related work is presented in Chapter 6. We conclude with a summary of our work on Chapter 7.

Chapter 2

Background

2.1 Challenges for Python

The analysis of Python source code presents the following challenges.

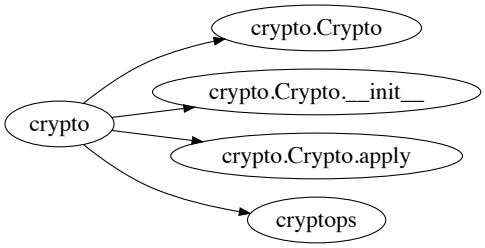
- *Higher-order Functions*: In a high-level language such as Python, a function can be assigned to a variable, passed as an argument to another function, or even serve as a return value.
- *Nested Definitions*: In Python, function and class definitions can be nested. This means that functions and classes can be defined and invoked within the context of other functions or classes.
- *Classes*: Python is an object-oriented language and allows for the creation of complex inheritance and composition schemes. The resolution of inherited methods from parent classes requires the computation of the Method Resolution Order (MRO) of each class.
- *Modules*: In a typical use case, a Python module imports many other modules. Furthermore, an import may be resolved in a different way for separate systems. Keeping track of the different imports of a module as well as their resolution can be a challenging task.
- *Dynamic Features*: In Python, variables may take different values during a program's execution and these values can have different types. Furthermore, a class can be dynamically modified during runtime.
- *Duck Typing*: Duck typing is a programming pattern common in dynamic languages such as Python [16]. Through this pattern, the presence of specific methods and properties determine the suitability of an object, instead of the type of the object itself. Duck typing makes identifying the origin of an invoked method that is defined by two (or more) classes in a given context a complex operation.

2.2 Limitations of Existing Call Graph Generators

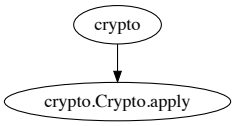
Despite Python's popularity [25], it is surprising that there are only few tools that aim to generate call graphs for programs written in the language. *Pyan* [26] traverses the program's Abstract Syntax

```
1  import cryptops
2
3  class Crypto:
4      def __init__(self, key):
5          self.key = key
6
7      def apply(self, msg, func):
8          return func(self.key, msg)
9
10 crp = Crypto("secretkey")
11 encrypted = crp.apply("hello world", cryptops.encrypt)
12 decrypted = crp.apply(encrypted, cryptops.decrypt)
```

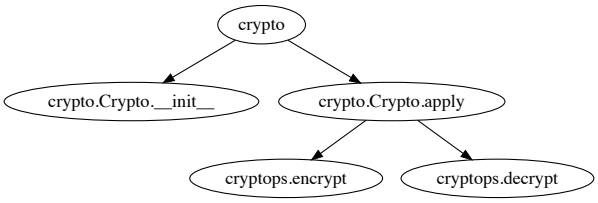
Figure 2.1: The `crypto` module. Existing tools fail to generate a corresponding call graph effectively.



(a) Pyan-generated call graph.



(b) *Depends*-generated call graph.



(c) Precise call graph.

Figure 2.2: Call graphs for the `crypto` module.

Tree (AST) to generate its call graph. However, it has issues in the way it handles module imports and the inter-procedural flow of values. *code2graph* [27, 28] has the same limitations, since it uses *Pyan*-generated call graphs to draw their visualizations. *Depends* [29] generates call graphs by inferring syntactical relations among source code entities. Nevertheless, it has drawbacks related to higher-order functions.

In this section, we discuss the limitations of the two existing analyzers in terms of efficiency and practicality. We do so by introducing a Python module named `crypto` (Figure 2.1), which implements the encryption and decryption of a "hello world" message. Initially, it imports an external Python module named `cryptops`, that defines the `encrypt(key, msg)` and `decrypt(key, msg)` functions. Afterwards, it defines the `Crypto` class, which is instantiated with an encryption key. The instance of the class can be then used to encrypt or decrypt messages by calling the `apply(self, msg, func)` method, in which `func` is one of `cryptops.encrypt(key, msg)` and `cryptops.decrypt(key, msg)`. Figure 2.2c shows the call graph of the module.

Pyan [26] produces the imprecise call graph shown in Figure 2.2a. *Pyan* does not track the inter-procedural flow of values, and therefore it is unable to infer which functions are passed as arguments to `apply(self, msg, func)` which leads to missing call edges. Furthermore, the graph includes false positives. Specifically, whenever an object is instantiated, *Pyan* adds call edges to both the class name and the `__init__()` method of the class. In addition, in the case of module imports, *Pyan* generates call edges with the source being the importing namespace and the destination being the module name.

Depends produces the call graph presented in Figure 2.2b. Initially, *Depends* generated an empty call graph since it does not track function calls originating from the module's namespace (e.g., `crypto.apply()`). To get a result, we wrapped those function calls within a new function. The resulting call graph misses most of the calls that are included in the `crypto` module. This happens because *Depends* does not capture calls to the `__init__()` function of a class. In addition, *Depends* does not track the inter-procedural flow of functions and does not infer which functions are passed as parameters to `apply(self, msg, func)` which leads to missing call edges to the parameter functions. In comparison with *Pyan*, *Depends* follows a more conservative approach. That is, it only includes a call edge when it has all the necessary information it needs to anticipate that the call will be realized. Contrary to *Pyan*, this can lead to a call graph without false positives.

Chapter 3

Analysis

Our approach uses a context-insensitive inter-procedural analysis that operates on an intermediate representation of the input Python program. The analysis employs a fixed-point iteration algorithm, and gradually builds the *assignment graph*, which shows the assignment relations between program identifiers (Section 3.1). The assignment graph is an essential component that we use for resolving the functions and literals that a variable can be assigned to. After the analysis terminates, we build the call graph and identify key errors by exploiting the assignment graph stemming from the analysis step (Section 3.2,Section 3.3).

3.1 The Core Analysis

Our approach initially computes the assignment graph using an inter-procedural analysis that works on an intermediate representation targeted for Python programs.

A key element of our analysis is the examination of attribute accesses based on the namespace in which every attribute is defined. As an example, consider the following code snippet:

```
1  class A:
2    def func():
3      pass
4
5  class B:
6    def func():
7      pass
8
9  a = A()
10 b = B()
11 a.func()
12 b.func()
```

Our analysis can distinguish the two functions defined at lines 2 and 6, since they belong to two distinct classes. Field-based approaches for JavaScript [13] will fail to differentiate between the two invocations, which causes imprecision. This happens because field-based approaches map all identically-named attribute accesses to the same object.

3.1.1 Syntax

The intermediate representation on which our analysis operates has the syntax of a simple imperative, object-oriented language. This syntax is shown in Figure 3.1. The evaluation contexts [30] for this language, to be explained shortly, are shown on the last rule of this figure.

Identifiers are an important element of this language. Each can be one of the following types: (1) **func** that corresponds to the name of a function (2) **var** that indicates the name of a variable, (3) **cls** for class names, and (4) **mod** when the identifier is a module name. Each pair $(x, \tau) \in \mathit{Identifier} \times \mathit{IdentType}$ forms a definition. An object is represented as a definition along with its namespace (see the *Obj* rule). A namespace is composed as a sequence of definitions and is essential for distinguishing

$$
\begin{array}{rcl}
\langle e \in \mathit{Expr} \rangle & ::= & o \mid x \mid f \mid x := e \mid \textbf{function}\ x\ (y\ldots)\ e \mid \textbf{return}\ e \mid \\
& & e(x{=}e\ldots) \mid \textbf{class}\ x\ (y\ldots)\ e \mid e.x \mid e.x := e \mid \\
& & e[x] \mid e[x] := e \mid \textbf{new}\ x\ (y = e\ldots) \mid \\
& & \textbf{import}\ x\ \textbf{from}\ m\ \textbf{as}\ y \mid \textbf{iter}\ x \mid e;e \\
\langle o \in \mathit{Obj} \rangle & ::= & n, v \\
\langle v \in \mathit{Definition} \rangle & ::= & x, \tau \\
\langle \tau \in \mathit{IdentType} \rangle & ::= & \textbf{func} \mid \textbf{var} \mid \textbf{cls} \mid \textbf{mod} \\
\langle n \in \mathit{Namespace} \rangle & ::= & (v)^* \\
\langle x, y \in \mathit{Identifier} \rangle & ::= & \textit{is the set of program identifiers} \\
\langle m \in \mathit{Modules} \rangle & ::= & \textit{is the set of modules} \\
\langle f \in \mathit{Literal} \rangle & ::= & \textit{is the set of literals} \\
\langle E \rangle & ::= & [] \mid x := E \mid \textbf{return}\ E \mid E(x = e\ldots) \mid \\
& & o(x = E\ldots) \mid \textbf{new}\ x(y{=}E) \mid E.x \mid E.x := e \mid \\
& & o.x := E \mid o[x] \mid o[x] := E \mid \textbf{iter}\ o \mid E;e \mid o;E
\end{array}
$$

Figure 3.1: The syntax that represents the input Python programs and the evaluation contexts.

objects that share the same identifier from one another. As an example, consider the following Python code snippet located in a module named `main`.

```
1  var = 10
2  class A:
3    var = 10
```

Our analysis can differentiate between the objects defined on lines 1 and 3, as the first one is defined in the namespace $[(\texttt{main}, \textbf{mod})]$, while the second one resides in the namespace $[(\texttt{main}, \textbf{mod}), (\texttt{A}, \textbf{cls})]$.

Our approach treats every object as the *value* that stems from the evaluation of the expressions supported by the language. Specifically, our representation contains expressions capturing the inter-procedural flow, assignment statements, class and function definitions, module imports, and iterators / generators (see the *Expr* rule). The language can abstract different features, including lambda expressions, keyword arguments, constructors, multiple inheritance, and more.

As with prior work that focuses on JavaScript [15, 31, 14], we use evaluation contexts [30] that describe in which order sub-expressions are evaluated. As an example, for an attribute assignment $E.x := e$, the E symbol indicates that the receiver of the attribute x is currently being evaluated, and $o.x := E$ denotes that the receiver has been already evaluated to an object $o \in \mathit{Obj}$ (recall that evaluating expressions results in objects), and then the evaluation will proceed to the right-hand side of the assignment.

Remarks. When Python functions that result in a generator are called (i.e., they include a `yield` statement instead of `return`), the calls take place only when the generator is actually used. When our analysis encounters these lazy calls (e.g., `gen = lazy_call(x)`), it models them by creating a thunk (e.g., `gen = lambda: lazy_call(x)`) that is evaluated only when the generator is iterated (through the **iter** construct).

3.1.2 State

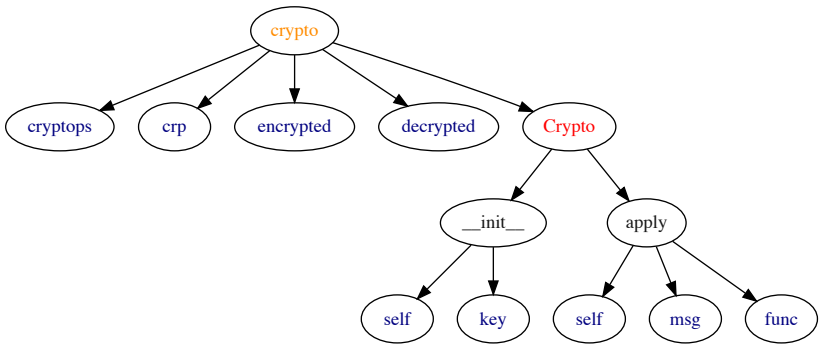
Following the conversion of the original Python program to our intermediate representation, our analysis begins evaluating each expression, and gradually builds the assignment graph. The analysis does that by maintaining a state consisting of four domains as shown in Figure 3.2, namely, *scope*, *class hierarchy*, *assignment graph*, *object literals map*, and *current namespace*.

The scope is a map of definitions to a set of definitions. In essence, a scope is a tree in which each node corresponds to a definition (e.g., a function), and each edge denotes the parent/child relations

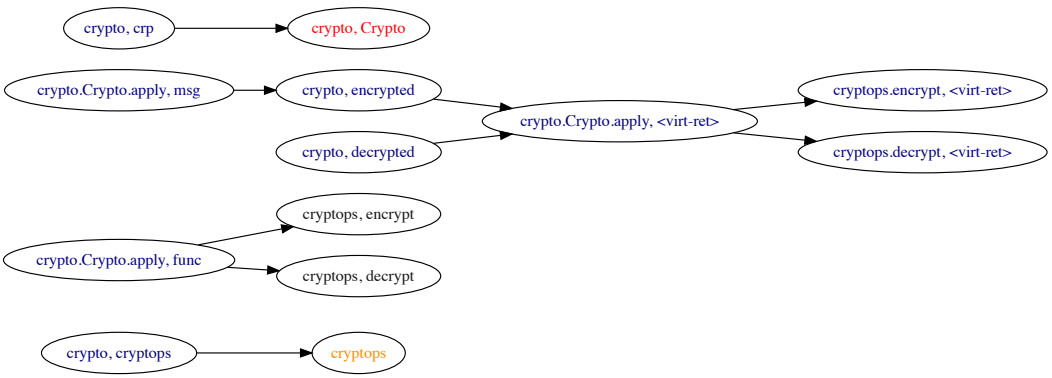
$$\pi \in AssignG = Obj \hookrightarrow \mathcal{P}(Obj)$$
$$s \in Scope = Definition \hookrightarrow \mathcal{P}(Definition)$$
$$h \in ClassHier = Obj \hookrightarrow Obj^*$$
$$l \in ObjLit = Obj \hookrightarrow \mathcal{P}(Literal)$$
$$\sigma \in State = AssignG \times Scope \times Namespace \times ClassHier \times ObjLit$$

Figure 3.2: Domains of the analysis.

between definitions, i.e., the target node is defined inside the definition of the source node. The domain of scopes helps to correctly resolve the definitions that are visible inside a specific namespace. Figure 3.3a shows the scope tree of the program that is depicted in Figure 2.1, and illustrates all program definitions and their inter-relations. Red nodes correspond to class definitions, orange nodes are module definitions, black nodes denote functions, while blue nodes indicate variables. Based on this scope tree, we infer that the function `apply` is defined inside the class `Crypto`, which is in turn defined inside the module `crypto`, i.e., notice the path `crypto` → `Crypto` → `apply` → `func`. This domain helps us to properly deal with Python features such as function closures and nested definitions.



(a) The scope tree of the `crypto` module.



(b) The assignment graph of the `crypto` module.

Figure 3.3: Analyzing the `crypto` module.

The class hierarchy is a tree that represents the inheritance relations among classes. An edge from node u to node v indicates that the class v is the parent of the class u. This domain is used by

our analysis for resolving class attributes (either methods or fields) defined in the base classes of the receiver object. Through this domain our analysis is able to handle the object-oriented features of Python, addressing multiple inheritance and the method resolution order.

The assignment graph is defined as a map of objects to an element of the power set of objects $\mathcal{P}(Obj)$. This graph contains the assignment relations between objects, and captures the assignments and the inter-procedural flow of the program. Figure 3.3b illustrates the assignment graph that corresponds to the program of Figure 2.1. Every node in the graph (e.g., `{crypto.Crypto.apply, func}`) represents an object. Each node label contains two components. The first (e.g., `crypto.Crypto.apply`) denotes the namespace where an identifier (e.g., `func`) is defined. Colors correspond to the type of the identifier as explained in a previous paragraph (e.g., the black color implies function definitions). An edge denotes the possible values that a variable may hold. As an example, the variable `func` defined in the `crypto.Crypto.apply` namespace may point to the functions `decrypt` and `encrypt`, which are both defined in the `cryptops` namespace. As another example, notice the edge originating from the node `{crypto.Crypto.apply, msg}` and leading to `{crypto, encrypted}`. This edge shows that the parameter `msg` of the function `crypto.Crypto.apply` points to the variable `encrypted` when the function is invoked on line 12. The assignment graph domain enables us to address the challenge regarding higher-order programming in Python.

The object literals map defines a relationship between objects and a set of literals. Specifically, for each object we maintain a set of literals that the object can point to during the program's execution. This domain is useful for efficiently resolving dictionary accesses. For example, consider the access `E[o]`. Through the object literals map we can identify the potential values that the object o may point to, and therefore resolve the dictionary entry that this access evaluates to.

Finally, the current namespace is used to track the location where new variables, classes, modules, and functions are declared. This domain allows us to establish a more precise analysis than field-based analysis employed by prior work. Through namespaces, objects and attribute accesses are differentiated based on their namespace, addressing challenges such as duck typing.

3.1.3 Analysis Rules

The analysis examines each expression that is contained in the intermediate representation of the initial program, and transitions the analysis state based on the semantics of each expression. The algorithm repeats this process until the state converges, and the assignment graph is given by the final state of the analysis.

In Figure 3.4 we show the state transition rules of our analysis. The rules follow the form:

$$\langle \pi, s, n, h, l, E[e] \rangle \rightarrow \langle \pi', s', n', h', l', E[e'] \rangle$$

In the following, we describe each rule in detail.

According to the [e-ctx] rule, for an expression e in the evaluation context E, an assignment graph π, a namespace n, a class hierarchy h, an object literals map l, and a scope s, an expression e' in the evaluation context E can be retrieved, if the initial expression e evaluates to e'. For the following, the binary operation $x \cdot y$ represents appending the element y to the list x.

The [compound] rule indicates that in the case of a compound expression that consists of two objects o_1, o_2, we return the last object o_2 as the result of the evaluation. Note that the evaluation of the compound expression requires every sub-term to have been evaluated to an object according to the evaluation contexts shown in Figure 3.1. The rest of the rules also follow this behavior.

The [ident] rule describes the case when the initial expression is an identifier x. The analysis retrieves the object o corresponding to the identifier x, in the namespace n, based on the scope tree s. To do so, the analysis uses the function `getObject`(s, n, x), which iterates every element y of the namespace n in the reverse order. Afterwards, it examines the scope tree s, to check if the element node y has any child that matches the identifier x. In the case that no such match exists, the function

e-ctx

$$\frac{\langle \pi, s, n, h, l, e\rangle \hookrightarrow \langle \pi', s', n', h', l', e'\rangle}{\langle \pi, s, n, h, l, E[e]\rangle \to \langle \pi', s', n', h', l', E[e']\rangle}$$

compound

$$\frac{}{\langle \pi, s, n, h, l, E[o_1; o_2]\rangle \to \langle \pi, s, n, h, l, E[o_2]\rangle}$$

ident

$$\frac{o = \texttt{getObject}(s, n, x)}{\langle \pi, s, n, h, l, E[x]\rangle \to \langle \pi, s, n, h, l, E[o]\rangle}$$

assign

$$\frac{s' = \texttt{addScope}(s, n, x, \textbf{var}) \qquad o' = (n, (x, \textbf{var})) \qquad \pi' = \pi[o' \to \pi(o') \cup \{o\}]}{\langle \pi, s, n, h, l, E[x := o]\rangle \to \langle \pi', s', n, h, l, E[o']\rangle}$$

assign-lit

$$\frac{s' = \texttt{addScope}(s, n, x, \textbf{var}) \qquad o' = (n, (x, \textbf{var})) \qquad l' = l[o' \to l(o') \cup \{f\}]}{\langle \pi, s, n, h, l, E[x := f]\rangle \to \langle \pi, s', n, h, l', E[o']\rangle}$$

func

$$\frac{\begin{array}{c} s' = \texttt{addScope}(s, n, x, \textbf{func}) \\ n' = n \cdot (x, \textbf{func}) \qquad s'' = \texttt{addScope}(s', n', \texttt{ret}, \textbf{var}) \qquad s^{(3)} = \texttt{addScope}(s'', n', y, \textbf{var}) \end{array}}{\langle \pi, s, n, h, l, E[\textbf{function}\ x\ (y \dots)\ e]\rangle \to \langle \pi, s^{(3)}, n', h, l, E[e]\rangle}$$

return

$$\frac{o' = (n \cdot x, (\texttt{ret}, \textbf{var})) \qquad \pi' = \pi[o' \to \pi(o') \cup \{o\}]}{\langle \pi, s, n \cdot x, h, l, E[\textbf{return}\ o]\rangle \to \langle \pi', s, n, h, l, E[o']\rangle}$$

call

$$\frac{\begin{array}{c} o_1 = (n', (fn, \textbf{func})) \qquad o_2' = (n' \cdot fn, (y, \textbf{var})) \\ o_3 = (n' \cdot fn, (x, \textbf{var})) \qquad \pi' = \pi[o_2' \to \pi(o_2') \cup \{o_2\}] \qquad l' = l[o_3 \to l(o_3) \cup \{f\}] \end{array}}{\langle \pi, s, n, h, l, E[o_1(x = f, y = o_2, \dots)]\rangle \to \langle \pi', s, n, h, l, (n' \cdot fn, (\texttt{ret}, \textbf{var}))\rangle}$$

class

$$\frac{\begin{array}{c} s' = \texttt{addScope}(s, n, x, \textbf{cls}) \\ t = \langle \texttt{getObject(s, n, b)} \mid b \in (y \dots)\rangle \qquad h' = h[(n, (x, \textbf{cls})) \to t] \qquad n' = n \cdot (x, \textbf{cls}) \end{array}}{\langle \pi, s, n, h, l, E[\textbf{class}\ x\ (y \dots)\ e]\rangle \to \langle \pi, s', n', h', l, E[e]\rangle}$$

attr

$$\frac{o' = \texttt{getClassAttrObject}(o, x, h)}{\langle \pi, s, n, h, l, E[o.x]\rangle \to \langle \pi, s, c, h, l, E[o']\rangle}$$

new

$$\frac{o_3 = \texttt{getObject}(s, n, x) \qquad o_2 = \texttt{getClassAttrObject}(o_3, \texttt{\_\_init\_\_}, h)}{\langle \pi, s, n, h, l, E[\textbf{new}\ x(y = o_1 \dots)]\rangle \to \langle \pi, s, n, h, l, E[o_2(y = o_1 \dots)]; o_3]\rangle}$$

attr-assign

$$\frac{o_3 = \texttt{getClassAttrObject}(o_1, x, h) \qquad \pi' = \pi[o_3 \to \pi(o_3) \cup \{o_2\}]}{\langle \pi, s, n, h, l, E[o_1.x := o_2]\rangle \to \langle \pi', s, n, h, l, E[o_3]\rangle}$$

import

$$\frac{\begin{array}{c} o_2 = \texttt{getObject(s, m, x)} \\ s' = \texttt{addScope}(s, n, y, \textbf{var}) \qquad o_1 = (n, (y, \textbf{var})) \qquad \pi' = \pi[o_1 \to \pi(o_1) \cup \{o_2\}] \end{array}}{\langle \pi, s, n, h, l, E[\textbf{import}\ x\ \textbf{from}\ m\ \textbf{as}\ y]\rangle \to \langle \pi', s', n, h, l, E[o_1]\rangle}$$

iter-iterable

$$\frac{o' = \texttt{getClassAttrObject}(o, \texttt{\_\_next\_\_}, h)}{\langle \pi, s, n, h, l, E[\textbf{iter}\ o]\rangle \to \langle \pi, s, n, h, l, E[o'()]\rangle}$$

iter-generator

$$\frac{\texttt{getClassAttrObject}(o, \texttt{\_\_next\_\_}, h) = \text{undefined}}{\langle \pi, s, n, h, l, E[\textbf{iter}\ o]\rangle \to \langle \pi, s, n, h, l, E[o()]\rangle}$$

Figure 3.4: Rules of the analysis.

`getObject` continues to the next part of the namespace. Note that this rule does not affect the analysis state.

[assign] assigns the object o to the identifier x. First, the identifier x is added in the namespace n of the scope tree s, using the function `addScope`(s, n, x, τ). This function works by adding an edge originating on the node accessed by the path n and leading to the node given by the definition (x, τ). Afterwards, the assignment graph is updated by adding an edge from the object corresponding to the left-hand side of the assignment (i.e., o') to that of the right-hand side (i.e., o). According to this update, the variable x defined in the namespace n can point to the object o.

In the case that a literal f is on the right-hand side of the assignment, we use the [assign-lit] rule, which works in a similar fashion as the [assign]. The main difference is that here, we update the object literals map l instead of the assignment graph. More precisely, we update the object literals map entry set for the identifier x, so that it contains f.

[func] updates the scope tree. In particular, a new scope s' is created by adding the function x to the current namespace n. Then, a new namespace n' is created by adding the function definition $(x, \textbf{func})$ to the top of the current namespace. Afterwards, it adds all function parameters, and a virtual variable named `ret`—which corresponds to the variable that holds the function's return value— to the new namespace n'. This results in a new scope tree $s^{(3)}$. Finally, the analysis proceeds with the evaluation of the body of the function x in the new namespace n', i.e., note that the rule evaluates to $E[e]$. The new namespace n' correctly identifies that any variable defined in e, is actually defined in the body of the function.

The [return] rule assigns the object o to the virtual variable `ret`, used for storing the return value of a function (recall the [func] rule). To accomplish this, the analysis updates the assignment graph by adding a new edge from the object o' that corresponds to the return variable `ret` to the object o which is the operand of **return**. Afterwards, the rule evaluates to the object o'. A similar approach is followed in the case that a literal is returned. Specifically, we update the objects literal map instead of the assignment graph in a similar manner as with the [assign-lit] rule. This case is omitted for brevity.

The inter-procedural flow is captured by the [call] rule. Specifically, when the analysis encounters a call expression $o_1(x = f, y = o_2, \dots)$, it examines the callee object o_1 associated with a function `fn` defined in a namespace n'. Then, each parameter of `fn` is connected with the appropriate argument that was passed during the function's invocation (e.g., the parameter y is assigned to o_2), leading to a new assignment graph π' and a new object literals map l'. For example, consider again the graph of Figure 3.3b. The edges originating at the {`crypto.Crypto.apply, func`} node are created by this rule. These edges denote that the parameter `func` of the `crypto.Crypto.apply` function may hold the functions `cryptops.encrypt` and `cryptops.decrypt` which were passed when `crypto.Crypto.apply` was called (Figure 2.1).

[class] handles class definitions. This rule initially adds the class x to the scope tree through the function `addScope`, and then retrieves every object related to the base classes of x (i.e., $y \dots$). To accomplish this, the rule consults the scope tree in the namespace n, and retrieves a sequence of objects t that follows the order in which base classes are defined upon class definition. We later explain why keeping the registration order of base classes is important. Then, this rule updates the class hierarchy so that the newly-defined class x is a child of the base classes pointed to by the identifiers $(y \dots)$. Next, the analysis proceeds on the body of the class x in a new namespace n'. The new namespace includes the class definition to the top of the current namespace (i.e., $n \cdot (x, \textbf{cls})$). Finally, the analysis starts examining the class' body using the new namespace.

The [attr] rule follows a similar approach with the [ident] rule. However, in this case, in order to correctly identify the object corresponding to the attribute x of the receiver object o, the analysis examines the hierarchy of classes h using the function `getClassAttrObject`(o, x, h). At this point our analysis is able to differentiate attributes according to the location (i.e., o) in which they are defined.

The function `getClassAttrObject` deals with multiple inheritance, by respecting the method resolution order implemented in Python. As an example, consider the following code snippet.

```
class A:
  def func():
    pass

class B:
  def func():
    pass

class C(B, A):
  pass

c = C()
c.func()
```

In the example above, the method resolution order is $C \rightarrow B \rightarrow A$, because the class `B` is the first parent class of `C`, while `A` is the second one. As a result, `c.func()` leads to the invocation of function `func` defined in class `B`, since it is the first matching function with the name `func` in the method resolution order. The correct resolution of class members explains why the domain of the class hierarchy maps every object to a *sequence* of objects rather than a set—we need to track the order in which the parents of a class are registered.

The [new] rule handles object initialization. This rule retrieves the object o_3 that is associated with the definition of the class x. Through the `getClassAttrObject` function, the rule inspects the method resolution order of the object o_3 to identify the first object o_2 that matches the function `__init__`. Recall that this function is called on the creation of a new object. Note how the **new** evaluates; it reduces to $o_2(y = o_1); o_3$. That is, the constructor of the class is first called with the same arguments passed as in the initial expression (i.e., $o_2(y = o_1)$), and then the object o_3 is returned, which corresponds to the class definition, which is eventually the result of the **new** expression.

The rule [attr-assign] handles attribute assignments of the form $o_1.x := o_2$. Specifically, it describes the case when the attribute x is defined on some location inside the class hierarchy of the receiver object o_1. In this case, `getClassAttrObject` retrieves the object o_3 that is associated with this attribute, and the rule updates the assignment graph so that o_3 points to o_2 from the right-hand side of the assignment. In the case that the attribute is not defined in the class hierarchy, (i.e., `getClassAttrObject` returns $\perp$) the attribute assignment works in a similar manner with [assign]. First, we add the attribute x to the current scope using `addScope`, and then the graph is updated. If a literal is assigned instead of an identifier, our approach does the same, but updates the object literals map instead of the assignment graph.

When the analysis encounters an **import** x **from** m **as** y expression, it retrieves the object o_2 corresponding to the imported identifier x, which is defined in the module m. Afterwards, it creates an alias y for x. It does so by adding y to the scope tree of the current namespace, and then updating the assignment graph by adding an edge from the object of y to that of x. Through this rule, we are able to deal with Python's module system.

Consuming iterables and generators is enabled through the **iter** x expression. When the identifier x points to an iterable, (i.e., the object that x points to has an attribute named `__next__`), we retrieve the object o' corresponding to `__next__`. Then, the **iter** expression evaluates to a call of $o'()$ (see the [iter-iterable] rule). Otherwise, x is treated as a generator ([iter-generator]), and **iter** reduces to a call of $x()$. Recall from Section 3.1.1 that generators are modeled as thunks. This scenario describes the evaluation of these thunks (generators) when they are actually iterated.

Analysis termination. The analysis traverses expressions, changing the analysis state according to the rules of Figure 3.4, until the state converges. The analysis is guaranteed to terminate, because the domains are finite. Even for the domains of class hierarchy $h \in ClassHier$ and the object literals map $l \in ObjLit$ (Figure 3.2), which are theoretically infinite, the analysis eventually terminates. This happens because a Python program cannot have an unbounded number of classes. On top of that, entries are added to the object literals map only for statements assigning (or returning) a literal. Literal results of constructs such as generators, which could lead to infinite literals, are ignored.

Algorithm 3.1: Call Graph Construction

Input : $p \in Program$
$\sigma \in State$
Output: $cg \in CallGraph$

```
1  foreach e in Program do
2      while e ∉ Obj do
3          ⟨σ, E[e]⟩ → ⟨σ′, E[e′]⟩
4          if e′ = o1(y = o2 ...) then // Call Expression
5              (π, s, n · f, h, l) ← σ′
6              c ← getReachableFuns(π, o1)
7              o3 ← getObject(s, n, f)
8              cg ← cg[f → cg(f) ∪ c] // Add Call Edges
9          end
10         e ← e′
11     end
12 end
13 return cg
```

3.2 Call Graph Construction

After the analysis terminates, we build the call graph by passing the intermediate representation of the given Python program one more time. Algorithm 3.1 describes the details of this pass. The algorithm takes two items as an input: (1) a program $p \in Program$ of the model language whose syntax is shown in Figure 3.1, and (2) the final state $\sigma \in State$ that stems from the analysis step. The algorithm produces a call graph:

$$cg \in CallGraph = Obj \hookrightarrow \mathcal{P}(Obj)$$

The graph contains only objects associated with functions. An element $o \in Obj$ mapped to a set of objects $t \in \mathcal{P}(Obj)$ means that the function o *may* call any function included in t.

The algorithm works by inspecting every expression e identified in the program (line 1), and each expression e is evaluated based on the state transition rules that are described in Figure 3.4. The algorithm iterates over the state transition rules, until e eventually reduces to an object (lines 2, 3). Each time e reduces to a call expression of the form $o_1(y = o_2 \dots)$ (line 4), the algorithm retrieves the namespace where this invocation happened and gets the top element of that namespace (see $n \cdot f$, line 5). Afterwards, the algorithm retrieves all functions that the callee object o_1 may point to. It does so, by consulting the assignment graph through the function `getReachableFuns(`π, o_1`)`, which implements a Depth-First Search (DFS) algorithm and retrieves the set of functions c that can be reached from the source node o_1. In turn, the call graph *cg* is updated by adding all edges from the top element of the current namespace to the set of the callee functions c (lines 7, 8). In other words, the object o_3 (line 7) that represents the top-level element of the namespace, where the call happens, is actually the caller of the functions pointed to by the object o_1.

3.3 Key Errors Identification

Our analysis identifies potential key errors by passing the intermediate representation one more time after the analysis terminates. Algorithm 3.2 outlines the steps of this pass. It takes as an input a program in our model language (see Figure 3.1) and the final state stemming from the analysis step. The algorithm's output is a list which contains the locations of potential key errors:

$$keyErr \in KeyErrorList = o \in Object$$

The algorithm initially inspects all the expressions identified in the program (line 1), until they are reduced to a dictionary access (lines 2-4). In that case, the algorithm first retrieves the potential dictionaries that the dictionary (i.e. o_1) can be assigned to (line 6). It does so with the use of the

Algorithm 3.2: Key Error Identification

```
   Input  : p ∈ Program
            σ ∈ State
   Output: cg ∈ KeyErrorList
1  foreach e in Program do
2      while e ∉ Obj do
3          ⟨σ, E[e]⟩ → ⟨σ′, E[e′]⟩
4          if e′ = o1[o2] then // Dictionary Access
5              (π, s, n, h, l) ← σ′
6              Dicts ← getReachableDicts(π, o1)
7              Vals ← getReachableLits(π, l, o2)
8              foreach d in Dicts do
9                  foreach v in Vals do
10                     if d · v ∉ π then // Dictionary entry does not exist
11                         KeyErrsList ← KeyErrsList ∪ [d · v]
12                     end
13                 end
14             end
15         end
16         e ← e′
17     end
18 end
19 return cg
```

`getReachableDicts` function, which traverses the assignment graph using a Depth-First Search operation. Afterwards, it uses the `getReachableLits` method to retrieve the potential literals that the dictionary accessor (i.e. o_2) can take as values (line 7). This method, first traverses the assignment graph to identify the objects that the accessor has been assigned to and then consults the objects literal map to retrieve their potential literal values. Finally, our algorithm examines all dictionary-literal combinations (lines 8,9) and checks whether such a combination exists as a key in the assignment graph (line 10). Recall, that the assignment graph contains all the objects of the Python program as keys, including dictionary entries. In the case that such a combination is not contained in the assignment graph our algorithm considers that a key error and adds the relevant information in the output list (line 11).

3.4 Discussion & Limitations

A major design decision is to ignore conditionals and loops. For instance, when our analysis comes across an `if` statement, it over-approximates the program's behavior and considers both branches of the statement. This design choice boosts efficiency without a high compromise of the analysis precision (as we will discuss in Section 5). Other static analyzers [9, 10, 8] choose to follow a more heavyweight approach and reason about conditionals. These static analyzers, attempt to compute the set of all states that can be reached based on an initial one. However, providing such an initial state that exercises all feasible paths is not straightforward, especially when analyzing libraries.

Python heavily relies on object-oriented features (e.g., dynamic method dispatch), duck typing [16], and modules. Hence, it is important to separate attribute accesses based on the namespace where each attribute is defined. This design choice boosts—in contrast with prior work [13]—the precision of our analysis without sacrificing its scalability.

Furthermore, our analysis does not fully support all of Python's features. We do not deal with code generation schemes, such as calls to the `eval` built-in method. In general, these dynamic constructs hinder the effectiveness of any static analysis, and dynamic approaches are often used as a countermeasure [32, 33]. Additionally, our method has not modeled the functions that correspond to variables'

built-in types. Therefore, attribute calls that depend on a specific built-in type (e.g., `list.append()`) are not resolved. Finally, our analysis can only analyze modules for which their source code has been provided. When a function—for which its code definition is not available—is called, our approach will add an edge to the function, but no edges stemming from that function will ever be added, and its return value will be ignored.

Chapter 4

Implementation

We have developed *PyCG*, a prototype of our approach in Python 3. *PyCG* does not have dependencies third-party packages and only relies on the *symtable* and *ast* modules of the Python standard library. These modules are used to create the scope tree and construct the intermediate representation respectively. Our prototype works as a command line interface (CLI) tool, and has been tested on UNIX environments.

4.1 Usage on Packages

Our prototype efficiently analyzes Python code ranging from single files to large libraries organized as a package. In the case of packages, our prototype can accept a CLI argument that specifies the root directory of the Python package. This argument enables the use of two features. First, the generation of internal objects' namespaces relative to the namespace of the package. Having these relative namespaces is important in the context of tools that aim to combine the call graphs of inter-dependent packages and then perform different kinds of analysis on them (e.g. dependency impact analysis). Second, our prototype implements a methodology that can automatically discover the locations of the package files that an *import* statement refers to. Therefore, for such cases, *PyCG* only requires the location of the package entry-point scripts as an input, and then, it will discover and analyze all imported modules. We go into detail about this mechanism on Section 4.3.

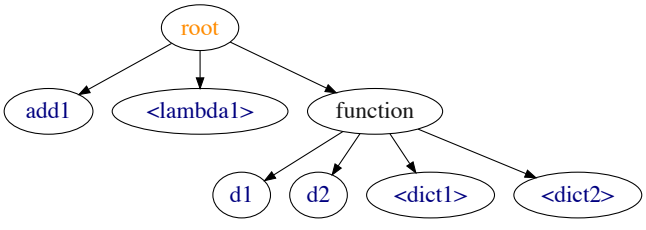
4.2 Internal Representation of State

PyCG's internal representation uses data structures that allow for the efficient retrieval and updates of the items stored by the domains of the analysis. In the following, we describe in detail the internal implementation of each domain.

Namespace. The current namespace is implemented as a stack of identifiers. Elements are pushed and popped from the stack when we enter and exit a scope respectively. The namespace is represented as a string by joining the elements of the stack using a dot (`.`) separator.

```
1  add1 = lambda x: x + 1
2  def function():
3      d1 = {
4          "key1": "val1",
5          "key2": "val2"
6      }
7      d2 = {"key3": "val3"}
```

(a) Python module that defines anonymous elements.



(b) Scope tree of the Python module.

Figure 4.1: Scope tree containing anonymous elements as thunks.

Scope Tree. The scope tree is implemented as a hashmap that maps namespaces to scope entries. Each scope entry corresponds to a specific scope (e.g. the scope of a function) and maintains two items. The first one, is a mapping that is used to directly retrieve the objects that the scope's identifiers refer to. The second, is a set of counters corresponding to the anonymous elements that our analysis supports (i.e. lambdas, dictionaries, and lists), which allows us to differentiate between them. When our analysis encounters these elements it creates a new identifier that is the combination of a thunk and the corresponding counter, and then adds it as a child of the current scope. As an example, consider the code snippet of Figure 4.1a. It implements a module that defines a lambda and a function that initializes two dictionaries. Its corresponding scope tree is shown on Figure 4.1b. In this case, the two dictionaries defined in lines 3 and 4 are differentiated based on the counter corresponding to the anonymous element of dictionaries.

We use the *symtable* built-in Python library to access the compiler's symbol tables of each module and initialize the scope tree. Our process works as follows. First, we access the symbol entries for the root level scope of the module and add all function and class definitions as its children in the tree. Then, for each child, we iterate its symbol entries in a recursive manner. Note that this process does not add variable names and anonymous elements to the tree. These items are accessed during the analysis step.

Class Hierarchy. The class hierarchy domain is implemented as a hashmap mapping namespaces to their class hierarchy sequences which are implemented as lists. We construct the class hierarchy of each class definition by combining the class hierarchies of its parent classes based on Python's Method Resolution Order (MRO)[1].

Object Literals Map. Once again, we implement the object literals map as a map of object namespaces to sets of literals. In the case of a literal assignment, we retrieve the set maintained for the left hand side object and update it to include the literal.

Assignment Graph. We maintain a mapping of object namespaces to assignment graph entries. Each entry includes its type and a set that contains the namespaces that the entry can point to. Recall that available types are `function`, `module`, `variable`, and `class`.

4.3 Identifying the Location of Imported Modules

We use the built-in *importlib* Python library to discover the file locations of imported modules. Note that this library is used internally by Python to resolve import statements. Specifically, when Python identifies an import statement it proceeds with the following process. Initially, the file location of the imported module is identified, and then a *loader* is used to import and execute the module's code. However, Python allows for the implementation of custom loaders that replace the built-in one. We took advantage of this functionality, and implemented a loader that does not execute the imported module's code, but stores its file location in an internal structure that we have implemented and then exits. This structure, which we call the import graph, is implemented as a mapping of module names to a set of file locations. Then, *PyCG* uses the import graph to analyze the code of imported modules. Through this methodology, our approach only requires a list of entry-points to fully analyze a Python package.

4.4 Output Format

Our prototype implements corresponding output formats for the two operations that it performs, namely call graph generation and the identification of key errors.

```
{
  "cryptops.decrypt": [],
  "cryptops": [],
  "crypto.Crypto.apply": [
    "cryptops.encrypt",
    "cryptops.decrypt"
  ],
  "crypto.Crypto.__init__": [],
  "cryptops.encrypt": [],
  "crypto": [
    "crypto.Crypto.__init__",
    "crypto.Crypto.apply"
  ]
}
```

Figure 4.2: Call graph of the `crypto` module in JSON format.

```
1    def func(dct, name):
2        dct[name]
3
4    dct = {"key1": "val1"}
5
6    dct["nokey"]
7    func(dct, "key2")
8    func(dct, "key1")
```

(a) A sample Python program with dictionary accesses.

```
[{
    "filename": "mod.py",
    "lineno": 2,
    "namespace": "mod.<dict1>",
    "key": "key2"
},
{
    "filename": "mod.py",
    "lineno": 8,
    "namespace": "mod.<dict1>",
    "key": "nokey"
}]
```

(b) Key errors in *PyCG*'s format.

Figure 4.3: A sample Python program with key errors in *PyCG*'s output format.

4.4.1 Call Graph Format

PyCG generates call graphs as an adjacency list in JSON format. Each object type that can invoke functions (i.e. module, function, class) has its namespace depicted as a node, and each node maintains a list of the namespaces of the functions that it invoked. Figure 4.2 displays the output call graph for the `crypto` module (Figure 2.1).

4.4.2 Key Errors Format

PyCG outputs the potential key errors that were identified in the input Python program as a list of dictionaries. Each dictionary contains (1) the file path in which the error was identified, (2) the line number, (3) the namespace of the dictionary being accessed, and (4) the access key. As an example, consider the sample Python program of Figure 4.3a. This program defines a function that takes two parameters: a dictionary and an access key. Then, it defines a dictionary with a single key-value pair. On line 6 the sample module accesses the dictionary with an invalid key, which leads to a key error. In addition, on lines 7 and 8 it calls the function defined on line 1 with the dictionary and two different access keys– an invalid and a valid one respectively – serving as parameters. *PyCG* produces the output outlined in Figure 4.3b, which contains the two key errors of the input program.

[1] More details about Python's Method Resolution Order can be found on https://www.python.org/download/releases/2.3/mro/.

Chapter 5

Evaluation

In this chapter we evaluate our call graph generation procedure and the mechnism that we use to identify invalid dictionary accesses. To do so, we use micro- and macro-benchmarks. For call graph generation, our micro-benchmark suite contains 112 minimal Python programs, and the macro-benchmark suite contains five popular real-world Python packages. We evaluate our key error identification mechanism, using a micro-benchmark of 25 minimal Python programs that lead to key errors and a macro-benchmark of 62 student submissions for a postgraduate course related to data analysis. Our experiments were ran on a Debian 9 host that has 16 CPUs and 16 GBs of RAM.

5.1 Call Graphs

5.1.1 Micro-benchmark Suite

We propose a test suite for benchmarking call graph generation in Python. Researchers can use this suite to evaluate and compare their approaches against a common basis. Reif et al. [34] have implemented similar benchmarks for Java, which contain unique call graph test cases that are grouped into different categories.

Our suite covers a wide range of Python's functionality, and is composed of 112 unique and minimal micro-benchmarks. The benchmarks contained in the micro-test suite are designed to have a limited scope and cover specific functionalities (such as decorators and lambdas). They are organized into 16 distinct categories, that range from simple function calls to more complex features such as higher order functions. Each category is composed of a number of tests. Every test includes (1) the source code, (2) its call graph (in JSON format, see 4.4.1), and (3) a description (in Markdown format). For each test case we have implemented only a single execution path (i.e., no conditionals and loops) in order to have a straightforward correspondence to its call graph. Table 5.1 lists the categories along with the number of benchmarks they contain and a corresponding description.

Table 5.2 lists our evaluation results. For every benchmark belonging to a specific category, we check if *PyCG* and *Pyan* generated complete or sound call graphs. Note that we consider a call graph complete when all the edges that it contains actually exist (no false positives), and sound when it contains every call edge that happens during the program's execution (no false negatives).

PyCG produces a complete call graph in almost all cases (111/112). Furthermore, it produces sound call graphs for 103 of the 112 benchmarks. The missing points for soundness are attributed to not fully covered functionalities, e.g. Python's starred assignments.

Pyan produces lower marks for both metrics. Nonetheless, *Pyan* better supports assignments and produces more sound results. To get some insights about *Pyan*'s scores, we performed a qualitative analysis on its generated call graphs. We found that *Pyan* produces incomplete graphs because it creates call edges to class names and their `__init__` methods (see also Section 2.2). In addition, it generates imprecise results because it has no support for some of Python's functionality, ($0/6$ generators and $0/3$ exceptions), does not handle the inter-procedural flow of functions ($0/6$ parameters and $0/4$ returns), does not capture calls to imported functions ($4/14$), and has limited support for classes ($10/22$).

Table 5.1: Call graph micro-benchmark suite categories.

Category	#tests	Description
`parameters`	6	Positional arguments that are functions
`assignments`	4	Assignment of functions to variables
`built-ins`	3	Calls to built in functions and data types
`classes`	22	Class construction, attributes, methods
`decorators`	7	Function decorators
`dicts`	12	Hashmap with values that are functions
`direct calls`	4	Direct call of a returned function (`func()()`)
`exceptions`	3	Exceptions
`functions`	4	Vanilla function calls
`generators`	6	Generators
`imports`	14	Imported modules, functions classes
`kwargs`	3	Keyword arguments that are functions
`lambdas`	5	Lambdas
`lists`	8	Lists with values that are functions
`mro`	7	Method Resolution Order (mro)
`returns`	4	Returns that are functions

The evaluation of *Depends* indicates both its fundamental limitations and strengths. Recall that each benchmark always includes a call coming from the module's namespace. Our results show that *Depends* does not identify such calls, and therefore soundness is never achieved (0/112). In terms of completeness, *Depends* achieves an almost perfect score (110/112) due to its conservative nature—i.e., it adds an edge when it has high confidence that it will be realized.

5.1.2 Macro-benchmarks

We have manually generated call graphs for five popular real-world packages. The packages were collected using the following process. First, we retrieved Python repositories from the GitHub API and sorted them by their number of stars. Then, we downloaded the source code of each repository and counted the number of lines of Python code. We included in our benchmark the first 5 repositories that contained less than 3.5k lines of Python code. Table 5.3 presents the GitHub repositories we chose along with their lines of code, GitHub stars and forks, and a short description.

Currently, there is no available implementation that generates Python call graphs in an effective manner, so we manually inspected the projects and generated their call graphs in JSON format. We selected medium sized projects (less than 3.5k LoC on average), in order to minimize human errors.

We use our macro-benchmark, to examine the three tools in terms of precision and recall. Precision measures the percentage of valid generated calls over the total number of generated calls. Recall measures the percentage of valid generated calls over the total number of calls.

Table 5.4 presents our results. On two cases, *Pyan* crashed during the call graph generation for the Python package and we could not get a result. Our results indicate that *PyCG* generates high precision call graphs. On all cases, more than 98% of the generated call edges are true positives, while on one case none of the generated call edges are false positives. The results for recall show that on average, 69.9% of all call edges are successfully retrieved. The missing call edges are owed to the approach's limitations (recall Section 3.4), and missing support for some Python functionalities.

Pyan shows average precision and low recall. The average precision appears because *Pyan* adds call edges to both class names and their `__init__()` functions. Furthermore, it does not capture the inter-procedural flow of functions, leading to its low recall. For instance, the source code of the `face_classification` package mostly depends on functions that are imported from external packages. *Pyan* ignores these calls which leads to a 7.6% recall.

Table 5.2: Call graph micro-benchmark results for *PyCG* and *Pyan*. *Depends* is unsound in all cases and complete in 110/112 cases and is omitted.

Category	PyCG		Pyan	
	Complete	**Sound**	**Complete**	**Sound**
`assignments`	4/4	3/4	4/4	4/4
`built-ins`	3/3	1/3	2/3	0/3
`classes`	22/22	22/22	6/22	10/22
`decorators`	6/7	5/7	4/7	3/7
`dicts`	12/12	11/12	6/12	6/12
`direct calls`	4/4	4/4	0/4	0/4
`exceptions`	3/3	3/3	0/3	0/3
`functions`	4/4	4/4	4/4	3/4
`generators`	6/6	6/6	0/6	0/6
`imports`	14/14	14/14	10/14	4/14
`kwargs`	3/3	3/3	0/3	0/3
`lambdas`	5/5	5/5	4/5	0/5
`lists`	8/8	7/8	3/8	4/8
`mro`	7/7	5/7	0/7	2/7
`parameters`	6/6	6/6	0/6	0/6
`returns`	4/4	4/4	0/4	0/4
Total	**111/112**	**103/112**	**43/112**	**36/112**

Table 5.3: Call graph macro-benchmark suite project details.

Project	LoC	Stars	Forks	Description
`fabric`	3,236	12.1k	1.8k	Remote execution & deployment
`autojump`	2,662	10.8k	530	Directory navigation tool
`asciinema`	1,409	7.9k	687	Terminal session recorder
`face_classification`	1,455	4.7k	1.4k	Face detection & classification
`Sublist3r`	1,269	4.4k	1.1k	Subdomains enumeration tool

Finally, *Depends* generates call graphs that have high precision (98.7%) and low recall. The precision results can be attributed to the conservative approach implemented by *Depends*. In addition, *Depends* does not track higher order functions and misses calls originating in module namespaces, leading to its low recall.

5.2 Key Errors

5.2.1 Key Errors Micro-Benchmark Suite

We have implemented a test suite of 25 unique and minimal Python programs that contain source code that will lead to an invalid dictionary access. Our test cases are split into 6 different categories, that range from different assignment methods to the inter-procedural flow of dictionaries and their accessors. We use this micro-benchmark to evaluate our key error identification mechanism on the different settings that can arise in a typical Python program.

The key error micro-benchmark suite is structured in a similar manner as the call graph micro-benchmarks. Specifically, each category implements a number of tests. Each one of these tests includes (1) the Python source code, (2) the key errors that will arise from that program in the same format that we use for *PyCG*'s output (see Section 4.4.2), and (3) a short description (in Markdown

Table 5.4: Call graph macro-benchmark results and tool comparison.

Project	Precision (%)			Recall (%)		
	PyCG	Pyan	Depends	PyCG	Pyan	Depends
`autojump`	99.5	66.5	99.2	68.2	28.5	22.5
`fabric`	98.3	-	100	61.9	-	6.3
`asciinema`	100	-	98.1	68	-	15.5
`face_classification`	99.5	86.8	96.2	89.7	7.6	5.7
`Sublist3r`	98.8	69.8	100	61.6	25.6	21.9
Average	**99.2**	**74.4**	**98.7**	**69.9**	**20.6**	**14.4**

Table 5.5: Key errors micro-benchmark suite categories.

Category	#tests	Description
`assignments`	3	Assignment of dictionaries to variables
`classes`	6	Class construction, attributes, methods
`dicts`	9	Dictionary operations
`kwargs`	2	Keyword arguments that are dictionaries
`lists`	2	Lists with values that are dictionaries
`parameters`	3	Positional arguments that are dictionaries

Table 5.6: Key errors micro-benchmark results.

	Complete	Sound
`assignments`	2/3	2/3
`classes`	6/6	6/6
`dicts`	9/9	8/9
`kwargs`	2/2	2/2
`lists`	2/2	2/2
`parameters`	3/3	3/3
Total	**24/25**	**23/25**

format). Table 5.5 lists the categories, the number of tests they contain, and a corresponding description.

We outline the results of our evaluation on Table 5.6. We check if our implementation can generate complete and sound results.

PyCG produces a complete list of key errors on 24/25 test cases and a sound list on 23/25 cases. The missing marks for the assignments category are due to *PyCG* not supporting starred assignments. Furthermore, *PyCG* produces an unsound key errors list in one test case of the dictionaries category, due to a call to the built-in method `update` of the `dictionary` type. Recall (Section 3.4) that *PyCG* ignores calls on built-in functions.

5.2.2 Key Errors Macro-Benchmarks

To evaluate our key error identification mechanism on real-world Python programs, we have collected the student submissions implemented in Python for an assignment of a post-graduate course related to practical data analysis. Specifically, the students had to implement a stock trading application. They were given access to text files containing historical trading data of the New York Stock Exchange and were asked to produce a sequence of trades that would lead to the maximum profit.

Table 5.7: Time and memory comparison.

Project	Time (sec)			Memory (MB)		
	PyCG	Pyan	Depends	PyCG	Pyan	Depends
`autojump`	0.76	0.42	2.37	62.7	37.8	27.1
`fabric`	0.77	-	1.83	60.9	-	18.5
`asciinema`	0.87	-	2	61.6	-	19.4
`face_classification`	0.92	0.38	2.49	60.9	35.3	25.6
`Sublist3r`	0.51	0.33	2.01	60	35.8	19.4
Average	**0.77**	**0.38**	**2.14**	**61.2**	**36.3**	**22**

We analyzed a total of 62 submissions and found a total of 11 key errors from 3 of those. We performed a qualitative analysis on the submissions with the assumption that the submitted solutions did not lead to invalid dictionary accesses when run on the student's machines. Our analysis was focused on the 3 submissions for which our mechanism found key errors, while we also analyzed many of the submissions that gave a clean result to identify potential false negatives.

Our qualitative analysis showed that our mechanism identified key errors when the input programs initialized their dictionaries based on external factors (e.g. the existence of a file) and then worked with the assumption that the external factors would have specific properties (e.g. a file has a specific name). As an example, consider the following simplified approach of one of those submissions.

```
1  files = os.listdir("stocks")
2  stock_contents = {}
3
4  for file in files:
5      with open(file) as f:
6          stock_contents[file] = f.read()
7
8  apple = stock_contents["AAPL.txt"]
```

Initially, on line 1 a list of file names is retrieved and on line 2 a dictionary is created which will later store the contents of those files. Then, on the loop of lines 4 to 6, the contents of each file are read and stored in the dictionary with the file name serving as key. Finally, on line 8 the contents of the file named `AAPL.txt` are retrieved. This source code makes the assumption that the `stocks` directory will contain a file named `AAPL.txt`, because on line 8 it directly accesses that dictionary entry without consulting the list of files contained in the directory. In such cases, *PyCG* identifies a key error because it does not have any information about the environment those scripts are meant to run in. Interestingly, the submissions for which *PyCG* did not identify a key error, worked with the same environment of files, but took care to initialize their dictionaries based on the state of the environment, and without making any assumptions — a good coding practice.

5.3 Time and Memory Performance

We perform our time and memory evaluation using the call graph macro-benchmark suite (Table 5.3) as our base. With regards to key error identification, our tool performs the same underlying analysis, and thus produces similar time and memory results. Table 5.7 presents the time and memory performance metrics of *PyCG*, *Pyan*, and *Depends*. We used the UNIX *time* command to calculate the execution times, and the unix *pmap* command to measure memory consumption. The metrics presented are the average out of 20 runs.

According to our results, *Pyan* is more efficient in terms of execution time, and *Depends* is more memory efficient. *PyCG* and *Pyan* generate a call graph for the programs in the benchmark ($\leq$ 3.5k LoC) in less than a second, while *Depends* requires more than two seconds on average. In addition,

all tools use a reasonable amount of memory, with *PyCG*, *Pyan* and *Depends* using on average ∼61.4, ∼36.3, and ∼22 MBs of memory respectively. *Pyan* is on average 2 times faster than *PyCG*, and *PyCG* uses 2.8x more memory than *Depends*. We attribute these differences between *Pyan* and *PyCG* to the fact that *Pyan* passes the ast two times compared to *PyCG* performing a fixed-point iteration (Chapter 3). *Depends* is slower, since it spends most of its execution time parsing the source files. Regarding memory, *Pyan* and *Depends* maintain less information about the state of the analysis which leads to better memory performance.

Chapter 6

Related Work

Static Analysis for Dynamic Languages. In the case of JavaScript, there are many frameworks that aim to statically analyze programs written in the language. The SAFE [9] framework implements a scalable, pluggable, and flexible static analysis methodology. JSAI [10] proposes a provably sound and formally specified analysis that uses abstract interpretation.

Other JavaScript analyzers target different aspects of the language's functionality. Madsen et al. propose RADAR [35] a tool for finding bugs in event-driven programs. Sotiropoulos et al. [14] implement an analysis that targets asynchronous functions. $\text{SAFE}_{\text{WAPI}}$ is a tool that can identify potential API misuses. $\text{SAFE}_{\text{WApp}}$, implements a static analysis for JavaScript code that resides on the client-side.

Call Graph Generation. Methods that generate call graphs are split into dynamic [36] and static [37]. Dynamic approaches usually deliver fewer false positives, but lack in performance. Furthermore, they can analyze a single execution path, with their effectiveness relying on the provided input. On the other hand, static approaches are more time efficient and typically attempt to capture all possible program behaviors, leading to larger coverage of execution paths. The combination of the two to get improved results has been tried by several approaches [38, 39, 40].

There are many methods and tools that attempt call graph generation for statically-typed programming languages. DOOP [41] and WALA [42] follow a context-sensitive, points-to analysis method for Java. PADDLE [43], a similar approach, employs Binary Decision Diagrams (BDDs) [44]. Finally, OPAL [45] is a lattice-based approach written in Scala. Ali et al. [46], implement CGC, a partial call graph generator for Java, that mainly focuses on efficiency. The tool does not infer calls originating in external libraries, and focuses on the analysis of the source code of the given input package. We are currently following a similar approach. However, our aim is to efficiently analyze function calls coming from external libraries in the future.

Regarding dynamic languages, Ali et al. [47] implement an approach that converts Python source code into JVM bytecode, and then generate its call graph through the use of the existing implementations for Java [42, 48, 49]. Their results show that generating precise call graphs with this method is infeasible, since the output sometimes contains more than 96% of false positives. *pycallgraph* [50] is dynamic approach that generates Python call graphs for a single execution path. Therefore, the analysis requires pairing with another method (e.g., fuzzing) to retrieve meaningful results. On the JavaScript front, Feldthaus et al. [13] propose a flow-based approach for call graphs generation. Their evaluation benchmark includes call graphs generated by a dynamic approach paired with instrumentation, with their analysis achieving $\geq$ 66% precision and $\geq$ 85% recall. Other JavaScript call graph generators include, NPM call graph [51], IBM WALA [42], Approximate Call Graph (ACG) [13], Google closure compiler [52], and Type Analyzer for JavaScript (TAJS) [8]. TAJS implements a lattice-based flow-sensitive approach that uses abstract interpretation. Even though this approach yields more promising results, it comes with a cost in performance.

Call Graph Benchmarking and Comparison. Reif et al. propose Judge [34], a toolchain for the analysis of call graph generators for Java. This toolchain implements a test suite with benchmarks for a range of different Java features. The authors use it to compare Java call graph generators, namely Soot [48, 49], WALA [42], DOOP [41], and OPAL [45]. Sui et al. [53], present another test suite of Java benchmarks, and use it to evaluate and compare Soot [48, 49], WALA [42], and DOOP [41].

The aforementioned benchmark suites are very similar, leading to Judge consolidating them into one benchmark suite. Recall that we have implemented a similar micro-benchmark suite (Chapter 5).

Error Checking for Python. Fromherz et al. [54] propose an implementation that can soundly identify run-time errors by evaluating the data types of Python variables through the use of abstract interpretation. However, their approach is still in development and does not support many Python features, including recursion. In comparison, we have implemented a prototype that can handle most of Python's features and reasons about literal values in a flow insensitive manner which allows us to efficiently identify invalid dictionary accesses.

There are several tools that aim to check Python programs for potential errors during the development process. *pyflakes* [20] works on the syntax tree of every file to identify errors including undefined variables and invalid formatting literals. *pylint* [21] is another popular approach that provides integration with several IDEs as well as continuous integration pipelines. It can identify many potential errors including undefined variables and invalid return types. *PyChecker* [55] focuses on the identification of errors that are usually caught by compilers for languages like C and C++. However, none of these tools deal with the identification of invalid accesses on dictionaries.

Other tools focus on the identification of common security issues [22, 23], type checking [56, 57, 58], and code enhancements [59, 60, 24]. Compared to the aforementioned tools, our approach employs an inter-procedural analysis and focuses on the identification of key errors.

Chapter 7

Conclusion

We have introduced a practical static analysis framework for Python programs and have implemented a corresponding prototype that we call *PyCG*. Our method performs a context-insensitive inter-procedural analysis that identifies the flow of values through the construction of a graph that stores all assignment relationships among program identifiers. We propose two applications that employ our analysis, namely call graph generation and the identification of key errors. For call graph generation, we found that our implementation marks high rates in both precision and recall. With respect to key error identification, we showed that our approach can efficiently identify key errors that can occur in many different Python setups.

In the future, we aim to extend our analysis to support more Python features and to include a model of built-in methods, types, and their effects. Furthermore, we will analyze the source code of external packages by adding heuristics to identify their source code's location.

Βιβλιογραφία

[1] N. Heath, "Tiobe index," 2021, [Online; accessed 30-May-2021]. [Online]. Available: https://www.tiobe.com/tiobe-index/

[2] ——, "Python is eating the world: How one developer's side project became the hottest programming language on the planet," 2019, [Online; accessed 30-May-2021]. [Online]. Available: https://tinyurl.com/yxt9ujnb

[3] J. Hejderup, A. van Deursen, and G. Gousios, "Software ecosystem call graph for dependency management," in *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*, ser. ICSE-NIER '18. New York, NY, USA: ACM, 2018, pp. 101–104.

[4] R. Kikas, G. Gousios, M. Dumas, and D. Pfahl, "Structure and evolution of package dependency networks," in *Proceedings of the 14th International Conference on Mining Software Repositories*, ser. MSR '17. IEEE Press, 2017, pp. 102–112.

[5] (2016) The npm blog: changes to npm's unpublish policy. [Online; accessed 30-May-2021]. [Online]. Available: https://blog.npmjs.org/post/141905368000/changes-to-npms-unpublish-policy

[6] (2020) npm(1)—a JavaScript package manager. [Online; accessed 30-May-2021]. [Online]. Available: https://github.com/npm/cli

[7] (2020) pip 20.0.2: The PyPA recommended tool for installing Python packages. [Online; accessed 30-May-2021]. [Online]. Available: https://pypi.org/project/pip/

[8] S. H. Jensen, A. Møller, and P. Thiemann, "Type analysis for JavaScript," in *International Static Analysis Symposium*. Springer, 2009, pp. 238–255.

[9] H. Lee, S. Won, J. Jin, J. Cho, and S. Ryu, "SAFE: Formal specification and implementation of a scalable analysis framework for ECMAScript," in *FOOL 2012: 19th International Workshop on Foundations of Object-Oriented Languages*. Citeseer, 2012, p. 96.

[10] V. Kashyap, K. Dewey, E. A. Kuefner, J. Wagner, K. Gibbons, J. Sarracino, B. Wiedermann, and B. Hardekopf, "JSAI: A static analysis platform for JavaScript," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, pp. 121–132.

[11] Y. Ko, H. Lee, J. Dolby, and S. Ryu, "Practically tunable static analysis framework for large-scale JavaScript applications," in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '15. IEEE Press, 2015, pp. 541–551.

[12] M. Madsen, B. Livshits, and M. Fanning, "Practical static analysis of javascript applications in the presence of frameworks and libraries," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: Association for Computing Machinery, 2013, pp. 499–509.

[13] A. Feldthaus, M. Schäfer, M. Sridharan, J. Dolby, and F. Tip, "Efficient construction of approximate call graphs for JavaScript IDE services," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. IEEE Press, 2013, pp. 752–761.

[14] T. Sotiropoulos and B. Livshits, "Static analysis for asynchronous JavaScript programs," in *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), A. F. Donaldson, Ed., vol. 134. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, pp. 8:1–8:30. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2019/10800

[15] M. Madsen, F. Tip, and O. Lhoták, "Static analysis of event-driven node.js JavaScript applications," *SIGPLAN Not.*, vol. 50, no. 10, pp. 505–519, Oct. 2015.

[16] N. Milojkovic, M. Ghafari, and O. Nierstrasz, "It's duck (typing) season!" in *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, May 2017, pp. 312–315.

[17] Valgrind, "Callgrind: a call-graph generating cache and branch prediction profiler," 2020. [Online]. Available: http://valgrind.org/docs/manual/cl-manual.html

[18] H. Shahriar and M. Zulkernine, "Mitigating program security vulnerabilities: Approaches and challenges," *ACM Comput. Surv.*, vol. 44, no. 3, Jun. 2012.

[19] A. Feldthaus, T. Millstein, A. Møller, M. Schäfer, and F. Tip, "Tool-supported refactoring for JavaScript," in *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '11. New York, NY, USA: Association for Computing Machinery, 2011, pp. 119–138.

[20] "Pyflakes – a simple program which checks python source files for errors." https://github.com/PyCQA/pyflakes, 2005, [Online; accessed 30-May-2021].

[21] "Pylint – code analysis for python." https://www.pylint.org/, 2001, [Online; accessed 30-May-2021].

[22] "A security linter from pycqa," https://bandit.readthedocs.io/en/latest/, 2015, [Online; accessed 30-May-2021].

[23] "Pysa: An open source static analysis tool to detect and prevent security issues in python code," https://engineering.fb.com/2020/08/07/security/pysa/, 2020, [Online; accessed 30-May-2021].

[24] "A lint framework that writes better python code for you." https://github.com/Instagram/Fixit, 2020, [Online; accessed 30-May-2021].

[25] GitHub, "The state of the octoverse," https://octoverse.github.com/, 2019, [Online; accessed 30-May-2021].

[26] D. Fraser, E. Horner, J. Jeronen, and P. Massot, "Pyan3: Offline call graph generator for Python 3," https://github.com/davidfraser/pyan, 2018, [Online; accessed 30-May-2021].

[27] G. Gharibi, R. Tripathi, and Y. Lee, "Code2graph: Automatic generation of static call graphs for Python source code," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018. New York, NY, USA: Association for Computing Machinery, 2018, pp. 880–883.

[28] G. Gharibi, R. Alanazi, and Y. Lee, "Automatic hierarchical clustering of static call graphs for program comprehension," in *IEEE International Conference on Big Data, Big Data 2018, Seattle, WA, USA, December 10-13, 2018*. IEEE, 2018, pp. 4016–4025.

[29] G. Zhang and J. Wuxia, "Depends is a fast, comprehensive code dependency analysis tool," https://github.com/multilang-depends/depends, 2018, [Online; accessed 30-May-2021].

[30] M. Felleisen, R. B. Findler, and M. Flatt, *Semantics engineering with PLT Redex*. Mit Press, 2009.

[31] M. Madsen, O. Lhoták, and F. Tip, "A model for reasoning about JavaScript promises," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, Oct. 2017. [Online]. Available: https://doi.org/10.1145/3133910

[32] S. Guarnieri and B. Livshits, "GATEKEEPER: Mostly static enforcement of security and reliability policies for JavaScript code," in *Proceedings of the 18th Conference on USENIX Security Symposium*, ser. SSYM'09. USA: USENIX Association, 2009, pp. 151–168.

[33] C.-A. Staicu, M. Pradel, and B. Livshits, "SYNODE: Understanding and automatically preventing injection attacks on Node. js." in *NDSS*, 2018.

[34] M. Reif, F. Kübler, M. Eichberg, D. Helm, and M. Mezini, "Judge: Identifying, understanding, and evaluating sources of unsoundness in call graphs," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2019. New York, NY, USA: Association for Computing Machinery, 2019, pp. 251–261.

[35] M. Madsen, F. Tip, and O. Lhoták, "Static analysis of event-driven Node.js JavaScript applications," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA 2015. New York, NY, USA: Association for Computing Machinery, 2015, pp. 505–519.

[36] T. Xie and D. Notkin, "An empirical study of Java dynamic call graph extractors," *University of Washington CSE Technical Report 02-12*, vol. 3, 2002.

[37] G. C. Murphy, D. Notkin, W. G. Griswold, and E. S. Lan, "An empirical study of static call graph extractors," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 7, no. 2, pp. 158–191, 1998.

[38] T. Eisenbarth, R. Koschke, and D. Simon, "Aiding program comprehension by static and dynamic feature analysis," in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*. IEEE Computer Society, 2001, p. 602.

[39] N. Grech, G. Fourtounis, A. Francalanza, and Y. Smaragdakis, "Heaps don't lie: Countering unsoundness with heap snapshots," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, Oct. 2017.

[40] J. Liu, Y. Li, T. Tan, and J. Xue, "Reflection analysis for Java: Uncovering more reflective targets precisely," in *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2017, pp. 12–23.

[41] M. Bravenboer and Y. Smaragdakis, "Strictly declarative specification of sophisticated points-to analyses," in *ACM SIGPLAN Notices*, vol. 44, no. 10. ACM, 2009, pp. 243–262.

[42] S. Fink and J. Dolby, "WALA—the T.J. Watson libraries for analysis," 2012.

[43] O. Lhoták and L. Hendren, "Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 18, no. 1, p. 3, 2008.

[44] M. Berndl, O. Lhoták, F. Qian, L. Hendren, and N. Umanee, "Points-to analysis using BDDs," *SIGPLAN Not.*, vol. 38, no. 5, pp. 103–114, May 2003.

[45] M. Eichberg, F. Kübler, D. Helm, M. Reif, G. Salvaneschi, and M. Mezini, "Lattice based modularization of static analyses," in *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*, ser. ISSTA '18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 113–118.

[46] K. Ali and O. Lhoták, "Application-only call graph construction," in *Proceedings of the 26th European Conference on Object-Oriented Programming*, ser. ECOOP'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 688–712.

[47] K. Ali, X. Lai, Z. Luo, O. Lhotak, J. Dolby, and F. Tip, "A study of call graph construction for JVM-hosted languages," *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.

[48] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A Java bytecode optimization framework," in *CASCON First Decade High Impact Papers*, ser. CASCON '10. USA: IBM Corp., 2010, pp. 214–224.

[49] O. Lhoták and L. Hendren, "Scaling Java points-to analysis using spark," in *International Conference on Compiler Construction*. Springer, 2003, pp. 153–169.

[50] GitHub user gak, "pycallgraph is a Python module that creates call graphs for Python programs." https://github.com/gak/pycallgraph, 2014, [Online; accessed 30-May-2021].

[51] G. Gessner, "npm call graph," https://www.npmjs.com/package/callgraph, 2019, [Online; accessed 28-May-2021].

[52] M. Bolin, *Closure: The Definitive Guide: Google Tools to Add Power to Your JavaScript*. " O'Reilly Media, Inc.", 2010.

[53] L. Sui, J. Dietrich, M. Emery, S. Rasheed, and A. Tahir, "On the soundness of call graph construction in the presence of dynamic language features—a benchmark and tool evaluation," in *Asian Symposium on Programming Languages and Systems*. Springer, 2018, pp. 69–88.

[54] A. Fromherz, A. Ouadjaout, and A. Miné, "Static value analysis of Python programs by abstract interpretation," in *NASA Formal Methods Symposium*. Springer, 2018, pp. 185–202.

[55] "Pychecker: a python source code checking tool," http://pychecker.sourceforge.net/, 2003, [Online; accessed 30-May-2021].

[56] "Optional static typing for python 3 and 2 (pep 484)," http://www.mypy-lang.org/, 2012, [Online; accessed 30-May-2021].

[57] "A performant type-checker for python 3," https://pyre-check.org/, 2018, [Online; accessed 30-May-2021].

[58] "Static type checker for python," https://github.com/Microsoft/pyright, 2019, [Online; accessed 30-May-2021].

[59] "The uncompromising code formatter," https://github.com/psf/black, 2018, [Online; accessed 30-May-2021].

[60] "Safe code refactoring for modern python," https://pybowler.io/, 2018, [Online; accessed 30-May-2021].

[61] Y. Li, "Empirical study of Python call graph," in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '19. IEEE Press, 2019, pp. 1274–1276.

[62] D. Spinellis, "CScout: A refactoring browser for C," *Sci. Comput. Program.*, vol. 75, no. 4, pp. 216–231, Apr. 2010.

[63] (2020) Django: The Web framework for perfectionists with deadlines. [Online; accessed 30-May-2021]. [Online]. Available: https://www.djangoproject.com/

[64] (2020) pyyaml: The next generation yaml parser and emitter for Python. [Online; accessed 30-May-2021]. [Online]. Available: https://github.com/yaml/pyyaml/

[65] (2017) CVE-2017-18342. [Online; accessed 28-May-2021]. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2017-18342

[66] (2020) Paramiko: The leading native Python sshv2 protocol library. [Online; accessed 30-May-2021]. [Online]. Available: https://github.com/paramiko/paramiko/

[67] (2018) CVE-2018-7750. [Online; accessed 28-May-2021]. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2018-7750

[68] (2020) GitHub advisory database. [Online; accessed 28-May-2021]. [Online]. Available: https://github.com/advisories

[69] (2020) symtable. [Online; accessed 30-May-2021]. [Online]. Available: https://docs.python.org/3/library/symtable.html

[70] (2020) AST in Python. [Online; accessed 28-May-2021]. [Online]. Available: https://docs.python.org/3/library/ast.html

[71] Y. Sui and J. Xue, "Pointer analysis and program depedence analysis in LLVM," https://svf-tools.github.io/SVF/, 2019, [Online; accessed 30-May-2021].

[72] G. Antal, P. Hegedus, Z. Tóth, R. Ferenc, and T. Gyimóthy, "Static JavaScript call graphs: A comparative study," in *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2018, pp. 177–186.

[73] P. Bhattacharya, M. Iliofotou, I. Neamtiu, and M. Faloutsos, "Graph-based analysis and prediction for software evolution," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. IEEE Press, 2012, pp. 419–429.

[74] A. Rahman, C. Parnin, and L. Williams, "The seven sins: Security smells in infrastructure as code scripts," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19. IEEE Press, 2019, pp. 164–175. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00033

[75] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini, "Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: Association for Computing Machinery, 2011, pp. 241–250. [Online]. Available: https://doi.org/10.1145/1985793.1985827

[76] O. Lhoták, "Comparing call graphs," in *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, ser. PASTE '07. New York, NY, USA: Association for Computing Machinery, 2007, pp. 37–42.

[77] S. Bae, H. Cho, I. Lim, and S. Ryu, "SAFEWAPI: Web API misuse detector for web applications," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, pp. 507–517.

[78] C. Park, S. Won, J. Jin, and S. Ryu, “Static analysis of JavaScript web applications in the wild via practical DOM modeling,” in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE ’15. IEEE Press, 2015, pp. 552–562.

[79] S. H. Jensen, M. Madsen, and A. Møller, “Modeling the HTML DOM and browser API in static analysis of JavaScript web applications,” in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE ’11. New York, NY, USA: Association for Computing Machinery, 2011, pp. 59–69.

[80] G. Richards, C. Hammer, B. Burg, and J. Vitek, “The eval that men do,” in *ECOOP 2011—Object-Oriented Programming*, M. Mezini, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 52–78.