ΠΟΛΥΤΕΧΝΕΙΟ ΚΡΗΤΗΣ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ & ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Δυναμική Ανάλυση για Python

Συγγραφέας:
Γιόελ Γιάννη

Εξεταστική Επιτροπή:
Καθηγητής Σωτήριος Ιωαννίδης
Αναπληρωτής Καθηγητής Βασίλης Σαμολαδάς
Διδάκτωρ Γεώργιος Χρήστου

Οκτώμβριος 2025

Περίληψη

Η δυναμική ανάλυση προγραμμάτων αποτελεί μια κρίσιμη τεχνική για την κατανόηση, παρακολούθηση και τον έλεγχο της συμπεριφοράς εφαρμογών κατά τον χρόνο εκτέλεσης. Στην παρούσα διπλωματική εργασία παρουσιάζεται το **Cerbex**, ένα νέο πλαίσιο δυναμικής ανάλυσης που στοχεύει ειδικά στη γλώσσα προγραμματισμού Python. Το εργαλείο αξιοποιεί τον μηχανισμό εισαγωγών της Python ώστε να παρεμβάλλει *κώδικα παρακολούθησης* κατά τη φόρτωση των βιβλιοθηκών, εισάγοντας άγκιστρα και περιτυλίγματα (wrappers) σε συναρτήσεις χωρίς να τροποποιεί τον πηγαίο κώδικα. Συνδυάζοντας ενσωμάτωση κώδικα παρακολούθησης (instrumentation) στο επίπεδο εισαγωγής Python αρθρωμάτων και με την αξιοποίηση του μηχανισμού `sys.setprofile` για C-επεκτάσεις, το Cerbex παρέχει κάλυψη τόσο στον Python κώδικα όσο και σε βιβλιοθήκες χαμηλού επιπέδου, όπως οι math, numpy και pandas. Ο πυρήνας του Cerbex υποστηρίζει δύο διακριτές λειτουργίες: στη λειτουργία μάθησης καταγράφει τα γεγονότα εκτέλεσης και τις εξαρτήσεις των αρθρωμάτων σε αρχεία όπως τα `events.json` και `dependencies.json`, ενώ στη συνέχεια τα συνδυάζει για να παραχθεί το `allowlist.json`. Στη λειτουργία επιβολής αξιοποιεί το ενιαίο αυτό αρχείο ώστε να αποτρέπει μη εξουσιοδοτημένες εισαγωγές και κλήσεις συναρτήσεων σε πραγματικό χρόνο. Παράλληλα, οι αναλύσεις εξάγουν πρόσθετα δεδομένα όπως αρχεία `perf.log` για επιδόσεις και `types.log` για τύπους επιστροφής. Η αρχιτεκτονική του εργαλείου είναι αρθρωτή, βασισμένη σε πρόσθετα εργαλεία ανάλυσης, επιτρέποντας την εύκολη επέκταση με νέες μορφές ελέγχου. Ενδεικτικά, το `PerfAnalyzer` καταγράφει χρόνους εκτέλεσης συναρτήσεων, ενώ το `TypeExtractor` συλλέγει τύπους επιστροφής, δείχνοντας την ευελιξία της προσέγγισης. Η αξιολόγηση του Cerbex καταδεικνύει ότι το εργαλείο εισάγει αισθητή σχετική επιβάρυνση σε μικρής κλίμακας ή βραχύβιες εκτελέσεις, όπου το σταθερό κόστος του μηχανισμού ενσωμάτωσης γίνεται κυρίαρχο. Αντίθετα, σε πιο σύνθετες εφαρμογές, μεγάλα πλαίσια και υπολογιστικά εντατικά σενάρια, η σχετική επιβάρυνση μειώνεται σημαντικά, με αποτέλεσμα το Cerbex να αποδίδει ιδιαίτερα καλά. Σε τέτοια περιβάλλοντα, η δυνατότητα για ορατότητα, επιβολή πολιτικών και εις βάθος κατανόηση της εκτέλεσης συνδυάζεται με ικανοποιητική απόδοση, καθιστώντας το εργαλείο ιδανικό για μεσαία και βαριά φορτία.

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τον καθηγητή Σωτήριο Ιωαννίδη, διευθυντή του Εργαστηρίου Μικροεπεξεργαστών και Υλικού, για την ευκαιρία να ασχοληθώ με την ασφάλεια λογισμικού. Είμαι επίσης ευγνώμων στους Διδάκτωρ Γεώργιο Χρήστου και Γρηγόρη Ντουσάκη για την καθοδήγηση και επιτήρηση της διπλωματικής μου εργασίας καθώς και τον Αναπληρωτή Καθηγητή Βασίλη Σαμολαδά για τη συμμετοχή του στην εξεταστική επιτροπή και για την αξιολόγηση της εργασίας μου. Τέλος, τίποτα απ' όλα αυτά δεν θα ήταν δυνατό χωρίς την στήριξη των γονιών μου και των φίλων μου που στάθηκαν δίπλα μου όλα αυτά τα χρόνια.

Περιεχόμενα

Κατάλογος Σχημάτων **5**

Κατάλογος Κώδικα **6**

1 Εισαγωγή **7**
1.1 Περιγραφή Προβλήματος . . . 7
1.2 Στόχοι της Εργασίας . . . 8
1.3 Κύρια Συνεισφορά . . . 8
1.4 Επισκόπηση της Εργασίας . . . 9

2 Θεωρητικό Υπόβαθρο **11**
2.1 Στατική, δυναμική και υβριδική ανάλυση προγραμμάτων . . . 11
2.2 Το σύστημα εισαγωγής (import) της Python . . . 11
2.3 Τεχνικές ενδοσκόπησης και πολιτικές ασφάλειας . . . 12
2.4 Μηχανισμοί καταγραφής εκτέλεσης στην Python . . . 12
2.5 Επεκτάσεις C και native modules . . . 12
2.6 Thread-local δεδομένα . . . 13
2.7 Τεχνικές περιτύλιξης συναρτήσεων και διακοσμητών . . . 13

3 Παράδειγμα Εφαρμογής και Βασικές Έννοιες **15**
3.1 Βασικές Έννοιες . . . 15
3.2 Κώδικας Παραδείγματος . . . 16
3.2.1 Εκτέλεση με κώδικα παρακολούθησης . . . 17
3.2.2 Αποτελέσματα με κώδικα παρακολούθησης . . . 18
3.2.3 Εκτέλεση σε Λειτουργία Επιβολής (Enforce Mode) . . . 23

4 Αρχιτεκτονική και Σχεδίαση **25**
4.1 Επισκόπηση της αρχιτεκτονικής του συστήματος . . . 25
4.2 Βασικά τμήματα και υλοποίηση . . . 26
4.2.1 Διεπαφή Γραμμής Εντολών (CLI) . . . 26
4.2.2 Διαμόρφωση (Config) . . . 27
4.2.3 Φορτωτής Hooks (Hook Loader) . . . 27
4.2.4 HookManager (Κεντρικός συντονιστής) . . . 28
4.2.5 Μηχανισμός Εισαγωγής και Wrappers (Importer / Wrappers) . . . 28
4.2.6 Πρόσθετα Εργαλεία Ανάλυσης . . . 29
4.2.7 Αναφορές και Καταγραφές (Reports &Logs) . . . 29
4.3 Πλαίσιο Ανάλυσης Ροής Γεγονότων (Event Flow Analysis Framework) . . . 29
4.3.1 Αρχιτεκτονική και Ροή Γεγονότων . . . 30

4.3.2 Ροή Κλήσεων Συναρτήσεων . . . 31
4.4 Διαχείριση Κόστους Εκτέλεσης . . . 32

5 Αναλύσεις **34**
5.1 Ανάλυση Ασφάλειας . . . 34
5.2 Ανάλυση Απόδοσης . . . 39
5.3 Εξαγωγή τύπων . . . 43
5.4 Δημιουργία Νέων Αναλύσεων . . . 45

6 Χρήση Εργαλείου **48**
6.1 Ενσωμάτωση μέσω Προγραμματιστικού API . . . 48
6.2 Χρήση εργαλείου μέσω της γραμμής εντολών . . . 49
6.2.1 Εγκατάσταση του εργαλείου ως CLI . . . 49
6.2.2 Εκτέλεση σε λειτουργία εκμάθησης . . . 49
6.2.3 Εκτέλεση σε λειτουργία επιβολής . . . 50

7 Αξιολόγηση **51**
7.1 Στόχοι και Επισκόπηση . . . 51
7.2 Μηχάνημα μετρήσεων . . . 52
7.2.1 Μεθοδολογία μετρήσεων . . . 52
7.3 Ανάλυση Επιβάρυνσης Απόδοσης . . . 53
7.3.1 FastAPI . . . 54
7.3.2 Flask . . . 54
7.3.3 Μικρο-Πειράματα . . . 56
7.3.4 Μακρο-πείραμα: Pandas GroupBy Benchmark . . . 58
7.3.5 Μακρο-πείραμα: NumPy Linear Algebra . . . 60
7.4 Σχετικά έργα με συγκριτική αξιολόγηση . . . 61
7.5 Διαφορές της παρούσας εργασίας από υπάρχουσες προσεγγίσεις . . . 64
7.6 Συζήτηση &Περιορισμοί . . . 64
7.6.1 Συζήτηση . . . 64
7.6.2 Περιορισμοί . . . 64
7.6.3 Μελλοντική Εργασία . . . 65

8 Συμπεράσματα **67**

Βιβλιογραφία **69**

Κατάλογος Σχημάτων

3.1 Loaded vs Unloaded modules workflow 22

4.1 High level architecture of Cerbex. 25
4.2 Event flow during module import. 30
4.3 Event flow during a function call (learn mode). 31
4.4 Event flow during a function call (enforce mode). 32

7.1 FastAPI performance results. Left: relative runtime slowdowns (%) compared to baseline. Right: absolute execution times with error bars (mean ± std) and 95th percentile markers. 55
7.2 Flask performance results. Left: relative runtime slowdowns (%) compared to baseline. Right: absolute execution times with error bars (mean ± std) and 95th percentile markers. 55
7.3 57
7.4 Micro-package results: (a) packages with shorter baseline runtimes suffer disproportionately larger relative slowdowns, (b) distribution of overheads across analysis configurations highlights the wide variance. 57
7.5 Pandas macro-benchmark results: comparison of execution time between baseline and Cerbex. 59
7.6 NumPy macro-benchmark results: comparison of execution time between baseline and Cerbex. 61

Κατάλογος Κώδικα

3.1 HTTP request and JSON parsing in Python . . . 16
3.2 Terminal output from executing the request_example script . . . 17
3.3 Runner script to execute request_example in learn mode . . . 17
3.4 Config.json file . . . 18
3.5 Runner script to execute request_example in enforce mode . . . 23
3.6 Modified request_example with disallowed import . . . 24
3.7 Terminal output in enforce mode with disallowed import . . . 24
5.1 Vulnerable deserialization script (unsafe_deserialize.py). . . . 34
5.2 Malicious payload generator (make_exploit.py). . . . 35
5.3 Benign payload generator (make_benign.py). . . . 35
5.4 Recording expected behavior in learn mode. . . . 36
5.5 Excerpt from `allowlist.json`. . . . 37
5.6 Execution in enforce mode. . . . 38
5.7 Error output under enforce mode. . . . 38
5.8 Batch image-resizer (`image_resizer.py`). . . . 39
5.9 `PerfAnalyzer` class that logs execution time of each hooked function to a file. . . . 40
5.10 `config.json` for performance analysis. . . . 41
5.11 Run learn+measure pass with `PerfAnalyzer`. . . . 41
5.12 Aggregated execution times from `perf.log`. . . . 42
5.13 Utility module `string_utils.py` . . . 43
5.14 Main program `main.py` with Cerbex hooks . . . 43
5.15 Cerbex analysis `TypeExtractor` . . . 44
5.16 Extracted return types in `type.log` . . . 45
5.17 Παράδειγμα ανάλυσης `CustomDataFlowAnalyzer` . . . 46
5.18 Runner για `CustomDataFlowAnalyzer` . . . 46
6.1 runner.py – Προγραμματιστική Ενσωμάτωση του Cerbex . . . 48
6.2 Εκτέλεση μέσω CLI σε learn mode . . . 49
6.3 Εκτέλεση μέσω CLI σε enforce mode . . . 50
7.1 Pandas macro-benchmark: groupby/agg on 100M rows, 10k groups, 350 iterations, 10 runs. . . . 58
7.2 NumPy macro-benchmark with repeated matrix multiplications and eigenvalue–eigenvector decompositions. . . . 60

Κεφάλαιο 1

Εισαγωγή

Στον συνεχώς εξελισσόμενο κόσμο της τεχνολογίας και της κυβερνοασφάλειας, η ανάγκη για ανάλυση και παρακολούθηση εκτελέσιμου κώδικα σε πραγματικό χρόνο έχει καταστεί επιτακτική. Τα σύγχρονα συστήματα λογισμικού βασίζονται όλο και περισσότερο σε εξωτερικές βιβλιοθήκες και πλαίσια [56, 25], τα οποία επιταχύνουν την ανάπτυξη, διευκολύνουν την επαναχρησιμοποίηση δοκιμασμένου κώδικα και ενσωματώνουν βέλτιστες πρακτικές. Ωστόσο, η εξάρτηση από τρίτο κώδικα εισάγει αδιαφάνεια, αυξάνοντας την επιφάνεια επιθέσεων και επιτρέποντας σε κακόβουλους προγραμματιστές να εκμεταλλεύονται ευπάθειες σε ευρέως χρησιμοποιούμενες βιβλιοθήκες.

Οι παραδοσιακές τεχνικές στατικής ανάλυσης παρέχουν χρήσιμες πληροφορίες για τον πηγαίο κώδικα, αλλά αποτυγχάνουν να συλλάβουν τη δυναμική φύση γλωσσών όπως η Python [16, 12]. Αντίστοιχα, οι μέθοδοι εκ των υστέρων ελέγχου (post-mortem) στερούνται μηχανισμών πραγματικής επιβολής πολιτικών κατά το χρόνο εκτέλεσης [52]. Οι υπάρχουσες τεχνικές δυναμικής ανάλυσης, αν και ακριβέστερες, επιφέρουν δυσβάσταχτη επιβάρυνση που τις καθιστά ακατάλληλες για χρήση σε παραγωγικά συστήματα [43, 7, 23, 15].

Συνεπώς, ανακύπτει η ανάγκη για έναν ελαφρύ και ευέλικτο μηχανισμό, ο οποίος θα επιτρέπει την παρακολούθηση και τον έλεγχο των αλληλεπιδράσεων των προγραμμάτων σε πραγματικό χρόνο, χωρίς την ανάγκη τροποποιήσεων στον υπάρχοντα κώδικα. Αυτή η ανάγκη επιβεβαιώνεται και από πρόσφατες εξελίξεις στη γλώσσα Python, όπως η εισαγωγή των audit hooks (PEP 578) για παρακολούθηση και ασφάλεια [6].

1.1 Περιγραφή Προβλήματος

Η ραγδαία αύξηση της εξάρτησης από βιβλιοθήκες τρίτων έχει μεταβάλει ριζικά τον τρόπο ανάπτυξης λογισμικού. Ενώ στο παρελθόν οι εφαρμογές ήταν κατά κύριο λόγο «μονολιθικές», σήμερα κάθε εφαρμογή Python ενσωματώνει δεκάδες ή και εκατοντάδες εξωτερικά πακέτα [5, 13]. Αυτό μεταφέρει την εμπιστοσύνη από τον προγραμματιστή στο οικοσύστημα, καθιστώντας την ορατότητα και την ασφάλεια ιδιαίτερα κρίσιμες.

Η εξάρτηση αυτή δημιουργεί συγκεκριμένες προκλήσεις:

- **Αδιαφάνεια και έλλειψη ορατότητας:** Ο κώδικας τρίτων συχνά λειτουργεί ως «μαύρο κουτί», καθιστώντας δύσκολη την παρακολούθηση των εσωτερικών ροών δεδομένων.
- **Ανεξέλεγκτη εξέλιξη:** Οι συνεχείς ενημερώσεις βιβλιοθηκών μπορεί να εισάγουν σφάλματα ή κενά ασφαλείας που δεν είναι άμεσα ορατά.

- **Αβεβαιότητες απόδοσης:** Αλλαγές στις εξαρτήσεις ή στη χρήση βιβλιοθηκών μπορούν να οδηγήσουν σε απροσδόκητες καθυστερήσεις ή αυξημένη κατανάλωση πόρων.

Η στατική ανάλυση αποτυγχάνει να καλύψει πλήρως αυτά τα ζητήματα σε περιβάλλοντα με έντονη δυναμική συμπεριφορά [16], ενώ οι υπάρχουσες τεχνικές δυναμικής ανάλυσης συχνά επιφέρουν δυσβάσταχτο κόστος χρόνου εκτέλεσης [43, 7, 23]. Αυτό καθιστά την ανάγκη για πιο αποδοτικούς μηχανισμούς ακόμη πιο επιτακτική.

1.2 Στόχοι της Εργασίας

Η επιτυχής παρακολούθηση και ανάλυση της εκτέλεσης προγραμμάτων Python απαιτεί την ικανοποίηση ορισμένων κριτηρίων που θα επιτρέψουν στο εργαλείο να είναι πρακτικά χρήσιμο, αποδοτικό και εύκολα επεκτάσιμο. Οι στόχοι αυτοί δεν περιορίζονται απλώς σε θεωρητικές απαιτήσεις, αλλά συνδέονται άμεσα με τα προβλήματα που αναλύθηκαν στην προηγούμενη ενότητα: την αδιαφάνεια, την ανεξέλεγκτη εξέλιξη και την αβεβαιότητα απόδοσης. Επομένως, το πλαίσιο που προτείνεται στην παρούσα εργασία στοχεύει να γεφυρώσει το χάσμα ανάμεσα στην ακαδημαϊκή έρευνα και τις πρακτικές ανάγκες των προγραμματιστών, ορίζοντας συγκεκριμένα χαρακτηριστικά που πρέπει να πληροί.

Η παρούσα εργασία αποσκοπεί στην ανάπτυξη ενός πλαισίου δυναμικής ανάλυσης που να ικανοποιεί τις παρακάτω προϋποθέσεις:

- **Διαφάνεια:** Το εργαλείο πρέπει να λειτουργεί χωρίς τροποποίηση του κώδικα του χρήστη, αξιοποιώντας τους μηχανισμούς του Python runtime.
- **Χαμηλό overhead:** Η παρακολούθηση να είναι επαρκώς αποδοτική ώστε να μπορεί να χρησιμοποιηθεί ακόμη και σε περιβάλλοντα παραγωγής.
- **Επεκτασιμότητα:** Οι χρήστες να μπορούν να αναπτύσσουν νέες αναλύσεις με ελάχιστη προσπάθεια.
- **Επιβολή πολιτικών:** Δυνατότητα μετάβασης από παθητική καταγραφή (learn mode) σε ενεργή επιβολή (enforce mode) με χρήση επιτρεπόμενων λιστών (allowlists).

1.3 Κύρια Συνεισφορά

Η συνεισφορά της παρούσας διπλωματικής εργασίας έγκειται στη σχεδίαση και υλοποίηση ενός νέου εργαλείου δυναμικής ανάλυσης για την Python, το οποίο ονομάζεται **Cerbex**. Το εργαλείο αυτό δεν αποτελεί απλώς μια εφαρμογή υπαρχόντων ιδεών, αλλά συνδυάζει τεχνικές από διάφορες προσεγγίσεις ώστε να καλύψει το ερευνητικό κενό που αναδείχθηκε στη σχετική βιβλιογραφία. Πιο συγκεκριμένα, το Cerbex στοχεύει να προσφέρει μια ισορροπία ανάμεσα στην αποδοτικότητα και την ευελιξία, υλοποιώντας ένα μοντέλο λειτουργίας που μπορεί να χρησιμοποιηθεί τόσο για απλή παρακολούθηση όσο και για την επιβολή πολιτικών ασφαλείας. Επιπλέον, η αρθρωτή του σχεδίαση επιτρέπει σε τρίτους να επεκτείνουν εύκολα τη λειτουργικότητά του με νέες αναλύσεις.

Η εργασία προτείνει και υλοποιεί το **Cerbex**, ένα εργαλείο δυναμικής ανάλυσης για την Python το οποίο:

- Εισάγει άγκιστρα κατά τη φόρτωση αρθρωμάτων μέσω προσαρμοσμένων `MetaPathFinder` και `Loader` [33, 20].
- Τυλίγει συναρτήσεις με περιτυλίγματα που παρακολουθούν κλήσεις και επιστροφές, χωρίς να αλλοιώνουν τη σημασιολογία [3, 9].

- Υποστηρίζει ανάλυση τόσο Python όσο και C-επεκτάσεων (μέσω `sys.setprofile`) [37, 55].
- Ενσωματώνει μηχανισμό εκμάθησης και επιβολής για την κατασκευή και αξιοποίηση επιτρεπόμενων λιστών.
- Προσφέρει αρθρωτή αρχιτεκτονική βασισμένη σε πρόσθετα εργαλεία ανάλυσης (π.χ. `PerfAnalyzer`, `TypeExtractor`).

1.4 Επισκόπηση της Εργασίας

Η παρούσα διπλωματική εργασία είναι οργανωμένη σε οκτώ κεφάλαια, καθένα από τα οποία επιτελεί έναν διακριτό ρόλο στην παρουσίαση του θεωρητικού υποβάθρου, της σχεδίασης και της αξιολόγησης του προτεινόμενου εργαλείου.

- **Κεφάλαιο 2:** Παρουσιάζεται το θεωρητικό υπόβαθρο που συνδέεται με το αντικείμενο της μελέτης. Αρχικά, αναλύονται οι βασικές έννοιες της στατικής, δυναμικής και υβριδικής ανάλυσης προγραμμάτων και τα πλεονεκτήματα/μειονεκτήματα καθεμιάς. Στη συνέχεια, περιγράφεται σε βάθος το σύστημα εισαγωγών (import) της Python, οι τεχνικές ενδοσκόπησης και οι μηχανισμοί ασφαλείας που διατίθενται κατά τον χρόνο εκτέλεσης. Εξετάζονται οι διαθέσιμες διεπαφές καταγραφής εκτέλεσης (ιχνηλάτηση, εργαλεία προφίλ, PEP 669), καθώς και οι ιδιαιτερότητες παρακολούθησης επεκτάσεων σε C/C++. Επιπλέον, γίνεται αναφορά στη χρήση `thread-local` δεδομένων και `contextvars` για ασύγχρονες εφαρμογές, ενώ παρουσιάζονται οι τεχνικές περιτύλιξης συναρτήσεων και οι διακοσμητές `(decorators)` ως βασικός μηχανισμός μεταπρογραμματισμού στην Python.
- **Κεφάλαιο 3:** Σε αυτό το κεφάλαιο δίνεται ένα παράδειγμα χρήσης του προτεινόμενου εργαλείου και παρουσιάζονται οι βασικές έννοιες που θα χρησιμοποιηθούν στα επόμενα κεφάλαια. Μέσα από το παράδειγμα αυτό αποσαφηνίζεται το πρακτικό πρόβλημα που καλείται να επιλύσει το Cerbex, ενώ γίνεται εισαγωγή στις θεμελιώδεις αρχές της λειτουργίας του.
- **Κεφάλαιο 4:** Σε αυτό το κεφάλαιο περιγράφεται η αρχιτεκτονική και ο σχεδιασμός του συστήματος. Αναλύονται τα δομικά του τμήματα, η εσωτερική τους διάδραση, καθώς και οι μηχανισμοί εισαγωγής και παρακολούθησης κλήσεων. Παρουσιάζονται επίσης οι σχεδιαστικές επιλογές που έγιναν ώστε να επιτευχθεί ισορροπία ανάμεσα στην απόδοση και την επεκτασιμότητα.
- **Κεφάλαιο 5:** Σε αυτό το κεφάλαιο εστιάζουμε στις αναλύσεις που υλοποιήθηκαν με χρήση του Cerbex. Παρουσιάζονται ενδεικτικά πρόσθετα εργαλεία, όπως ο `PerfAnalyzer` και ο `TypeExtractor`, τα οποία αναδεικνύουν τις δυνατότητες της πλατφόρμας και επιδεικνύουν πώς μπορεί να αξιοποιηθεί η αρθρωτή αρχιτεκτονική της για διαφορετικούς τύπους μελετών.
- **Κεφάλαιο 6:** Σε αυτό το κεφάλαιο παρουσιάζεται η χρήση του εργαλείου τόσο από την πλευρά της γραμμής εντολών (CLI) όσο και μέσω προγραμματιστικής διεπαφής. Δίνονται παραδείγματα εκτέλεσης και διαχείρισης των διαθέσιμων λειτουργιών, ώστε να καταστεί σαφές πώς μπορεί να ενσωματωθεί το Cerbex σε διαφορετικά σενάρια χρήσης.
- **Κεφάλαιο 7:** Σε αυτό το κεφάλαιο πραγματοποιείται η αξιολόγηση του συστήματος και παρουσιάζεται η σχετική ερευνητική εργασία που συνδέεται με το αντικείμενο της παρούσας μελέτης. Παρουσιάζονται τα αποτελέσματα από πειράματα μέτρησης επιβάρυνσης, καθώς και συγκρίσεις με άλλες υπάρχουσες προσεγγίσεις. Επιπλέον, εξετάζεται πώς το εργαλείο ανταποκρίνεται στους στόχους που τέθηκαν και πόσο αποτελεσματικά επιτυγχάνει την ισορροπία

ανάμεσα στη λειτουργικότητα και την απόδοση.Παράλληλα, εξετάζονται υπάρχοντα συστήματα και εργαλεία (όπως τα Lya, Jalangi, DynaPyt, Wasabi,Valgrind και PySecu), επισημαίνοντας τα πλεονεκτήματα και τους περιορισμούς τους, ώστε να αναδειχθεί το ερευνητικό κενό που επιχειρεί να καλύψει το Cerbex.

- **Κεφάλαιο 8:** Σε αυτό το κεφάλαιο συνοψίζονται τα συμπεράσματα της εργασίας και προτείνονται κατευθύνσεις για μελλοντική έρευνα και ανάπτυξη.

Κεφάλαιο 2

Θεωρητικό Υπόβαθρο

2.1 Στατική, δυναμική και υβριδική ανάλυση προγραμμάτων

Η στατική ανάλυση εξετάζει τον κώδικα ενός προγράμματος χωρίς να τον εκτελεί, εξάγοντας πληροφορίες όπως πιθανά σφάλματα ή νεκρό κώδικα (dead code). Η δυναμική ανάλυση, αντίθετα, παρακολουθεί τη συμπεριφορά του προγράμματος κατά την εκτέλεσή του, συλλέγοντας δεδομένα από πραγματικές εκτελέσεις (π.χ. κλήσεις συναρτήσεων, προσπελάσεις μνήμης). Η στατική ανάλυση καλύπτει θεωρητικά όλες τις πιθανές διαδρομές εκτέλεσης, ενώ η δυναμική καλύπτει μόνο τη τρέχουσα διαδρομή αλλά έχει ξεκάθαρη εικόνα για τις πραγματικές τιμές των μεταβλητών και τη διάταξη της μνήμης [48, 12]. Στην πράξη, η κάθε ανάλυση έχει συμβιβασμούς: η στατική ανάλυση μπορεί να βγάλει ψευδώς θετικά και δεν παρακολουθεί τιμές κατά το χρόνο εκτέλεσης, ενώ η δυναμική ανάλυση μπορεί να μην επισκεφθεί όλες τις διαδρομές εκτέλεσης και επιβαρύνει σημαντικά τον χρόνο εκτέλεσης [2]. Για παράδειγμα, το εργαλείο Jalangi αναφέρει περίπου 26× επιβάρυνση κατά την καταγραφή της εκτέλεσης [43].

2.2 Το σύστημα εισαγωγής (import) της Python

Η Python διαθέτει ευέλικτο μηχανισμό εισαγωγής (import) βασισμένο σε `ευρετές (finders)` και `φορτωτές (loaders)`. Κάθε φορά που καλείται import X, ελέγχεται πρώτα η κρυφή μνήμη sys.modules. Αν δεν βρεθεί, ενεργοποιείται το πρωτόκολλο εισαγωγής. Το PEP 302 εισήγαγε τους `meta-path finders` και `path hooks`: η λίστα sys.meta_path περιέχει αντικείμενα ευρετών που παρεμβαίνουν πριν από το κανονικό σύστημα sys.path [51, 46]. Κάθε ευρετής διαθέτει μια μέθοδο find_spec(fullname, path, target) (ή την find_module για παλιότερες εκδόσεις) που εντοπίζει αν μπορεί να φορτώσει το ζητούμενο άρθρωμα. Οι ευρετές επιστρέφουν ένα ModuleSpec ή έναν Loader που καθορίζει πώς θα φορτωθεί ο κώδικας. Το PEP 451 (Python 3.4) πρόσθεσε την κλάση ModuleSpec στο importlib, ώστε να περιγράφονται πλήρως οι πληροφορίες του αρθρώματος (αρχείο, φορτωτές κ.ά.) χωρίς να χρειάζεται να φορτωθεί εκ των προτέρων [46]. Συνολικά, αυτό επιτρέπει σε εφαρμογές να εγγράφουν προσαρμοσμένα άγκιστρα εισαγωγής (π.χ. sys.meta_path.insert(0, my_finder)), ώστε ο προσαρμοσμένος ευρετής να καλείται πρώτος σε κάθε εισαγωγή [51, 49, 20, 18].

2.3 Τεχνικές ενδοσκόπησης και πολιτικές ασφάλειας

Η ενδοσκόπηση (introspection) στην Python αναφέρεται στην ικανότητα ενός προγράμματος να εξετάζει και να τροποποιεί τον εαυτό του σε πραγματικό χρόνο. Η Python παρέχει εσωτερικές (built-in) συναρτήσεις (π.χ. getattr(), setattr(), hasattr()) και τη βιβλιοθήκη inspect για να ανακτά πληροφορίες σχετικά με κλάσεις, συναρτήσεις, υπογραφές μεθόδων και τα χαρακτηριστικά τους [40]. Με αυτόν τον τρόπο μπορούμε δυναμικά να ελέγχουμε ποια χαρακτηριστικά έχει ένα αντικείμενο, να δημιουργούμε περιτυλίγματα (wrappers) γύρω από μεθόδους με μηχανισμούς παρεμβολής(proxies) ή να δούμε τη στοίβα κλήσεων, κάτι που έχει παρατηρηθεί και σε γλώσσες όπως η JavaScript μέσω της χρήσης μηχανισμών παρεμβολής [4]. Παράλληλα, στο πεδίο της ασφάλειας, η Python 3.8 εισήγαγε (PEP 578) audit hooks εκτέλεσης: Συγκεκριμένα, οι sys.addaudithook και sys.audit επιτρέπουν την αναφορά σημαντικών γεγονότων κατά το χρόνο εκτέλεσης(π.χ. άνοιγμα αρχείων, δημιουργία υποδοχέων (sockets), εισαγωγή κώδικα) [6]. Ο στόχος είναι η καταγραφή και πιθανή αποτροπή κακόβουλων ενεργειών στα scripts.

2.4 Μηχανισμοί καταγραφής εκτέλεσης στην Python

Η Python παρέχει ενσωματωμένους μηχανισμούς για παρακολούθηση της εκτέλεσης μέσω των συναρτήσεων `sys.settrace` και `sys.setprofile`. Η `sys.settrace` καταγράφει γεγονότα σε επίπεδο γραμμής κώδικα (π.χ. κάθε φορά που εκτελείται μία νέα γραμμή), ενώ η `sys.setprofile` λειτουργεί σε επίπεδο κλήσης συναρτήσεων και γεγονότων C, και ως εκ τούτου χρησιμοποιείται συνήθως από εργαλεία προφίλ (profilers) και εργαλεία δυναμικής ανάλυσης [38, 37]. Σε αντίθεση με το tracing (ιχνηλάτηση), το profiling (προφίλ εκτέλεσης) δεν παρακολουθεί κάθε γραμμή κώδικα αλλά μόνο τις κλήσεις και τις επιστροφές συναρτήσεων, μειώνοντας έτσι το υπολογιστικό κόστος (overhead). Το PEP 669 εισήγαγε ένα νέο API "low impact monitoring" με σκοπό την περαιτέρω ελαχιστοποίηση του κόστους τέτοιων μηχανισμών [44].

2.5 Επεκτάσεις C και native modules

Η γλώσσα Python παρέχει τη δυνατότητα ενσωμάτωσης βιβλιοθηκών που έχουν υλοποιηθεί σε C ή C++ υπό τη μορφή *native modules*, με κατάληξη `.so` (Unix) ή `.pyd` (Windows). Η λειτουργικότητα αυτή αποτελεί κρίσιμο χαρακτηριστικό του οικοσυστήματος της Python, καθώς επιτρέπει την ανάπτυξη βιβλιοθηκών υψηλής απόδοσης και τη γεφύρωση με κώδικα συστήματος ή βιβλιοθήκες χαμηλού επιπέδου. Χαρακτηριστικά παραδείγματα συνιστούν τόσο οι ενσωματωμένες βιβλιοθήκες της γλώσσας (π.χ. `math`, `json`) όσο και ιδιαίτερα διαδεδομένα πακέτα τρίτων, όπως τα `NumPy`, `Pandas` και `TensorFlow`, τα οποία βασίζονται σε εκτενή υλοποίηση κρίσιμων λειτουργιών σε C/C++ [29].

Τα αρθρώματα αυτού του τύπου φορτώνονται μέσω του μηχανισμού `importlib.machinery.ExtensionFileLoader` , παρακάμπτοντας τη συνήθη διαδικασία ερμηνείας Python bytecode [33, 30]. Ως εκ τούτου, απουσιάζει από αυτά αναγνώσιμη μορφή bytecode, γεγονός που καθιστά μη εφαρμόσιμες τις μεθόδους στατικής ανάλυσης πηγαίου κώδικα, αλλά και δυσχεραίνει την άμεση αξιοποίηση μηχανισμών δυναμικής ενσωμάτωσης κώδικα παρακολούθησης οι οποίοι στηρίζονται στην αντικατάσταση συναρτήσεων σε επίπεδο Python.

Για την παρακολούθηση της εκτέλεσης τέτοιων αρθρωμάτων απαιτείται η αξιοποίηση χαμηλού επιπέδου διεπαφών που παρέχει ο ίδιος ο διερμηνέας CPython. Ειδικότερα, μέσω του μηχανισμού `sys.setprofile` καθίσταται δυνατή η ανίχνευση γεγονότων που σχετίζονται με κλήσεις και επιστροφές συναρτήσεων υλοποιημένων σε C (`c_call` και `c_return`) [37]. Με τον τρόπο αυτό, το *επίπεδο ανάλυσης* αποκτά τουλάχιστον ορατότητα στην αλληλεπίδραση μεταξύ του κώδικα Python

και του χαμηλότερου επιπέδου αρθρώματος σε C, παρότι η εσωτερική λογική του τελευταίου παραμένει ουσιαστικά αδιαφανής.

Η προσέγγιση αυτή συνεπάγεται έναν αναπόφευκτο συμβιβασμό: αν και δεν είναι εφικτή η εις βάθος ανάλυση της υλοποίησης σε C, εξασφαλίζεται πρακτική δυνατότητα παρακολούθησης της χρήσης των βιβλιοθηκών αυτών με περιορισμένο κόστος απόδοσης. Αυτό καθιστά εφικτή την ενσωμάτωση δυναμικών τεχνικών ανάλυσης σε περιβάλλοντα που στηρίζονται σε κρίσιμες native βιβλιοθήκες, διατηρώντας ταυτόχρονα την αποδοτικότητα του εκτελούμενου προγράμματος.

2.6 Thread-local δεδομένα

Κατά την παρακολούθηση κλήσεων συναρτήσεων σε πολυνηματικά (multi-threaded) προγράμματα, είναι σημαντικό οι πληροφορίες να αποθηκεύονται μεμονωμένα ανά νήμα ώστε να αποφευχθεί η ανάμιξη δεδομένων. Η Python παρέχει την κλάση `threading.local`, η οποία δημιουργεί "thread-local storage" — κάθε νήμα βλέπει ανεξάρτητες τιμές για τις ίδιες μεταβλητές [39]. Έτσι, διαφορετικά νήματα μπορούν να αποθηκεύουν πλαίσιο (context) (π.χ. ID χρήστη ή session token) χωρίς να αλληλεπιδρούν μεταξύ τους.

Αυτό είναι κρίσιμο για την ορθή λειτουργία εργαλείων καταγραφής εκτέλεσης (logging/tracing) σε πολυνηματικά προγράμματα, αλλά και σε εξυπηρετητές που χειρίζονται πολλαπλούς χρήστες ταυτόχρονα. Για παράδειγμα, σε έναν εξυπηρετητή ιστού μπορούμε να αποθηκεύουμε προσωρινά πληροφορίες για τον τρέχοντα χρήστη χωρίς να κινδυνεύουμε να τις μοιραστούμε με άλλα αιτήματα.

Ωστόσο, τα `thread-local` δεδομένα δεν επαρκούν για ασύγχρονες εφαρμογές που βασίζονται στο event loop (asyncio). Σε αυτή την περίπτωση χρησιμοποιείται το άρθρωμα `contextvars` (εισήχθη στην Python 3.7) [42], το οποίο παρέχει παρόμοιο μηχανισμό, αλλά λειτουργεί σωστά με υπορουτίνες και ασύγχρονες διεργασίες, εξασφαλίζοντας ότι κάθε διεργασία βλέπει το δικό της απομονωμένο πλαίσιο (context).

2.7 Τεχνικές περιτύλιξης συναρτήσεων και διακοσμητών

Στην Python, ένας *διακοσμητής (decorator)* είναι μια συνάρτηση που δέχεται ως είσοδο μια άλλη συνάρτηση ή κλάση και επιστρέφει μια νέα, συνήθως τροποποιημένη, εκδοχή της. Με αυτόν τον τρόπο επιτρέπει την προσθήκη ή μεταβολή συμπεριφορών χωρίς αλλαγές στον αρχικό κώδικα [45].

Η Python υποστηρίζει τη δημιουργία "περιτυλιγμένων" συναρτήσεων (wrappers) μέσω διακοσμητών, ώστε να εκτελείται πρόσθετος κώδικας πριν ή μετά την κανονική κλήση. Η βιβλιοθήκη `functools` παρέχει τη συνάρτηση `update_wrapper` που διατηρεί μεταδεδομένα (metadata) (όπως `__name__`, `__doc__`, `__annotations__`) κατά την περιτύλιξη [31]. Επιπλέον, η βιβλιοθήκη `inspect` επιτρέπει την ανάκτηση και διατήρηση των υπογραφών συναρτήσεων μέσω του `inspect.signature`, ώστε τα περιτυλίγματα να είναι διαφανή για τον προγραμματιστή [40].

Οι διακοσμητές αποτελούν βασικό εργαλείο της μεταπρογραμματιστικής (metaprogramming) ικανότητας της Python. Χρησιμοποιούνται ευρέως σε πλαίσια όπως το Flask για τον ορισμό HTTP routes (`@app.route`) ή στο Django για έλεγχο πρόσβασης σε views (`@login_required`). Ένα χαρακτηριστικό παράδειγμα είναι και το `@functools.lru_cache`, που επιτρέπει caching αποτελεσμάτων συναρτήσεων χωρίς να αλλάξει ο κώδικας της συνάρτησης.

Σε επίπεδο αρχιτεκτονικής, οι διακοσμητές μπορούν να συγκριθούν με το γνωστό *Decorator design pattern* [11], ενώ παράλληλα σχετίζονται με έννοιες του θεματοστρεφή προγραμματισμού (aspect-oriented programming, AOP), όπου cross-cutting concerns όπως logging, authorization, caching ή metrics υλοποιούνται εκτός της βασικής επιχειρησιακής λογικής αλλά εφαρμόζονται "τυλιγμένα" γύρω από αυτήν [14, 17]. Αυτό κάνει τον κώδικα πιο αρθρωτό και επεκτάσιμο, χωρίς

να θυσιάζεται η καθαρότητα της κύριας λογικής. Έτσι, οι διακοσμητές δεν είναι μόνο εργαλείο συντομίας αλλά και σχεδιαστικό μοτίβο που διευκολύνει τη διαχείριση επαναλαμβανόμενων λειτουργιών σε μεγάλα συστήματα.

Κεφάλαιο 3

Παράδειγμα Εφαρμογής και Βασικές Έννοιες

Σε αυτό το κεφάλαιο θα παρουσιάσουμε ένα χαρακτηριστικό παράδειγμα που χρησιμοποιεί δημοφιλείς βιβλιοθήκες της Python και θα δείξουμε πώς το εργαλείο μας καταγράφει τη συμπεριφορά του. Ξεκινάμε με ένα απλό πρόγραμμα Python που χρησιμοποιεί τις βιβλιοθήκες requests και json για να αντλήσει και να αναλύσει κάποια δεδομένα. Αρχικά παρουσιάζουμε τη βασική (χωρίς κώδικα παρακολούθησης (uninstrumented)) εκτέλεση του κώδικα και στη συνέχεια παρουσιάζουμε την έξοδο της εκτέλεσης που παράγεται απο το εργαλείο με κώδικα παρακολούθησης. Στη συνέχεια εξηγούμε, σε υψηλό επίπεδο, πώς πραγματοποιείται η ενσωμάτωση κώδικα παρακολούθησης (μέσω άγκιστρων εισαγωγής και περιτύληξης συναρτήσεων). Αυτή η ενότητα παρέχει μόνο το βασικό υπόβαθρο που απαιτείται για την κατανόηση του παραδείγματος - περαιτέρω λεπτομέρειες θα αναληθούν σε μεταγενέστερα κεφάλαια.

3.1 Βασικές Έννοιες

Οι σύγχρονες εφαρμογές Python βασίζονται σε ολοένα και πιο πολύπλοκα δίκτυα βιβλιοθηκών τρίτων και πακέτων, γεγονός που καθιστά την ανάλυση προγραμμάτων και την επιβολή πολιτικών ασφαλείας πιο απαιτητική [19]. Για παράδειγμα, κάποιος μπορεί να χρησιμοποιήσει τη βιβλιοθήκη `requests` [41] για να κάνει HTTP αιτήσεις, και τη βιβλιοθήκη `json` [34] για να διαχειριστεί JSON δεδομένα.

Το εργαλείο μας παρεμβάλλει με διαφανή τρόπο τις κλήσεις τέτοιων βιβλιοθηκών κατά τον χρόνο εκτέλεσης, χωρίς να απαιτείται τροποποίηση του αρχικού τους κώδικα. Σε επίπεδο υλοποίησης, καταχωρούμε ειδικά άγκιστρα εισαγωγής (import hooks) μέσω ενός μηχανισμού `MetaPathFinder` [51] που προστίθεται στο `sys.meta_path`, και συμπληρωματικά χρησιμοποιούμε αντικατάσταση της συνάρτησης `__import__` [32], ώστε να καταγράφονται και οι εισαγωγές που δεν περνούν από τον μηχανισμό εύρεσης, όπως περιπτώσεις caching ή φορτώσεις μέσω του C API της Python.

Κάθε φορά που φορτώνεται ένα άρθρωμα, αν αυτό περιλαμβάνεται στους στόχους ανάλυσης, εφαρμόζεται προσαρμοσμένος φορτωτής (custom loader) που εκτελεί τον κώδικα του αρθρώματος και εγκαθιστά περιτυλίγματα (wrappers) γύρω από τις συναρτήσεις του. Τα περιτυλίγματα αυτά ενεργοποιούν κώδικα ανάλυσης που έχει ορίσει ο χρήστης πριν και μετά την εκτέλεση κάθε συνάρτησης — για παράδειγμα, την καταγραφή της κλήσης, τη μέτρηση χρόνου, ή την επιβολή πολιτικών ασφάλειας.

Επιπλέον, αξιοποιούμε τον μηχανισμό `sys.setprofile` για την καταγραφή κλήσεων σε χαμηλό

επίπεδο (όπως κλήσεις σε εσωτερικές συναρτήσεις ή C επεκτάσεις), με στόχο την πληρέστερη δυνατή κάλυψη του χρόνου εκτέλεσης. Η σχεδίαση του μηχανισμού εξισορροπεί ακρίβεια και απόδοση, καθώς εφαρμόζεται περιτύλιγμα μόνο στα αρθρώματα που έχουν δηλωθεί ως στόχοι, ενώ παράλληλα αποφεύγεται πλεονάζουσα επεξεργασία μέσω κρυφής μνήμης.

Παρακάτω, θα δείξουμε την εκτέλεση ενός μικρού προγράμματος χωρίς και με τη χρήση του εργαλείου μας.

3.2 Κώδικας Παραδείγματος

Όπως και στην αρχή θα χρησιμοποιήσουμε τις βιβλιοθήκες 'requests' και 'json' για να κάνουμε ένα αίτημα μέσω HTTP και να αποσυνθέσουμε την απάντηση χρησιμοποιώντας μορφή JSON.

Παρόμοια απλά παραδείγματα χρησιμοποιούνται συχνά στη βιβλιογραφία για να αναδειχθούν οι δυνατότητες εργαλείων δυναμικής ανάλυσης, όπως για παράδειγμα στο Jalangi και το Lya για JavaScript [43, 24] ή στο DynaPyt για Python [7]. Ο σκοπός τέτοιων παραδειγμάτων δεν είναι να προσομοιώσουν μεγάλης κλίμακας εφαρμογές, αλλά να δείξουν με διαφανή τρόπο πώς οι μηχανισμοί κώδικα παρακολούθησης καταγράφουν εισαγωγές και κλήσεις συναρτήσεων σε ένα γνώριμο περιβάλλον.

```python
#requests_example.py
import requests
import json

def main():
    response = requests.get(
        "https://httpbin.org/anything",
        params={"msg": "hello", "n": 42}
    )
    print("Status code:", response.status_code)

    data = json.loads(response.text)
    print("Parsed JSON data:", data)

if __name__ == "__main__":
    main()
```

Κώδικας 3.1: HTTP request and JSON parsing in Python

```
$ python3 request_example.py
Status code: 200
Parsed JSON data: {
  'args': {'msg': 'hello', 'n': '42'},
  'data': '',
  'files': {},
  'form': {},
  'headers': {
    'Accept': '*/*',
    'Accept-Encoding': 'gzip, deflate, br, zstd',
    'Host': 'httpbin.org',
    'User-Agent': 'python-requests/2.31.0',
    'X-Amzn-Trace-Id': '<trace-id>'
  },
  'json': None,
  'method': 'GET',
  'origin': 'X.X.X.X',
  'url': 'https://httpbin.org/anything?msg=hello&n=42'
}
```

Κώδικας 3.2: Terminal output from executing the request_example script

3.2.1 Εκτέλεση με κώδικα παρακολούθησης

Για να αναλύσουμε τον παραπάνω κώδικα θα χρησιμοποιήσουμε το εργαλείο Cerbex. Συγκεκριμένα, με το εργαλείο θα δούμε ποιες βιβλιοθήκες εισήχθησαν και τι συναρτήσεις χρησιμοποιήθηκαν. Αφού εκτελεστεί θα παραχθούν τρία αρχεία (events.json, dependencies.json, allowlist.json). Ο κώδικας που χρειάζεται για να εφαρμοστεί η ανάλυση που αναφέραμε είναι ο παρακάτω:

```
# --- Hook setup ---
from Cerbex.hook_loader import install_hooks

# Load hooks in 'learn' mode: this writes dependencies.json,
    events.json, allowlist.json.
install_hooks(
    config_path='config.json',
    analyses=[],
    mode='learn'
)
# Run the target instrumented script
import request_example
request_example.main()
```

Κώδικας 3.3: Runner script to execute request_example in learn mode

Το αρχείο config.json για το παράδειγμα μας.

```
{
    "targets": [
      "requests_example",
      "requests",
      "json"
    ]
  }
```

Κώδικας 3.4: Config.json file

Βλέπουμε ότι ο κώδικας πραγματοποιεί τις εξής λειτουργίες:

- Στη **γραμμή 1** εισάγουμε από το αρχείο `hook_loader` τη συνάρτηση `install_hooks`.
- Στις **γραμμές 5–9** καλείται η `install_hooks`, η οποία ενεργοποιεί τον μηχανισμό σε λειτουργία μάθησης (`learn`) και ορίζει την τοποθεσία του αρχείου `config.json`, όπου περιλαμβάνονται οι βιβλιοθήκες των οποίων θέλουμε να γίνει ανάλυση των συναρτήσεων τους.
- Στη **γραμμή 11** εισάγεται το αρχείο `request_example`, το οποίο δείξαμε στον Πίνακα 3.1, και στη συνέχεια εκτελείται καλώντας τη συνάρτηση `main()`.

Παρακάτω παραθέτουμε αποσπάσματα από τα αρχεία που παρήχθησαν κατά την ανάλυση, ώστε να επιβεβαιώσουμε τη λειτουργία του μηχανισμού καταγραφής.

3.2.2 Αποτελέσματα με κώδικα παρακολούθησης

Παρατηρείται ότι το αρχείο `events.json` περιλαμβάνει και πολλές ακόμη εγγραφές που αφορούν εσωτερικές ή έμμεσες βιβλιοθήκες, όπως `urllib3`, `ssl`, `http.client`, `email.parser`, καθώς και διάφορα αρθρώματα του standard library ή C επεκτάσεις (π.χ. `_socket`, `_json`, `struct`).

Αυτές οι καταγραφές προκύπτουν επειδή ο μηχανισμός κώδικα παρακολούθησης ακολουθεί αναδρομικά τις εξαρτήσεις των αρθρωμάτων που καλεί το πρόγραμμα, όπως και εκείνες που φορτώνονται δυναμικά κατά την εκτέλεση. Έτσι διασφαλίζεται πλήρης ορατότητα στις πραγματικές κλήσεις που εκτελούνται κατά το χρόνο εκτέλεσης, ακόμη και όταν αυτές δεν είναι ορατές στον αρχικό πηγαίο κώδικα του χρήστη.

Για λόγους παρουσίασης, στο παρόν παράδειγμα εστιάζουμε μόνο στα γεγονότα που σχετίζονται άμεσα με τον κώδικα του χρήστη, ενώ το πλήρες `events.json` διατίθεται για αναλυτική μελέτη. Αντίστοιχα, θα εστιάσουμε και στα άλλα δύο αρχεία.

Για το αρχείο παραδείγματος `requests_example.py`[3.1], παρατηρούμε τα εξής:

- Κλήση της συνάρτησης `main()` και οι εισαγωγές των `json` και `requests`:

```
  "requests_example": {
    "call:main": true,
    "return:main": true,
    "import:json": true,
    "import:requests": true
  },
```

- Χρήση της `requests.get()` και δυναμικές εισαγωγές από την ίδια τη βιβλιοθήκη:

```
  "requests": {
    "call:get": true,
    "return:get": true,
    "import:sessions": true,
    "import:exceptions": true,
    "import:urllib3": true,
    "import:status_codes": true,
    "import:__version__": true,
    ...
  },
```

- Χρήση της βιβλιοθήκης `json`, με κλήσεις στις συναρτήσεις `loads()` και `dump()`, οι οποίες στηρίζονται εσωτερικά στο C άρθρωμα `_json`:

```
"json": {
    "return:loads": true,
    "call:loads": true,
    "return:dump": true,
    "call:dump": true
  },
"_json": {
  "call:encode_basestring_ascii": true,
  "return:encode_basestring_ascii": true
}
```

- Εκτελέστηκαν επίσης πολλές ενσωματωμένες συναρτήσεις της Python, όπως `str.upper()`, `list.append()`, `len()`, `print()`, κ.ά., οι οποίες καταγράφηκαν σε δύο διαφορετικά namespaces:

```
  "__builtins__": {
    "call:append": true,
    "return:upper": true,
    ...
  },
  "builtins": {
    "call:len": true,
    "call:print": true,
    ...
  }
```

Χάρη στην ενιαία προσέγγιση ιχνηλάτησης εισαγωγών, το εργαλείο καταγράφει με ακρίβεια τόσο βιβλιοθήκες του χρήστη όσο και βιβλιοθήκες της Python κατά το χρόνο εκτέλεσης, χωρίς περαιτέρω επεξεργασία ή παραλείψεις. Αυτή η πληροφορία μπορεί να αξιοποιηθεί σε αναλύσεις τύπων, ιχνηλάτησης κλήσεων ή εφαρμογή πολιτικών ασφάλειας.

Εξαγωγή Εξαρτήσεων (`dependencies.json`)
Το εργαλείο μας κατασκευάζει έναν γράφο εξαρτήσεων βάσει των καταγεγραμμένων εισαγωγών και τον αποθηκεύει στο αρχείο `dependencies.json`.
Ενδεικτικά για το παράδειγμα `requests_example.py` περιλαμβάνει:

```
"requests_example": [
  "json",
  "requests"
],
"requests": [
      "",
      "__version__",
      "api",
      "chardet",
      "exceptions",
      "logging",
      "models",
      "sessions",
      "ssl",
      "status_codes",
      "urllib3",
      "urllib3.exceptions",
      "warnings"
    ]
```

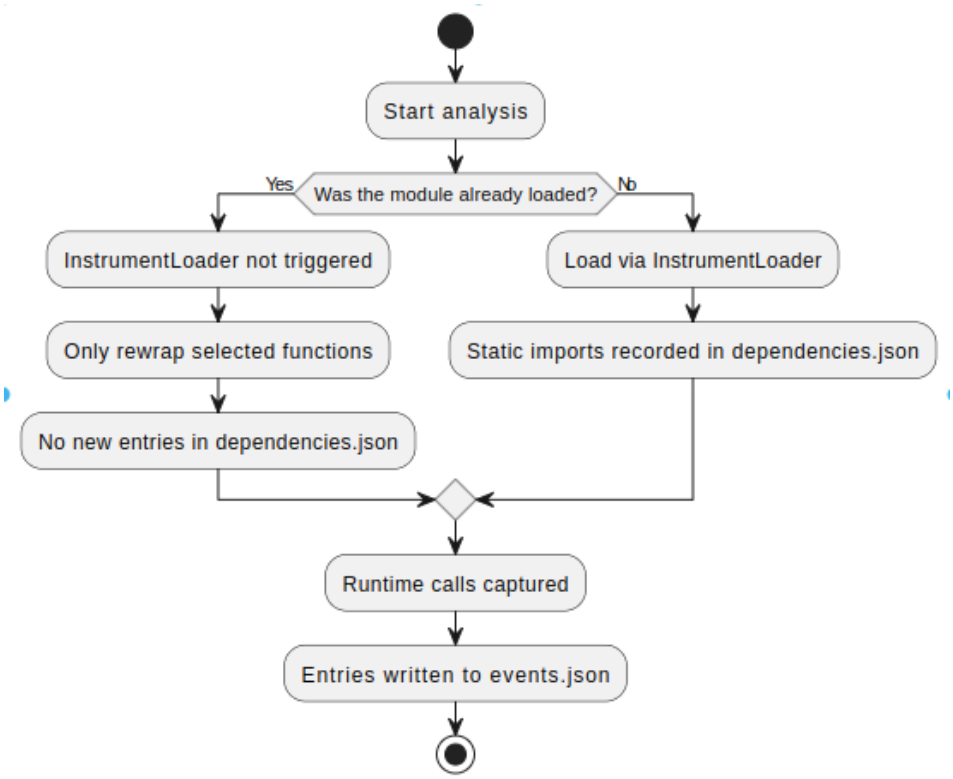
Λίστα Επιτρεπόμενων (`allowlist.json`)
Υποστηρίζεται η επιβολή πολιτικών μέσω allowlist, όπου στο αρχείο `allowlist.json` ορίζουμε ποια αρθρώματα και συναρτήσεις επιτρέπεται να καλούνται. Ενδεικτικά για το παράδειγμα:

```
"requests_example": [
  "json",
  "main",
  "requests"
],
"requests": [
      "",
      "__version__",
      "api",
      "chardet",
      "exceptions",
      "get",
      "logging",
      "models",
      "sessions",
      "ssl",
      "status_codes",
      "urllib3",
      "urllib3.exceptions",
      "warnings"
    ],
    "json": [
      "dump",
      "loads"
    ],
"_json": [
      "encode_basestring_ascii"
    ]
```

Σε `λειτουργία επιβολής`, κάθε συμβάν (εισαγωγή ή κλήση) ελέγχεται *σε πραγματικό χρόνο* έναντι της λίστας επιτρεπόμενων που έχει φορτωθεί στη μνήμη, και σε περίπτωση μη επιτρεπόμενης ενέργειας διακόπτεται.

Απουσία εσωτερικών εξαρτήσεων της `json`
Παρά το γεγονός ότι το άρθρωμα `json` εμφανίζεται στο αρχείο `dependencies.json` ως δηλωμένη εξάρτηση του `requests_example`, οι ενδογενείς του εισαγωγές (`_json`, `re`, `decimal`) δεν αποτυπώνονται. Το φαινόμενο αυτό ερμηνεύεται από το γεγονός ότι το άρθρωμα είχε ήδη ενσωματωθεί στο εκτελεστικό περιβάλλον πριν από την εγκατάσταση του `InstrumentLoader`. Στη συνέχεια, εφαρμόστηκε επανατύλιξη αποκλειστικά σε επιλεγμένες συναρτήσεις (`loads`, `dump`) μέσω της `rewrap_existing_targets`, χωρίς να ανακληθεί εκ νέου η διαδικασία φόρτωσης του αρθρώματος υπό τον μηχανισμό οργάνωσης εισαγωγών.

Δεδομένου ότι το `dependencies.json` παράγεται βάσει συμβάντων `on_import`, τα οποία ενεργοποιούνται αποκλειστικά κατά τη στιγμή της αρχικής φόρτωσης, οι ήδη επιλυθείσες εξαρτήσεις του `json` δεν καταγράφηκαν. Αντιθέτως, το `events.json` περιλαμβάνει δυναμικές κλήσεις προς συναρτήσεις του `_json`, καθώς οι εν λόγω εκτελέσεις εντοπίστηκαν σε πραγματικό χρόνο. Η ενσωμάτωση των στατικών εξαρτήσεων στο `dependencies.json` θα απαιτούσε είτε εξαναγκασμένη επαναφόρτωση του αρθρώματος `json` υπό τον `InstrumentLoader`, είτε άμεση εκτέλεση του `exec_module` επί του κώδικά του, ώστε να αναλυθούν εκ νέου οι εσωτερικές εισαγωγές.



Γράφημα 3.1: Loaded vs Unloaded modules workflow

Συμπέρασμα
Αξίζει να σημειωθεί ότι με τις ελάχιστες γραμμές που προσθέσαμε στο παράδειγμα για την προσθήκη του εργαλείου καταφέραμε να έχουμε εικόνα για τις εισαγωγές βιβλιοθηκών που έγιναν στο πρόγραμμα, τις κλήσεις διαφόρων συναρτήσεων, τις εξαρτήσεις των αρθρωμάτων που εισήχθησαν καθώς και επίσης τη δημιουργία του αρχείου allowlist. Όλα αυτά χωρίς το αποτέλεσμα της εκτέλεσης να επηρεαστεί. Ωστόσο, όπως θα δούμε στο κεφάλαιο 7, αυτό δεν γίνεται με σημαντικό κόστος στην ταχύτητα εκτέλεσης.

3.2.3 Εκτέλεση σε Λειτουργία Επιβολής (Enforce Mode)

Στην ενότητα 3.2.1 είδαμε πώς το εργαλείο καταγράφει εισαγωγές και κλήσεις συναρτήσεων σε *λειτουργία μάθησης* (`learn mode`), παράγοντας τα αρχεία `events.json`, `dependencies.json` και `allowlist.json`. Στη συνέχεια μπορούμε να ενεργοποιήσουμε τη *λειτουργία επιβολής* , ώστε κατά την εκτέλεση του προγράμματος να επιτρέπονται μόνο τα γεγονότα (εισαγωγές ή κλήσεις συναρτήσεων) που βρίσκονται καταγεγραμμένα στο `allowlist.json`. Κάθε μη επιτρεπόμενη ενέργεια διακόπτει άμεσα την εκτέλεση του προγράμματος.

Παρακάτω φαίνεται το σενάριο εκτέλεσης του παραδείγματός μας σε λειτουργία επιβολής:

```
from Cerbex.hook_loader import install_hooks

# Load hooks in 'enforce' mode: all imports/calls must match
    allowlist.json
install_hooks(
    config_path='config.json',
    analyses=[],
    mode='enforce',
    allowlist_path="allowlist.json"
)

# Run the target instrumented script
import request_example
request_example.main()
```

Κώδικας 3.5: Runner script to execute request_example in enforce mode

Για να δοκιμάσουμε την πολιτική επιβολής, τροποποιούμε το αρχικό πρόγραμμα εισάγοντας επιπλέον το άρθρωμα `math`, το οποίο δεν περιλαμβάνεται στο `allowlist.json`:

```python
#requests_example.py
import requests
import json
import math   # Not present in allowlist.json

def main():
    response = requests.get(
        "https://httpbin.org/anything",
        params={"msg": "hello", "n": 42}
    )
    print("Status code:", response.status_code)

    data = json.loads(response.text)
    print("Parsed JSON data:", data)

if __name__ == "__main__":
    main()
```

Κώδικας 3.6: Modified request_example with disallowed import

Η εκτέλεση του παραπάνω κώδικα αποτυγχάνει με το ακόλουθο μήνυμα σφάλματος:

```
Traceback (most recent call last):
  File "runner.py", line 11, in <module>
  File ".../cerbex/importer.py", line 208, in fallback_import
    return orig_import(name, globals, locals, fromlist, level)
  File ".../cerbex/importer.py", line 135, in exec_module
    exec(code, module.__dict__, module.__dict__)
  File "request_example.py", line 4, in <module>
    import math
  File ".../cerbex/importer.py", line 207, in fallback_import
    hook_mgr.on_import(parent, name)
  File ".../cerbex/hook_manager.py", line 68, in on_import
    raise ImportError(f"Import of {name} not allowed in module {parent}")
ImportError: Import of math not allowed in module request_example
```

Κώδικας 3.7: Terminal output in enforce mode with disallowed import

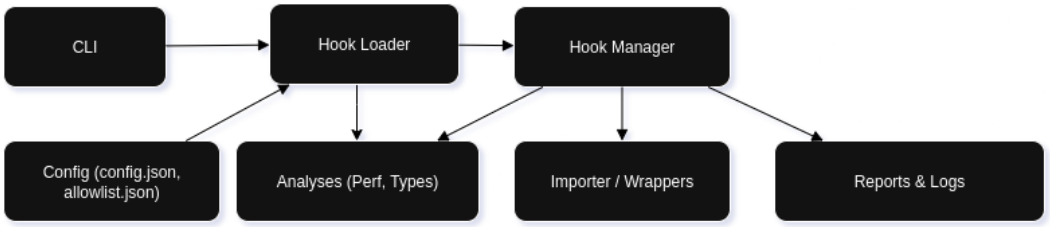
Παρατηρούμε ότι το εργαλείο εντόπισε την εισαγωγή της βιβλιοθήκης `math` και διέκοψε την εκτέλεση, καθώς αυτή δεν περιλαμβάνεται στη λίστα επιτρεπόμενων (allowlist). Με τον τρόπο αυτό αποδεικνύεται ότι η λειτουργία επιβολής μπορεί να λειτουργήσει ως μηχανισμός *πραγματικού χρόνου* για τον περιορισμό των βιβλιοθηκών και συναρτήσεων που χρησιμοποιούνται σε ένα πρόγραμμα, εφαρμόζοντας έτσι πολιτικές ασφαλείας.

Κεφάλαιο 4

Αρχιτεκτονική και Σχεδίαση

4.1 Επισκόπηση της αρχιτεκτονικής του συστήματος

Η αρχιτεκτονική του Cerbex ακολουθεί μια **αρθρωτή και πολυεπίπεδη προσέγγιση**, η οποία διαχωρίζει τις ευθύνες των επιμέρους τμημάτων, διατηρώντας παράλληλα την επεκτασιμότητα και την αποδοτικότητα του εργαλείου. Στον πυρήνα της, η σχεδίαση επιτρέπει την παρακολούθηση και τον περιορισμό της συμπεριφοράς ενός προγράμματος Python κατά τον χρόνο εκτέλεσης, αξιοποιώντας τον μηχανισμό εισαγωγής και την παρακολούθηση σε κλήσεις συναρτήσεων.



Γράφημα 4.1: High level architecture of Cerbex.

Η αρχιτεκτονική οργανώνεται γύρω από επτά βασικά τμήματα:

1. **Διεπαφή Γραμμής Εντολών (CLI)** Αποτελεί το κύριο σημείο εισόδου για τον χρήστη. Μέσω παραμέτρων εντολών (π.χ. `-mode learn`, `-mode enforce`) καθορίζει τη λειτουργία του συστήματος, τα αρχεία ρυθμίσεων, τις ενεργές αναλύσεις και τους καταλόγους εξόδου. Η υλοποίηση βρίσκεται στο αρχείο `cli.py`.

2. **Διαμόρφωση (Config)** Περιλαμβάνει τα αρχεία ρυθμίσεων σε μορφή JSON. Το `config.json` καθορίζει τα αρθρώματα-στόχους που θα παρακολουθούνται, ενώ το `allowlist.json` ορίζει ποιες εισαγωγές και κλήσεις επιτρέπονται σε λειτουργία επιβολής. Τα αρχεία αυτά φορτώνονται κατά την εκκίνηση από τον Hook Loader.

3. **Φορτωτής Άγκιστρων (Hook Loader)** Είναι υπεύθυνος για την προετοιμασία του περιβάλλοντος εκτέλεσης. Φορτώνει τα αρχεία διαμόρφωσης και τις λίστες επιτρεπόμενων κλήσεων και εγκαθιστά τους μηχανισμούς παρακολούθησης (άγκιστρα εισαγωγής, άγκιστρα του εργαλείου προφίλ). Η βασική συνάρτηση `install_hooks` (στο `hook_loader.py`) δημιουργεί

το αντικείμενο `HookManager` και καταχωρεί τις απαραίτητες δομές καθαρισμού για την παραγωγή αναφορών.

4. **Διαχειριστής Άγκιστρων (Hook Manager)** Αποτελεί τον κεντρικό συντονιστή του συστήματος (`hook_manager.py`). Διαχειρίζεται όλα τα γεγονότα εκτέλεσης (εισαγωγές, κλήσεις συναρτήσεων, επιστροφές) και τα προωθεί προς τα εγγεγραμμένα πρόσθετα εργαλεία ανάλυσης. Επιπλέον, διατηρεί τον γράφο εξαρτήσεων, την λίστα επιτρεπτών και το αρχείο γεγονότων. Σε λειτουργία επιβολής ελέγχει αν οι ενέργειες είναι επιτρεπτές, ενώ σε λειτουργία εκμάθησης καταγράφει τα γεγονότα.

5. **Μηχανισμός Εισαγωγής και Περιτυλίγματα (Importer / Wrappers)** Υλοποιεί προσαρμοσμένους `MetaPathFinder` και `Loader` (στο `importer.py`), που παρεμβάλλονται κατά τη φόρτωση των αρθρωμάτων. Οι συναρτήσεις και οι μέθοδοι τυλίγονται μέσω της συνάρτησης `wrap_value` (στο `utils.py`), ώστε οι κλήσεις τους να περνούν μέσα από τα άγκιστρα του `HookManager`.

6. **Τα Πρόσθετα Εργαλέια (plugins) Ανάλυσης** Υλοποιούνται ως υποκλάσεις της κλάσης `Analysis` (στο `analysis.py`) και παρακολουθούν γεγονότα όπως `on_call`, `on_return` και `on_import`. Στην τρέχουσα έκδοση περιλαμβάνονται δύο πρόσθετα εργαλεία: ο `PerfAnalyzer`, που μετρά χρόνους εκτέλεσης συναρτήσεων, και ο `TypeExtractor`, που καταγράφει τύπους επιστροφής.

7. **Αναφορές και Καταγραφές (Reports & Logs)** Στο τέλος της εκτέλεσης, ο `HookManager` μέσω της συνάρτησης `write_reports()` παράγει τα τελικά αρχεία εξόδου: `events.json`, `dependencies.json`, `allowlist.json`, καθώς και αρχεία καταγραφής όπως `perf.log` και `types.log`.

Σε υψηλό επίπεδο, η διεπαφή CLI εκκινεί τη διαδικασία, ο Hook Loader προετοιμάζει το περιβάλλον, ο Hook Manager συντονίζει τα γεγονότα, οι μηχανισμοί εισαγωγής και τυλιγμάτων παρεμβάλλονται στον κώδικα του χρήστη, τα πρόσθετα εργαλεία ανάλυσης καταναλώνουν τα γεγονότα, και τελικά παράγονται αναφορές και αρχεία καταγραφής.

4.2 Βασικά τμήματα και υλοποίηση

Στην ενότητα αυτή παρουσιάζονται αναλυτικά τα βασικά τμήματα της αρχιτεκτονικής του Cerbex, όπως φαίνονται στο Σχήμα 4.1. Κάθε τμήμα αντιστοιχεί σε συγκεκριμένο αρχείο πηγαίου κώδικα και εκτελεί διακριτές ευθύνες, ώστε να διασφαλίζεται η επεκτασιμότητα και η συντηρησιμότητα του συστήματος.

4.2.1 Διεπαφή Γραμμής Εντολών (CLI)

Η διεπαφή γραμμής εντολών (`cli.py`) αποτελεί το σημείο εισόδου του εργαλείου. Υλοποιείται με χρήση της `argparse` και δίνει τη δυνατότητα στον χρήστη να επιλέξει τη λειτουργία (`learn` ή `enforce`), να καθορίσει αρχεία διαμόρφωσης και καταλόγους εξόδου, καθώς και να ενεργοποιήσει συγκεκριμένα πρόσθετα εργαλεία ανάλυσης.

Μετά την ανάλυση των παραμέτρων, η CLI καλεί την `install_hooks()` του `hook_loader.py`. Έτσι διασφαλίζεται ότι οι μηχανισμοί παρακολούθησης είναι ενεργοί από την έναρξη της εκτέλεσης του προγράμματος-στόχου.

4.2.2 Διαμόρφωση (Config)

Η διαμόρφωση υλοποιείται ως εξωτερικά αρχεία JSON:

- `config.json`: ορίζονται τα αρθρώματα που θέλουμε να παρακολουθήσουμε.
- `allowlist.json`: περιλαμβάνει τις επιτρεπτές εξαρτήσεις και κλήσεις συναρτήσεων, όπως έχουν παρατηρηθεί από τη λειτουργία εκμάθησης. Σε λειτουργία επιβολής, το αρχείο αυτό αποτελεί το σημείο αναφοράς για την εφαρμογή περιορισμών.

Με τον τρόπο αυτό, ο χρήστης μπορεί να προσαρμόσει το εργαλείο χωρίς να επέμβει στον πηγαίο κώδικα. Ο `Hook Loader` αναλαμβάνει να φορτώσει τα αρχεία αυτά και να τα μετατρέψει σε δομές δεδομένων που χρησιμοποιεί ο `HookManager`.

4.2.3 Φορτωτής Hooks (Hook Loader)

Ο `Hook Loader` (`hook_loader.py`) αποτελεί τον μηχανισμό εκκίνησης του συστήματος. Η κύρια συνάρτηση `install_hooks()` δεν περιορίζεται μόνο στη δημιουργία του `HookManager`, αλλά συντονίζει όλα τα βήματα που απαιτούνται ώστε το πρόγραμμα-στόχος να εκτελεστεί με ενεργή την ενσωμάτωση κώδικα παρακολούθησης σε κλήσεις συναρτήσεων. Συγκεκριμένα, εκτελεί τέσσερις κατηγορίες ενεργειών:

1. Φόρτωση διαμόρφωσης. Αρχικά καλούνται οι βοηθητικές συναρτήσεις `_load_config()` και `_load_allowlist()`, ώστε να διαβαστούν τα αρχεία `config.json` και `allowlist.json` (σε λειτουργία επιβολής). Οι πληροφορίες αυτές μετατρέπονται σε κατάλληλες δομές δεδομένων που θα χρησιμοποιήσει ο `HookManager`.

2. Δημιουργία του Hook Manager. Στη συνέχεια δημιουργείται ένα αντικείμενο της κλάσης `HookManager`, στο οποίο περνιούνται οι στόχοι προς παρακολούθηση, η λίστα αναλύσεων, η λειτουργία (learn/enforce), καθώς και η allowlist, αν υπάρχει. Το αντικείμενο αυτό θα αποτελέσει το κεντρικό σημείο συντονισμού καθ' όλη τη διάρκεια της εκτέλεσης.

3. Εγκατάσταση μηχανισμών παρακολούθησης κώδικα. Ακολουθεί η ενεργοποίηση όλων των μηχανισμών παρακολούθησης κώδικα σε κλήσεις συναρτήσεων (instrumentation):

- Κλήση της `install_import_hook()`, η οποία εγγράφει έναν `InstrumentFinder` στο `sys.meta_path`, αντικαθιστά την ενσωματωμένη συνάρτηση `__import__`, και τυλίγει την `importlib.import_module`.
- Κλήση της `mark_loaded_c_exts()`, που καταγράφει της ήδη φορτωμένες C επεκτάσεις στο γράφο εξαρτήσεων.
- Κλήση της `rewrap_existing_targets()`, που εφαρμόζει εκ νέου το `wrap_value()` σε αρθρώματα που βρίσκονται ήδη στο `sys.modules`, ώστε να μην διαφύγουν από την παρακολούθηση συναρτήσεων.
- Ενεργοποίηση του profiler άγκιστρου μέσω `sys.setprofile(hook_mgr.c_profile)`, ώστε να παρακολουθούνται και οι κλήσεις σε συναρτήσεις C (`c_call`/`c_return`).

4. Καταχώρηση καθαρισμού και αναφορών. Τέλος, ο `Hook Loader` καταχωρεί με `atexit.register()` τη μέθοδο `write_reports()` του `HookManager`. Έτσι εξασφαλίζεται ότι κατά τον τερματισμό της εκτέλεσης, ακόμη και αν προκύψει σφάλμα, θα παραχθούν τα αρχεία `events.json`, `dependencies.json`, `allowlist.json` και οι καταγραφές των πρόσθετων εργαλείων.

Συνοπτικά. Ο `Hook Loader` λειτουργεί ως *bootstrapper* (μηχανισμός εκκίνησης) του Cerbex: φορτώνει τις ρυθμίσεις, δημιουργεί τον `HookManager`, εγκαθιστά άγκιστρα στο μηχανισμό εισαγωγής, στο εργαλείο προφίλ, και καταχωρεί τις διαδικασίες καθαρισμού. Χωρίς το στάδιο αυτό, τα υπόλοιπα μέρη του συστήματος δεν θα μπορούσαν να συνδεθούν με την εκτέλεση του προγράμματος.

4.2.4 HookManager (Κεντρικός συντονιστής)

Ο `HookManager` (`hook_manager.py`) είναι το κεντρικό σημείο συντονισμού. Διατηρεί τις εσωτερικές δομές δεδομένων:

- `self.analyses`: λίστα με τα ενεργά πρόσθετα εργαλεία ανάλυσης.
- `self.mode`: σημαία λειτουργίας (`learn` ή `enforce`).
- `self.allowlist`: λεξικό που χαρτογραφεί αρθρώματα σε επιτρεπόμενες συναρτήσεις.
- `self.dep_graph`: γράφος εξαρτήσεων (αρθρώματα → εισαγόμενα αρθρώματα).
- `self.events`: καταγραφή γεγονότων (εισαγωγές, κλήσεις, επιστροφές).
- `self._local`: `threading.local` για ασφάλεια νήματος (π.χ. σημαία `in_hook`).

Οι κύριες μέθοδοι του είναι:

- `on_import(parent, name)`: ενημερώνει τον γράφο εξαρτήσεων και καταγράφει γεγονότα εισαγωγής. Σε λειτουργία επιβολής ελέγχει την `λίστα επιτρεπτών`.
- `on_call(module, func, args, kwargs)`: καταγράφει κλήσεις συναρτήσεων ή ελέγχει την επιτρεπτότητά τους σε λειτουργία επιβολής.
- `on_return(module, func, result)`: ενημερώνει τα πρόσθετα εργαλεία μετά από κάθε επιστροφή.
- `c_profile(frame, event, arg)`: συνδέεται με το `sys.setprofile` για καταγραφή κλήσεων/επιστροφών(`c_call`/`c_return`) από συναρτήσεις C.
- `write_reports()`: παράγει τα τελικά αρχεία JSON (dependencies, events, allowlist).

Ο `HookManager` εφαρμόζει το *Observer pattern*, καθώς ειδοποιεί όλα τα πρόσθετα εργαλεία ανάλυσης για κάθε γεγονός, ενώ με το *Strategy-like* μοτίβο μεταβάλλει συμπεριφορά ανάλογα με τη λειτουργία (learn/enforce) [10].

4.2.5 Μηχανισμός Εισαγωγής και Wrappers (Importer / Wrappers)

Το υποσύστημα αυτό βρίσκεται στα αρχεία `importer.py` και `utils.py` και υλοποιεί τη δυναμική παρακολούθησης κώδικα συναρτήσεων. Αποτελείται από:

- **InstrumentFinder**: εγγράφεται στο `sys.meta_path` και καλεί τον `HookManager.on_import` για κάθε νέο άρθρωμα. Αν ταιριάζει με το στόχο, τότε μεταβιβάζει τη διαδικασία φόρτωσης στον `InstrumentLoader`.
- **InstrumentLoader**: φορτώνει και εκτελεί το άρθρωμα. Στη συνέχεια τυλίγει όλα τα globals με τη `wrap_value()`.

- **wrap_value()**: αποφασίζει αν ένα αντικείμενο πρέπει να τυλιχθεί. Για συναρτήσεις και μεθόδους δημιουργεί περιτυλίγματα που καλούν της `on_call`/`on_return`, ενώ για κλάσεις εφαρμόζει περιτύλιξη στα χαρακτηριστικά τους.
- **make_wrapper()**: υλοποιεί τον *Decorator pattern* δημιουργώντας νέες συναρτήσεις (sync ή async) που εκτελούν κώδικα παρακολούθησης γύρω από την αρχική κλήση.

Ο μηχανισμός περιλαμβάνει fallback στην `__import__` και στο `importlib.import_module`, ώστε να καλύπτονται όλα τα πιθανά σενάρια εισαγωγής, ακόμη και για αρθρώματα που ήταν ήδη φορτωμένα στο `sys.modules`.

4.2.6 Πρόσθετα Εργαλεία Ανάλυσης

Οι αναλύσεις υλοποιούνται στο `analysis.py` ως υποκλάσεις της αφηρημένης κλάσης `Analysis` που βρίσκεται στο `HookManager`. Υποστηρίζονται οι callbacks: `on_import`, `on_call`, `on_return`.

Τα διαθέσιμα πρόσθετα εργαλεία περιλαμβάνουν:

- **PerfAnalyzer**: μετρά τον χρόνο εκτέλεσης κάθε κλήσης, χρησιμοποιώντας `threading.local` για να διαχειρίζεται στοίβες χρονικών σημάνσεων ανά νήμα.
- **TypeExtractor**: καταγράφει τον τύπο επιστροφής συναρτήσεων στο `on_return`, παρέχοντας πληροφορίες για τη συμπεριφορά τύπων κατά την εκτέλεση.

Ο σχεδιασμός με βάση αφηρημένη κλάση εξασφαλίζει επεκτασιμότητα: νέα πρόσθετα εργαλεία μπορούν να προστεθούν χωρίς ριζικές αλλαγές στον πυρήνα.

4.2.7 Αναφορές και Καταγραφές (Reports & Logs)

Κατά τον τερματισμό, η `write_reports()` του `HookManager` παράγει τα εξής αρχεία:

- `events.json`: πλήρης καταγραφή γεγονότων (imports, calls, returns).
- `dependencies.json`: γράφος εξαρτήσεων αρθρωμάτων.
- `allowlist.json`: λίστα επιτρεπόμενων εξαρτήσεων και κλήσεων.
- `perf.log`, `types.log`: αρχεία που παράγονται από τις αναλύσεις των πρόσθετων εργαλείων.

Η ύπαρξη αυτών των αναφορών επιτρέπει την αναδρομική ανάλυση και την επαναχρησιμοποίηση των δεδομένων για μελλοντική επιβολή.

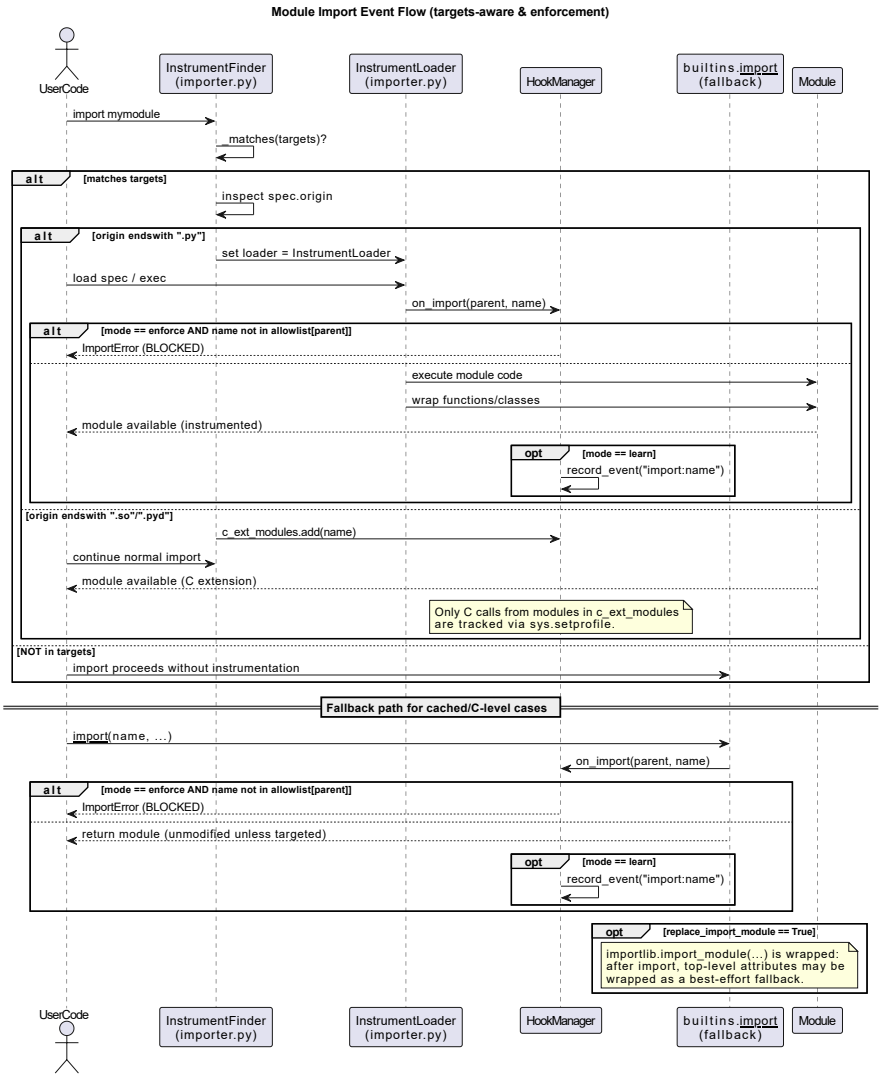
4.3 Πλαίσιο Ανάλυσης Ροής Γεγονότων (Event Flow Analysis Framework)

Το υποσύστημα ανάλυσης γεγονότων του Cerbex υλοποιεί την αρχή *παρατηρητών* (observer pattern) [10]. Ο `HookManager` συλλέγει γεγονότα εκτέλεσης (εισαγωγές, κλήσεις, επιστροφές) και τα προωθεί σε ένα σύνολο από *πρόσθετα εργαλεία ανάλυσης* τα οποία υλοποιούν την αφηρημένη κλάση `Analysis`. Με τον τρόπο αυτό, ο πυρήνας παραμένει ουδέτερος ως προς την ίδια την ανάλυση: τα πρόσθετα εργαλεία καθορίζουν ποιά δεδομένα θα συλλεχθούν, πώς θα επεξεργαστούν, και πού θα αποθηκευτούν.

4.3.1 Αρχιτεκτονική και Ροή Γεγονότων

Η κλάση `Analysis` (στο `analysis.py`) ορίζει callback μεθόδους όπως `on_import`, `on_call`, και `on_return`. Κάθε υποκλάση υλοποιεί την αντίστοιχη λογική ανάλυσης. Ο `HookManager`, όταν εντοπίσει γεγονός, καλεί όλες τις εγγεγραμμένες υλοποιήσεις.

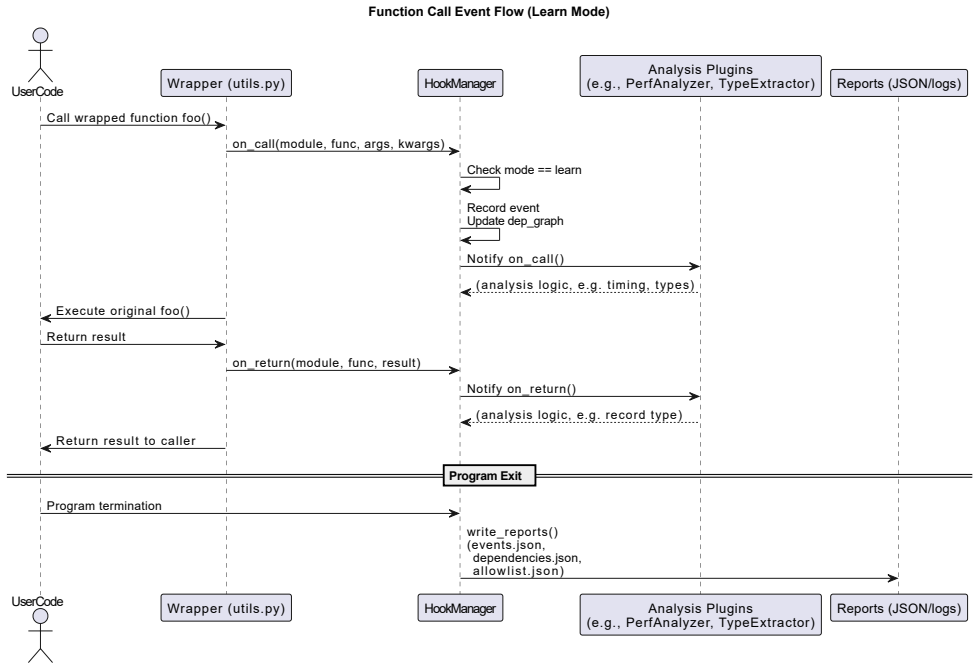
Η ροή για εισαγωγές αρθρώματα απεικονίζεται στην Εικόνα 4.2. Κατά την εισαγωγή ενός νέου αρθρώματος, η ενσωμάτωση κώδικα παρακολούθησης ενεργοποιεί το `HookManager.on_import`, ο οποίος ενημερώνει τον γράφο εξαρτήσεων και ειδοποιεί τα πρόσθετα εργαλεία. Στη λειτουργία εκμάθησης καταγράφονται γεγονότα και λίστες επιτρεπόμενων εξαρτήσεων, ενώ στη λειτουργία επιβολής ελέγχεται η `allowlist` και μπορεί να 'πετάξει' `ImportError`.



Γράφημα 4.2: Event flow during module import.

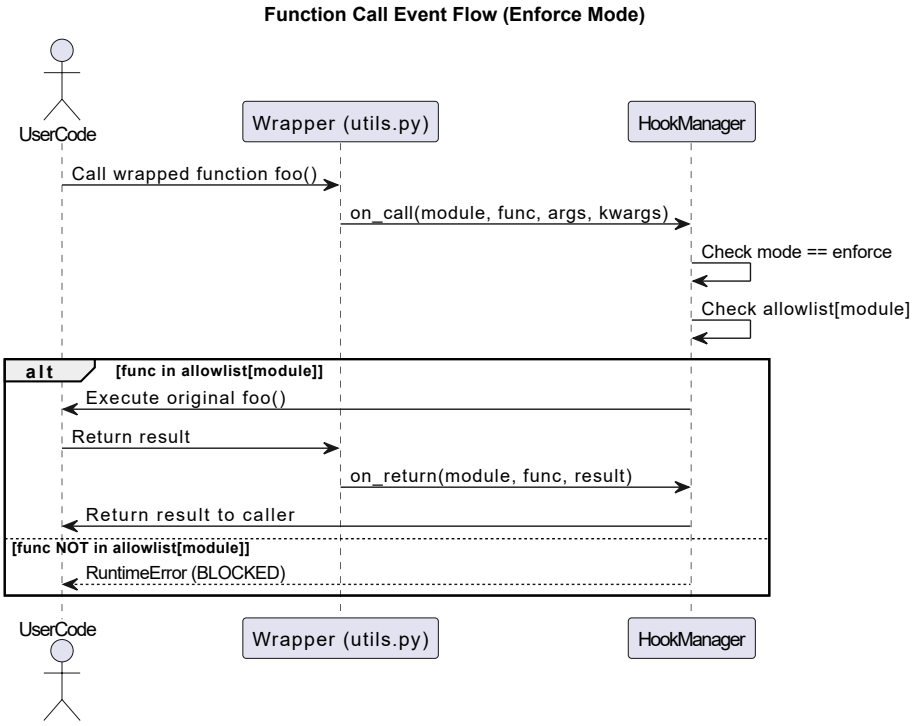
4.3.2 Ροή Κλήσεων Συναρτήσεων

Η κλήση συναρτήσεων ακολουθεί μια αντίστοιχη ροή. Στη λειτουργία εκμάθησης (Εικόνα 4.3), το περιτύλιγμα που έχει παραχθεί από το `wrap_value()` καλεί πρώτα το `HookManager.on_call`, έπειτα εκτελεί την αρχική συνάρτηση, και στο τέλος καλεί το `HookManager.on_return`. Ο `HookManager` καταγράφει τα δεδομένα και τα προωθεί σε όλα τα πρόσθετα εργαλεία.



Γράφημα 4.3: Event flow during a function call (learn mode).

Στη λειτουργία επιβολής (Εικόνα 4.4), η ροή διαφοροποιείται με την προσθήκη ενός βήματος ελέγχου πριν την εκτέλεση: ο `HookManager` εξετάζει την `allowlist`. Εάν η κλήση δεν επιτρέπεται, εγείρεται `RuntimeError` και η συνάρτηση δεν εκτελείται. Διαφορετικά, η εκτέλεση συνεχίζεται κανονικά και τα πρόσθετα εργαλεία ενημερώνονται όπως στη λειτουργία εκμάθησης.



Γράφημα 4.4: Event flow during a function call (enforce mode).

4.4 Διαχείριση Κόστους Εκτέλεσης

Η δυναμική ενσωμάτωση κώδικα παρακολούθησης εισάγει αναπόφευκτα κόστη εκτέλεσης, καθώς κάθε κλήση ή εισαγωγή παρεμβάλλεται με επιπλέον λογική. Το Cerbex υιοθετεί μια σειρά από τεχνικές βελτιστοποίησης, ώστε να περιορίσει το κόστος εκτέλεσης χωρίς να θυσιάζει τη διαφάνεια και την πληρότητα της παρακολούθησης.

Επιλεκτική ενσωμάτωση κώδικα παρακολούθησης. Μόνο τα αρθρώματα-στόχοι που ορίζονται στη διαμόρφωση υπόκεινται σε τύλιγμα. Οι υπόλοιπες βιβλιοθήκες —συμπεριλαμβανομένων των ενσωματωμένων αρθρωμάτων της Python και των τρίτων βιβλιοθηκών— παραμένουν αμετάβλητες. Επίσης, κατά το τύλιγμα αγνοούνται primitive τιμές και μη κλήσιμα (non callable) αντικείμενα, ώστε να αποφεύγεται περιττή επιβάρυνση.

Προσωρινή αποθήκευση περιτυλιγμάτων. Κάθε συνάρτηση τυλίγεται μία μόνο φορά, με χρήση κρυφής μνήμης (`WeakKeyDictionary`). Αυτό αποτρέπει τη δημιουργία πολλαπλών περιτυλιγμάτων για την ίδια συνάρτηση και μειώνει τόσο τη μνήμη όσο και τον χρόνο δημιουργίας.

Έλεγχοι Νήματος. Με τη χρήση του `threading.local` μπορούμε να κρατάμε ξεχωριστή κατάσταση (`in_hook`) για κάθε νήμα. Έτσι αποτρέπεται ο κίνδυνος επαναληπτικής ενσωμάτωσης

κώδικα παρακολούθησης, με ένα ελαφρύ λογικός έλεγχο (boolean check) που δεν προσθέτει αισθητό κόστος.

Άγκιστρα χαμηλού επιπέδου. Η αξιοποίηση της `sys.setprofile` για τα γεγονότα `c_call`/`c_return` σημαίνει ότι δεν γίνεται ενσωμάτωση κώδικα παρακολούθησης (*instrumentation*) για όλα τα opcodes της Python, αλλά μόνο οι κλήσεις συναρτήσεων. Αυτό μειώνει σημαντικά το βάθος της παρεμβολής.

Διαχείριση Καταγραφών. Τα γεγονότα και οι μετρήσεις δεν εγγράφονται άμεσα σε δίσκο· αντίθετα συσσωρεύονται στη μνήμη κατά την εκτέλεση και αποθηκεύονται μαζικά στο τέλος μέσω του `HookManager.write_reports()`. Η προσέγγιση αυτή αποφεύγει το σημαντικό κόστος του συνεχούς I/O και μειώνει την επιβάρυνση στην εκτέλεση. Αν απαιτείται *real-time* παρακολούθηση, τα πρόσθετα εργαλεία μπορούν να επεκταθούν ώστε να κάνουν περιοδικό καθαρισμό (flush) ή εγγραφές σε παρτίδες (batch). Ωστόσο, η προσέγγιση αυτή συνεπάγεται αυξημένο κόστος στον χρόνο εκτέλεσης, καθώς εισάγει πρόσθετες λειτουργίες I/O κατά τη διάρκεια της εκτέλεσης.

Φιλτράρισμα Μεθόδων. Κατά το τύλιγμα παραλείπονται ειδικές μέθοδοι όπως `__repr__` και `__str__`, ώστε απλές λειτουργίες απεικόνισης ή καταγραφής αντικειμένων να μην παράγουν γεγονότα *instrumentation* ούτε να προκαλούν αναδρομικούς βρόχους.

Συνοπτικά. Οι παραπάνω τεχνικές επιτρέπουν στο Cerbex να εξισορροπεί την ακρίβεια παρακολούθησης με την αποδοτικότητα. Με τον τρόπο αυτό, το εργαλείο παραμένει διαφανές για τον προγραμματιστή, στοχεύοντας μόνο τον κώδικα που έχει οριστεί από τον χρήστη και ελαχιστοποιώντας την επίδραση στην απόδοση.

Κεφάλαιο 5

Αναλύσεις

5.1 Ανάλυση Ασφάλειας

Πρόβλημα Το εσωτερικό άρθρωμα (builtin module) `pickle` (πίκλα) της Python είναι γνωστό για την ανασφαλή αποσειριοποίηση: η φόρτωση ενός κακόβουλου αρχείου pickle μπορεί να εκτελέσει αυθαίρετο κώδικα [35, 22]. Το OWASP έχει από καιρό κατατάξει την ανασφαλή αποσειριοποίηση ως μία από τις δέκα κορυφαίες κατηγορίες ευπαθειών, προειδοποιώντας ότι τα μη αξιόπιστα payloads pickle οδηγούν άμεσα σε απομακρυσμένη εκτέλεση κώδικα σε εφαρμογές που κατά τα άλλα είναι ακίνδυνες [27]. Η στατική ανάλυση δεν μπορεί να ανιχνεύσει αυτά τα payloads κατά την εκτέλεση, ενώ η πλήρης δυναμική ανίχνευση είναι πολύ κοστοβόρα για παραγωγή [43]. Επομένως, είναι απαραίτητη μια ελαφριά επιβολή επιτρεπόμενων/απαγορευμένων ενεργειών που μπλοκάρει απροσδόκητες λειτουργίες (π.χ. κλήσεις προς `os.system`).

Στην πράξη. Ας εξετάσουμε ένα απλό πρόγραμμα `unsafe_deserialize.py` που διαβάζει από το `data.pkl` και το αποσειριοποιεί χωρίς κάποιον έλεγχο της εισόδου:

```
import pickle

def main():
    with open('data.pkl', 'rb') as f:
        obj = pickle.load(f)   #  unsafe!
    print("Loaded object:", obj)

if __name__ == "__main__":
    main()
```

Κώδικας 5.1: Vulnerable deserialization script (unsafe_deserialize.py).

Ένας κακόβουλος χρήστης θα μπορούσε να δημιουργήσει ένα κακόβουλο payload που εκτελεί μια shell εντολή κατά τη διάρκεια της αποσυμπίεσης, για παράδειγμα:

```
import pickle, os

class Exploit:
    def __reduce__(self):
        return (os.system, ("echo 'Owned!'; rm -rf /tmp/*",))

with open('data.pkl', 'wb') as f:
    pickle.dump(Exploit(), f)
```

Κώδικας 5.2: Malicious payload generator (make_exploit.py).

Η εκτέλεση του ευάλωτου προγράμματος με αυτό το payload θα εκτελούσε την shell εντολή, διαγράφοντας αρχεία στο `/tmp`.

Ανάλυση. Το εργαλείο λειτουργεί σε δύο λειτουργικές καταστάσεις:

1. *Λειτουργία εκμάθησης (Learn mode)*: καταγράφει κάθε *εισαγωγή* και κλήση συνάρτησης (π.χ. `pickle.load`), προκειμένου να κατασκευάσει έναν κατάλογο επιτρεπόμενων (*allowlist*).
2. *Λειτουργία επιβολής (Enforce mode)*: επανα-ενσωματώνει τα ίδια άγκιστρα (*hooks*), αλλά εμποδίζει οποιαδήποτε *εισαγωγή* ή κλήση δεν περιλαμβάνεται στον κατάλογο επιτρεπόμενων (π.χ. `os.system`).

Πρώτα, εκτελούμε το πρόγραμμα make_benign.py για να δημιουργήσουμε το αρχείο data.pkl:

```
# make_benign.py
import pickle
from types import SimpleNamespace

class Benign:
    def __reduce__(self):
        # Reconstruct a harmless SimpleNamespace at load time
        state = {
            "user": "alice",
            "id": 42,
            "items": ["apple", "banana", "cherry"]
        }
        # On unpickle, Python will call SimpleNamespace(**state)
        return (SimpleNamespace, (), state)

# Build the benign pickle file using the same __reduce__ pattern
with open('data.pkl', 'wb') as f:
    pickle.dump(Benign(), f)

print("Benign payload written to data.pkl")
```

Κώδικας 5.3: Benign payload generator (make_benign.py).

Με το παραγόμενο αρχείο `data.pkl` διαθέσιμο, προχωράμε στην εκτέλεση του `unsafe_deserialize.py` με το εργαλείο σε λειτουργία εκμάθησης, ώστε να καταγράψει την αναμενόμενη συμπεριφορά.

```
  Cerbex \
--mode learn \
--config config.json\
-- unsafe_deserialize.py
```

Κώδικας 5.4: Recording expected behavior in learn mode.

Η διαδικασία αυτή δημιουργεί το αρχείο `allowlist.json`, το οποίο περιλαμβάνει μια «λευκή λίστα» με όλα τα αρθρώματα και τις συναρτήσεις που παρατηρήθηκαν κατά την εκτέλεση. Στο παρακάτω απόσπασμα φαίνεται η δομή του αρχείου που παρήχθη.

```
{
  "allowlist": {
  ...
    "__main__": [
      "_compat_pickle",
      "_pickle",
      "pickle",
      "org",
      "pkgutil",
      "types"
    ],
  ...
    "pkgutil": [
          ...
          "os",
          "os.path",
          ...
        ],
  ...
    "pickle": [
      "_compat_pickle",
      "_pickle",
      "codecs",
      "copyreg",
      "functools",
      "io",
      "itertools",
      "org.python.core",
      "re",
      "struct",
      "sys",
      "types"
    ],
  ...
    "posix": [
          "_path_normpath",
          "fspath",
          "getcwd",
          "stat"
        ],
  ...
    "_json": [
          "encode_basestring_ascii"
        ]
  }
}
```

Κώδικας 5.5: Excerpt from `allowlist.json`.

Λειτουργία επιβολής. Στη συνέχεια, εκτελούμε το ίδιο σενάριο σε λειτουργία επιβολής, αλλά πρώτα πρέπει να εκτελέσουμε το `make_exploit.py` 5.2:

```
    Cerbex\
  --mode enforce \
  --allowlist allowlist.json \
  unsafe_deserialize.py
```

Κώδικας 5.6: Execution in enforce mode.

Αποτέλεσμα. Όταν το κακόβουλο payload προσπαθεί να καλέσει το `os.system`, η επιβολή δεν το επιτρέπει και πετάει σφάλμα:

```
Traceback (most recent call last):
    ...
    File "/home/X-X-X-X/Thesis/Cerbex/Cerbex/hook_manager.py", line 43,
        in on_import
    raise ImportError(f"Import of {name} not allowed in module {parent}")
ImportError: Import of posix not allowed in module __main__
```

Κώδικας 5.7: Error output under enforce mode.

Συμπέρασμα. Παρατηρούμε ότι ενώ το posix υπάρχει στη λίστα επιτρεπτών, το εργαλείο το απέριψε. Αυτό γίνεται διότι το posix που καταγράφηκε κατά την εκμάθηση προέρχεται από το άρθρωμα pkgutil, το οποίο εξαρτάται από το os, το οποίο με τη σειρά του δίνει πρόσβαση στο posix. Αλλά αυτό δεν έγινε ποτέ από την __main__ κατά τη διάρκεια της εκμάθησης. Αυτό δείχνει ότι δεν αρκεί απλά να υπάρχει μία εισαγωγή ή μια κλήση στη λίστα αποδοχών, αλλά έχει σημασία και από πού προήλθε, όπως σε αυτή την περίπτωση. Έτσι το εργαλείο Cerbex απέτρεψε την αυθαίρετη κακόβουλη εκτέλεση.

5.2 Ανάλυση Απόδοσης

Πρόβλημα. Σε μεγάλες εφαρμογές με πολλές εξαρτήσεις, μια απότομη επιβράδυνση μπορεί να είναι δύσκολο να διαγνωστεί. Μια βιβλιοθήκη βαθιά στο δέντρο εξαρτήσεων μπορεί να εισάγει κακογραμμένο κώδικα. Για παράδειγμα, σε διαδικασίες μαζικής επεξεργασίας (batch processing) μεγάλων συνόλων δεδομένων — όπως εικόνες υψηλής ανάλυσης — τα σημεία συμφόρησης απόδοσης μπορεί να είναι απροσδόκητα ασαφή [21, 47]. Μια ρουτίνα βαθιά μέσα σε μια βιβλιοθήκη επεξεργασίας εικόνων (για παράδειγμα, ο αλγόριθμος αλλαγής μεγέθους που υπάρχει στη βιβλιοθήκη Pillow [28]) μπορεί να προκαλέσει σημαντική καθυστέρηση, υποβαθμίζοντας τη συνολική απόδοση. Επομένως, η ταυτοποίηση των συναρτήσεων που καταναλώνουν τον περισσότερο χρόνο είναι κρίσιμη για στοχευμένη βελτιστοποίηση της απόδοσης [36].

Στη πράξη. Ας υποθέσουμε ότι έχουμε μια εφαρμογή Python `image_resizer.py` και θέλουμε να μετρήσουμε πόσο χρόνο απαιτεί η εκτέλεση κάθε συνάρτησης βιβλιοθήκης. Μπορούμε να γράψουμε μια ανάλυση απόδοσης που καταγράφει χρονοσφραγίδες (timestamps) σε κάθε κλήση και επιστροφή συνάρτησης. Η κλάση `PerfAnalyzer` στο 5.9 υλοποιεί αυτή τη λειτουργία υπερκαλύπτοντας τα `on_call` και `on_return` άγκιστρα: κάθε φορά που καλείται μια συνάρτηση, ωθούμε στη στοίβα την τρέχουσα χρονοσφραγίδα· όταν η συνάρτηση επιστρέφει, ανακτούμε (pop) τον χρόνο έναρξης από τη στοίβα, υπολογίζουμε τη διάρκεια που πέρασε και το καταγράφουμε.

```
from pathlib import Path
from PIL import Image

INPUT_DIR  = Path("photos/")
OUTPUT_DIR = Path("thumbnails/")
SIZE       = (200, 200)

def main():
    OUTPUT_DIR.mkdir(exist_ok=True)
    for img_path in INPUT_DIR.glob("*.jpg"):
        img = Image.open(img_path)
        img.thumbnail(SIZE)
        img.save(OUTPUT_DIR / img_path.name)

if __name__ == "__main__":
    main()
```

Κώδικας 5.8: Batch image-resizer (`image_resizer.py`).

Σε ένα σύνολο δεδομένων 100 μεγάλων έγχρωμων εικόνων (4000×3000 px), αυτό το σενάριο ολοκληρώνεται σε περίπου 3 δευτερόλεπτα πραγματικού χρόνου (με το επιπλέον φορτίο που προσθέτει το εργαλείο μας στον συνολικό χρόνο εκτέλεσης) — αλλά ποια βήματα καταναλώνουν τον περισσότερο χρόνο;

Υλοποίηση. Χρησιμοποιούμε τον `PerfAnalyzer` (Listing 5.9) για να καταγράψουμε τους χρόνους κλήσης και επιστροφής κάθε παρατηρούμενης συνάρτησης. Διαμορφώνουμε το Cerbex ώστε να παρακολουθεί μόνο το πρόγραμμα μας και το άρθρωμα `PIL.Image`:

Κώδικας 5.9: `PerfAnalyzer` class that logs execution time of each hooked function to a file.

```
import threading
import time
import atexit
from typing import Any, List, Tuple
from Cerbex.hook_manager import Analysis

class PerfAnalyzer(Analysis):
    """
    Measures execution time of each function call with zero I/O overhead
       during execution.
    Buffers timings in memory and dumps to file at program exit.
    """
    def __init__(self, outfile: str = "perf.log") -> None:
        self.outfile = outfile
        self._local = threading.local()
        # Buffer for (module.func, duration) tuples
        self._buffer: List[Tuple[str, float]] = []
        # Register dump at program exit
        atexit.register(self._dump)

     def on_call(self, module, func, args, kwargs):
        stack = getattr(self._local, "stack", None)
        if stack is None:
            stack = []
            self._local.stack = stack
        stack.append(perf_counter())

    def on_return(self, module, func, result):
        stack = getattr(self._local, "stack", None)
        if not stack:
            return
        start = stack.pop()
        duration = perf_counter() - start
        self._buffer.append((f"{module}.{func}", duration))

    def results(self) -> List[Tuple[str, float]]:
        """
        Returns the list of ("module.func", duration) tuples.
        """
        return list(self._buffer)

    def _dump(self) -> None:
        """
        Writes all buffered timings to the output file in one batch.
        """
        if not self._buffer:
            return
        lines = [f"[Perf] {name} took {dur:.6f}s\n" for name, dur in
           self._buffer]
        with open(self.outfile, "a") as f:
            f.writelines(lines)
```

```
{
  "targets": [
    "image_resizer",
    "PIL.Image",
    "PIL.ImageFile"
  ]
}
```

Κώδικας 5.10: `config.json` for performance analysis.

Στη συνέχεια εκτελούμε:

```
python3 runner.py
```

Κώδικας 5.11: Run learn+measure pass with `PerfAnalyzer`.

όπου το `runner.py` περιέχει:

```
// runner.py
from Cerbex.hook_loader import install_hooks
from Cerbex.analysis import PerfAnalyzer

install_hooks(
    config_path="config.json",
    mode="learn",
    analyses=[PerfAnalyzer(outfile="perf.log")]
)

import image_resizer
image_resizer.main()
```

Λειτουργία του PerfAnalyzer. Το πρόσθετο εργαλείο `PerfAnalyzer` παρεμβάλλεται (*hooks*) στις κλήσεις και στις επιστροφές συναρτήσεων, με στόχο τη μέτρηση του χρόνου εκτέλεσης. Συγκεκριμένα, καταγράφει τις ακόλουθες ενέργειες:

- `on_call`: Όταν καλείται μια συνάρτηση, ωθεί την τρέχουσα χρονοσφραγίδα σε μια στοίβα (thread-local).
- `on_return`: Όταν η συνάρτηση επιστρέφει, αφαιρεί (pop) τον χρόνο έναρξης από τη στοίβα και υπολογίζει τη διάρκεια που παρήλθε. Το ζεύγος `(module.func, duration)` προστίθεται σε έναν πίνακα μνήμης(buffer), αποφεύγοντας I/O κατά τη διάρκεια της εκτέλεσης.
- Τερματισμός προγράμματος: Κατά τον τερματισμό, ο αναλυτής αποθηκεύει όλους τους καταγεγραμμένους χρόνους στο αρχείο εξόδου (χρησιμοποιώντας `atexit`), έτσι ώστε οι λειτουργίες I/O να γίνουν μόνο μία φορά.

Τρέχουμε το `image_resizer.py` υπό τον `PerfAnalyzer` με 100 μεγάλες (4000×3000) εικόνες που περιέχουν μόνο χρώματα. Η εντολή `time` στο shell αναφέρει:

```
real 0m3.197s
user 0m3.073s
sys 0m0.121s
```

Συγκεντρώνοντας τα περιεχόμενα του `perf.log` προκύπτει ο συνολικός χρόνος που δαπανάται σε κάθε συνάρτηση:

```
=== PIL.Image.open ===
TOTAL PIL.Image.open: 0.305033s
=== PIL.Image.save ===
TOTAL PIL.Image.save: 0.243947s
=== PIL.Image.thumbnail ===
TOTAL PIL.Image.thumbnail: 2.136417s
=== image_resizer.main ===
TOTAL image_resizer.main: 2.721153s
```

Κώδικας 5.12: Aggregated execution times from `perf.log`.

Συμπέρασμα. Τα αποτελέσματα της ανάλυσης προφίλ εντοπίζουν με σαφήνεια το σημείο συμφόρησης: η συνάρτηση `PIL.Image.thumbnail` κυριαρχεί στον χρόνο εκτέλεσης (2.136s συνολικά). Αντίθετα, η φόρτωση (open) και η αποθήκευση (save) εικόνων διαρκούν μόνο περίπου 0.3s η κάθε μία. Η συνάρτηση `image_resizer.main` αντιστοιχεί σε 2.72s, συμφωνώντας με το άθροισμα των κλήσεων. Αυτά τα ευρήματα υποδεικνύουν ότι η βελτιστοποίηση ή η παράλληλη εκτέλεση του υπολογισμού της αλλαγής μεγέθους εικόνων (thumbnail) θα μπορούσε να αποδώσει τα μεγαλύτερα οφέλη απόδοσης. Συνοψίζοντας, ο `PerfAnalyzer` εντοπίζει με επιτυχία τις συναρτήσεις που καταναλώνουν τον περισσότερο χρόνο, καθοδηγώντας τη στοχευμένη βελτιστοποίηση της ροής εργασιών επεξεργασίας εικόνων με σχετικά λίγες γραμμές επιπλέον κώδικα (49 LoC).

5.3 Εξαγωγή τύπων

Πρόβλημα. Δυναμικές γλώσσες προγραμματισμού όπως η Python δεν περιλαμβάνουν ρητές δηλώσεις τύπων για τις εισόδους ή τις εξόδους των συναρτήσεων [50]. Επομένως, η κατανόηση των τύπων δεδομένων που διακινούνται μέσω του API μιας βιβλιοθήκης πρέπει να βασίζεται στην παρατήρηση κατά το χρόνο εκτέλεσης [54, 1]. Η εξαγωγή τύπων είναι η διαδικασία καταγραφής των τύπων των τιμών που δίνονται ως ορίσματα και επιστρέφονται από τις συναρτήσεις ενός αρθρώματος. Αυτές οι πληροφορίες μπορούν να βοηθήσουν στην τεκμηρίωση της χρήσης της βιβλιοθήκης, στην ανίχνευση απρόσμενων σφαλμάτων τύπων ή στην αυτοματοποιημένη δημιουργία προδιαγραφών τύπων (type specifications).

Στη πράξη. Ας θεωρήσουμε ένα απλό βοηθητικό άρθρωμα `string_utils.py` που παρέχει βασικές λειτουργίες για συμβολοσειρές (Listing 5.13). Στη συνέχεια, καλούμε αυτές τις συναρτήσεις από ένα κεντρικό πρόγραμμα (Listing 5.14). Παρατηρήστε ότι καμία από τις συναρτήσεις δεν έχει δηλωμένους τύπους, οπότε ένας προγραμματιστής ή ένα εργαλείο πρέπει να συμπεράνει τη συμπεριφορά τους δοκιμάζοντάς.

```
def concat(a, b):
    return a + b

def shout(message):
    return message.upper() + '!'

def count_chars(s):
    return len(s)
```

Κώδικας 5.13: Utility module `string_utils.py`

```
from Cerbex.hook_loader import install_hooks
from Cerbex.analysis import TypeExtractor

install_hooks(
    config_path="config.json",
    mode="learn",
    analyses=[TypeExtractor(outfile="type.log")]
)

import string_utils

_ = string_utils.concat("foo", "bar")
_ = string_utils.shout("hello")
_ = string_utils.count_chars("baz")
```

Κώδικας 5.14: Main program `main.py` with Cerbex hooks

Υλοποίηση. Στο Cerbex, οι αναλύσεις είναι υποκλάσεις της βασικής κλάσης `Analysis` και υπερκαλύπτουν (override) τις μεθόδους άγκιστρα. Ο `TypeExtractor` (Listing 5.15) υπερκαλύπτει τη μέθοδο `on_return` ώστε να καταγράφει τον τύπο Python κάθε επιστρεφόμενης τιμής συνάρτησης, εκτελώντας άμεσο καθαρισμό (flush) για να αποφευχθεί η απώλεια δεδομένων.

```python
from Cerbex.hook_manager import Analysis
from typing import Any

class TypeExtractor(Analysis):
    """
    Extracts return types of each function call with zero I/O overhead during execution.
    Buffers type info in memory and dumps to file at program exit.
    """
    def __init__(self, outfile: str = "types.log") -> None:
        self.outfile = outfile
        # Buffer for (module.func, type_name) tuples
        self._buffer: List[Tuple[str, str]] = []
        # Register dump at program exit
        atexit.register(self._dump)

    def on_return(self, module: str, func: str, result: Any) -> None:
        # Buffer the type information instead of writing immediately
        self._buffer.append((f"{module}.{func}", type(result).__name__))

    def results(self) -> List[Tuple[str, str]]:
        """
        Returns the list of ("module.func", type_name) tuples.
        """
        return list(self._buffer)


    def _dump(self) -> None:
        """
        Writes all buffered type information to the output file in one batch.
        """
        if not self._buffer:
            return
        lines = [f"[Type] {name} returned {type_name}\n" for name, type_name in self._buffer]
        with open(self.outfile, "a") as f:
            f.writelines(lines)
```

Κώδικας 5.15: Cerbex analysis `TypeExtractor`

Μόλις αυτή η ανάλυση καταχωρηθεί στον διαχειριστή άγκιστρων του Cerbex (για παράδειγμα, ρυθμίζοντας τον `TypeExtractor` ως ανάλυση για το άρθρωμα `string_utils`), ενεργοποιείται αυτόματα κάθε φορά που καλείται μια συνάρτηση σε αυτό το άρθρωμα. Στο παράδειγμά μας, η ανάλυση εκτελεί τις ακόλουθες ενέργειες:

- Το άγκιστρο `on_return` αλείται μετά την ολοκλήρωση κάθε συνάρτησης στο `string_utils`.

- Μέσα στο `on_return`, η ανάλυση γράφει μια εγγραφή στο αρχείο καταγραφής με το όνομα της συνάρτησης και τον τύπο Python της επιστρεφόμενης τιμής.
- Η έξοδος γίνεται flush στο αρχείο (`types.log`) άμεσα, διασφαλίζοντας την καταγραφή των πληροφοριών ακόμα κι αν το πρόγραμμα τερματιστεί απροσδόκητα.

Αποτελέσματα. Αφού εκτελέσουμε το κύριο πρόγραμμα με το Cerbex και αυτή την ανάλυση, λαμβάνουμε τους εξαγόμενους τύπους. Για τις τρεις κλήσεις παραπάνω, το αρχείο καταγραφής περιέχει:

```
[Type] string_utils.concat returned str
[Type] string_utils.shout returned str
[Type] string_utils.count_chars returned int
```

Κώδικας 5.16: Extracted return types in `type.log`

Αυτές οι εγγραφές δείχνουν καθαρά ότι οι `concat` και `shout` επιστρέφουν τύπο `str`, ενώ η `count_chars` επιστρέφει τύπο `int`.

Συμπέρασμα. Με ελάχιστο επιπλέον κώδικα (boilerplate), ο `TypeExtractor` παρέχει χρήσιμα μεταδεδομένα σχετικά με τους τύπους επιστρεφόμενων τιμών συναρτήσεων. Αυτή η έξοδος μπορεί να αξιοποιηθεί από downstream εργαλεία για τη δημιουργία υποδείξεων τύπων (type annotations), την επαλήθευση συμβολαίων API ή την αυτόματη παραγωγή τεκμηρίωσης (documentation), ενισχύοντας έτσι την ποιότητα του κώδικα και τη συντηρησιμότητά του.

5.4 Δημιουργία Νέων Αναλύσεων

Πρόβλημα. Πέρα από τις ενσωματωμένες αναλύσεις (ασφάλειας, απόδοσης, τύπων, ιδιοτήτων), οι χρήστες μπορεί να έχουν εξειδικευμένες απαιτήσεις παρακολούθησης — π.χ. καταγραφή συγκεκριμένων ροών δεδομένων (data-flows), εντοπισμός αλλαγών σε global state ή άλλες μετρικές ειδικού τομέα. Η πλατφόρμα πρέπει να επιτρέπει τη γρήγορη συγγραφή νέων πρόσθετων εργαλείων ανάλυσης χωρίς τροποποίηση του πυρήνα.

Στη πράξη. Η βασική κλάση `Analysis` ορίζει όλα τα διαθέσιμα άγκιστρα:

- `on_import(parent, name)`
- `on_call(module, func, args, kwargs)`
- `on_return(module, func, result)`

Για να δημιουργήσει κανείς νέα ανάλυση, αρκεί να κληρονομήσει την `Analysis` και να υπερκαλύψει τα άγκιστρα που τον ενδιαφέρουν.

Υλοποίηση. Στο παράδειγμα του 5.17 βλέπουμε μια ανάλυση `CustomDataFlowAnalyzer` που μετράει πόσες φορές καλείται η συνάρτηση `process_item` και ποιοι τύποι δεδομένων της δίνονται ως όρισμα:

```
from Cerbex.hook_manager import Analysis
import atexit

class CustomDataFlowAnalyzer(Analysis):
    """
    Example of a user-defined analysis that tracks calls to
        'process_item'
    and logs the type of its first argument.
    """
    def __init__(self, outfile="dataflow.log"):
        self.outfile = outfile
        self._buffer = []  # store (type_name, module, func)
        atexit.register(self._dump)

    def on_call(self, module, func, args, kwargs):
        if func == "process_item":
            arg_type = type(args[0]).__name__ if args else "None"
            # Add an entry to the in-memory buffer
            self._buffer.append((module, func, arg_type))

    def _dump(self):
        if not self._buffer:
            return
        counts = {}
        for _, _, t in self._buffer:
            counts[t] = counts.get(t, 0) + 1

        with open(self.outfile, "w") as f:
            f.write(f"process_item called {len(self._buffer)} times\n")
            for t, cnt in counts.items():
                f.write(f"  {t}: {cnt}\n")
```

Κώδικας 5.17: Παράδειγμα ανάλυσης `CustomDataFlowAnalyzer`

Ο χρήστης ενεργοποιεί την ανάλυση με ένα runner ή με τη χρήση του εργαλείου μέσω της γραμμής εντολών όπως δείχνουμε στο Κεφάλαιο 6:

```
from Cerbex.hook_loader import install_hooks
from myanalysis import CustomDataFlowAnalyzer

install_hooks(
    config_path="config.json",
    mode="learn",
    analyses=[CustomDataFlowAnalyzer()]
)

import target_script
target_script.main()
```

Κώδικας 5.18: Runner για `CustomDataFlowAnalyzer`

Σημείωση για επεκτασιμότητα. Στη φάση που βρίσκεται το εργαλείο, τα άγκιστρα που παρέχονται είναι αυτά που αναφέρθηκαν στην αρχή του υποκεφαλαίου. Εάν χρειάζεται κάποιο άγκιστρο που δεν παρέχεται από την κλάση `Analysis`, ο χρήστης πρέπει να επεκτείνει το εργαλείο.

Αποτέλεσμα. Με αυτήν τη μεθοδολογία, οι προγραμματιστές μπορούν σε μερικές γραμμές κώδικα να δημιουργήσουν πολύπλοκες, ειδικού τομέα αναλύσεις εξασφαλίζοντας πλήρη ευελιξία στην παρακολούθηση συμπεριφοράς κατά τον χρόνο εκτέλεσης.

Κεφάλαιο 6

Χρήση Εργαλείου

6.1 Ενσωμάτωση μέσω Προγραμματιστικού API

Η ενσωμάτωση του Cerbex σε μια υπάρχουσα εφαρμογή Python μπορεί να γίνει απευθείας στον κώδικα, χωρίς ανάγκη ξεχωριστού CLI. Ακολουθεί ένα παράδειγμα «runner» πρόγραμμα που φορτώνει άγκιστρα, εκτελεί τις αναλύσεις και στο τέλος παράγει τα αρχεία αναφοράς:

```python
# runner.py
# 1. Installation of hooks in learn mode
from Cerbex.hook_loader import install_hooks
from Cerbex.analysis import PerfAnalyzer, TypeExtractor

hook_mgr = install_hooks(
    config_path="config.json",      )
    mode="learn",                   # learn or enforce
    analyses=[PerfAnalyzer(),
              TypeExtractor()],
    allowlist_path="allowlist.json" # Used only in enforce mode
    log_events=True #or False.
)

# 2. Import our app and if a main exists run main
import app
app.main()
```

Κώδικας 6.1: runner.py – Προγραμματιστική Ενσωμάτωση του Cerbex

- **install_hooks(...)**: Ρυθμίζει τις εισαγωγές, περιτυλίγματα και το εργαλείο προφιλ, φορτώνει τους αναλυτές, και καταχωρεί στο `atexit` τη μέθοδο `write_reports` για παραγωγή των JSON αρχείων.
- **config.json**: Ορίζει τα patterns των αρθρωμάτων στα οποία θα εφαρμοστεί ενσωμάτωση κώδικα παρακολούθησης (π.χ. `"targets":["app", "requests", "json"]`).
- **analyses**: Θέτονται οι αναλύσεις που θα τρέξουν.
- **allowlist**: Φορτώνεται μόνο σε *λειτουργία επιβολής* και χρησιμοποιείται για τον έλεγχο εισαγωγών και κλήσεων.

- **log_events**: Χρησιμοποιείται για την ενεργοποιήση ή απενεργοποιήση των καταγραφών στις αναλύσεις για να υπολογιστεί ακριβώς η επιβάρυνση του εργαλείου χωρίς πρόσθετει επεξεργασία. Δεν είναι απαραίτητο για την λειτουργία του εργαλείου.

- Με την `import app; app.main()` εκτελείται η κύρια συνάρτηση της εφαρμογής, διατηρώντας το σχήμα `if __name__ == "__main__"` στο αρχείο `app.py`. Αν δεν υπάρχει συνάρτηση main που εκτελεί το πρόγραμμα τότε απλά εισάγωντας το άρθρωμα θα εκτελεστεί το top level.

Με αυτόν τον τρόπο, η εφαρμογή `app.py` τρέχει κανονικά, αλλά όλες οι εισαγωγές και οι κλήσεις συναρτήσεων περνούν από τον HookManager και τους αναλυτές του Cerbex. Τα παραγόμενα αρχεία `dependencies.json`, `events.json` και `allowlist.json` αποθηκεύονται στο τρέχον φάκελο και μπορούν να χρησιμοποιηθούν σε επόμενη εκτέλεση σε *λειτουργία επιβολής* για την επιβολή των πολιτικών.

6.2 Χρήση εργαλείου μέσω της γραμμής εντολών

Το Cerbex υποστηρίζει εκτέλεση και μέσω γραμμής εντολών, παρέχοντας αντίστοιχη λειτουργικότητα με την προγραμματιστική προσέγγιση, αλλά με πιο απλή χρήση. Η εντολή `Cerbex` εγκαθίσταται μέσω του `setup.py` και καθιστά δυνατή την εκτέλεση οποιουδήποτε Python αρχείου υπό καθεστώς καταγραφής ή επιβολής.

6.2.1 Εγκατάσταση του εργαλείου ως CLI

Για να γίνει η εντολή `Cerbex` διαθέσιμη στο σύστημα, απαιτείται η εγκατάσταση του εργαλείου με την εντολή:

```
pip install -e .
```

Η παραπάνω εντολή πρέπει να εκτελεστεί από τον βασικό φάκελο του project (εκεί όπου βρίσκεται το `setup.py`) και ιδανικά μέσα σε κάποιο εικονικό περιβάλλον (virtual environment). Με αυτόν τον τρόπο, το εργαλείο εγκαθίσταται σε λειτουργία επεξεργασίας και όλες οι αλλαγές στον πηγαίο κώδικα λαμβάνονται άμεσα υπόψη. Αν η εντολή εκτελεστεί εκτός του φακέλου όπου βρίσκεται ο πηγαίος κώδικας (δηλ. εκτός του φακέλου που περιέχει το `setup.py`), τότε θα πρέπει να δηλωθεί ρητά το path προς το εργαλείο κατά την εγκατάστασή του, π.χ.:

```
pip install -e /full/path/to/the/tool
```

6.2.2 Εκτέλεση σε λειτουργία εκμάθησης

Η βασική σύνταξη είναι:

```
Cerbex --mode learn \
      --config config.json \
      --analyses perf types \
      --outdir logs \
      -- \
      requests_example.py
```

Κώδικας 6.2: Εκτέλεση μέσω CLI σε learn mode

- `-mode learn`: Εκτελεί το πρόγραμμα σε λειτουργία μάθησης (learn), καταγράφοντας εξαρτήσεις, γεγονότα και επιτρεπόμενες κλήσεις.
- `-config config.json`: Αρχείο παραμετροποίησης που ορίζει ποια αρθρώματα θα καλυφθούν με άγκιστρα.
- `-analyses perf types`: Ορίζει ποιες αναλύσεις θα ενεργοποιηθούν. Οι επιλογές είναι `perf` για μέτρηση χρόνου εκτέλεσης συναρτήσεων και `types` για καταγραφή των τύπων μεταβλητών.
- `-outdir logs`: Φάκελος στον οποίο θα γραφούν τα αρχεία καταγραφής των αναλύσεων (όχι τα JSON).
- `-`: Διαχωρίζει τις σημαίες του Cerbex από τα ορίσματα του στοχευμένου προγράμματος.

Τα αρχεία `events.json`, `dependencies.json` και `allowlist.json` αποθηκεύονται αυτόματα στο τρέχον φάκελο κατά την έξοδο του προγράμματος. Το ίδιο ισχύει και για τα αρχεία των αναλύσεων εφόσον δεν χρησιμοποιηθεί `-outdir`.

6.2.3 Εκτέλεση σε λειτουργία επιβολής

Μετά από μια φάση μάθησης, το Cerbex μπορεί να τρέξει το ίδιο πρόγραμμα υπό καθεστώς επιβολής. Στην περίπτωση αυτή, το εργαλείο μπλοκάρει μη εγκεκριμένες εισαγωγές και κλήσεις βάσει του `allowlist.json`.

```
Cerbex --mode enforce \
      --allowlist allowlist.json \
      -- \
      requests_example.py
```

Κώδικας 6.3: Εκτέλεση μέσω CLI σε enforce mode

Κατά την εκτέλεση, αν το πρόγραμμα επιχειρήσει να καλέσει συναρτήσεις ή να εισάγει αρθρώματα που δεν περιλαμβάνονται στη λίστα επιτρεπτών (`allowlist.json`), το εργαλείο θα αποτύχει με εξαίρεση, ανάλογα με την παραμετροποίηση.

Παρατηρήσεις

- Η χρήση CLI καθιστά το εργαλείο πιο εύχρηστο για προγραμματιστές που θέλουν να ελέγχουν υπάρχοντα προγράμματα χωρίς να τα τροποποιούν.
- Το CLI είναι πλήρως παραμετροποιήσιμο και επεκτάσιμο, με υποστήριξη για σύνθετες ροές εργασιών, π.χ. αυτοματοποιημένο έλεγχο πολιτικής ασφάλειας.
- Αν δεν προστεθεί το διαχωριστικό - πριν το όνομα του αρχείου, μπορεί να υπάρξει σφάλμα στην ερμηνεία των παραμέτρων.
- Όταν χρησιμοποιείται το CLI για ένα αρχείο όπως το 3.1, για την σωστή λειτουργία του εργαλείου θα πρέπει να χρησιμοποιείθει ένα runner αρχείο όπως το 3.3 (χωρίς τη συνάρτηση install_hooks) ώς όρισμα στο CLI ώστε να γίνεται εισαγωγή το άρθρωμα που θέλουμε να αναλυθεί. Αλλιώς δεν περνάει από την λογική παρακολούθησης που έχει το εργαλείο επειδή χρειάζεται να γίνει κάπως επαναφόρτωση. Αυτό δεν είναι πρόβλημα στη χρήση του εργαλείου μέσω προγραμματιστικού API γιατί φορτώνουμε το άρθρωμα ακριβώς μετά την εγκατάσταση του άγκιστρου (συνάρτηση install_hooks).

Κεφάλαιο 7

Αξιολόγηση

7.1 Στόχοι και Επισκόπηση

Σε αυτή την ενότητα παρουσιάζουμε τους κύριους στόχους της αξιολόγησης και δίνουμε μια συνοπτική επισκόπηση των μελετών περίπτωσης και των πειραματικών σεναρίων.

Στόχοι:

1. *Ορθότητα:* Να επιβεβαιωθεί ότι το εργαλείο δεν αλλοιώνει τη συμπεριφορά των εφαρμογών, διατηρώντας την επιτυχία των test suites και την ισοδυναμία των αποτελεσμάτων. Η συζήτηση για τον στόχο αυτό γίνεται στο Κεφ. 7.6.
2. *Απόδοση:* Ποιο επιπλέον κόστος χρόνου εκτέλεσης εισάγει η χρήση του εργαλείου

Μελέτες Περίπτωσης:

- *Πλαίσια (Frameworks):* FastAPI & Flask — εκτέλεση των πλήρων test suites μέσω `pytest`.
- *Μικρο-πειράματα:* 20 δημοφιλή μικρά πακέτα — εκτέλεση ενός test από το test suite κάθε πακέτου.
- *Μακρο-πειράματα:*
  (α) επαναλαμβανόμενα `groupby/agg` πειράματα με Pandas σε μεγάλα δεδομένα,
  (β) επαναλαμβανόμενοι πολλαπλασιασμοί πινάκων και εύρεση ιδιοτιμών–ιδιοδιανυσμάτων με NumPy.

7.2 Μηχάνημα μετρήσεων

Όλα τα πειράματα εκτελέστηκαν στο παρακάτω μηχάνημα:

Hardware:

- CPU: Intel Core i9-10900K @ 3.70 GHz (10 cores, 20 threads; max boost 5.30 GHz)
- Architecture: `x86_64`, 39 bit physical / 48 bit virtual addresses
- Caches: L1d 320 KiB, L1i 320 KiB, L2 2.5 MiB, L3 20 MiB
- RAM: 62 GiB total (16 GiB free, 44 GiB buff/cache)

Software:

- OS: Debian GNU/Linux 12 ("bookworm"), kernel 6.1.0-21-amd64
- Python: 3.11.2
- Pytest: 8.4.1
- Fastapi: 0.116.1
- Flask: 3.2.0.dev0
- Numpy: 2.3.3
- Pandas: 2.3.2

Όλοι οι χρόνοι αναφέρονται σε δευτερόλεπτα, εκτός αν αναφέρεται διαφορετικά.

7.2.1 Μεθοδολογία μετρήσεων

Για την αξιολόγηση του Cerbex χρησιμοποιήθηκαν διαφορετικά σενάρια εκτέλεσης (micro-benchmarks, ολόκληρες βιβλιοθήκες/πλαίσια, καθώς και δύο macro-benchmarks). Σε κάθε περίπτωση πραγματοποιήθηκαν μετρήσεις τόσο χωρίς το εργαλείο (baseline), όσο και με το εργαλείο ενεργό σε διαφορετικές παραλλαγές (π.χ. με `PerfAnalyzer`, `TypeExtractor`).

Flask και FastAPI. Για τα πλαίσια Flask και FastAPI εκτελέστηκε ολόκληρη η σουίτα από τέστ που παρέχουν, χρησιμοποιώντας `pytest`. Στην baseline περίπτωση οι δοκιμές εκτελέστηκαν κανονικά, ενώ στην περίπτωση του Cerbex πραγματοποιήθηκε παρέμβαση στο αρχείο `conftest.py`, ώστε να εγκαθίσταται το API του εργαλείου και να ενεργοποιείται η ενσωμάτωση κώδικα παρακολούθησης, όπως περιγράφεται στο Κεφάλαιο 6. Στη συνέχεια εκτελέστηκαν τα test με την εντολή pytest κανονικά. Ο χρόνος εκτέλεσης μετρήθηκε από το συνολικό χρόνο που αναφέρει το ίδιο το `pytest` μετά την ολοκλήρωση όλων των tests. Για κάθε πλαίσιο, το test suite εκτελέστηκε 1000 φορές σε κάθε διαμόρφωση.

Μικρο-πειράματα. Για τα είκοσι μικρο-πακέτα επιλέχθηκε ένα χαρακτηριστικό τεστ από τη σουίτα κάθε πακέτου, ώστε να υπάρχει αντιπροσωπευτική αλλά ελαφριά φόρτωση. Όπως και με τα πλαίσια, οι μετρήσεις χρόνου προήλθαν από το συνολικό χρονόμετρο του `pytest`. Κάθε μικροπείραμα εκτελέστηκε 100 φορές ανά διαμόρφωση.

Μακρο-πειράματα (Pandas, NumPy). Για τα μακρο-πειράματα δεν χρησιμοποιήθηκε το `pytest`, αλλά η λειτουργία CLI του Cerbex (βλ. Κεφάλαιο 6). Συγκεκριμένα, εκτελέστηκαν δύο αντιπροσωπευτικά σενάρια: (α) επαναλαμβανόμενα `groupby/agg` πειράματα σε μεγάλα δεδομένα με χρήση `Pandas`, και (β) επαναλαμβανόμενοι πολλαπλασιασμοί πινάκων και εύρεση ιδιοτιμών–ιδιοδιανυσμάτων μεγάλων πινάκων με χρήση `NumPy`. Κάθε πείραμα επαναλήφθηκε 10 φορές για κάθε διαμόρφωση (*baseline*, και πλήρης χρήση του Cerbex με ενεργοποιημένα τα `PerfAnalyzer` και `TypeExtractor`). Ο χρόνος εκτέλεσης μετρήθηκε απευθείας μέσα από τον κώδικα Python, χρησιμοποιώντας τη βιβλιοθήκη `time`, λαμβάνοντας τον συνολικό χρόνο εκπαίδευσης/υπολογισμού. Αξίζει να σημειωθεί ότι οι ενδιάμεσες διαμορφώσεις (μόνο `PerfAnalyzer` ή μόνο `TypeExtractor`) δεν μετρήθηκαν, καθώς η επιβάρυνση είναι αμελητέα και δεν παρουσιάζει ιδιαίτερο ενδιαφέρον αφού γνωρίζουμε ήδη την επιβάρυνση του χειρότερου σεναρίου.

Γενικές παρατηρήσεις. Οι τιμές που αναφέρονται στα διαγράμματα και στους πίνακες αντιστοιχούν στον μέσο όρο των επαναλήψεων. Κατά τις μετρήσεις δεν εκτελούνταν άλλες υπολογιστικά εντατικές διεργασίες στο σύστημα, ώστε να μειωθεί ο εξωτερικός θόρυβος.

7.3 Ανάλυση Επιβάρυνσης Απόδοσης

Σημείωση για micro vs macro-benchmarks: Τα micro-benchmarks εξετάζουν λειτουργίες εξαιρετικά μικρής διάρκειας, με χρόνους εκτέλεσης που μετρώνται σε κλάσματα του δευτερολέπτου, ενώ τα macro-benchmarks αναπαριστούν υπολογιστικά εντατικά, ρεαλιστικά workloads. Η σταθερή επιβάρυνση που εισάγει το εργαλείο σε κάθε συνάρτηση προκαλεί υψηλότερα ποσοστιαία slowdowns στα micro-benchmarks, τα οποία όμως είναι ουσιαστικά αμελητέα σε σχέση με τα macro-benchmarks. Ως εκ τούτου, τα macro-benchmarks παρέχουν πιο αντιπροσωπευτική εκτίμηση της πραγματικής επίδρασης του εργαλείου στην απόδοση.

Σε αυτή την ενότητα, ποσοτικοποιούμε την επίδραση του πλαισίου ενσωμάτωσης κώδικα παρακολούθησης στο χρόνο εκτέλεσης σε τρία αντιπροσωπευτικά φορτία εργασίας: FastAPI, Flask, τη μελέτη Micro-Package Echosystem και τη μελέτη Macro-Benchmark.

1. **Baseline** (no tool)
2. **Tool, no analyses, no log** (instrumentation only, `write_reports()` disabled)
3. **Tool, no analyses** (instrumentation + JSON output)
4. **Tool + PerfAnalyzer**
5. **Tool + TypeExtractor**
6. **Tool + PerfAnalyzer + TypeExtractor**

7.3.1 FastAPI

Για κάθε εκτέλεση, αναφέρουμε τον μέσο χρόνο εκτέλεσης, την τυπική απόκλιση, την καθυστέρηση του 95ου εκατοστημορίου και την επιβράδυνση σε σχέση με τη βάση.

Πίνακας 7.1: FastAPI Performance Overhead Results

Configuration	Mean (s)	Std Dev (s)	95th %ile (s)	Overhead (%)
Baseline (no tool)	18.181	0.062	18.270	–
Tool, no analyses, no I/O	35.871	0.146	36.095	+97.3%
Tool, no analyses	39.362	0.144	39.578	+116.5%
Tool + PerfAnalyzer	43.649	0.267	44.027	+140.0%
Tool + TypeExtractor	46.087	0.818	47.322	+153.5%
Tool + PerfAnalyzer + TypeExtractor	51.531	0.939	52.968	+183.4%

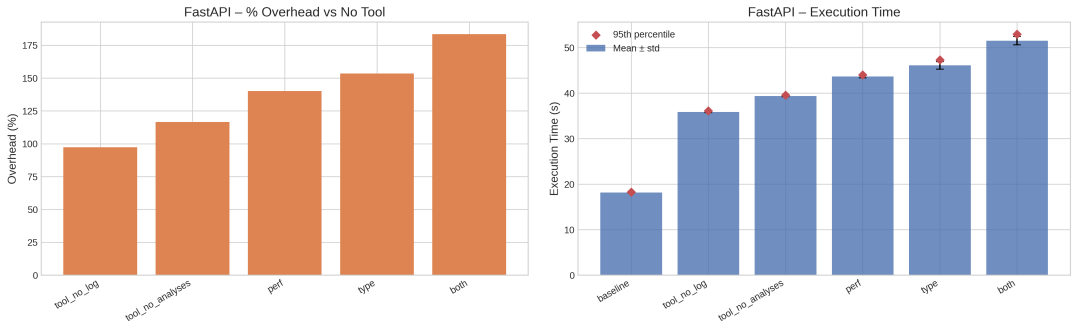
Σχολιασμός: Παρατηρούμε ότι το βασικό κόστος της ενσωμάτωσης (*instrumentation only*) σχεδόν διπλασιάζει τον χρόνο εκτέλεσης σε σχέση με τη βάση (+97%), πριν ακόμη ενεργοποιηθεί οποιαδήποτε ανάλυση ή καταγραφή. Η προσθήκη απλής καταγραφής σε JSON αυξάνει περαιτέρω την επιβάρυνση (+116%), ενώ κάθε επιπλέον ανάλυση εισάγει σταδιακά αυξημένο κόστος: το `PerfAnalyzer` φθάνει το +140%, το `TypeExtractor` το +153%, και ο συνδυασμός τους οδηγεί σε +183%. Η συμπεριφορά αυτή δείχνει ότι το μεγαλύτερο μέρος της επιβάρυνσης προέρχεται από τον ίδιο τον μηχανισμό περιτυλίγματος συναρτήσεων, ενώ οι αναλύσεις προσθέτουν συγκριτικά μικρότερο αλλά σωρευτικό βάρος. Παρ' όλα αυτά, για ένα μεγάλο framework όπως το FastAPI, η επιβάρυνση αυτή παραμένει διαχειρίσιμη σε απόλυτους χρόνους (18s → 52s), κάτι που καθιστά το εργαλείο κατάλληλο για μελέτες σεναρίων μεγάλης κλίμακας.

7.3.2 Flask

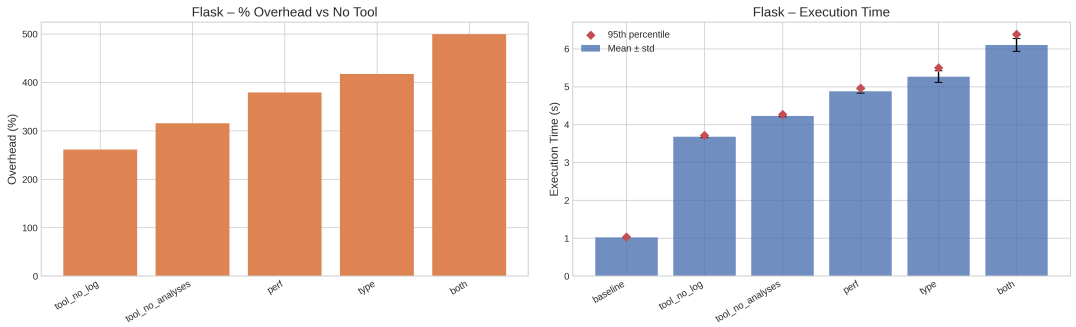
Πίνακας 7.2: Flask Performance Overhead Results

Configuration	Mean (s)	Std Dev (s)	95th %ile (s)	Overhead (%)
Baseline (no tool)	0.991	0.006	1.004	–
Tool, no analyses, no I/O	3.678	0.022	3.716	+271.2%
Tool, no analyses	4.228	0.025	4.267	+326.6%
Tool + PerfAnalyzer	4.880	0.052	4.962	+392.4%
Tool + TypeExtractor	5.432	0.905	5.372	+448.2%
Tool + PerfAnalyzer + TypeExtractor	6.287	0.155	6.521	+534.5%

Σχολιασμός: Στην περίπτωση του Flask, οι απόλυτοι χρόνοι εκτέλεσης είναι πολύ μικρότεροι (1 δευτ. στη βάση), με αποτέλεσμα οι σχετικές επιβραδύνσεις να φαίνονται εξαιρετικά υψηλές. Ακόμη και χωρίς αναλύσεις, η επιβάρυνση φθάνει το +271%, ενώ με πλήρη ενεργοποίηση και των δύο αναλύσεων αγγίζει το +534%. Ωστόσο, η απόλυτη καθυστέρηση είναι μόλις μερικά δευτερόλεπτα (0.99s → 6.28s), πράγμα που δείχνει ότι οι υψηλές ποσοστιαίες τιμές ανταναικλούν κυρίως το πολύ μικρό baseline των micro-benchmarks. Το ίδιο εργαλείο σε μεγάλα, υπολογιστικά εντατικά workloads (macro-benchmarks) προσθέτει πολύ μικρότερο σχετικό overhead, όπως φαίνεται παρακάτω.



Γράφημα 7.1: FastAPI performance results. Left: relative runtime slowdowns (%) compared to baseline. Right: absolute execution times with error bars (mean ± std) and 95th percentile markers.



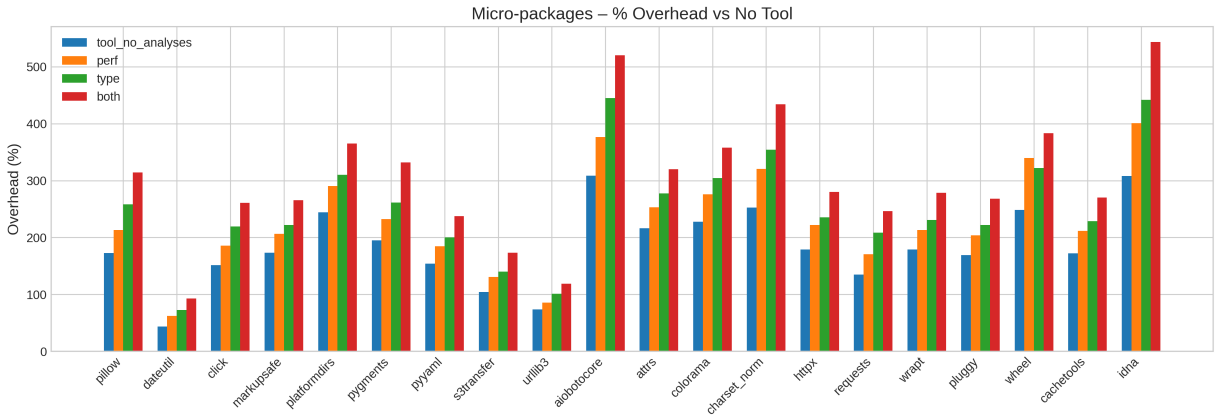
Γράφημα 7.2: Flask performance results. Left: relative runtime slowdowns (%) compared to baseline. Right: absolute execution times with error bars (mean ± std) and 95th percentile markers.

7.3.3 Μικρο-Πειράματα

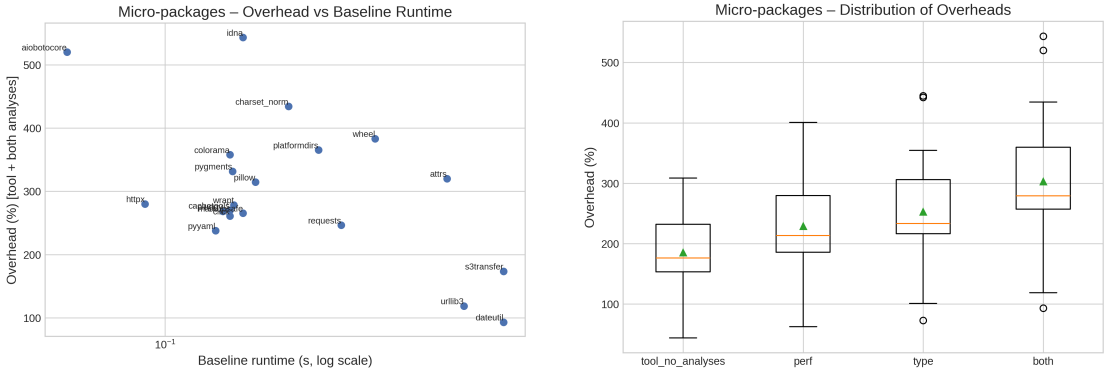
Package	Baseline (s)	No Analyses	PerfAnalyzer	TypeExtractor	Both
pillow	0.160	0.436 (+172%)	0.501 (+213%)	0.573 (+258%)	0.663 (+314%)
dateutil	0.581	0.836 (+44%)	0.943 (+62%)	1.003 (+73%)	1.121 (+93%)
click	0.140	0.352 (+151%)	0.400 (+186%)	0.447 (+219%)	0.505 (+261%)
markupsafe	0.150	0.410 (+173%)	0.460 (+207%)	0.483 (+222%)	0.548 (+265%)
platformdirs	0.222	0.764 (+244%)	0.867 (+291%)	0.911 (+310%)	1.033 (+365%)
pygments	0.142	0.419 (+195%)	0.472 (+232%)	0.513 (+261%)	0.613 (+332%)
pyyaml	0.130	0.330 (+154%)	0.370 (+185%)	0.390 (+200%)	0.439 (+238%)
s3transfer	0.581	1.186 (+104%)	1.342 (+131%)	1.395 (+140%)	1.589 (+173%)
urllib3	0.473	0.821 (+74%)	0.879 (+86%)	0.951 (+101%)	1.034 (+119%)
aiobotocore	0.060	0.245 (+308%)	0.286 (+377%)	0.327 (+445%)	0.372 (+520%)
attrs	0.433	1.369 (+216%)	1.529 (+253%)	1.635 (+278%)	1.819 (+320%)
colorama	0.140	0.459 (+228%)	0.526 (+276%)	0.566 (+304%)	0.641 (+358%)
charset_norm.	0.190	0.670 (+253%)	0.799 (+321%)	0.863 (+354%)	1.015 (+434%)
httpx	0.090	0.251 (+179%)	0.290 (+222%)	0.302 (+236%)	0.342 (+280%)
requests	0.250	0.587 (+135%)	0.677 (+171%)	0.771 (+208%)	0.866 (+246%)
wrapt	0.143	0.399 (+179%)	0.448 (+213%)	0.473 (+231%)	0.541 (+278%)
pluggy	0.135	0.363 (+169%)	0.410 (+204%)	0.435 (+222%)	0.497 (+268%)
wheel	0.298	1.038 (+248%)	1.310 (+340%)	1.258 (+322%)	1.440 (+383%)
cachetools	0.140	0.381 (+172%)	0.436 (+212%)	0.460 (+229%)	0.518 (+270%)
idna	0.150	0.612 (+308%)	0.751 (+401%)	0.813 (+442%)	0.965 (+543%)

Πίνακας 7.3: Mean runtime overhead for 20 representative packages. Values in parentheses are slowdowns relative to baseline.

Σχολιασμός: Στο οικοσύστημα των μικροπακέτων παρατηρείται ετερογένεια: βιβλιοθήκες που εκτελούν πιο «βαριές» λειτουργίες με λιγότερες κλήσεις (π.χ. `dateutil`, `urllib3`) εμφανίζουν μέτριες επιβραδύνσεις της τάξης του +40–100%. Αντίθετα, βιβλιοθήκες που αποτελούνται κυρίως από πολλές μικρές και ταχύτατες κλήσεις συναρτήσεων (π.χ. `idna`, `aiobotocore`, `charset_normalizer`) υφίστανται δυσανάλογα υψηλές επιβραδύνσεις, που ξεπερνούν το +400% ή ακόμη και το +500%. Ο λόγος είναι ότι το σταθερό κόστος που εισάγει το περιτύλιγμα (instrumentation) γύρω από κάθε συνάρτηση γίνεται συγκριτικά πολύ μεγαλύτερο όταν η ίδια η συνάρτηση ολοκληρώνεται σε ελάχιστο χρόνο, χαρακτηριστικό των micro-benchmarks. Συνεπώς, οι υψηλές ποσοστιαίες τιμές δεν αντιστοιχούν σε πραγματικά προβλήματα απόδοσης σε ρεαλιστικά workloads. Το εργαλείο είναι πολύ πιο αποδοτικό σε μεσαία και βαριά φορτία, ενώ σε βιβλιοθήκες με πολύ υψηλή συχνότητα κλήσεων παρουσιάζει σημαντικές σχετικές καθυστερήσεις.



Γράφημα 7.3



(a) Overhead vs. baseline runtime (log scale).

(b) Distribution of overheads across packages.

Γράφημα 7.4: Micro-package results: (a) packages with shorter baseline runtimes suffer disproportionately larger relative slowdowns, (b) distribution of overheads across analysis configurations highlights the wide variance.

7.3.4 Μακρο-πείραμα: Pandas GroupBy Benchmark

Για την αξιολόγηση της συμπεριφοράς του εργαλείου σε υπολογιστικά εντατικά σενάρια, πραγματοποίησαμε ένα μακρο-πείραμα χρησιμοποιώντας `pandas`. Συγκεκριμένα, δημιουργήσαμε ένα συνθετικό dataset με 100,000,000 γραμμές και 10,000 ομάδες, και εκτελέσαμε 350 επαναλήψεις μιας βαριάς λειτουργίας `groupby/agg` για κάθε run. Κάθε πείραμα επαναλήφθηκε 10 φορές για μεγαλύτερη ακρίβεια.

```python
import pandas as pd
import numpy as np
import time

n_rows = 100_000_000
n_groups = 10_000
iterations = 350
runs = 10

df = pd.DataFrame({
    "group": np.random.randint(0, n_groups, size=n_rows, dtype=np.int32),
    "val1": np.random.randn(n_rows).astype(np.float32),
    "val2": np.random.randn(n_rows).astype(np.float32),
    "val3": np.random.randn(n_rows).astype(np.float32),
})

times = []

for run in range(1, runs + 1):
    start = time.time()
    for i in range(iterations):
        df.groupby("group").agg({
            "val1": "mean",
            "val2": "sum",
            "val3": "std",
        })
    end = time.time()
    elapsed = end - start
    times.append(elapsed)
    print(f"Run {run}: {elapsed/60:.2f} minutes")

print(f"Average: {np.mean(times):.2f} ± {np.std(times):.2f} seconds")
print(f"Average: {np.mean(times)/60:.2f} ± {np.std(times)/60:.2f}
    minutes")
```

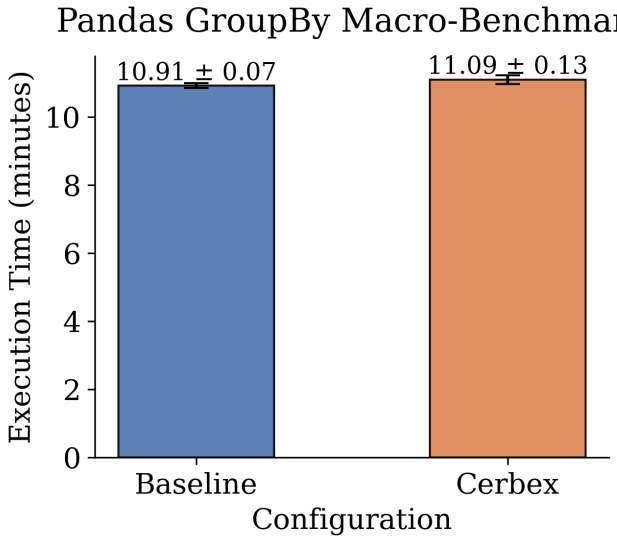
Κώδικας 7.1: Pandas macro-benchmark: groupby/agg on 100M rows, 10k groups, 350 iterations, 10 runs.

Πίνακας 7.4: Pandas GroupBy Macro-Benchmark Times (average over 10 runs).

Configuration	Execution Time	Overhead (%)
Baseline (no tool)	654.81 ± 4.11 s (10.91 ± 0.07 min)	–
Cerbex (instrumented)	665.43 ± 7.68 s (11.09 ± 0.13 min)	+1.6%

Σχολιασμός: Η μέτρηση δέκα επαναλήψεων δείχνει σταθερούς χρόνους εκτέλεσης, με τυπική απόκλιση μικρή σε σχέση με τον μέσο χρόνο. Η επιβάρυνση που εισάγει το εργαλείο είναι μόλις

1.6%, πολύ χαμηλή σε απόλυτο χρόνο, επιβεβαιώνοντας ότι σε μεγάλης κλίμακας workloads η επίδραση της σταθερής επιβάρυνσης ανά συνάρτηση είναι αμελητέα. Αυτή η συμπεριφορά αντιπαραβάλλεται με τα micro-benchmarks, όπου η ίδια σταθερή επιβάρυνση μπορεί να εμφανιστεί ως μεγάλος ποσοστιαίος χρόνος καθυστέρησης.



Γράφημα 7.5: Pandas macro-benchmark results: comparison of execution time between baseline and Cerbex.

7.3.5 Μακρο-πείραμα: NumPy Linear Algebra

Για να εξετάσουμε επιπλέον υπολογιστικά απαιτητικά σενάρια, πραγματοποιήσαμε ένα δεύτερο μακρο-πείραμα με χρήση της βιβλιοθήκης `NumPy`. Η εργασία αυτή βασίστηκε σε επαναλαμβανόμενους πολλαπλασιασμούς πινάκων και εύρεση ιδιοτιμών και ιδιοδιανυσμάτων (`np.linalg.eigh`), ώστε να προσομοιωθεί ένα υπολογιστικά βαρύ φορτίο γραμμικής άλγεβρας. Η διάσταση των πινάκων ορίστηκε σε 2000×2000 και εκτελέστηκαν 800 επαναλήψεις, παράγοντας συνολικά απαιτητικό υπολογιστικό φόρτο. Κάθε πείραμα επαναλήφθηκε 10 φορές για μεγαλύτερη ακρίβεια, όπως φαίνεται στο Παράδειγμα 7.2.

```python
import numpy as np
import time

N = 2000
iterations = 800
runs = 10
times = []

for run in range(1, runs + 1):
    A = np.random.rand(N, N)
    B = np.random.rand(N, N)

    start = time.time()
    for i in range(iterations):
        C = A @ B
        vals, vecs = np.linalg.eigh(C @ C.T)
        A = vecs @ np.diag(vals) @ vecs.T
    end = time.time()

    elapsed = end - start
    times.append(elapsed)
    print(f"Run {run}: {elapsed:.2f} seconds")

print(f"Average: {np.mean(times):.2f} ± {np.std(times):.2f} seconds")
```

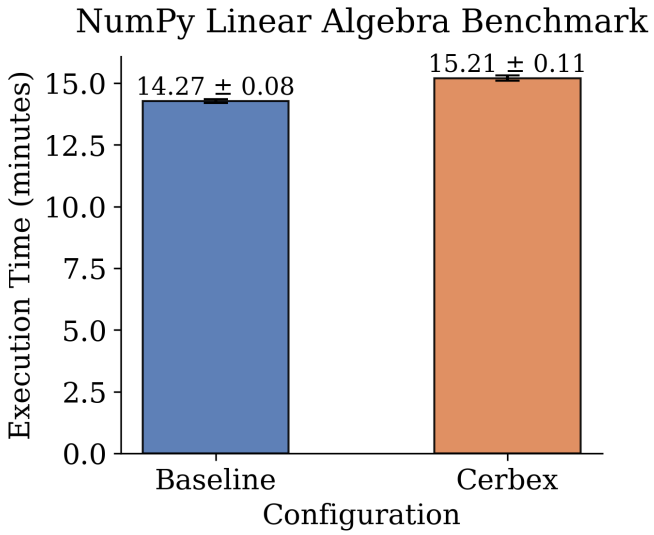
Κώδικας 7.2: NumPy macro-benchmark with repeated matrix multiplications and eigenvalue–eigenvector decompositions.

Ο χρόνος εκτέλεσης μετρήθηκε και εδώ τόσο χωρίς το εργαλείο (*baseline*), όσο και με ενεργοποιημένο το Cerbex.

Πίνακας 7.5: NumPy Linear Algebra Benchmark Times (average over 10 runs).

Configuration	Execution Time	Overhead (%)
Baseline (no tool)	856.3 ± 4.7 s (14.27 ± 0.08 min)	–
Cerbex (instrumented)	912.3 ± 6.7 s (15.20 ± 0.11 min)	+6.5%

Σχολιασμός: Οι επαναλήψεις δείχνουν ότι οι χρόνοι εκτέλεσης παραμένουν σταθεροί, με μικρή τυπική απόκλιση (<1%). Το μέσο κόστος εκτέλεσης κυμαίνεται γύρω στο 5–7%, επιβεβαιώνοντας ότι το εργαλείο είναι κατάλληλο για βαριά workloads. Συγκρίνοντας με τα micro-benchmarks, το χαμηλό ποσοστιαίο overhead σε macro-benchmarks αποδεικνύει ότι οι υψηλές επιβραδύνσεις σε μικρές λειτουργίες οφείλονται κυρίως στη σταθερή επιβάρυνση ανά συνάρτηση και όχι στη συνολική απόδοση του εργαλείου σε πραγματικά σενάρια.



Γράφημα 7.6: NumPy macro-benchmark results: comparison of execution time between baseline and Cerbex.

7.4 Σχετικά έργα με συγκριτική αξιολόγηση

- **Lya:** Το *Lya* αποτελεί ένα coarse-grained πλαίσιο δυναμικής ανάλυσης για JavaScript, το οποίο λειτουργεί σε επίπεδο βιβλιοθηκών: κατά την `εισαγωγή` αναλύει και επανεγγράφει τον πηγαίο κώδικα, εισάγοντας άγκιστρα σε συναρτήσεις και αντικείμενα. Παρέχει έτοιμες αναλύσεις για ασφάλεια, απόδοση και τύπους, ενώ η ανάπτυξη νέων είναι απλή (≈ 100 γραμμές κώδικα). Σε πειράματα με 50 δημοφιλή πακέτα npm, η επιβάρυνση κυμάνθηκε γύρω στο 3.6–4.1%, σημαντικά χαμηλότερη από αυτή του Jalangi, με βελτίωση έως και δύο τάξεων μεγέθους σε ορισμένα σενάρια [53, 24]. Το πλαίσιο προσφέρει χαμηλό κόστος εκτέλεσης και πρακτική δυνατότητα χρήσης σε παραγωγή, εις βάρος όμως της λεπτομέρειας που παρέχουν πιο fine-grained προσεγγίσεις.

- **Jalangi:** Το *Jalangi* αποτελεί, επίσης, ένα ευρέως διαδεδομένο πλαίσιο δυναμικής ανάλυσης για JavaScript. Παρέχει περίπου 28 άγκιστρα γεγονότων (π.χ. αναγνώσεις/εγγραφές μεταβλητών, κλήσεις συναρτήσεων, εξαιρέσεις). Υποστηρίζει προηγμένες λειτουργίες όπως *record–replay*, εντοπισμό σφαλμάτων και ανάπτυξη προσαρμοσμένων αναλύσεων. Ωστόσο, το πλούσιο αυτό σύνολο δυνατοτήτων συνοδεύεται από ιδιαίτερα υψηλό υπολογιστικό κόστος: αναφέρεται επιβράδυνση περίπου 26× κατά την καταγραφή και 30× κατά την αναπαραγωγή, ακόμη και για αναλύσεις χωρίς ουσιαστικό υπολογιστικό φορτίο (*no-op*) [43]. Κατά συνέπεια, το πλαίσιο προσφέρει ευελιξία σε βάρος της απόδοσης.

- **Valgrind:** Το *Valgrind* συνιστά ένα γενικού σκοπού πλαίσιο δυναμικής ανάλυσης σε επίπεδο δυαδικών εκτελέσιμων. Η λειτουργία του βασίζεται στη δυναμική δυαδική ενορχήστρωση (*dynamic binary instrumentation*), κατά την οποία ο κώδικας μηχανής αναδομείται σε μία ενδιάμεση αναπαράσταση (VEX IR) και εκτελείται σε ελεγχόμενο περιβάλλον. Η αρχιτεκτονική του επιτρέπει την ανάπτυξη εξειδικευμένων εργαλείων, όπως τα `memcheck`, `cachegrind` και `callgrind`, τα οποία καλύπτουν ελέγχους μνήμης, καταγραφή και ανάλυση επιδόσεων. Ωστόσο, το γενικευμένο αυτό μοντέλο εισάγει σημαντικό υπολογιστικό κόστος: τυπικά 4–20×, ανάλογα με το εργαλείο που χρησιμοποιείται [23]. Η αξία του έγκειται στην καθολική

του εφαρμοσιμότητα, ανεξαρτήτως γλώσσας προγραμματισμού ή εκτελεστικού περιβάλλοντος, με αντάλλαγμα όμως τη μειωμένη απόδοση.

- **Wasabi:** Το *Wasabi* αποτελεί ένα σύγχρονο πλαίσιο δυναμικής ανάλυσης για τον χώρο του WebAssembly (Wasm). Υποστηρίζει λεπτομερή (*fine-grained*) ανάλυση μέσω εισαγωγής άγκιστρων σε επίπεδο bytecode, καλύπτοντας γεγονότα όπως κλήσεις συναρτήσεων, προσβάσεις μνήμης και χειρισμό ελέγχου ροής. Ξεχωρίζει για τον αποδοτικό σχεδιασμό του: ο τροποποιημένος κώδικας παραμένει σε μορφή WebAssembly και εκτελείται με μέτρια–υψηλή επιβάρυνση (τυπικά 1.3–20×, έως 163× στη χειρότερη περίπτωση), η οποία θεωρείται ανταγωνιστική συγκριτικά με προγενέστερες προσεγγίσεις [15]. Το πλαίσιο εστιάζει κυρίως σε διαδικτυακές εφαρμογές και θέματα ασφάλειας, παρέχοντας ταυτόχρονα ευέλικτα APIs για την ανάπτυξη εξειδικευμένων αναλύσεων.

- **DynaPyt:** Το *DynaPyt* αποτελεί το πρώτο γενικού σκοπού πλαίσιο δυναμικής ανάλυσης για την Python. Βασίζεται στη μέθοδο *source-to-source instrumentation*, εισάγοντας έως και 97 άγκιστρα σε ιεραρχική δομή για ποικίλα γεγονότα εκτέλεσης (π.χ. κλήσεις συναρτήσεων, προσπελάσεις μνήμης, εξαιρέσεις). Παρέχει δυνατότητες επιλεκτικής ενεργοποίησης γεγονότων (*pay-per-use*) καθώς και άμεσης τροποποίησης της εκτέλεσης (π.χ. αλλαγή τιμών). Το υπολογιστικό κόστος εξαρτάται από το πλήθος των ενεργών γεγονότων: με πλήρες instrumentation (*TraceAll*), η εκτέλεση είναι 1.2–16× βραδύτερη σε σχέση με το μη τροποποιημένο πρόγραμμα, αλλά παραμένει 5.6–88.6% ταχύτερη συγκριτικά με την ενσωματωμένη λειτουργία `sys.settrace` της Python [7].

- **PySecu:** Το *PySecu* είναι ένα εργαλείο αδρομερούς δυναμικής ανάλυσης για Python, το οποίο ακολουθεί τη λογική της «επαναπλαισιοποίησης βιβλιοθηκών» κατά τα πρότυπα του Lya για JavaScript. Η λειτουργία του βασίζεται στην αγκιστροποίηση του μηχανισμού `εισαγωγής`, ώστε κατά τη φόρτωση ενός αρθρώματος να εφαρμόζεται περιτύλιξη (*wrapping*) σε συναρτήσεις, κλάσεις και μεθόδους. Μέσω αυτής της τεχνικής, το PySecu επιβάλλει απλή πολιτική ελέγχου τύπου *allow/deny*, καταγράφοντας ή και περιορίζοντας τις προσβάσεις που πραγματοποιεί ο κώδικας. Σε πειράματα με πληθώρα βιβλιοθηκών αναφέρθηκε μέση επιβάρυνση περίπου 3× σε σχέση με την εκτέλεση χωρίς ανάλυση, τιμή που είναι συγκρίσιμη με το `sys.settrace` αλλά και με αντίστοιχα εργαλεία αδρομερούς ανάλυσεις για άλλες γλώσσες [8]. Η συμβολή του έγκειται στην απλότητα χρήσης και την πρακτική αξιοποίηση σε σενάρια ασφάλειας, αν και υστερεί σε λεπτομερή κάλυψη γεγονότων συγκριτικά με πιο fine-grained προσεγγίσεις όπως το DynaPyt.

Εργαλείο	**Τεχνική Instrumentation**	**Κάλυψη Γεγονότων**	**Επιβάρυνση / Απόδοση**	**Ιδιαιτερότητες**
Lya	Μετασχηματισμοί σε επίπεδο βιβλιοθηκών (*module recontextualization*)	Εισαγωγές modules, κλήσεις συναρτήσεων, προσβάσεις σε αντικείμενα	Πολύ χαμηλή επιβάρυνση (≈ 3.6–4.1%)	Απλή συγγραφή αναλύσεων (≈ 100 γραμμές), διαθέσιμες αναλύσεις για ασφάλεια/απόδοση/τύπους, λιγότερο fine-grained λεπτομέρεια συγκριτικά με Jalangi
Jalangi	AST μετασχηματισμοί με εισαγωγή hooks σε bytecode	Πλήρες φάσμα (≈28 γεγονότα: reads/writes, calls, exceptions)	Πολύ υψηλή επιβάρυνση (≈26–30×)	Υποστηρίζει record–replay και debugging· προσφέρει λεπτομερή ανάλυση με σημαντικό κόστος
DynaPyt	Source-to-source AST instrumentation	Εκτεταμένη κάλυψη (έως 97 hooks σε ιεραρχική δομή)	Μέτρια–υψηλή επιβάρυνση (1.2–16×)· ταχύτερο από `sys.settrace`	Επιλεκτική ενεργοποίηση γεγονότων (*pay-per-use*), δυνατότητα άμεσης τροποποίησης εκτέλεσης
PySecu	Import-based wrapping σε βιβλιοθήκες	Κλήσεις συναρτήσεων και μεθόδων σε imported modules	Μέση επιβάρυνση (≈3×)	Coarse-grained έλεγχος τύπου allow/deny· απλή χρήση για σενάρια ασφάλειας, υστερεί σε λεπτομέρεια σε σχέση με fine-grained πλαίσια
Valgrind	Dynamic binary instrumentation (μετατροπή σε VEX IR)	Εντολές μηχανής, προσπελάσεις μνήμης, caches, κλήσεις συναρτήσεων	Υψηλή επιβάρυνση (≈4–20×)	Ανεξάρτητο από τη γλώσσα· παρέχει εργαλεία υψηλής ακρίβειας όπως `memcheck`, `cachegrind`
Wasabi	Binary Instrumentation σε επίπεδο WebAssembly bytecode	Κλήσεις συναρτήσεων, μνήμη, control-flow events	Μέτρια–Υψηλή επιβάρυνση (≈1.3–20×, έως 163× στη χειρότερη περίπτωση)	Σχεδιασμένο για WebAssembly· προσφέρει APIs για ανάπτυξη custom αναλύσεων
Cerbex (προτεινόμενο)	Import-based instrumentation με `MetaPathFinder` + `sys.setprofile`	Εισαγωγές modules, κλήσεις συναρτήσεων Python και C-επεκτάσεων	Επιβάρυνση από πολύ χαμηλή (1–7% σε macro workloads) έως μετρήσιμη σε frameworks· υψηλή σε micro-packages	Υποστηρίζει Learn/Enforce modes· modular plugins· εστιάζει σε πολιτικές ασφαλείας

Πίνακας 7.6: Συγκριτική παρουσίαση υπαρχόντων πλαισίων δυναμικής ανάλυσης και του προτεινόμενου *Cerbex*.

7.5 Διαφορές της παρούσας εργασίας από υπάρχουσες προσεγγίσεις

Το προτεινόμενο εργαλείο συνδυάζει βασικές ιδέες από υπάρχοντα πλαίσια δυναμικής ανάλυσης, με έμφαση στην εκμάθηση και επιβολή πολιτικών ασφαλείας. Όπως τα Lya και PySecu, παρακολουθεί γεγονότα (events) σε επίπεδο βιβλιοθηκών και αρθρωμάτων Python, χρησιμοποιώντας μετασχηματισμούς κατά την εισαγωγή (import-time) για την ενσωμάτωση άγκιστρων. Αντίθετα με άλλα εργαλεία όπως τα Jalangi, DynaPyt ή Valgrind, που πραγματοποιούν πλήρη τροποποίηση του κώδικα, bytecode ή δυαδικών εκτελέσιμων, το εργαλείο αξιοποιεί τις εγγενείς δυνατότητες του Python runtime.

Συγκεκριμένα, ένας προσαρμοσμένος `MetaPathFinder` στο `sys.meta_path` παρεμβαίνει κατά την εισαγωγή των στόχων, παρέχοντας έναν φορτωτή (loader) που τυλίγει ή παρακολουθεί τον κώδικα των αρθρωμάτων σε πραγματικό χρόνο [51]. Έτσι, επιτυγχάνεται παρακολούθηση Python άρθρωμάτων, συμπεριλαμβανομένων των εισαγόμενων αρθρωμάτων και C-επεκτάσεων, χωρίς τροποποίηση του διερμηνέα.

Σε σύγκριση με πλαίσια όπως το Wasabi, που παρέχει fine-grained ενσωμάτωση κώδικα παρακολούθησης σε περιβάλλον WebAssembly, το Cerbex εφαρμόζει παρόμοιο coarse-to-fine έλεγχο σε Python, με χαμηλή επιβάρυνση και δυνατότητα ανάπτυξης προσαρμοσμένων αναλύσεων. Πέραν της παρακολούθη-σης, το εργαλείο ενσωματώνει λογική πολιτικών: μπορεί να «μάθει» τυπικές αλληλεπιδράσεις, όπως αλληλουχίες κλήσεων μεταξύ βιβλιοθηκών, και στη συνέχεια να επιβάλλει κανόνες επιτρεπτών ή απαγορευμένων ενεργειών σε πραγματικό χρόνο. Με αυτόν τον τρόπο διαφοροποιείται από εργαλεία που περιορίζονται σε απλή καταγραφή, όπως τα Jalangi, DynaPyt και PySecu.

7.6 Συζήτηση & Περιορισμοί

7.6.1 Συζήτηση

Τα αποτελέσματα δείχνουν ότι το εργαλείο επιτυγχάνει τον στόχο της ορθότητας, καταγράφοντας εισαγωγές, κλήσεις και επιστροφές συναρτήσεων με τρόπο ισοδύναμο με τους εγγενείς μηχανισμούς της Python. Επιπλέον, η αρχιτεκτονική του πλαισίου επιτρέπει την εύκολη ανάπτυξη αναλύσεων (π.χ. `PerfAnalyzer`, `TypeExtractor`) με ελάχιστο κώδικα, επιβεβαιώνοντας τον στόχο της συντομίας.

7.6.2 Περιορισμοί

Παρά τα θετικά αποτελέσματα, η αξιολόγηση ανέδειξε ορισμένους περιορισμούς:

Διακοσμητές (decorators) και Πλαίσια (frameworks). Ένα θεμελιώδες πρόβλημα που έχει αναφερθεί ως πιθανό αφορά συναρτήσεις που τυλίγονται με *decorators* σε πλαίσια όπως το FastAPI. Οι *decorators* εκτελούνται κατά το *import-time*, με αποτέλεσμα η αρχική συνάρτηση να αντικαθίσταται ή να τυλίγεται πριν προλάβει να εφαρμοστεί ο κώδικας παρακολούθησης μετά την *εισαγωγή*. Έτσι, *handlers* που δηλώνονται με `@app.get(...)` ή παρόμοιες δηλώσεις μπορεί να μην καταγράφονται από *post-import* προσεγγίσεις. Το ζήτημα αυτό δεν έχει πλήρως τεκμηριωθεί για όλα τα πλαίσια, αλλά υπάρχουν συζητήσεις και περιορισμοί στη βιβλιογραφία [26] και σε τεχνολογίες παρακολούθησης (π.χ. OpenTelemetry) που δείχνουν παρόμοια συμπεριφορά.

Αλλοίωση στοίβας κλήσεων(call stack) και ενδοσκόπησης (introspection). Ο μηχανισμός περιτυλίγματος του εργαλείου μεταβάλλει τη στόιβα κλήσεων και τα μεταδεδομένα των

συναρτήσεων (π.χ. `inspect.signature`, annotations, default τιμές). Αυτό οδηγεί σε προβλήματα με όσαα πλάισια βασίζονται σε ακριβή introspection ή σε ανάλυση του traceback stack:

- **FastAPI:** Η έγχυση εξαρτήσεων και η λογική επικύρωσης του FastAPI απαιτούν ακριβείς υπογραφές συναρτήσεων και συνεπή αναπαράσταση των frames. Η αλλοίωση αυτή προκάλεσε 24 αποτυχίες σε 2.469 tests (˜0.97%), με διαφορές σε κωδικούς κατάστασης (π.χ. 422 αντί 200) και αποκλίσεις σε απαντήσεις σε μορφή JSON.
- **Flask:** Ο μηχανισμός CLI του Flask χρησιμοποιεί `sys.exc_info()` και `traceback` για να ταξινομήσει σφάλματα φόρτωσης εφαρμογών. Η μεταβολή του βάθους των frames από το fallback importer οδήγησε σε 4 αποτυχίες σε 450 tests (˜0.88%), με `NoAppException` να εγείρεται σε διαφορετικά σημεία και με απουσία αναμενόμενων μηνυμάτων (`FLASK_APP`) στο stderr.

Οι αποτυχίες αυτές δεν οφείλονται στα ίδια τα πλαίσια αλλά στις παρενέργειες του Cerbex, ο οποίος μεταβάλλει τη στοίβα σε κρίσιμα σημεία όπου απαιτείται ακρίβεια.

Δυναμικά παραγόμενες συναρτήσεις (pytest). Ένα ακόμη ζήτημα αναδείχθηκε στη χρήση του `pytest`. Για κλάσεις που κληρονομούν από `unittest.TestCase`, οι μέθοδοι εκτελούνται απευθείας από τη βιβλιοθήκη `unittest`. Αυτό σημαίνει ότι η κλήση περνά από το ίδιο το αντικείμενο-μέθοδο που έχει ήδη τυλιχθεί από το εργαλείο και επομένως καταγράφεται. Αντίθετα, για «απλές» κλάσεις Python (π.χ. `class TestInputTypes(object)`), το `pytest` στο στάδιο *collection* αποσπά τις μεθόδους και δημιουργεί νέα `Function` αντικείμενα, τα οποία εκτελούνται αντί των αρχικών μεθόδων. Επειδή η ενσωμάτωση κώδικα παρακολούθησης είχε εφαρμοστεί μόνο στις αρχικές μεθόδους, οι κλήσεις αυτών των κλάσεων δεν καταγράφονται. Το πρόβλημα είναι εγγενές στον τρόπο που το `pytest` διαχειρίζεται τις μεθόδους και δεν μπορεί να αντιμετωπιστεί με γενικές post-import τεχνικές.

Επίδραση στις μετρήσεις. Αν και τα αποτυχημένα tests εκτελούνταν υπό πλήρη instrumentation και επομένως συνέβαλαν στη μέτρηση του κόστους εκτέλεσης, σε ορισμένες περιπτώσεις οι εκτελέσεις τερματίστηκαν νωρίτερα από ό,τι χωρίς το εργαλείο. Αυτό μπορεί να οδηγεί σε ελαφριά υποεκτίμηση του κόστους για συγκεκριμένα μονοπάτια κώδικα. Ωστόσο, τα αποτελέσματα θεωρούνται αντιπροσωπευτικά του κόστους που θα προέκυπτε και σε επιτυχείς αλλά παρόμοιες ροές εκτέλεσης.

Στατική φύση του μοντέλου εκμάθησης/επιβολής. Το Cerbex στηρίζεται σε δύο φάσεις: κατά τη *λειτουργία μάθησης* καταγράφει γεγονότα και δημιουργεί ένα σύνολο «κανόνων», ενώ κατά το *λειτουργία επιβολής* εφαρμόζει αυστηρά αυτούς τους κανόνες. Αυτό σημαίνει ότι συμπεριφορές που δεν παρατηρήθηκαν στη φάση εκμάθησης —ακόμη και αν είναι ακίνδυνες— θα επισημανθούν ως παραβιάσεις. Η προσέγγιση αυτή μπορεί να οδηγήσει σε *ψευδώς θετικά* στα πραγματικά περιβάλλοντα, όπου οι εκτελέσεις σπάνια αναπαράγουν πλήρως όλα τα πιθανά μονοπάτια. Το ζήτημα αυτό περιορίζει την πρακτική χρήση του εργαλείου σε σενάρια όπου υπάρχει σαφής και πλήρης βάση εκπαίδευσης.

7.6.3 Μελλοντική Εργασία

Για να ξεπεραστούν τα παραπάνω όρια, μελλοντική εργασία μπορεί να κινηθεί στις εξής κατευθύνσεις:

- **Στοχευμένη ενσωμάτωση:** Το εργαλείο ήδη υποστηρίζει επιλεκτικό περιτύλιγμα μέσω `config.json`, όπου ο χρήστης μπορεί να ορίσει ποια αρθρώματα θα παρακολουθούνται. Μελλοντική δουλειά μπορεί να εστιάσει στη βελτίωση αυτής της δυνατότητας με πιο εκφραστικά φίλτρα ή με δυναμική επιλογή στόχων κατά το χρόνο εκτέλεσης.

- **Ειδικές επεκτάσεις πλαισίων:** Ανάπτυξη ειδικών πρόσθετων εργαλείων για FastAPI, Flask και άλλα decorator-heavy πλαίσια, με στόχο την καταγραφή διαχειριστών (handlers) που δηλώνονται κατά την εισαγωγή. Παρόμοιες προσεγγίσεις έχουν υλοποιηθεί σε συστήματα παρακολούθησης όπως το OpenTelemetry [26], όπου απαιτείται ειδική επέκταση στην ενσωμάτωση κώδικα παρακολούθησης για να διασφαλιστεί η πλήρης ορατότητα των διαχειριστών.

- **Διατήρηση μεταδεδομένων:** Βελτίωση του μηχανισμού περιτύλιξης ώστε να διατηρούνται πλήρως οι υπογραφές, τα annotations και τα docstrings των συναρτήσεων, αποφεύγοντας ασυμβατότητες.

- **Βελτίωση επιδόσεων:** Μείωση του κόστους εκτέλεσης σε σενάρια με ιδιαίτερα υψηλή συχνότητα κλήσεων, αξιοποιώντας πιο αποδοτικούς μηχανισμούς ιχνηλάτησης [44].

- **Επέκταση με πρόσθετα εργαλεία:** Σχεδίαση και ανάπτυξη περισσότερων αναλύσεων (π.χ. παρακολούθηση ρύπων δεδομένων (taint tracking), ανάλυση χρήσης μνήμης (memory profiling), ανίχνευση ανωμαλιών (anomaly detection)), ώστε να διευρυνθεί το φάσμα των χρήσεων του εργαλείου.

- **Υποστήριξη εγγενής αρθρωμάτων:** Αν και το εργαλείο χρησιμοποιεί ήδη μηχανισμούς τύπου `sys.setprofile` για παρακολούθηση σε Python-level frames, μελλοντική εργασία μπορεί να εστιάσει στην ενσωμάτωση άγκιστρων σε εγγενή modules (C/C++ επεκτάσης), ώστε να παρακολουθείται η εκτέλεση και η ροή δεδομένων σε χαμηλού επιπέδου κώδικα που δεν καλύπτεται από το Python runtime tracing. Αυτό θα διευρύνει την εφαρμοσιμότητα του εργαλείου σε βιβλιοθήκες υψηλής απόδοσης και εγγενής επεκτάσεις (native extensions).

- **Εξέλιξη μοντέλου μάθησης/επιβολής:** Στην παρούσα μορφή, το εργαλείο καταγράφει γεγονότα σε αρχείο κατά τη *λειτουργία μάθησης* και, σε *λειτουργία επιβολής*, εφαρμόζει τους κανόνες που προέκυψαν. Αυτό σημαίνει ότι συμπεριφορές που δεν εμφανίστηκαν στη φάση εκμάθησης —ακόμη και αν είναι αθώες— θα σημειώνονται ως παραβιάσεις. Μελλοντική εργασία μπορεί να επεκτείνει αυτή την προσέγγιση προς πιο σύγχρονες τεχνικές δυναμικής ανάλυσης και ασφάλειας, όπως προσαρμοστικά μοντέλα, ανίχνευση αποκλίσεων (deviation detection) ή συνδυασμό με μηχανισμούς μηχανικής μάθησης, ώστε να μειωθούν τα ψευδώς θετικά και να βελτιωθεί η ακρίβεια στην ανίχνευση κακόβουλων συμπεριφορών.

- **Συνδυασμός στατικής και δυναμικής ανάλυσης:** Εξερεύνηση υβριδικών προσεγγίσεων που θα επιτρέπουν κάλυψη περισσότερων μονοπατιών με χαμηλότερο κόστος εκτέλεσης κατά το χρόνο εκτέλεσης.

Κεφάλαιο 8

Συμπεράσματα

Σε αυτή τη διπλωματική εργασία παρουσιάσαμε το **Cerbex**, ένα νέο εργαλείο δυναμικής ανάλυσης για την Python. Το Cerbex αξιοποιεί τον μηχανισμό εισαγωγών και το προφίλ κλήσεων συναρτήσεων ώστε να παρεμβάλλει κώδικα παρακολούθησης κατά τον χρόνο εκτέλεσης, χωρίς να απαιτείται τροποποίηση του πηγαίου κώδικα ή του διερμηνέα. Παρέχει δύο λειτουργίες: **μάθησης**, όπου καταγράφεται με ακρίβεια η συμπεριφορά και οι εξαρτήσεις ενός προγράμματος, και **επιβολής**, όπου εφαρμόζονται πολιτικές ασφάλειας σε πραγματικό χρόνο. Η αρθρωτή του αρχιτεκτονική, βασισμένη σε plugins, επιτρέπει την εύκολη ανάπτυξη νέων αναλύσεων και μελλοντικών επεκτάσεων. Το εργαλείο είναι γραμμένο στην ίδια γλώσσα που αναλύει —την Python— και διατίθεται ελεύθερα για εγκατάσταση και πειραματισμό μέσω GitHub.

Η εργασία κινήθηκε με βάση τέσσερις κύριες επιδιώξεις που τέθηκαν στο Κεφάλαιο 1: **διαφάνεια**, **χαμηλό κόστος εκτέλεσης**, **επεκτασιμότητα** και **δυνατότητα επιβολής πολιτικών**. Από την ανάπτυξη και αξιολόγηση του Cerbex προέκυψαν τα εξής βασικά συμπεράσματα:

- **Διαφάνεια:** Το εργαλείο λειτουργεί χωρίς να αλλοιώνει τη σημασιολογία του προγράμματος, καταγράφοντας με ακρίβεια τόσο άμεσες όσο και έμμεσες κλήσεις, καθώς και εξαρτήσεις modules που φορτώνονται δυναμικά.
- **Επιβολή πολιτικών:** Η διάκριση μεταξύ λειτουργίας μάθησης και λειτουργίας επιβολής παρέχει έναν πρακτικό μηχανισμό ελέγχου, ο οποίος μπορεί να περιορίσει την εκτέλεση σε επιτρεπόμενα imports και κλήσεις συναρτήσεων.
- **Απόδοση:** Η επιβάρυνση χρόνου εκτέλεσης είναι αισθητή σε μικρής κλίμακας ή έντονα κλητοκεντρικά σενάρια, αλλά παραμένει διαχειρίσιμη σε μεγαλύτερες εφαρμογές και πλαίσια, καθιστώντας το εργαλείο αξιοποιήσιμο σε πραγματικές συνθήκες.
- **Επεκτασιμότητα:** Η αρθρωτή αρχιτεκτονική του Cerbex, βασισμένη σε πρόσθετα εργαλεία, επιτρέπει την εύκολη ανάπτυξη νέων αναλύσεων (π.χ. PerfAnalyzer, TypeExtractor), δείχνοντας την ευελιξία και τη δυνατότητα προσαρμογής του σε διαφορετικά σενάρια χρήσης.

Η αξιολόγηση (Κεφάλαιο 7) έδειξε ότι το Cerbex ανταποκρίνεται ικανοποιητικά στους στόχους που είχαν τεθεί. Σε σχέση με υπάρχουσες προσεγγίσεις, προσφέρει έναν πιο ισορροπημένο συνδυασμό ακρίβειας, πρακτικότητας και δυνατότητας επιβολής σε πραγματικό χρόνο. Με τον τρόπο αυτό, καλύπτει ένα ερευνητικό κενό που αφορά εργαλεία τα οποία μπορούν να χρησιμοποιηθούν τόσο για **ανάλυση** όσο και για **ασφάλεια** σε παραγωγικά περιβάλλοντα.

Παράλληλα, η εργασία ανέδειξε ορισμένους περιορισμούς και κατευθύνσεις για μελλοντική έρευνα:

- **Βελτίωση επιδόσεων** σε σενάρια με εξαιρετικά υψηλή συχνότητα κλήσεων, μέσω βελτιστοποίησης του μηχανισμού παρακολούθησης κώδικα ή αξιοποίησης πιο αποδοτικών tracing APIs[44].
- **Υποστήριξη πιο σύνθετων πολιτικών**, όπως context-aware κανόνες που εξαρτώνται από την ακολουθία κλήσεων ή το περιβάλλον χρόνου εκτέλεσης.
- **Συνδυασμός στατικής και δυναμικής ανάλυσης**, ώστε να επιτευχθεί καλύτερη κάλυψη εκτελέσιμων μονοπατιών με χαμηλότερο κόστος.
- **Ενσωμάτωση με σύγχρονους μηχανισμούς της Python**, όπως τα audit hooks [6], με στόχο τη βαθύτερη σύνδεση με το ίδιο το runtime.
- **Επέκταση με περισσότερα plugins**, ώστε να καλύπτονται επιπλέον σενάρια ανάλυσης (π.χ. παρακολούθηση μνήμης, data flow tracking, ανίχνευση ανωμαλιών).
- **Διερεύνηση πραγματικών εφαρμογών**, μέσα από deployment σε μεγαλύτερα έργα ή παραγωγικά συστήματα, ώστε να αξιολογηθεί περαιτέρω η πρακτική αξία του εργαλείου.

Εν κατακλείδι, η εργασία αυτή κατέδειξε ότι είναι εφικτό να αναπτυχθεί ένα εργαλείο δυναμικής ανάλυσης για την Python που να είναι **απλό στη χρήση**, **μη παρεμβατικό** ως προς τον κώδικα και ταυτόχρονα να προσφέρει **χρήσιμη πληροφορία** και **δυνατότητες ελέγχου**. Το Cerbex συμβάλλει στην κατανόηση και στην ασφάλεια εφαρμογών Python, ανοίγοντας τον δρόμο για περαιτέρω έρευνα και ανάπτυξη στην περιοχή της δυναμικής ανάλυσης.

Βιβλιογραφία

[1] D. Ancona, M. Ancona, A. Cuni, and N. D. Matsakis, "Rpython: A step towards reconciling dynamically and statically typed oo languages," in *Proceedings of the 2007 Symposium on Dynamic Languages*, ser. DLS '07. New York, NY, USA: ACM, 2007, pp. 53–64. [Online]. Available: https://doi.org/10.1145/1297081.129709

[2] E. Andreasen, L. Gong, A. Møller, M. Pradel, M. Selakovic, K. Sen, and C.-A. Staicu, "A survey of dynamic analysis and test generation for javascript," *ACM Comput. Surv.*, vol. 50, no. 5, Sep. 2017. [Online]. Available: https://doi.org/10.1145/3106739

[3] Z. Chen, L. Chen, Y. Zhou, Z. Xu, W. C. Chu, and B. Xu, "Dynamic slicing of python programs," in *2014 IEEE 38th Annual Computer Software and Applications Conference*, 2014, pp. 219–228.

[4] L. Christophe, C. De Roover, and W. De Meuter, "Dynamic analysis using javascript proxies," in *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ser. ICSE '15. IEEE Press, 2015, p. 813–814.

[5] A. Decan, T. Mens, and M. Claes, "On the topology of package dependency networks: a comparison of three programming language ecosystems," in *Proccedings of the 10th European Conference on Software Architecture Workshops*, ser. ECSAW '16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2993412.3003382

[6] S. Dower, "Pep 578: Python runtime audit hooks," Python Software Foundation, Tech. Rep., 2018, python Enhancement Proposal.

[7] A. Eghbali and M. Pradel, "Dynapyt: a dynamic analysis framework for python," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 760–771. [Online]. Available: https://doi.org/10.1145/3540250.3549126

[8] K. Filopoulos, "Coarse-grained dynamic analysis for python," Diploma Thesis, Technical University of Crete, Greece, 2022. [Online]. Available: https://dias.library.tuc.gr/view/94244

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

[10] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[11] J. Hunt, *Decorators*. Cham: Springer International Publishing, 2023, pp. 339–351. [Online]. Available: https://doi.org/10.1007/978-3-031-35122-8_29

[12] Intel, "Dynamic analysis vs. static analysis," https://www.intel.com/content/www/us/en/develop/documentation/inspector-user-guide-windows/top/getting-started/dynamic-analysis-vs-static-analysis.html, 2022, accessed: 2025-09-21.

[13] R. Kikas, G. Gousios, M. Dumas, and D. Pfahl, "Structure and evolution of package dependency networks," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, 2017, pp. 102–112.

[14] R. Laddad, *AspectJ in Action: Practical Aspect-Oriented Programming*. USA: Manning Publications Co., 2003.

[15] D. Lehmann and M. Pradel, "Wasabi: A framework for dynamically analyzing webassembly," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*, 2019, pp. 1045–1058. [Online]. Available: https://doi.org/10.1145/3297858.3304068

[16] L. Li, J. Wang, and H. Quan, "Scalpel: The python static analysis framework," 2022. [Online]. Available: https://arxiv.org/abs/2202.11840

[17] C. Lopes, G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, J.-m. Loingtier, and J. Irwin, "Aspect-oriented programming," *ACM Computing Surveys*, vol. 28, 10 1999.

[18] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," *SIGPLAN Not.*, vol. 40, no. 6, p. 190–200, Jun. 2005. [Online]. Available: https://doi.org/10.1145/1064978.1065034

[19] J. Mahon, C. Hou, and Z. Yao, "Pypitfall: Dependency chaos and software supply chain vulnerabilities in python," 2025. [Online]. Available: https://arxiv.org/abs/2507.18075

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

[21] P. Mehta, S. Dorkenwald, D. Zhao, T. Kaftan, A. Cheung, M. Balazinska, A. Rokem, A. Connolly, J. Vanderplas, and Y. AlSayyad, "Comparative evaluation of big-data systems on scientific image analytics workloads," *arXiv preprint arXiv:1612.02485*, 2016. [Online]. Available: https://doi.org/10.48550/arXiv.1612.02485

[22] MITRE, "Cwe-502: Deserialization of untrusted data," 2023, accessed: 2025-09-21. [Online]. Available: https://cwe.mitre.org/data/definitions/502.html

[23] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 89–100. [Online]. Available: https://doi.org/10.1145/1250734.1250746

[24] G. Ntousakis, "Lya: A general-purpose dynamic program analysis tool for javascript," Diploma Thesis, Technical University of Crete, 2020. [Online]. Available: https://dias.library.tuc.gr/view/87056

[25] G. Ntousakis, S. Ioannidis, and N. Vasilakis, "Demo: Detecting third-party library problems with combined program analysis," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21)*, 2021, pp. 2429–2431.

[26] OpenTelemetry Project, "Instrumentation libraries (monkey-patching via library hooks)," https://opentelemetry.io/docs/concepts/instrumentation/libraries/, 2025, accessed: 2025-09-06.

[27] OWASP Foundation, "Insecure deserialization," https://owasp.org/www-community/vulnerabilities/Deserialization_of_untrusted_data, accessed July 2025.

[28] Pillow contributors, *Pillow (PIL Fork) Documentation*, 2023, accessed: 2025-09-21. [Online]. Available: https://pillow.readthedocs.io/en/stable/

[29] Python Software Foundation, "Python/c api reference manual," https://docs.python.org/3/c-api/index.html, accessed: 2025-09-28.

[30] ——, *Extending and Embedding the Python Interpreter*, 2023, accessed: 2025-09-21. [Online]. Available: https://docs.python.org/3/extending/

[31] ——, *functools — Higher-order functions and operations on callable objects*, 2023, accessed: 2025-09-21. [Online]. Available: https://docs.python.org/3/library/functools.html

[32] ——, *import system — Python import internals*, 2023, accessed: 2025-09-21. [Online]. Available: https://docs.python.org/3/reference/import.html

[33] ——, *importlib.machinery — Import machinery*, 2023, accessed: 2025-09-21. [Online]. Available: https://docs.python.org/3/library/importlib.html

[34] ——, *json — JSON encoder and decoder*, 2023, accessed: 2025-09-21. [Online]. Available: https://docs.python.org/3/library/json.html

[35] ——, *pickle — Python object serialization*, 2023, accessed: 2025-09-21. [Online]. Available: https://docs.python.org/3/library/pickle.html

[36] ——, *profile, cProfile, and pstats — Python profiling tools*, 2023, accessed: 2025-09-21. [Online]. Available: https://docs.python.org/3/library/profile.html

[37] ——, *sys.setprofile — Profiling functions*, 2023, accessed: 2025-09-21. [Online]. Available: https://docs.python.org/3/library/sys.html#sys.setprofile

[38] ——, *sys.settrace — Trace functions*, 2023, accessed: 2025-09-21. [Online]. Available: https://docs.python.org/3/library/sys.html#sys.settrace

[39] ——, *threading.local — Thread-local data*, 2023, accessed: 2025-09-21. [Online]. Available: https://docs.python.org/3/library/threading.html#threading.local

[40] ——, *inspect — Inspect Live Objects*, Python Software Foundation, 2025, online; accessed 2025-07-07. [Online]. Available: https://docs.python.org/3/library/inspect.html

[41] K. Reitz and contributors, *Requests: HTTP for Humans*, 2023, accessed: 2025-09-21. [Online]. Available: https://docs.python-requests.org/en/latest/

[42] Y. Selivanov, "Pep 567: Context variables," Python Software Foundation, Python Enhancement Proposal, 2017, accessed: 2025-09-28. [Online]. Available: https://peps.python.org/pep-0567/

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

[44] M. Shannon, "Pep 669: Low impact monitoring for cpython," Python Software Foundation, Python Enhancement Proposal, 2021, python-Version: 3.12; accessed 2025-07-08. [Online]. Available: https://peps.python.org/pep-0669/

[45] K. D. Smith *et al.*, "Pep 318 – decorators for functions and methods," https://peps.python.org/pep-0318/, 2003, accessed: 2025-09-27.

[46] E. Snow, "Pep 451: A modulespec type for the import system," https://peps.python.org/pep-0451/, Python Software Foundation, Tech. Rep., 2013, python Enhancement Proposal.

[47] G. Teodoro, T. Kurc, J. Kong, L. Cooper, and J. Saltz, "Accelerating large scale image analyses on parallel, cpu-gpu equipped systems," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. IEEE, 2012, pp. 1093–1104. [Online]. Available: https://ieeexplore.ieee.org/document/6267914

[48] P. Thomson, "Static analysis: An introduction," *ACM Queue*, vol. 19, no. 4, 2021.

[49] Dunes. S. user), "How to implement an import hook that can modify the source code on the fly using importlib?" StackOverflow question #43571737, 2017, online; accessed 2025-07-07.

[50] G. van Rossum, J. Lehtosalo, and Łukasz Langa, "Pep 484 – type hints," 2014, accessed: 2025-09-21. [Online]. Available: https://peps.python.org/pep-0484/

[51] G. van Rossum and P. Moore, "Pep 302: New import hooks," Python Software Foundation, Tech. Rep., 2002, python Enhancement Proposal. [Online]. Available: https://peps.python.org/pep-0302/

[52] N. Vasilakis, B. Karel, N. Roessler, N. Dautenhahn, A. Dehon, and J. Smith, "Breakapp: Automated, flexible application compartmentalization," in *25th Annual Network and Distributed System Security Symposium, NDSS*, 01 2018.

[53] N. Vasilakis, G. Ntousakis, V. Heller, and M. C. Rinard, "Efficient module-level dynamic analysis for dynamic languages with module recontextualization," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 1202–1213. [Online]. Available: https://doi.org/10.1145/3468264.3468574

[54] M. M. Vitousek, A. M. Kent, J. G. Siek, and J. Baker, "Design and evaluation of gradual typing for python," in *Proceedings of the 2014 ACM SIGPLAN Symposium on Partial Evaluation and Program Manipulation*, ser. PEPM '14. New York, NY, USA: ACM, 2014, pp. 45–56. [Online]. Available: https://doi.org/10.1145/2543728.2543736

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

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