ΤΕΧΝΟΛΟΓΙΕΣ ΑΝΑΠΤΥΞΗΣ ΑΣΦΑΛΟΥΣ ΛΟΓΙΣΜΙΚΟΥ

ΔΙΑΤΡΙΒΗ ΓΙΑ ΤΗΝ ΑΠΟΚΤΗΣΗ ΔΙΔΑΚΤΟΡΙΚΟΥ ΔΙΠΛΩΜΑΤΟΣ
ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

2014

Δημήτριος Ι. Μητρόπουλος
Τμήμα Διοικητικής Επιστήμης και Τεχνολογίας
Οικονομικό Πανεπιστήμιο Αθηνών


ATHENS UNIVERSITY OF ECONOMICS & BUSINESS
ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

Η παρούσα έρευνα έχει συγχρηματοδοτηθεί από την Ευρωπαϊκή Ένωση (Ευρωπαϊκό Κοινωνικό Ταμείο - ΕΚΤ) και από εθνικούς πόρους μέσω του Επιχειρησιακού Προγράμματος «Εκπαίδευση και Δια Βίου Μάθηση» του Εθνικού Στρατηγικού Πλαισίου Αναφοράς (ΕΣΠΑ) – Ερευνητικό Χρηματοδοτούμενο Έργο: Ηράκλειτος ΙΙ. Επένδυση στην κοινωνία της γνώσης μέσω του Ευρωπαϊκού Κοινωνικού Ταμείου.

Με τη συγχρηματοδότηση της Ελλάδας και της Ευρωπαϊκής Ένωσης

Περίληψη

Παρόλο που η «ασφάλεια υπολογιστών και δικτύων» είναι σήμερα βασικό μάθημα στα προγράμματα σπουδών πληροφορικής σε όλο τον κόσμο, λίγες φορές συμπεριλαμβάνει στην ύλη του ασφαλείς τεχνικές προγραμματισμού. Οι προγραμματιστές έχουν εκπαιδευτεί έτσι ώστε όταν υλοποιούν εφαρμογές να έχουν στο μυαλό τους την απαιτούμενη λειτουργικότητα χωρίς να λαμβάνουν υπόψη την ασφάλεια. Ένα συνηθισμένο λάθος που κάνουν συνήθως, αφορά τα δεδομένα που εισάφουν οι χρήστες της εφαρμογής, υποθέτοντας, για παράδειγμα, ότι ο χρήστης θα δώσει σε συγκεκριμένα πεδία εισόδου μόνο αριθμητικούς χαρακτήρες, είτε ότι η είσοδος δεν θα υπερβαίνει ένα ορισμένο μήκος. Τέτοια λάθη μπορεί να επιτρέψουν σε έναν κακόβουλο χρήστη να «αναγκάσει» μια εφαρμογή να εκτελέσει κακόβουλο κώδικα. Οι επιθέσεις αυτού του είδους είναι γνωστές και ως «επιθέσεις ένεσης (ή έγχυσης) κώδικα».

Οι επιθέσεις αυτές μπορεί να αποδειχθούν ιδιαίτερα επιζήμιες για μια εφαρμογή μιάς και μπορούν λάβουν χώρα σε διαφορετικά μέρη της, όπως στη βάση δεδομένων, στον εγγενή της κώδικα, στις βιβλιοθήκες της και αλλού. Η ανίχνευση και αντιμετώπιση αυτού του είδους επιθέσεων δεν αποτελεί εύκολη υπόθεση. Ακόμα και αν ο επιτιθέμενος δεν καταφέρει να πετύχει τον στόχο του την πρώτη φορά, τα μηνύματα λάθους που πιθανόν να του επιστραφούν θα λειτουργήσουν σαν ανάδραση και στην επόμενη επίθεσή του το πιθανότερο είναι να πετύχει αυτό που θέλει.

Οι επιθέσεις ένεσης κώδικα, μπορούν να χωριστούν σε δύο βασικές κατηγορίες. Η πρώτη περιλαμβάνει δυαδικό κώδικα και η δεύτερη εκτελέσιμο κώδικα. Στη διατριβή αυτή παρουσιάζουμε μια ενοποιημένη προσέγγιση που ανιχνεύει αυτόματα επιθέσεις που ανήκουν στην δεύτερη κατηγορία. Ειδικότερα, επιθέσεις που χρησιμοποιούν ως μέσο, κώδικα SQL, XPath και JavaScript. Εδώ να σημειώσουμε πως οι επιθέσεις ένεσης SQL και XPath έχουν σαν στόχο την εφαρμογή που βρίσκεται στο διακομιστή ενώ οι επιθέσεις ένεσης JavaScript αποσκοπούν στην παραβίαση της ασφάλειας των φυλλομετρητών που χρησιμοποιούν οι χρήστες για την πλοήγησή τους στο διαδίκτυο. Επιλέξαμε να εστιάσουμε σε τέτοιου τύπου επιθέσεις διότι βρίσκονται για πολλά συνεχόμενα χρόνια στην κορυφή των διάφορων καταλόγων ευπαθειών των διαφόρων φορέων παροχής ασφάλειας.

Για να αναπτύξουμε την προσέγγισή μας, αρχικά μελετήσαμε τις υπάρχουσες προσεγγίσεις και προσδιόρισαμε τα θετικά και αρνητικά χαρακτηριστικά τους. Συγκεκριμένα, αναλύσαμε και αυτές και τους μηχανισμούς που τις υλοποιούν. Έπειτα, τις αξιολογήσαμε βάση συγκεκριμένων κριτηρίων όπως η η χρηστικότητα, η ακρίβεια, η επιβάρυνση σε σχέση με την λειτουργία της προστατευόμενης οντότητας και άλλα. Στη συνέχεια, αναλύσαμε ένα οικοσύστημα λογισμικού αποτελούμενο από αλληλοεξαρτώμενες εκδόσεις εφαρμογών και παρατηρήσαμε πως οι αδυναμίες του λογισμικού που οδηγούν σε επιθέσεις ένεσης κώδικα, εξελίσσονται με την πάροδο του χρόνου. Αναλύσαμε επίσης τη σχέση τους με άλλες κατηγορίες λαθών και τη σχέση τους με το μέγεθος της εφαρμογής. Τέλος, χρησιμοποιήσαμε μια μινιμαλιστική σχεδιαστική φιλοσοφία για να καθορίσουμε τις βασικές οντότητες του προβλήματος που οδηγεί στις επιθέσεις ένεσης κώδικα. Αυτό έγινε με τη χρήση μια γλώσσας προγραμματισμού υψηλού επιπέδου.

Η προσέγγισή μας είναι μια δυναμική μέθοδος ανίχνευσης που προστατεύει διαδικτυακές εφαρμογές με τη χρήση «υπογραφών». Οι υπογραφές αυτές είναι μοναδικά αναγνωριστικά που συνδυάζουν σταθερά στοιχεία ευάλωτου κώδικα. Τέτοια στοιχεία θα μπορούσαν να είναι η δομή του κώδικα, οι λέξεις-κλειδιά που εμφανίζονται μέσα σε αυτόν και τα χαρακτηριστικά που εξαρτώνται από το πλαίσιο εκτέλεσης του (π.χ. τα ίχνη της στοίβας και οι μέθοδοι που κάλεσαν το συγκεκριμένο κομμάτι κώδικα). Η προσέγγιση δουλεύει σε δύο φάσεις: «εκμάθησης» και «παραγωγής». Κατά τη διάρκεια της εκμάθησης του συστήματος οι βιβλιοθήκες εφαρμόζουν μια κρυπτογραφική συνάρτηση κατακερματισμού στα στοιχεία που συνδυάζονται. Το αποτέλεσμα που είναι η «υπογραφή», αποθηκεύεται σε έναν πίνακα. Όταν

η προσέγγιση βρίσκεται στην φάση της παραγωγής, τότε τα βήματα που ακολουθούνται είναι ίδια μέχρι να φτιαχτεί και πάλι μια υπογραφή. Εαν αυτή η υπογραφή υπάρχει στον πίνακα που δημιουργήθηκε κατά την φάση της παραγωγής τότε ο κώδικας εκτελείται κανονικά. Εαν δεν υπάρχει τότε πιθανότατα μια επίθεση ένεσης λαμβάνει χώρα.

Για να επικυρώσουμε την προσέγγισή μας, υλοποιήσαμε τρεις μηχανισμούς που προστατεύουν τις εφαρμογές από επιθέσεις ένεσης κώδικα SQL, XPath και JavaScript αντίστοιχα. Επιπλέον, εξετάσαμε τους μηχανισμούς μας σε σχέση με την ακρίβεια τους. Κατα πόσο είναι αποτελεσματικοί δηλαδή στην ανίχνευση επιθέσεων. Ακόμη εξετάσαμε και την υπολογιστική επιβάρυνση τους σε σχέση με την εφαρμογή που προστατεύουν. Στις δοκιμές μας, δεν προέκυψαν ψευδή θετικά ή αρνητικά αποτελέσματα. Επίσης, σε όλες τις περιπτώσεις οι μηχανισμοί μπορούν να έχουν πρακτική αξία μιας και επιβαρύνουν ελάχιστα την προστατευόμενη εφαρμογή.

Με την ραγδαία ανάπτυξη των εφαρμογών σε κινητά και την υιοθέτηση σύγχρονων τρόπων για την δημιουργία διαδικτυακών εφαρμογών οι επιθέσεις ένεσης κώδικα θα συνεχίσουν να είναι ένα κρίσιμο ζήτημα στον τομέα της ασφάλειας στον κυβερνοχώρο και θα συνεχίσουν να απασχολούν παρόμοια τους ερευνητές και τους επαγγελματίες. Ελπίζουμε ότι με την διατριβή αυτή θα συνεισφέρουμε θετικά στο να καταπολεμηθούν οι επιθέσεις αυτές αποτελεσματικότερα.

Secure Software Development Technologies

Dimitris Mitropoulos

Department of Management Science and Technology,
Athens University of Economics and Business,
76 Patission Street, 10434, Athens, Greece.
Email: dimitro@aueb.gr

Supervised by Professor **Diomidis Spinellis**

To Titiki and Mr Rado.

Contents

List of Figures

List of Tables


ATHENS UNIVERSITY OF ECONOMICS & BUSINESS
ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

Preface

"The aeroplane flies high, turns left, looks right. The aeroplane knows that it is alone in its drama bones. Madness, preconceptions, ray gun logics run and spit and rationalized until a whole chorus of mug wumps, blue in the face from yelling their divisive mantras, run out of young breath and just plain give in to the spirit of the whole damn apple. Face it, you love it, it's fun for one and all, and for all you know the earth spins on it's rusty axis just because of it. The aeroplane moves whether you want it to or not. Cram packed with fuel injected jet missile action, this is war motherfucker and don't you forget it for one second. It is us versus them, and if you're giving in then you are giving up. All the names don't mean shit. Ugly, beautiful, pretentious, arrogant, old, tired, happy, sell outs, careerists, transcendent, hypnotic, trippy, spellbinding, numb, egocentric, solipsistic, empty, hollow, shallow, lost, 70's, 60's, 80's, 20's, long winded, phony, grand, the worst, the best, creepy, cranky, desperate... The aeroplane just flies higher, faster, stronger. There isnt much time for maybes, even goodbyes sometimes. Dust settles, the arcwelders come out and reconstruct the obvious, and we are all left holding the blur. Life will always be a sentimental way, you can vivisect it all you want. Blood and will are indivisible. The aeroplane flies high, turns left, looks right. The world pisses a silver stream to let you know it is there. On the other side of the slipstream of countless thoughtless thoughts. It shatters and divides into a million fragments because life is not a lifestyle choice. We are not a fashion accessory. Music is god's bones creaking pleasure, amusement, even occasional approval. We salut you all with a crack of the back, a baseball bat and a smile. God bless us all, for what we think and feel is all we really have. But when is too far far enough. No limit that I ever knew really matters. There is strength in the dirt of your garden sorrows, there are no more tomorrows, only blissed todays, purple and immeasurable in stature and stealth, because the sun is always sneaking around behind your sneaky back, can you hear us because if you can't we will turn it up till your ears bleed nascent approving harmony. It's all good, and don't you forget it. The fourth wall is down and deserves to stay down, because all you are really watching in others is yourself, the third generation t.v. reflection. Time is never time at all. There is no time, no heartbeats, no babies, no frnech fries, just spider webs strung to oscillate the fever pitch of blandkind, oops I mean mankind. Once the sonic dart leaves your fingers, it is hard to get back. Scratch, sniff, observe, obey, deceive, distort, disarm it all, the bomb is on and ticking. We know but we ain't telling anyone anything, because we know nothing. "T.V. generation X.Y.U.", zero and is playing on a single bill, one night only at the bottom of the ocean. Once it is gone there is no going back, and it is never ever the same. Wave to the magic balloons with your names attached, 5 zillion strong circling the precious earth in search of a friend in search of another. I hope you all find what you need in whatever hole you peer down, whatever cloud you peek behind, let the disaster dukes masticate on the green grass of hope and love. This year is the most joyous and happy, mournful and sad year I have known. Life is good bleats the bleating heart, and it keeps on bleating like an 808. Never ever forever tomorrow comes, new dawns blister, new songs to be sung. The aeroplane flies high, turns left, looks right. The aeroplane knows you know, sings the song of truth, of redemption, of sorrow. **Look no further than your dirty feet**."

Billy Corgan, summer of 1996.


ATHENS UNIVERSITY OF ECONOMICS & BUSINESS
ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

Acknowledgements

Grabbing audience attention: Research made me a happier person. I will not explain this because I cannot. You have to do it for yourself. Research is a really hard task to carry out though. Meet some people, who without their help this dissertation could not have been possible.

True mentoring is more than just answering occasional questions or providing ad hoc help. It is about an ongoing relationship of learning, dialogue, and challenge. Diomidis Spinellis is my mentor. A man that I know and collaborate since 2006. An exceptional teacher and an inspiring programmer. His training involved a lot of studying, hard hitting reviews and infinite support. Apart from supervising my research, Diomidis showed me how to be brief, accurate, whole, patient, regather my thoughts in no time and the best of them all: turn a really bad situation into a good one. I doubt that he will ever acknowledge this but he made me a better person.

Panos Louridas is a friend and a knowledge powerhouse. When I had an idea and no clue on how to implement it I asked for his help. His comments and reviews really helped me through my research and I quote: "*Once it is correct, then it can be sent, otherwise no deadline matters*". A coffee killer (Panos can actually drink a hot cappuccino at once) who can discuss a great range of subjects, from 90's music to types and programming languages.

Now there's this SENSE-lab[1] gang who have helped me through the years in more ways than they could imagine. The "*problem*" with these people is that they all want to become like their supervisor, dissenters, challengers and true software engineers. This provides you with a great advantage of course because their comments are always valuable and sharpen your skills.

Enter, Vassilios Karakoidas. *The great objector*. Everything you know is wrong. But if you are really prepared and confident about your idea, you will sense his acceptance as a true relief. A true friend who has always been there for me. Interestingly, when I sensed that he was not, after some time I realized that he has done it on purpose ("*trust me, I know what I'm doing*").

Kostantinos Stroggylos (a.k.a. *the trickster*) was the man that introduced me to Diomidis. A great friend and a gifted coder. We have spend a lot of hours together writing code, discussing about ideas, relationships and listen to tons of alternative records. Recently, we let another member in our music / coding hackathons, his son, Stratis!

Giorgos Gousios has this intensity when he asks you all the time:"*what is the contribution of this paper you're working on*" that makes you go home and work harder. *gousix* has a really unique sense of how systems work and he can provide you with great advice on your research. Pity that his games skills (digital or physical) do not match his research skills. (Sorry mate but you cannot mess with a left-handed Wii tennis player. I really love you though. P.S. Kisses to Elli!)

It's no secret that SENSE has a *secret member*. Giorgos Zouganelis provided me with excellent technical advice and support when I needed it the most. A loyal and wise friend that I am grateful to have.

Maria Kechagia is a wonderful person to work with. I want to thank her for sharing her ideas with me and introducing me to a whole new research area.

[1] Software Engineering and Security.

SENSE also includes some other brilliant researchers who I'd like to thank for their support: the upcoming guitar hero Stephanos Androutsellis-Theotokis, the suave and sophisticated academic celebrity Vassileios Vlachos, and the talented soccer player Marios Fragkoulis.

During my PhD years I have also worked with some other great people like Sotiris Ioannidis, professor Angelos Keromytis, professor Panagiotis Katsaros, Zaharias Tzermias, Stelios Pantelopoulos, Venelin Arnaudov, Jan Paul Posma, Vaggelis Giannikas, Periklis Gkolias and perhaps others that I may forget (my apologies!). Professor Giannis Ioannidis and professor Alex Delis might not know me, but I want to thank them for their guidance and teaching during my bachelor years. I would also like to thank the organisation that supported my research.[2]

The following persons have contributed to this dissertation without actually knowing it. Who are these people and what are they capable of:

My family. My mom got the best out of me. She would probably be a far more better researcher than I will ever be but she chose to raise me instead. Probably the best mathematician I will ever get to know and understand what actually says. Believe me I have been taught mathematics by too many teachers and I've never understood a word coming from their mouth. A mind sharpener that introduced me to hackathons by solving hundreds of calculus exercises within a day before my panhellenic examinations. My Dad was always been there for me in his own quiet way. He taught me how to be a good listener, a cunning linguist like he is, and passed me on his vinyl disease. Being a brother is better than being a superhero. My brother is my hero. One of the most powerful persons I know. Straight, clever, phlegmatic and politically correct. A book eater who can tear you apart about your political views and then go write C code for dinner. If I am considered as a decent researcher, my brother will be a far better one.

The family clan *thank-yous* also go to my uncle Giorgos who has been a second father to me. My grandmother Marika who is the most lovable person I know. My grandfather, maestro Lampis who showed me the true power of music through his masterful guitar playing. My late grandmother Sofie, who was a really bright mind and the best story teller I'll ever get to listen to. My windless grandfather Dimitris, my two cousins Giorgos and Dimitris, my aunt Lizy, my late grandmother Giorgia, Maria, my pappou George V., Ourania, Lia, Maria, uncle Thanassis and aunt Pia, Thanassis, Voula, Vlassis, Apostolis, Mr. Roussopoulos, my cousin Giorgia, Dimosthenis and Dimitra (my twins!), uncle Christos, Vicky, and of course my late aunt Roula who taught me what integrity means early on in my life. I carry her with me everywhere all the time (P.S. I guess you're having a ball up there with Sofie!).

The wild bunch. "*A true friend stabs you in the front*" said Oscar Wilde. That's what we have been doing with these jackasses for the past 25 years. Criticising ruthlessly one another is our favorite hobby and as I see it now this made our bonds stronger through the years. Here are these great men that I have the honor to call friends: Mr. Stefanos Tsibidis is one of the most handsome men you'll ever meet but for me he will always be this 9 year old kid who was crying because I got mad and shouted at during a football game. My *unofficial brother*. Technically, the man who can go to hell, beat the devil down and bring me back alive. Mr. Ilias Louziotis is what you have in mind when you think about the "*best friend*" term. You can call him from your house to bring you coffee from Starbucks while he is in Singapore playing the stock market. He'll do it. I cannot make any decision without discussing with this man first. Mr. Georgios Zaharias Tsibidis is my *laughter coach* ("*how can you attend a conference? Can you? You can't! Oh but you do! But how?*"). Probably he will never find out that he is mentioned here. In fact, he does not give A shit.

The weather underground. My music family, namely: *Kaloriferis*, *Beer Bilation* and *Tost*. The men I play music with for the past eleven years. People that know me better than anyone just because we have shared our feelings while writing and playing music.

[2]This research has been co-financed by the European Union (European Social Fund - ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework - Research Funding Program: Heracleitus II. Investing in knowledge society through the ESF.

I want to also thank the *jazzman* and dear friend Tassos, my "*cousin-uncle*" and one of my best friends *Lee* Deli, *Toni Fox*, my "*sis*" Eleni, Mrs. Ersi (who brought balance in my life for quite a while), Elena "*stranger*", the wonderful Artemis, my second mom Rania, the PAPAPA family (take good care of my godchild people), *Kokoro* (SHOOT-EM-UP-SOU-EIPA), Mr. Meimaris, Mr. Fightakis, Mrs. Dafni, my crazy ISDI 2013 students (sorry Maria, Faih and Pinelopi but I cannot use the term here), Fenia (and Elli of course), Diasakos Bros., TV, my basketball mates (Dionysis, Themis and Akis), Kozi, my crazy DI classmates (a.k.a. the FISKA boys – you know who you are), my master classmates (Akis, Pkaps, Paschalis, Sotiris, Christos), Pepi, Athens, Iro, Foivos Iason and Christos Iordanidis for being a friend and a teacher of how to become a teacher.

Last but not least. My constant inspiration: Konstantina, who is my mirror, my love and my pal at the same time. Clever as hell, sensitive, loving and always looking me with her clean eyes. She stands by me when I am under pressure, understands me like few people do and this is why I deeply love her (I didn't say thank ha!).

dimitro, winter of 2014.


ATHENS UNIVERSITY OF ECONOMICS & BUSINESS
ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

Summary

Although computer security is nowadays standard fare in academic curricula around the globe, few courses emphasize secure programming techniques. Programmers have been trained in terms of writing code that implements the required functionality without considering its many security aspects. One common trap into which programmers fall concerns user input, assuming, for example, that only numeric characters will be entered by the user, or that the input will never exceed a certain length. Mistakes like these can lead to the processing of invalid data that a malicious user can introduce into a program and cause it to execute malicious code. This kind of exploit is known as a *code injection attacks* (CIAs).

Code injection attacks are one of the most damaging class of attacks because they can occur in different layers, like databases, native code, applications, libraries, the browser and others. In addition, they span a wide range of security and privacy issues, like viewing sensitive information, destruction or modification of sensitive data, or even stopping the execution of the entire application.

CIAs can be divided into two basic categories. The first involves binary code and the second executable source code. In this dissertation we introduce a training approach that uses unique identifiers created by blending features coming from legitimate code statements with elements extracted from the application's execution environment, to dynamically detect executable source code injection attacks. In particular, attacks that involve SQL, XPath and JavaScript as attack vectors. Note that the the first two vectors involve attacks that target applications on the server side while JavaScript injection attacks target web users. We chose to focus on this CIA-category because it includes attacks that top the vulnerability lists of numerous bulletin providers for several years.

To develop our approach we studied the existing approaches that counter CIAs and identified their positive and negative aspects. Apart from the basic approaches used to counter CIAs we also analyzed the mechanisms that implement them. In addition, we evaluated them based on the following requirements: flexibility, usability, accuracy, implementation independence and overhead. Then, we analysed a software ecosystem of interdependent project versions to see how software bugs that can leat to CIAs evolve over time, their persistence, their relation with other bug categories, and their relationship with the project's size. Finally, we used a minimalist design philosophy to specify the basic entities of the code injection problem. This was done by using a high-level programming language.

Our approach, a dynamic detection method, identifies and registers vulnerable code statements by using unique signatures that are generated during a training phase. Signatures are based on features that when combined provide a unique identifier. Some of these features depend entirely on the code statement that is about to be executed while others are independent of the code statements, but depend on the execution flow and environment. Before generating and storing a signature our method analyzes the code. Code analysis involves the complete removal of what is expected as user input in which case it would be useless as a predictor. Then, at runtime, it checks all statements for compliance with the trained model and blocks code statements containing additional maliciously injected elements.

To validate our approach, we have implemented three mechanisms that protect applications from SQL, XPath and JavaScript injection attacks respectively. Our mechanisms are implementation independent and easy to use.

Furthermore, we have tested our mechanisms in terms of accuracy and computational overhead. In our tests we found out no false positives or false negatives. Also, our results in all cases indicate that the described solutions can be of practical value as the overhead it imposes is minimal both in time and in space.

With the increasing use of mobile applications and the adoption of modern ways that web applications are set up, CIAs wil continue to be a critical issue in the field of cyber security and draw the attention of researchers and practitioners alike. We hope that our contributions will help the research community to deal with CIAs more efficiently.

Chapter 1

Introduction

Software has numerous interdependencies on operating systems, enterprise applications and network infrastructures. This is why implementing high quality software in a fashion that meets security standards is becoming a basic point for designing applications [82]. The strong association between code quality and security has been realized quite recently. A formal definition of software quality was first presented in the ISO—9126:1/1998 standard, according to which software quality is the conformance to explicitly stated Non-Functional Requirements (NFRs).[1] According to the ISO-9126:1/1998 standard, software quality attributes are classified in a structured set of six further subdivided characteristics. These characteristics are functionality, reliability, usability, efficiency, maintainability and portability. In this definition, security is a feature of functionality. Still, there are more aspects of quality that are affected by the threats of security like maintainability, reliability and correctness [139, 216, 227]. In the new ISO/IEC—25010:2011[2] that replaced ISO—9126:1/1998, security has been added as a characteristic (as described in Figure 1.1), rather than a sub-characteristic of functionality, with the following sub-characteristics: confidentiality, integrity, non-repudiation, accountability and authenticity.

The development of a qualitative software system is a complex, costly and time-consuming activity. Fortunately, a number of methodologies have been proposed to manage it effectively such as top-down or bottom-up, waterfall model or spiral model [201]. Unfortunately, these software life cycle management models fail to take into consideration the security aspect of software development — even though there are some new ones that try to incorporate it [39, 79, 92].

Software security is a relatively new area of research. Publications, and books on the field appeared in the early 2000's [108, 145]. Before this, information security was mainly associated with

[1] http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=35733
[2] https://www.iso.org/obp/ui/#iso:std:iso-iec:25010:ed-1:v1:en

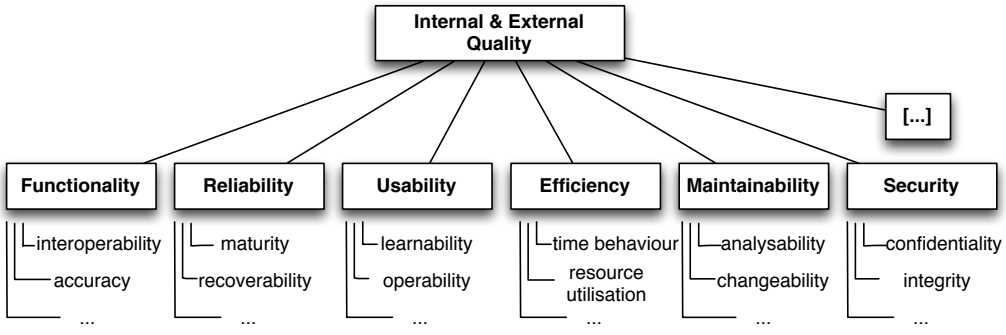


Figure 1.1: The quality characteristics classification as described in ISO/IEC—25010:2011.

network security, operating systems security and viral software. Software defects can severely affect an organization's infrastructure [196], and they can cause significant financial and reputational damage to an organization [21, 213]. Specifically, whereas a software bug can cause a software artifact to fail, a security bug can allow a malicious user to alter the execution of the entire application for his or her own gain. In this case, such bugs could give rise to a wide range of security and privacy issues, like the access of sensitive information, the destruction or modification of data, and denial of service. Moreover, security bug disclosures lead to a negative and significant change in market value for a software vendor [212].

Most software vulnerabilities derive from a relatively small number of common programming errors that lead to security holes [219, 235]. This is mainly because most programmers have been trained in terms of writing code that implements the required functionality without considering its many security aspects [108, 206]. Although computer security is nowadays standard fare in academic curricula around the globe, few courses emphasize secure programming techniques [214]. For instance, during a standard introductory C course, students may not learn that using the `gets` function could make code vulnerable to an exploit [134, 193]. The situation is similar in web programming. Programmers are not aware of security loopholes inherent to the code they write; in fact, knowing that they program using higher level languages than those prone to security exploits, they may assume that these render their application immune from exploits stemming from coding errors.

One common trap into which programmers fall concerns user input, assuming, for example, that only numeric characters will be entered by the user, or that the input will never exceed a certain length. Programmers may think, correctly, that a high-level (usually scripting) language in a web application will protect them against buffer overruns [124]. Programmers may also think, incorrectly, that input is not a security issue any more. That is wrong. Their assumptions can lead to the processing of invalid data that a malicious user can introduce into a program and cause it to execute malicious code. This kind of exploit is known as a code injection attack (CIA). In this dissertation we present an approach that counters a specific class of CIAs in a novel way.

1.1 Code Injection Attacks

Code injection is a technique to introduce code into a computer program or system by taking advantage of unchecked assumptions the system makes about its inputs. Bratus et al. [36] portray the issue in a generic fashion: *"unexpected (and unexpectedly powerful) computational models inside targeted systems, which turn a part of the target into a so-called "weird machine" programmable by the attacker via crafted inputs (a.k.a. "exploits")"*. An example of the above definition is the following: The code fragment below, defines the operation of addition in the Scheme programming language which is a high-level language and in particular, it is one of the two main dialects of Lisp [3, 69]:

```
(define (add x y) (+ x y))
```

Consider the case where instead of a number, a function that leads to an endless loop is passed as an argument by the user. This will cause the interpreter to enter an endless loop and lead to a denial of service. The intuition here is that *every application that copies untrusted input verbatim into an output program is vulnerable to CIAs*. Ray and Ligatti [183] actually proved the above claim based on *formal language theory*.

Code injection attacks are one of the most damaging class of attacks [84, 208] due to the following reasons:

- They can occur in different layers, like databases, native code, applications, libraries, the browser and others.

- They span a wide range of security and privacy issues, like viewing sensitive information, destruction or modification of sensitive data, or even stopping the execution of the entire application.

Despite many countermeasures that have been proposed the number of CIAs has been increasing[3] (see Section 2). Malicious users seem to find new ways to introduce compromised embedded executable code to applications by using a variety of languages and techniques as we will see in the forthcoming subsections. Figure 1.2 presents a taxonomy of CIAs divided into two basic categories. The first involves binary code and the second executable source code. We illustrate the attack categories and subcategories that have been extensively analyzed in other research papers, in grey colour. Note that a specific category, (binary code injection attacks in particular) has been extensively analyzed in other references, thus we do not provide a full description of attacks of this kind.

[3] http://www.sans.org/top-cyber-security-risks/, http://cwe.mitre.org/top25/, http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

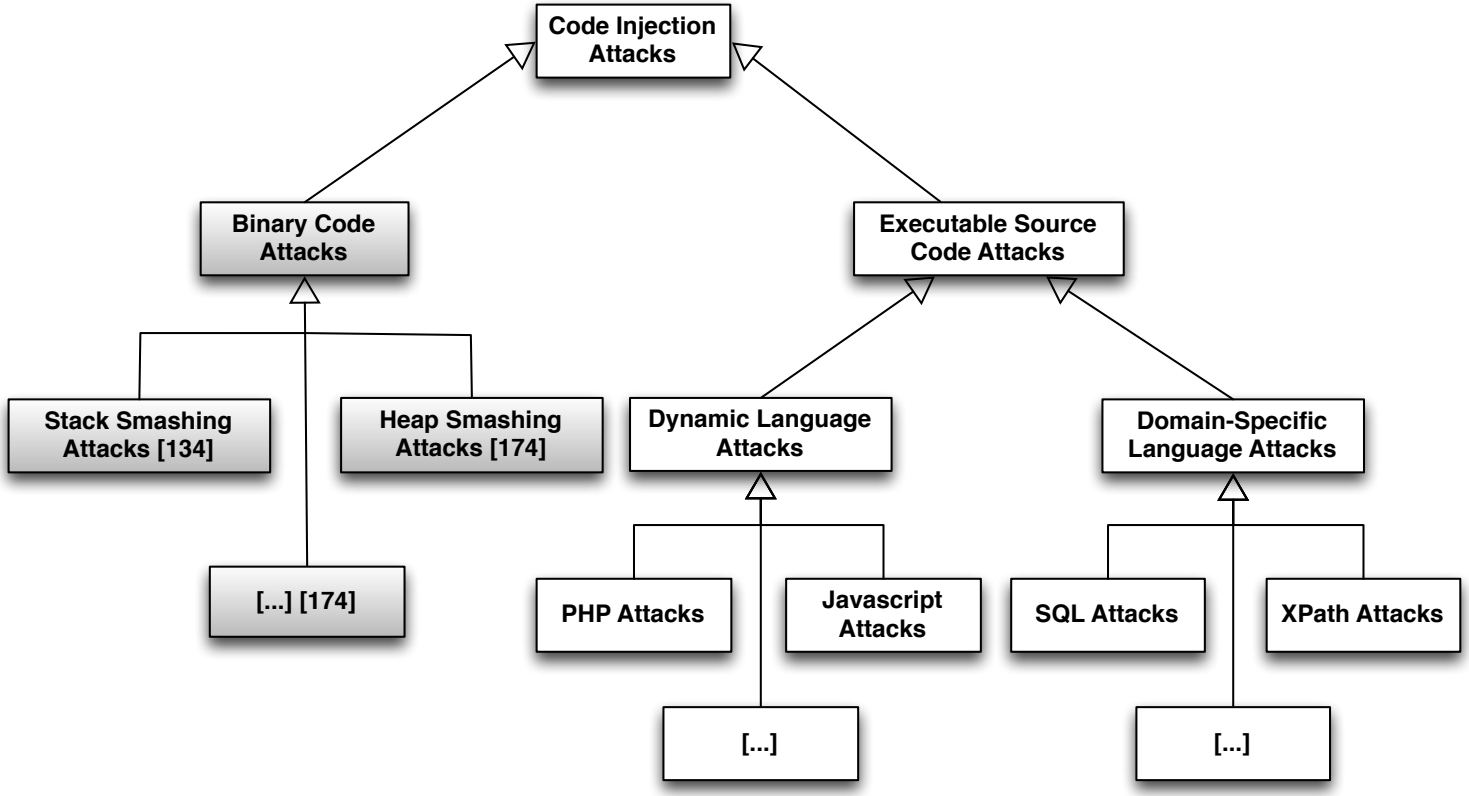


Figure 1.2: A taxonomy of code injection attacks. The subcategories that have been extensively analyzed in other research papers can be seen in grey colour. For each subcategory we provide the corresponding reference.

1.1.1 Binary Code Injection

Binary code injection involves the insertion of binary code into a target application to alter its execution flow and execute inserted compiled code. This category includes buffer-overflow attacks [60, 124], a staple of security problems. These attacks are possible when the bounds of memory areas are not checked, and access beyond these bounds is possible by the program. By taking advantage of this, attackers can inject additional data overwriting the existing data of adjacent memory. From there they can take control over a program, crash it or, even take control of the entire host machine.

Specifically, a common method for exploiting a stack-based buffer overflow is to overwrite the function return address with a pointer to attacker-controlled data. Consider the code below:

```
#include <string.h>

void foo(char *bar)
{
   char  c[12];
   strcpy(c, temp);
}

int main(int argc, char **argv)
{
   foo(argv[1]);
}
```

The code above trivially takes an argument from the command line and copies it to a local stack variable `c`. This may work fine for command line arguments smaller than 12 characters, but with any arguments larger than 11 characters long will result in corruption of the stack.

C and C++ are vulnerable to this kind of attacks since typical implementations lack a protection scheme against overwriting data in any part of the memory. Specifically, they do not check if the data written to an array is within its boundaries. In comparison, Java guards against such attacks by preventing access beyond array bounds, throwing a runtime exception. An extensive survey on binary code injection attacks can be found in reference [134].

There are two research papers that present various techniques that belong to this CIA category. In particular, an extensive survey on binary code injection attacks can be found in reference [134]. Furthermore, specific advances in exploiting such vulnerabilities (i.e. *heap smashing*, *arc injection* and others) have been presented in reference [174].

Finally, the countermeasures used to detect such defects have been already surveyed [239] (many of them are also included in a book: [61] – Section 13.8). Nevertheless, we include some of them in our research (see Subsection 2.1.2) because they prompted the development of some sophisticated countermeasures.

1.1.2 Executable Source Code Injection

Code injection also includes the use of source code, either of a DSL or a *Dynamic Language*. Note that binary code injection attacks can only occur when the target system is implemented in languages lacking array bounds checking, like C and C++. Contrariwise, source code-driven injection attacks can target applications written in various programming languages with different features and characteristics.

DSL-Driven Injection Attacks Code injection attacks that involve domain specific languages, constitute an important subset of code injection, as DSLs like SQL and eXtensible Markcup Language

(XML) play an important role in the development of web applications. For instance, many applications have interfaces where a user enters input to interact with the application's data, thereby interacting with the underlying Database Management System (DBMS). This input can become part of an SQL statement and executed on the target DBMS.

A code injection attack that exploits the vulnerabilities of these interfaces is called an "SQL injection attack" [45, 99]. There are many forms of SQL injection attacks. The most common involve taking advantage of:

- Incorrectly passed parameters,
- incorrectly filtered quotation characters, or
- incorrect type handling.

Consider a trivial example that takes advantage of incorrectly filtered quotation characters. In a login page, besides the user name and password input fields, there is usually a separate field where users can input their e-mail address, in case they forget their password. The statement that is executed can have the following form:

```
SELECT * FROM passwords WHERE email = 'theemailIgave@example.com';
```

If an attacker, inputs the string *anything' OR 'x'='x*, she could conceivably view every item in the table. In a similar way, the attacker could modify the database's contents or schema.

An "incorrect type handling" attack occurs when a user-supplied field is not strongly typed or is not checked for type constraints. For example, many web sites allow users, to access their older press releases. A Universal Resource Locator (URL) for accessing the site's fifth press release could look like this:

```
http://www.website.com/pressRelease.jsp?RelID=5
```

And the statement that is probably executed is:

```
SELECT description, issuedate, body FROM pressRel WHERE RelID = 5
```

If some attackers wished to find out if the application is vulnerable to SQL injection, they could change the URL into something like:

```
http://www.website.com/pressRelease.jsp?RelID=5%20AND%201=1
```

If the page displayed is the same page as before, it is clear that the field `RelID` is not strongly typed and end users can manipulate the statement as they choose. Note that, while the first attack could be countered by filtering out the quotation characters from the input data, countering the second attack would require code to ensure that the input data is a single integer. Savvy programmers could use a language's libraries, like PHP's `mysql_real_escape_string()` to detect malformed input; or they could use prepared SQL statements, instead of statement templates (see Sections 2.1 and 3.2). Unfortunately, the number of SQL injection attacks suggest that programmers are not always that careful.

The usage of XML documents instead of relational databases makes web applications vulnerable to XPath injection attacks [44]. This is because of the loose typing nature of the XPath language [26, 27, 125]. XPath injection attacks can be considerably dangerous because XPath 1.0[4] not only allows one to query all items of the database, but also offers no access control to protect it. Using XPath querying, a malicious user may extract a complete XML document, expose sensitive information, and compromise the integrity of the entire database. Consider the following XML file used by an e-commerce website to store customers' order history:

[4] http://www.w3.org/TR/xpath

```
<?xml version="1.0" encoding="UTF-8"?>
<orders>
    <customer id="1">
        <name>Dimitris Mitropoulos</name>
        <email>dimitro@aueb.gr</email>
        <creditcard>12345678</creditcard>
        <order>
        <item>
            <quantity>1</quantity>
            <price>1.000</price>
            <name>television</name>
            </item>
        <item>
            <quantity>2</quantity>
            <price>9.00</price>
            <name>CD-R</name>
        </item>
        </order>
    </customer>
...
</orders>
```

The web application allows its users to search for items in their order history based on the price. The query that the application performs could look like this:

```
string query =
"/orders/customer[@id='" + cId + "']" +
"/order/item[price >= '" + pFilter + "']";
```

Without any proper input validation a malicious user will be able to select the entire XML document by entering the following value:

```
'] | /* | /anything[bar='
```

With a simple request, the attacker can steal personal data for every customer that has ever used the application. XPath statements do not throw errors when the search elements are missing from the XML file in the same way that SQL queries do when the search table or columns are missing from the database. Because of this, it is easier for an attacker to use malicious code in order to perform an XPpath injection attack than an SQL injection attack.

Dynamic Language-Driven Injection Attacks Dynamic languages like Python, Perl, JavaScript, and PHP have the capability of interpreting and executing code at runtime through a method called `eval` [71, 194]. A simple example of a dynamic language-driven attack is an input string that is fed into an `eval()` function call, e.g., in PHP:

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

The user may pass into the `value` parameter code that will execute in the server. If `value` is `10 ; system(''touch foo'');` then a file will be created on the server; it is easy to imagine more

detrimental instances. Thus, special care must be taken when using `eval` with data coming from an untrusted source. As we will see in the following paragraphs, `eval` is extensively used in JS injection attacks.

JavaScript injection attacks comprise a wide subset of dynamic language-driven attacks. Such an attack occurs when an attacker manages to inject a script in the JS engine of a browser and alter its execution flow [77]. JS injection attacks are considered as a critical issue in web application security mainly because they are associated with major vulnerabilities such as: Cross-Site Scripting (XSS) attacks [200] and Cross-Channel Scripting (XCS) attacks [32, 226]. A JavaScript injection vulnerability is manifested when a web application accepts and redisplays data of uncertain origin without appropriate validation and filtering. Such content can compromise the security of these applications and the privacy of their corresponding users [66]. Many web sites allow registered users to post data which are stored on the server-side (i.e. a third-party comment on a blog page). If attackers hide a script in such data, they could manipulate the browser of another user. For example consider the following code snippet:

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

If a malicious user could post data containing the above script, web users visiting the page that contains this data could have their cookies stolen. Through this script the attacker calls an external Common Gateway Interface (CGI) script and passes all the cookies associated with the current document to it as an argument via the `document.cookie` property.

A common but rough way to stop malicious behaviors like this is server-side code filtering (i.e. the server strips out the word "javascript" from any external source) [112]. Still, there are many ways to bypass such defense mechanisms. For example, one could escape special characters to bypass simple filtering operations, or take advantage of issues in the implementation of Cascading Style Sheets (CSS) rendering engines of browsers like *Microsoft Internet Explorer* (versions prior to 7).

Consider the case where an attacker manages to hide the following code fragment in the CSS of a web page:

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

The attacker utilizes the `eval` function and a newline character ("java—*newline*—script") to bypass the security checks and manoeuvre the user's browser to execute the code contained in the `expr` variable. This is done by using the `document.all` array that contains all of the elements within a document.

Attacks like the above take advantage of the fact that `eval` executes the code passed to it in the same execution environment as the function's caller. Malicious users can also use `eval` to assemble innocuous-looking parts into harmful strings that the protecting mechanisms of a web page would normally consider dangerous and remove [186].

Another back-door that could allow a JS injection attack, is the legitimate use of the `eval` function in order to dynamically generate and execute JavaScript code on a web page. For example, a convenient way for a JavaScript program to parse JSON formatted data is by using the `eval` function [148, 211]. If this data is not validated by the server then an attacker can inject malicious code into the evaluated script and abuse the legitimate user's browser. Previous research has shown that in a total of 6,805 sites, there is a high percentage that uses the dangerous `eval` function (over 44.4%) [241].

In addition, a JS injection attack does not necessarily have to involve HyperText Markup Language (HTML) elements. A malicious user can embed a script within a PostScript file, upload it as a valid document and then use it to trigger the attack [23].

XCS attacks are an XSS variation. In an XCS attack, a malicious user utilizes a non-web channel such as Simple Network Management Protocol (SNMP) to inject code. For example, there are several Network-Attached Storage (NAS) devices and allow users to upload files via the SMB protocol (Server Message Block) [32]. A malicious user could upload a file with a filename that contains a well-crafted script. When a legitimate user connects over a web channel to the device to browse its contents, the device will send through an Hypertext Transfer Protocol (HTTP) response the list of all filenames, including the malicious one which is going to be interpreted as a valid script by the browser.

Finally, in dynamic pharming, an attacker provides a web document with a malicious JavaScript to the client. Then she exploits Domain Name System (DNS) rebinding defects existing in the client's browser in order to force it to connect to the legitimate server, but in a separate window or frame. If the client authenticates herself to the server, the attacker utilizes the injected script to hijack the session [121].

1.2 Rationale

A software ecosystem can be seen as a collection of components, which are developed and co-evolve in the same environment [138]. The rationale for our research was based on the examination of the security bugs of a large software ecosystem. In this section, we present observations concerning the security bugs of the *Maven central repository*[5] [179] (approximately 265 Gigabytes (GB) of data). Maven is a build automation tool used primarily for Java projects and it is hosted by the Apache Software Foundation. It uses XML to describe the software project being built, its dependencies on other external modules, the build order, and required plug-ins. To build a software component, it dynamically downloads Java libraries and Maven plug-ins from the Maven central repository, and stores them in a local cache. The repository can be updated with new projects and also with new versions of existing projects.

The Maven ecosystem has been previously analyzed by Raemaekers et al. [179] to produce the *Maven dependency dataset*. Apart from basic information like individual methods, classes, packages and lines of code for every JAR, this dataset also includes a database with all the connections between the aforementioned elements.

To statically analyze the software components of the repository, we used *FindBugs*,[6] a static analysis tool that examines bytecode to detect software bugs and has already been used in research [19, 107, 198, 204], to scan all the project versions of all the projects that exist in the Maven repository. For instance, FindBugs was used to analyze all available builds of Java Development Kit (JDK) [20] while Google has also incorporated it into its software development process [19]. It has also been extended to verify Application Programming Interface (API) calls [205] and discover bugs in AspectJ applications [198].

Contrary to the approaches that examine versions formed after every change that has been committed to a repository [70, 142, 169, 242], our observations are made from a different perspective. The versions examined in this work were actual releases of the projects. As a result we do not have an indication of how many changes have been made between the releases. In essence, these projects were the ones that were or still are, out there in the wild, being used either as applications, or dependencies of others.

Experiment The goal of our experiment was to retrieve all the bugs that FindBugs reports, from all the project versions existing on the Maven repository (in the Maven repository, versions are actual

[5] http://mvnrepository.com/
[6] http://findbugs.sourceforge.net/

Python Scripts	14
Lines of Code	1256
Code Size (KB)	92

Table 1.1: Size metrics concerning the tools created for the dataset construction process.

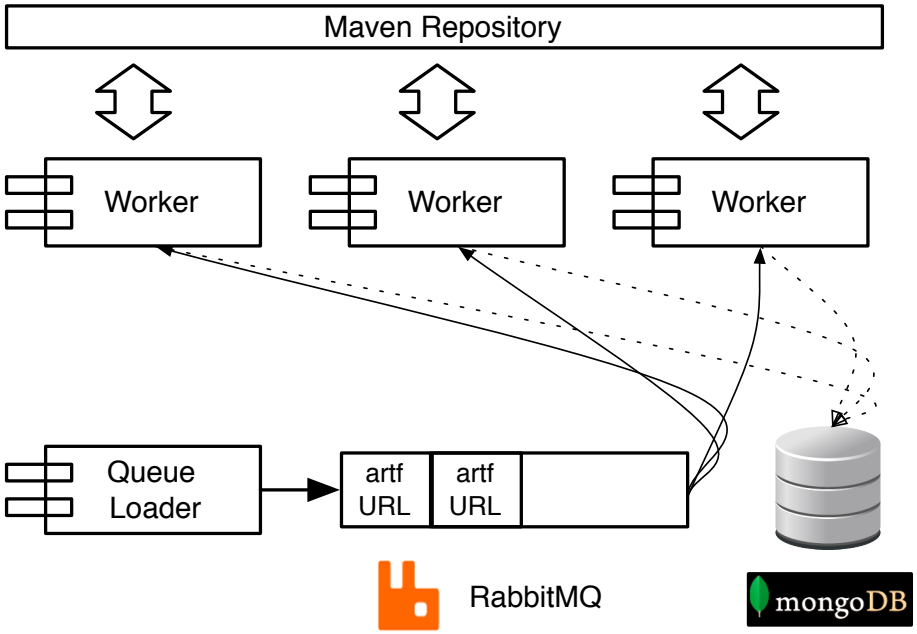


Figure 1.3: The data processing architecture.

releases). Table 1.1 presents some metrics concerning the tools created for the process. The experiment involved four entities: a number of workers (a custom Python script), a task queue mechanism (RabbitMQ—version 3.0.1),[7] a data repository (MongoDB—version 2.2),[8] and the code repository, which in our case it was the public Maven repository.

In particular, we obtained a snapshot (January 2012) of the Maven repository and handled it locally to retrieve a list of all the names of the project versions that existed in it. A project version can be uniquely identified by the triplet: *group id*, *artifact id* and *version*. Figure 1.4 illustrates the hierarchy of the Maven repository. First, we scanned the Maven repository for appropriate JAR files and created a list that included them. We discuss the JAR selection process in the next section. With the JAR list at hand, we created a series of processing tasks and added them to the task queue. Then we executed twenty five (UNIX-based) workers that checked out tasks from the queue, processed the data and stored the results to the data repository. A typical processing cycle of a worker included the following steps: after the worker spawned, it requested a task from the queue. This task contained the JAR name, which was typically a project version that was downloaded locally. First, specific JAR metadata were calculated and stored. Such metadata included the JAR's size (in terms of bytecode), its dependencies (derived from the *pom.xml* file), and a number that represented the ordinal version number of the release (see also Table 1.2). This number was derived from an XML file that accompanies every project in the Maven repository called *maven-metadata.xml*. Then, FindBugs was invoked by the worker and its

[7] http://www.rabbitmq.com/
[8] http://www.mongodb.org/

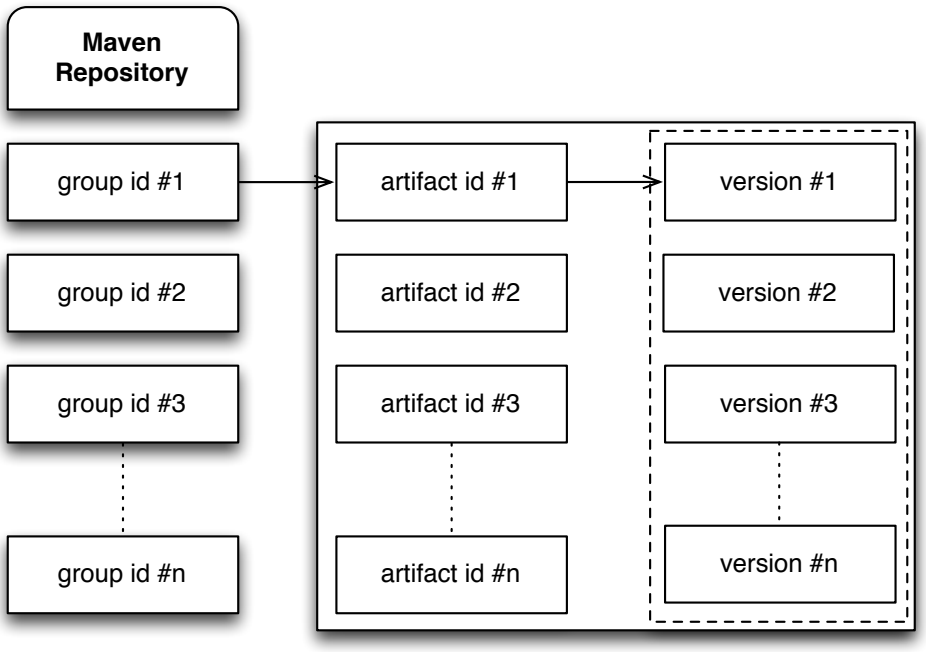


Figure 1.4: The Maven repository hierarchy.

results were also stored in the data repository. FindBugs, outputs its results in XML format. Hence, all the data were converted to the JSON format by mapping all XML elements to JSON objects. The following listing shows the format of the information we collected (note that the results of FindBugs are too large to show):

```
{"JarMetadata": {
    "version": "1.0.0",
    "version_order": "1",
    "jar_size": "34768",
    "dependencies": [
        {
            "version" : "2.0",
            "groupId" : "org.apache.maven",
            "artifactId" : "maven-project"
        },
        { /* other dependencies */ }
    ],
    "group_id": "org.apache.myfaces.buildtools",
    "jar_filename": "myfaces-jdev-plugin-1.0.0.jar",
    "artifact_id": "myfaces-jdev-plugin",
    },
 "BugCollection": { /* FindBugs data */ }
}
```

When the task was completed the queue was notified and the next task was requested. This process was

`version`	The version of the project in the repository.
`version_order`	The ordinal version number of the release.
`jar_size`	The size of the JAR file.
`dependencies`	List of all dependencies for the project.
`group_id`	The group ID of the project in Maven repository.
`jar_filename`	JAR's filename.
`artifact_id`	The artifact ID of the project in Maven repository.

Table 1.2: JAR metadata description.

executed for all the available JARs in the task queue. A schematic representation of the data processing architecture can be seen in Figure 1.3.

Measurement	Value
Projects	17,505
Versions (total)	115,214
Min (versions per project)	1
Max (versions per project)	338
Mean (versions per project)	6.58
Median (versions per project)	3
Range (over versions)	337
1^{st} Quartile (over versions)	1
3^{rd} Quartile (over versions)	8

Table 1.3: Descriptive statistics measurements for the Maven repository.

FindBugs works by examining the compiled Java virtual machine bytecodes of the programs it checks, using the Bytecode Engineering Library (BCEL). To detect a bug, FindBugs uses formal methods like *data-flow analysis* (see Subsection 2.1.3). It has also other detectors that employ visitor patterns [167] over classes and methods by using state machines to reason about values stored in variables or on the stack. Since FindBugs analyses applications written in the Java programming language, and the Maven repository hosts projects from languages other than Java such as Scala, Groovy, Clojure, etc., we filtered out such projects by performing a series of checks in the repository data and metadata.

In addition, we implemented a series of audits in the worker scripts that checked if the JARs are valid in terms of implementation. For instance, for every JAR the worker checked if there were any *.class* files before invoking FindBugs. After the project filtering, we narrowed down our data set to 17,505 projects with 115,214 versions. Table 1.3 summarises the data set information and provides the basic descriptive statistic measurements. The distribution of version count among the selected projects is presented in Figure 1.5.

The statistical measurements presented in Table 1.3 indicate that we have 17,505 projects and the data set's median is 3, which means that almost 50% (8,753 projects) of the project population have 1 to 3 versions. In general, most projects have a few number of versions, there are some projects with ten versions and only a few with hundreds of versions. The maximum number of versions for a project is 338. The 3^{rd} quartile measurement also indicated that 75% (13,129) of the projects have a maximum of 8 versions.

Threats to Validity A threat to the internal validity of our experiment could be the false alarms of the FindBugs tool [19, 106]. False positives and negatives of static analysis tools and how they can be reduced is an issue that has already been discussed in the literature [204]. In addition, reported security bugs may not be applicable to an application's typical use context. For instance, FindBugs could report an SQL injection vulnerability in an application that receives no external input. In this

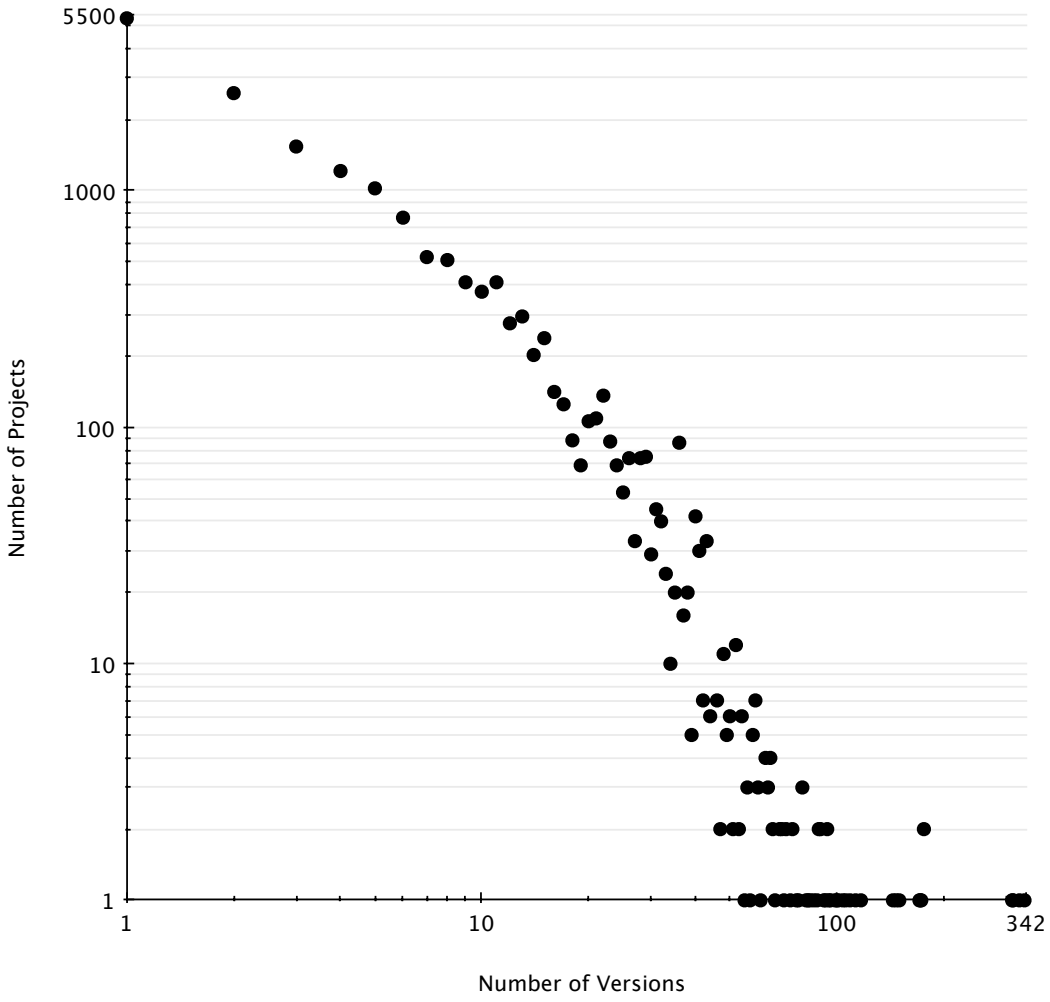


Figure 1.5: Distribution of version count among project population.

Category	Description
Bad Practice	Violations of recommended and essential coding practice.
Correctness	Involves coding misting a way that is particularly different from the other bug sakes resulting in code that was probably not what the developer intended.
Experimental	Includes unsatisfied obligations. For instance, forgetting to close a file.
Internationalization (i18n)	Indicates the use of non-localized methods.
Multi-Threaded (MT) Correctness	Thread synchronization issues.
Performance	Involves inefficient memory usage allocation, usage of non-static classes.
Style	Code that is confusing, or written in a way that leads to errors.
Malicious Code	Involves variables or fields exposed to classes that should not be using them.
Security	Involves input validation issues, unauthorized database connections and others.

Table 1.4: Bug categorisation according to FindBugs.

particular context, this would be a false positive alarm. Furthermore, given that our analysis is done on open-source projects written in the Java programming language and hosted on Maven, a threat to the external validity of our work is the fact that our results may not be applicable to other programming languages, ecosystems, and development cultures.

Overview and Results FindBugs separates software bugs into nine categories (see Table 1.4). Two of them involve security issues: *Security* and *Malicious Code*. From the total number of releases, 4,353 of them contained at least one bug coming from the first category and 45,559 coming from the second. Our first results involve the most popular bugs in the Maven repository. Figure 1.6 shows how software bugs are distributed among the repository. Together with the *Bad Practice* bugs and the *Style* bugs, security bugs (the sum of the *Security* and *Malicious Code* categories - 0.21% + 21.81%) are the most popular in the repository ($\geq 21\%$ each). This could be a strong indication that programmers write code that implements the required functionality without considering its many security aspects; an issue that we have previously mentioned in this chapter, and it has already been reported in literature [206].

Another observation involves bugs that we could call *Security High* and they are a subset of the *Security* category. Such bugs are related to vulnerabilities that we are interested in in this research and can lead to code injection attacks (see Section 1.1). Table 1.5 presents the number of releases where at least one of these bugs exists. In essence, 5,501 releases ($\approx 4,77\%$), contained at least one severe security bug. Given the fact that other projects include these versions as their dependencies, they are automatically rendered vulnerable if they use the code fragments that include the defects. The remaining bugs of the *Security* category are grouped together with the bugs of the *Malicious Code* category in another category that we call *Security Low*. This category contains for the most part, bugs that imply violations of good Object-Oriented Programming (OOP) design (i.e. keeping variables private to classes and others). The above categorization was done specifically to point out the the bugs that can lead to code injection attacks.

Linus's Law states that "given enough eyeballs, all bugs are shallow". In a context like this, we expect that the project versions that are dependencies to many other projects would have a small number of security bugs. To examine this variation of the Linus's Law and highlight the *domino effect* [213] we did the following: during the experiment we retrieved the dependencies of every version. Based on this information we created a graph that represented the snapshot of the Maven repository. The nodes of the graph represented the versions and the vertices their dependencies. The graph was not entirely

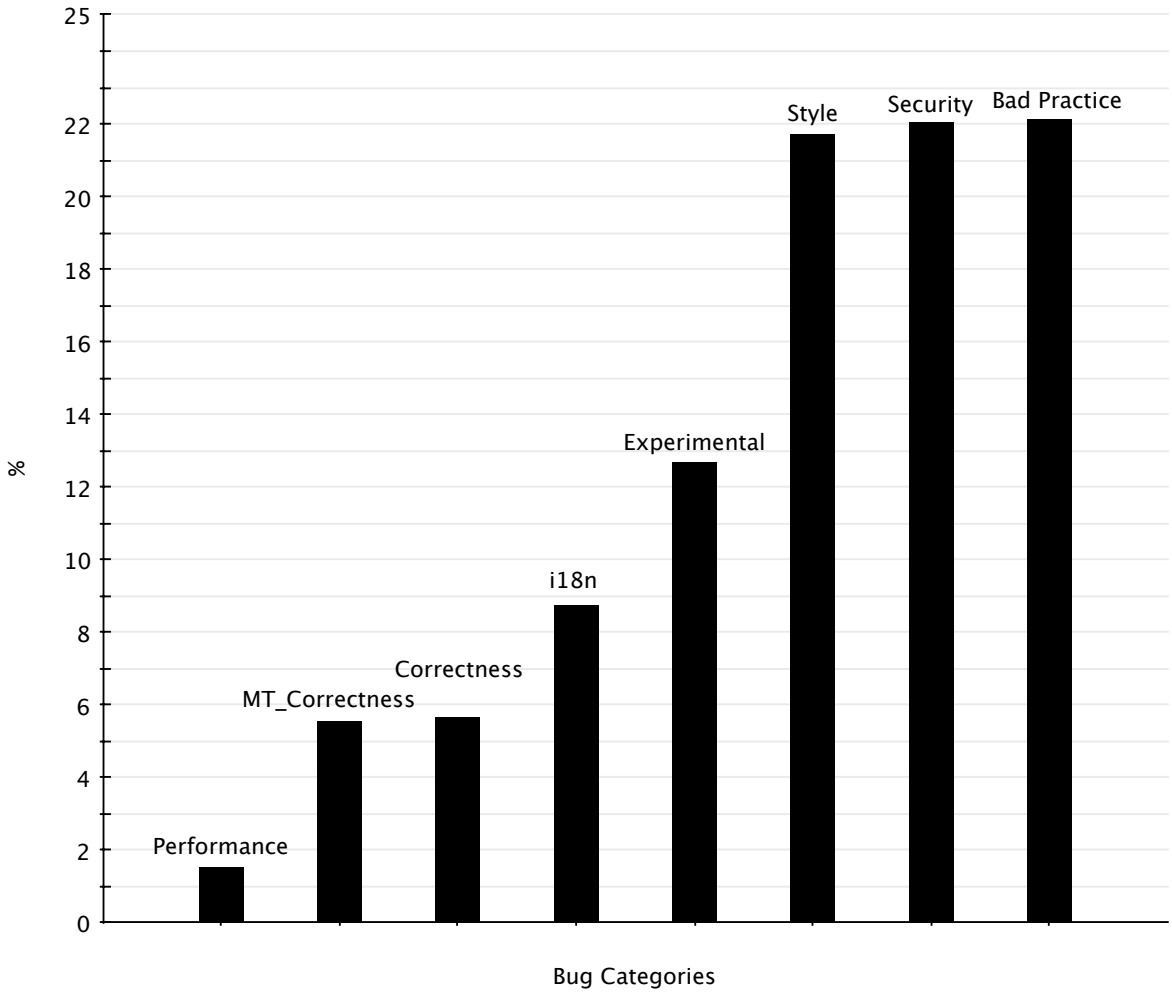


Figure 1.6: Bug percentage in Maven repository.

Bug Description	Number of Project Releases
HRS: HTTP cookie formed from untrusted input	151
HRS: HTTP response splitting vulnerability	1,579
SQL: non-constant string passed to execute method on an SQL statement	1,875
SQL: a prepared statement is generated from a non-constant String	1,486
XSS: JavaServer Pages (JSP) reflected cross site scripting vulnerability	18
XSS: Servlet reflected cross site scripting vulnerability in error page	90
XSS: Servlet reflected cross site scripting vulnerability	142

Table 1.5: Number of project releases that contain at least one bug coming from the *Security High* category.

accurate. For instance, if a dependency was pointing only to a project (and not to a specific version), we chose to select the latest version found on the repository. Also, this graph is not complete. This is because there were missing versions. From the 565,680 vertices, 191,433 did not point to a specific version while 164,234 were pointing to missing ones. The graph contained 80,354 nodes. Obviously, the number does not correspond to the number of the total versions. This is because some versions did not contain any information about their dependencies so they are not represented in the graph. After creating the graph, we ran the PageRank algorithm [38] on it and retrieved all PageRanks for every node. Then we examined the security bugs of the fifty most popular nodes based on their PageRank. Contrary to Linus's Law, thirty three of them contained bugs coming from the *Security Low* category, while two of them contained *Security High* bugs. Twenty five of them were latest versions at the time. This also highlights the domino effect.

Our findings indicate that security bugs were one of the top two bug categories existing in a large ecosystem. Also, the existence of the domino effect indicated that no matter how well a programmer secures a software component it won't matter if he or she is using vulnerable code fragments coming from another library. In addition, as we saw in Table 1.5, there are thousands of projects that contain at least one software bug that can actually lead to a CIA attack. Finally, we showed evidence that Linus's Law does not apply in the case of the security bugs and there are latest versions of software that include bugs directly related with CIAs. The above findings indicate that *programmers do not seem to consider bugs that can lead to CIAs when they develop applications, thus rendering them vulnerable to such attacks*.

1.3 Hypothesis and Contributions

Over several years of efforts, a large body of knowledge has been assembled regarding code injection attacks consisting of countermeasures, novel ways of attacking, and others. Unfortunately, as we will see in the next chapter, many countermeasures are ad-hoc solutions that lack a stable theoretical background, while others are quite difficult to adopt. The ineffective countermeasures, the numerous attack vectors that can be used to perform a CIA, and the dominance of software bugs that lead to such attacks within large software ecosystems, indicate that *the creation of a new framework against CIAs is actually meaningful*. In this dissertation we show that:

unique identifiers created by blending *features coming from legitimate code statements* with *elements extracted from the application's execution environment*, can be used by a *training-based* approach to detect *executable source code injection attacks* in an *efficient* manner.

To prove this assertion, we have implemented corresponding mechanisms that counter CIAs that use SQL, XPath and JavaScript as attack vectors. Note that such attacks top the vulnerability lists of numerous bulletin providers for several years.[9] Consider the The Open Web Application Security Project (OWASP)[10] Top Ten project whose main goal is to raise awareness about web application security by identifying some of the most critical risks facing organizations. It is referenced by Payment Card Industry Security Standards Council (PCI DSS),[11] Defense Information Systems Agency (DISA)[12] and other numerous researchers. In its three consecutive Top Ten lists (2007, 2010, 2013), different source code-driven injection attacks dominate the top five positions.

The main contributions of this thesis can be summarized as follows:

1. We surveyed and analyzed the countermeasures developed to detect CIAs. Our analysis was based on criteria like flexibility, accuracy, overhead, usability and implementation dependencies (see Chapter 2).

2. We analyzed the evolution and the behavior of security bugs that are related to input validation and could lead to CIAs. Our analysis involved a data set containing 17,505 projects with 115,214 versions. Our results indicate that although such bugs do close over time in particular projects, there is no indication that they decrease as projects mature. We also found that bugs related to CIAs: a) are not eliminated in a way that is particularly different from the other bugs, b) they are not proportionally related to the project's size and c) they do not appear together with bugs coming from other categories like performance and style (see Section 3.1).

3. We developed a unified approach that detects executable source code injection attacks (see Section 3.3). Our approach operates in two modes, namely: training and production. During training mode, our method analyzes the code to be executed and creates unique identifiers ("signatures"). All signatures are stored in an auxiliary table. Then, during production our method generates a signature and validates it by checking if it exists in the table of valid signatures. Our main contribution lies in the fact that during signature generation, the approach combines features that can depend on the code statement that is about to be executed, with elements extracted from the execution context of the protecting entity. In this manner our method differs from the traditional training methods (see Section 2.2.4).

4. We provided prototypes that implement our approach for different attack subcategories (SQL, XPath and JavaScript injection attacks – see Chapter 4).

5. We deployed the prototypes in real world conditions and tested them in terms of accuracy, operation cost and maintenance cost. Our tests indicated that our mechanisms can be of practical value as the overhead they impose is minimal and does not affect the user's experience. Also, our prototypes recognized and blocked all of the performed attacks, without producing any false positives or negatives (see Chapter 5).

1.4 Research Methodology

The methodological approach of this research is rather typical for a systems PhD. Figure 1.7 depicts our methodological approach as a Unified Modeling Language (UML) activity diagram. The process started with the selection of a specific research area: *information security*. After the problem definition (see Section 1.1) the next step was to study in depth the related work in the field and design the

[9] http://www.sans.org/top-cyber-security-risks/, http://cwe.mitre.org/top25/
[10] https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project
[11] https://www.pcisecuritystandards.org/security_standards/
[12] www.disa.mil/

framework that provides a solution to the stated problem. The implementation followed and during the testing phase we evaluated our prototypes.

In particular, our work consists of three research-oriented work items that share common ground work. Surveying the existing countermeasures and putting the problem into different contexts was critical for the approach that we developed. This involved the categorization of the countermeasures in terms of the method that they are based upon. Analyzing the evolution and the behavior of software bugs that could lead to CIAs was mostly quantitative work. It also had a qualitative aspect as we tried to study their association with other bug categories (performance or code style related bugs). To investigate the above we used freely available data from the Open Source Software (OSS) ecosystem (in particular, the Maven central repository mentioned in Section 1.2). This work item also included the studying of the relation between such bugs and a project's size, their persistence over time and others. By identifying the distinct characteristics of such bugs provided us with information that was essential to understand and cope with the problem of code injection.

The results of the above research provided a stable background to develop our approach and implement corresponding prototypes. When designing our method we also considered the following non-functional requirements [201]. Such requirements are critical when building security mechanisms that protect applications [10, 149, 187]:

- **Flexibility** We opted for an approach that could be adjusted in order to detect different CIA categories.
- **Ease of Use** We sought for an approach that can be easy to use by security experts.
- **False Positives and Negatives** Since we were developing an approach that counters a very important set of application attacks, the non-existence of false positive and negative alarms was a reasonable requirement.
- **Implementation Independence** One of our main goals was that the mechanisms based on the proposed approach do not depend either on the characteristics of the programming language that was used to develop it or on the implementation details of the protecting entity.
- **User Experience** Finally, we aimed for efficient mechanisms that have significantly low overhead.

This work item was tackled in three iterations: (i) the development of the approach and the prototype mechanisms, (ii) testing in terms of accuracy and computational overhead, and (iii) improvement of the approach leading to more robust implementations. In particular, to evaluate the effectiveness of our method we searched for applications that had a record of being vulnerable to CIAs and tested our prototypes against real-world attacks. For the prototype implementation we focused on the prevention of CIA categories that were *critical* and *popular* at the same time (for instance, JavaScript injection attacks are currently one of the most prominent attack categories exploiting both web and mobile application vulnerabilities). Finally, we aimed at prototypes that would be easy to adopt, have limited false alarm rates and incur minimum overhead.

1.5 Thesis Outline

The remainder of this dissertation is organised as follows:

- In Chapter 2, we describe and analyse previous work that relates to and motivates this dissertation. We provide a systematic literature review and we show the various approaches proposed to detect CIA defects. Then, based on specific requirements we evaluate the existing mechanisms, setting the basis for our own approach.

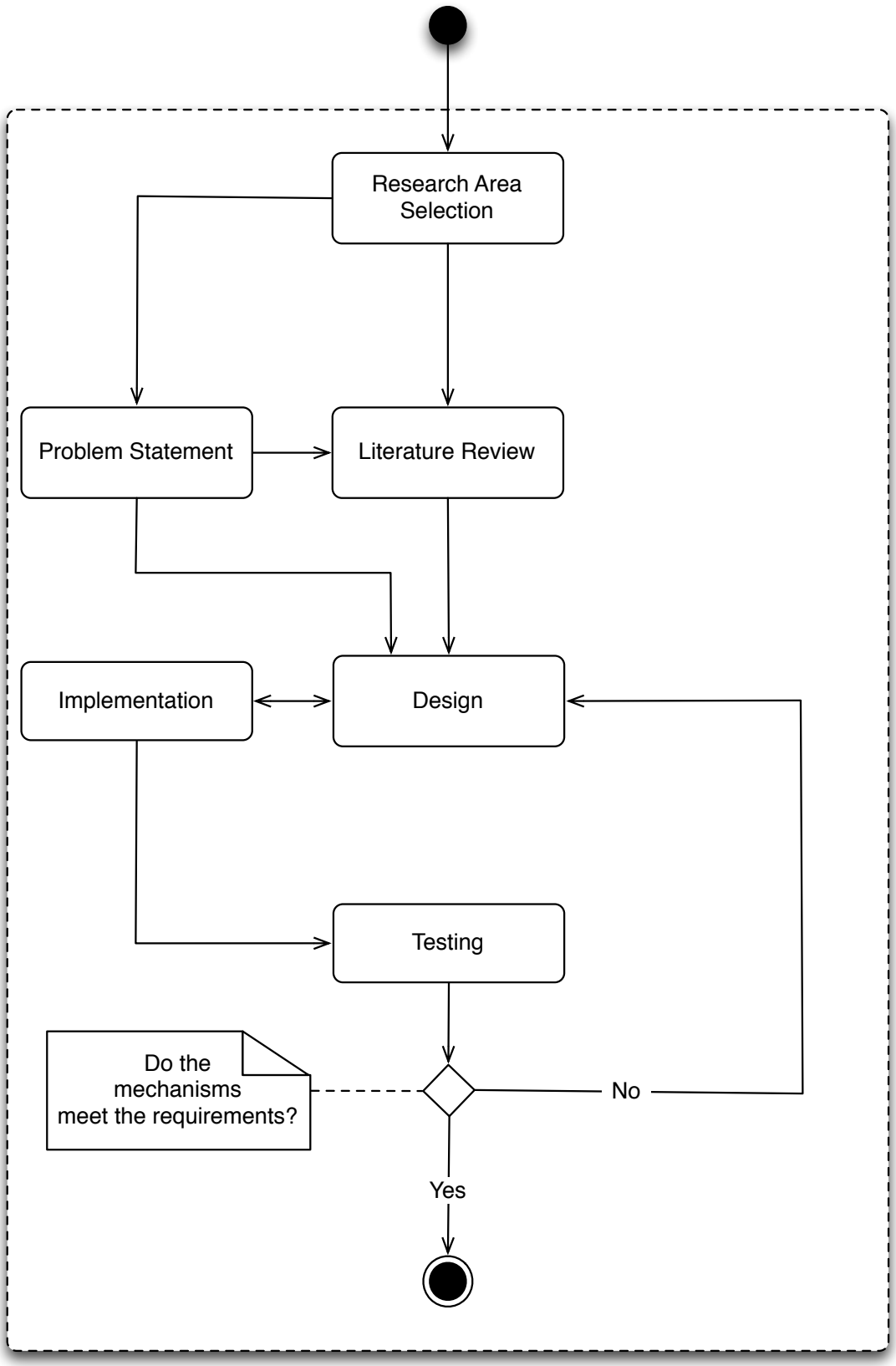


Figure 1.7: PhD methodology.

- In Chapter 3, we present our approach. First we analyze the evolution and the characteristics of bugs that lead to CIAs. Then we present the theoretical modeling of the code injection problem and how our approach deals with it with respect to the requirements presented in the previous chapter. Finally, we provide the design of three mechanisms. Our mechanisms are the realization of our approach in two different contexts. The first involves DSL-driven CIAs, and the second involves dynamic language-driven CIAs. The applicability of our approach on different CIA categories, hints at a possible generalisation to any kind of CIA.
- In Chapter 4, we present the implementation details of the three mechanisms based on the corresponding design of the previous chapter. Our mechanisms detect CIAs that counter SQL, XPath and JavaScript injection attacks respectively. They can be easily deployed and they are implementation independent.
- In Chapter 5, we provide results concerning the testing of our mechanisms in terms of accuracy and computational overhead which are established requirements for security mechanisms. Our tests involved real-world attacks previously used for evaluation in research and plenty of applications known to be vulnerable.
- Finally, in Chapter 6, we present a list of future research topics that emerge from this work and conclude the dissertation.

Chapter 2

Approaches for Countering Code Injection Attacks

Two different basic methods are used to deal with the code injection problem (see Figure 2.1):

- **Static Analysis** involves the inspection of computer code to find software bugs that could lead to a code injection attack without actually executing the program.
- **Dynamic Detection** observes the behavior of a running system in order to detect a code injection attack.

This distinction is based on an extensive survey on mitigating software vulnerabilities [196]. Both of the above approaches originate from two complementary security concepts, namely: *add-on security* [10] and *build-in security* [145]. The first, involves the development of methods and tools that secure systems after their deployment. In the second case, programmers try to eliminate software vulnerabilities while applications are created.

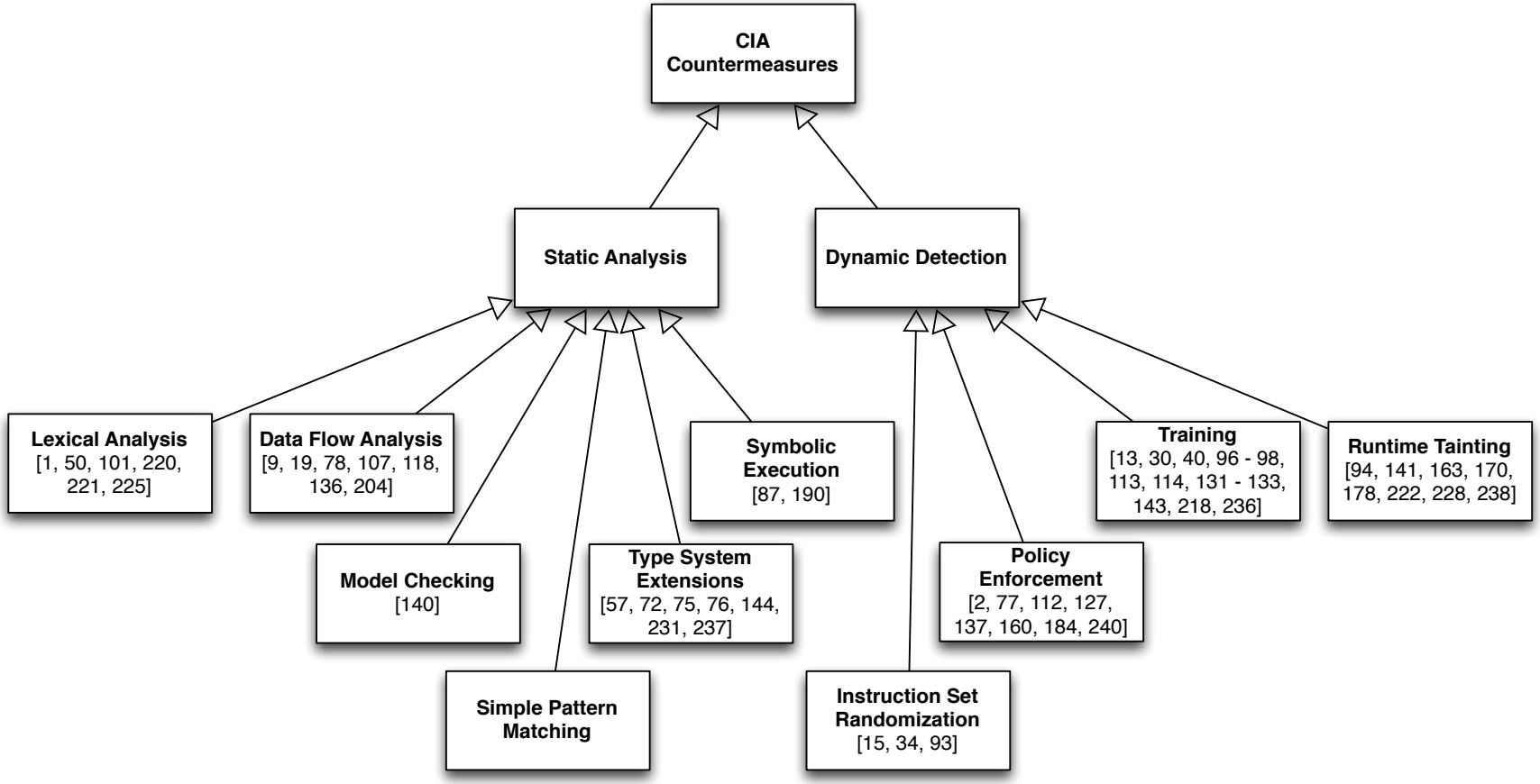


Figure 2.1: The basic categories of CIA countermeasures. For each approach we provide the references that present corresponding mechanisms.

Note that, countermeasures that prevent two subcategories of CIAs, binary code injection [239] and SQL injection [99] in particular, have already been surveyed. The body of work done on CIAs though, exceeds the boundaries of this research.

In the former case [239], the countermeasures that have been surveyed detect defects that lead to attacks that are language-dependent and concern the lack of memory-safety in two specific programming languages as we will see in the following sections. This means that they do not appear in all kinds of applications. Still, some of these countermeasures are classics that led to the development of many more sophisticated mechanisms, thus we mention them here. In particular, Subsections 2.1.2, 2.1.3 and 2.2.1 contain such classic tools.

In the latter case [99], the survey is quite old and since then the number of countermeasures that detect SQL injection attacks alone, has doubled. Besides, SQL injection attacks belong to a large attack category that is language-independent. In essence, attacks can be performed in numerous applications regardless of the programming language used to develop them, and by performing different attack techniques. Interestingly, the authors of this survey do not take accuracy into account in their research.

Finally, as we mentioned in Section 1.3 the approach that we propose in this thesis involves the detection of source code-driven injection attacks. Thus, it would be reasonable to present methods and tools that deal with similar defects.

2.1 Static Analysis

The main idea behind static analysis is to identify software defects during the development phase. Currently, there are many modern software development processes that include static checkers for security as their integral parts [39, 79, 92]. From the usage of utilities like `grep` to complex methods, static analysis has been an evolving approach to detect software vulnerabilities [51].

Initially, the most straightforward and sensible approach is the adoption of *secure coding practices* [108, 145, 219]. For instance, most SQL injection attacks can be prevented by passing user data as parameters of previously prepared SQL statements instead of intermingling user data directly in the SQL statement. However, this does not always happen, as programmers may not be aware of them, or time schedules may be tight, encouraging sloppy practices instead.

2.1.1 Simple Pattern Matching

During a manual, code review it is easy to look for functions associated with code injection defects. Using existing tools available in almost every operating system, security auditors can search through a set of files for an arbitrary text pattern. These patterns are commonly specified through a regular expression. Accompanied by a well organized list of patterns, the auditor can quickly identify locations at which a program might face security problems.

If auditors choose to use utilities like `grep` and `qgrep` though, they must check for every vulnerability manually. Apart from this, they must have an expert knowledge since there is a great number of such vulnerabilities [221]. Furthermore, the analysis these utilities perform is naive. For example there is no distinction between a vulnerable function call, a comment, and an unrelated identifier. Hence a higher prevalence of false positives should be expected [50]. Finally the output can be disorganized and overwhelming.

The distinct disadvantages of pattern scanning and the continuous emergence of new defects were some of the main reasons that led to more sophisticated approaches.

2.1.2 Lexical Analysis

Lexical analysis is one of the first approaches used for detecting security defects. This is because it is simple and easy to use. Lexical analysis is based upon formal language theory and finite state automata [6]. As a term, it is mostly used to describe the first phase of the compilation process. However, there is no difference between this phase and the method that we describe here [146]. The two differ only in the manipulation of their outcome.

There are three phases that can be distinguished here, namely: scanning, tokenizing and matching [120]. In the first two phases possible character sequences are recognized, classified and transformed into various tokens. Then the resulting sequences are associated with security vulnerabilities. Specifically, there are lists that contain entries of vulnerable constructs used during the matching phase. After a successful match an alert message warns auditors, describing the vulnerability and providing them with alternative usages. In this way, many false positives reported in pattern matching are avoided here. For example, the existence of the `scanf` character sequence in the following method declaration will never be reported:

```
float scanfornumbers(File f);
```

Comments that contain the specified sequence will be omitted too. To avoid false positives that have to do with variables specific checks are performed. For example the following variable name will not be reported, even though programmers do not generally use such names as variable names. This is because a left parenthesis is expected right after the occurrence of `scanf`.

```
int scanf;
```

The lexical analysis approach is implemented by security utilities like BOON [225], PS*can* [49, 101, 116], ITS*4* [1, 220, 221], *Flawfinder*[1] [1] and RATS[2] [1, 50, 129]. For the most part, these tools scan source code pointing out unsafe calls of string-handling functions, time-of-check and time-of-use race conditions, functions that replace process images (i.e. `exec`), poor random number retrieval and others. Then they provide a list of possible threats ranked by risk level. This level is usually specified by checking the arguments of such functions. The vulnerability lists are simply constructed making the addition, removal, and modification of an entry quite easy. Specifically ITS4 keeps a simple text file as a library while RATS preserves its rules in an XML based schema [193]. All the aforementioned tools scan C and C++. Only RATS can be also used on Perl, Python and PHP only to find Time Of Check, Time Of Use (TOCTOU) race condition vulnerabilities rather than other CIA defects. This exclusiveness, lies on the fact that C and its standard libraries are very susceptible to security defects [193]. For example, many buffer overflow exploitations and format string attacks come from using standard C library functions like `gets`, `strcpy` and `scanf` [134] (see Subsection 1.1.1).

Lexical analysis can be flexible, straightforward and extremely fast. With one or more non-processed files as input, and simple descriptions as output, developers can quickly check their code for vulnerabilities. Also they can easily update and edit their vulnerability library with new possible threats due to its simplistic nature. i.e. The library that RATS uses is formatted in XML while ITS4 employs a text file of vulnerabilities with a simple format. The best thing about lexical analysis utilities is that they helped the gathering and depiction of a tentative set of security rules in one place for the first time [146].

Although superior to manual pattern matching, this approach has no knowledge of the code's semantics or how data circulates throughout a program. As a result there are several false positive and negative reports [51, 59]. Specifically, based on simple assumptions and without considering context in any way this method could report every possible dangerous function call as a problem, no matter

[1] `http://www.dwheeler.com/flawfinder/`
[2] `http://www.security-database.com/toolswatch/RATS-v2-3-Rough-Auditing-Tool-for.html`

how carefully it is used in the program. Hence, auditors must be experienced programmers in order to interpret the results of lexical analysis tools and they must regard them as as an aid in the code review process and not as a firm solution to find software vulnerabilities [59, 243].

2.1.3 Data-Flow Analysis

Data flow analysis is another compiler-associated approach used to discover software defects. It is more sophisticated and more appropriate for a comprehensive code review than lexical analysis.

Data flow analysis can be described as a process that gathers details that concern the definition and dependencies of data within a program without executing it [83, 158]. In addition, data-flow analysis algorithms can document all sequences of specific types of events which might occur in a program execution. The key insight of this approach is a Control-Flow Graph (CFG). Based on the program's CFG, this method examines how data moves throughout a program by representing all its possible execution paths [43, 51]. A control flow graph depicts the logic structure of a program. Nodes represent statements or expressions, and edges represent the handover of control between the nodes. Each possible execution path of the program has a corresponding path from the entry to the exit node of the graph [83, 86]. Early applications of this method derive from the area of compiler optimization and include the elimination of invariant code from loops, constant folding, the elimination of dead code and other things [6].

In every program there is a number of variables that could be vulnerable and the transfer of information from such a variable to another one can be critical for the programs normal execution. By traversing the CFG of a program, data-flow analysis can determine where values are generated and where they are used. Hence this approach can be used to describe safety properties that are based on the flow of information [4]. As an example: it is very likely that the CFG of a program with an SQL injection defect, will include a data-flow path from an input function to a vulnerable operation. For instance, in the following code fragment, user input reaches a method that interacts with a database without any prior validation:

```
uName = request.getParameter("username");
String query = null;
if (uName != null) {
    query = "SELECT * FROM table WHERE uname = ' "+uName+" '";
    rs = stmt.executeQuery(query);
} else {
    ...
}
```

As a result, the danger of an SQL injection attack is prominent. In this case, data-flow analysis can be applied to connect the unchecked input with the execution of a query based on user input and issue a notification about the vulnerability. But to make such an association, the analysis algorithm should take into account that:

1. the return value of `getParameter()` can be tainted,
2. the result of a concatenated string is tainted if one of the substring is tainted, and
3. that a method that executes an SQL command has security repercussions.

In this case, the algorithm will determine if there is a validation logic between the program location that receives the data and the location that will actually call this data into play, and act accordingly i.e. show to the auditor the path for the issue.

The various adaptations of this approach depend on the kind of the vulnerability that each adaptation aims to report. Initially, data flow analysis was applied in the context of global program optimization. These early applications of the approach aimed at languages like C, Fortran and Jovial, and they were quite simple. To their majority, they performed interprocedural analysis and their primary target was to detect various programming errors just by using the CFG of a program. Such errors included dead and double definitions, non-initialized variables, incorrect sequences of program events and others. Three indicatory tools that use such light adaptations are DAVE, *Omega* and *Cesar* [166, 168, 232].

Data race detection relies in great part on this approach. Choi et al. [53] in particular, proposed a system, that performs data-flow analysis over a program's "interthread" call graph to detect race hazards. An "interthread" call graph is virtually a CFG that additionally presents threads spawns as directed edges. By using this graph this system builds up sets of abstractions of actual objects that would exist at runtime. Then, these abstractions are used together with other techniques to determine if there is a race condition. Data flow analysis serves as a basis to another framework (*Racer*X) that detects both race conditions and deadlocks [74]. Specifically this framework, starts by building a CFG of the entire program. Then performs a Depth First Search (DFS) traversal down the graph. When it encounters a locking statement it adds a lock element to a specified set and removes it when it comes upon the corresponding unlocking statement. Finally, by using heuristics, statistical analysis and ranking it produces its results.

As it is already clear from the aforementioned example, data-flow analysis is tailored to localize command injection vulnerabilities. This is why there are numerous adaptations that detect SQL injection defects, cross-site scripting vulnerabilities, buffer overflows and others. In addition, most of the creators of such frameworks, claim that with minor changes, their prototypes can be equally applied to also detect other kinds of such defects. But to encounter such anomalies, a data-flow analysis algorithm needs more than a vulnerability library that connects coding constructs with software defects. Furthermore, a rule-pack containing specific control flow rules and ad-hoc checkers that run upon the CFG are required. The most common rules used in this method, are the *source*, the *pass-through* and the *sink* rules [51]. A source rule denotes the starting point of a possible hazard while a sink rule depicts the coding construct where the hazard takes place. In the aforementioned example, a source rule will apply for the first line where input can come from a malicious user. The sink rule on the other hand will refer to the fifth line where attacked-controlled data can reach the database. The pass rule indicates the code that exists between the above two and carries the possibly corrupted data. For the most part, these rules are maintained in external files that use a specific format to describe them.

Livshits et al. [136] based their work on the functionality presented above to detect possible SQL and JavaScript injection defects in Java applications. Nagy et al. [162] have proposed a number of checkers that locate buffer overflow and format string defects. The idea behind their proposal is to mark all the user-input-related parts of the source code. These checkers are implemented as plug-ins to the *CodeSurfer*[3] tool [9], a commercial tool that performs data-flow analysis on C/C++ programs. Another two indicative tools used to detect injection anomalies are *Pixy* and XSS*detect* [118]. Both of these tools detect cross-site scripting vulnerabilities in web applications. The latter, released by Microsoft, runs as a Visual Studio plug-in and analyzes .NET Intermediate Language (IL) which is read directly from the compiled binaries. Pixy on the other hand, is a standalone open source tool, that examines PHP scripts. In many cases, rules can appear directly in the code of the program in the form of annotations. A tool that considers control flow graphs and uses annotations at the same time to find buffer overflows and memory leaks is *Splint* [78]. FindBugs [19, 107, 204] (see Subsection 1.2) is also a static analyzer based on data-flow analysis.

Many researchers, do not base their frameworks on data-flow analysis to detect vulnerabilities in the first place, but they utilize it after the detection, only to find the declarations of dubious variables. For example, *AutoPag* is a tool that searches the code to find out-of-bound violations [135]. After

[3] http://www.grammatech.com/research/technologies/codesurfer

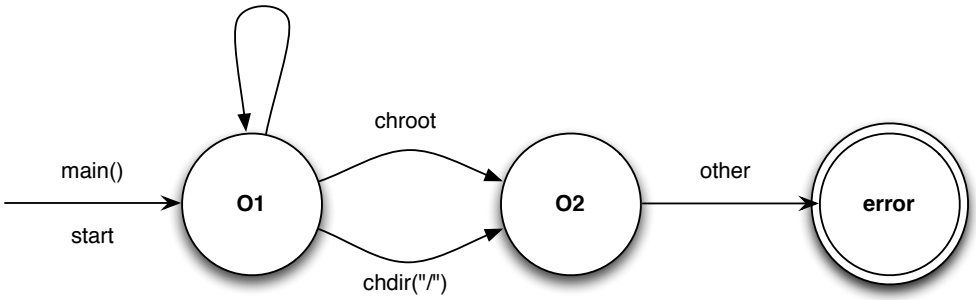


Figure 2.2: A finite-state automaton representing the secure usage of the `chroot` system call.

detecting these violations, it uses this approach to find the declaration statement, scope etc of the overflowed variable quickly and efficiently. Inter-procedural data-flow analysis has been also used for the automatic verification of smart card applications, which are security critical by their own nature [7].

As a more sophisticated approach than lexical analysis, data-flow analysis exhibits fewer false positives and negatives than the former. For example, many buffer overflows are not exploitable because the attacker can't handle the data that overflows the buffer. By using this method, an auditor can in fact distinguish exploitable from non-exploitable buffer overflows. The major advantage of data flow static analysis is that it can identify vulnerabilities that could actually occur when real application paths are exercised and not just dangerous coding constructs. This is why lexical analysis is an error-prevention practice and data-flow analysis is an error-detection practice. Another important feature of this approach is the direct identification of the variables affected by a vulnerability and the specific path that this vulnerability occur. This is very meaningful for a programmer that wants to write more secure code.

Many state that dynamic and static data-flow analysis are complementary [46]. This is because in static dynamic analysis, there is a high false alarm rate while dynamic data-flow analysis is more accurate than its static sibling since the flow values are tracked and managed at runtime. For example in Pixy the false positive rate is nearly 50% while on Splint the rate exceeds 60%. But still, the dynamic approach suffers from run time overhead [5]. Chang et al. [46] introduced a system that combined the two siblings with successful results. Specifically in this system, static data-flow analysis was used in order to reduce the cost of tracking all program objects at runtime.

Data flow analysis is an efficient and feasible way of analyzing a program. In practice though, even large programs execute only some small parts of their total possible paths. This is why for a given analysis, the nature of the problem may help to reduce the number of the paths to be checked. On the security perspective, it can be used to ensure conformance of source code to good programming practices and it can detect numerous vulnerabilities.

2.1.4 Model Checking

Model checking is a formal verification approach developed to check both hardware and software systems [25, 48, 55]. It has also been applied to detect flaws in network protocol implementations [100, 159]. Model checking is based on graph theory and finite state automata [56, 151].

A software model checking framework, accepts a system's source or binary code as input, and checks automatically if it satisfies specific properties. First, the framework analyzes statically the code to extract a high-level representation of the system, namely a model. This model usually corresponds to a control-flow graph or a pushdown automaton [28, 47]. In model checking, the nodes of the model represent the states of the system, while the edges portray the possible transitions that change the status of the system. The properties are often expressed either as assertions, as formulas of temporal logic,

or as finite state automata [154, 175]. By traversing every execution path of the model, the framework determines whether certain states represent a violation of the provided properties.

There is a great number of dangerous programming practices that can be accurately modeled with equivalent security properties. For example, the chroot system call should be immediately followed by a call to `chdir("/")`. Otherwise, the current working directory could be outside the isolated hierarchy and provide access to a malicious user via relative paths. A finite state automaton that represents this property can be seen in Figure 2.2. With a high-level representation of the system at hand and by using ad-hoc algorithms [185], properties like the above can be easily checked. Consider that the high-level representation of the system is a pushdown automaton. Following the previous example the goal would be to check if a risky state of the automaton presented in Figure 2.2 is reachable in the pushdown automaton. Likewise, it is possible to state the detection of code injection defects as a reachability problem [217].

There are many tools based on model checking to detect software vulnerabilities. Classic tools include SPIN [105], SMV [147] and MOPS [47, 192]. Their functionality is very similar to the one described in the aforementioned paragraph. Even though they are representative of the approach, they do not support the detection of CIA defects. XMC [181] is another tool that can be used to analyze the consistency of security pattern compositions [68] but it does not detect CIA vulnerabilities. A security pattern is a solution to a recurring information security problem, it encapsulates security expertise in the form of worked solutions to recurring problems. Finally, QED is a model checking system that accepts as input web application written in the standard Java servlet specification[4] and examine them for code injection vulnerabilities [140].

The users of a model checking tool do not need to construct a correctness proof. Instead, they just need to enter a description of the circuit or program to be verified and the specification to be checked. Still, writing specifications is hard and code reviewers with experience are needed. Model checking is faster compared to the aforementioned static analysis methods. One of the key features of model checking is that it can either reassure developers that the system is correct or provide them with a counterexample. As a result, together with the discover of a security issue, auditors are provided with a possible solution. A major problem in model checking is the state explosion issue [56, 151]. The number of all states of a system with many processes or complicated data structures can be enormous.

2.1.5 Symbolic Execution

Symbolic execution generalizes testing by using unknown symbolic variables during evaluation [42, 126]. In essence, symbolic execution is a means of analyzing a program in order to determine which inputs cause each part of a program to execute. An interpreter analyzes the program, assuming symbolic values for inputs rather than obtaining actual inputs. In essence, the concept of *symbolic execution* is tailored to detect CIAs. Nonetheless, it suffers from the state explosion issue like *model checking*. Due to this, tools that implement this method struggle to achieve scalability.

To counter SQL injection attacks, Fu et al. [87] have proposed SAFELI. First, SAFELI analyzes the code to detect code constructs used by the application, to interact with a database. At each location that submits an SQL query, an equation is constructed to find out the initial values of that could lead to a security breach. The equation is then solved by a hybrid string solver where the solution obtained is used to construct test cases. If a defect is detected, an attack is replayed by the tool to developers. Ruse et al. [188] detect DSL-driven injection vulnerabilities in a similar manner.

Saxena et al. [190] have proposed a framework called *Kudzu* to detect JavaScript injection attacks. Given a URL for a web application, Kudzu automatically generates test cases to explore its execution space. To handle JavaScript code's complex use of string operations, they have designed a new language of string constraints and implemented a solver that supports the specification of boolean, ma-

[4] https://jcp.org/aboutJava/communityprocess/final/jsr315/

chine integer, and string constraints, including regular expressions, over multiple variable-length string inputs.

Symbolic execution has also been used together either with genetic algorithms to detect JavaScript injection attacks [17] or with runtime tainting (see Subsection 2.2.1) to detect SQL injection attacks [58]. Symbolic execution shares a similar problem with model checking (see Subsection 2.1.4). Symbolically executing all program paths does not scale with large programs since the number of feasible paths grows exponentially.

2.1.6 Type System Extensions

A type system is a collection of rules that assign a property called a type to the various constructs of a program [173]. Type systems were formalized early in the twentieth century in an attempt to address problems that involved logical paradoxes in mathematics. Later on, they became standard tools in logic, especially in proof theory. The basic reasons that make a type system the foundation stone of a programming language are:

- **Error Detection** One of the most typical advantages of static type checking is the discovery of programming errors at compile time. As a result, numerous errors can then be detected immediately, rather than discovered later upon execution.
- **Abstraction** The existence of a type system enforces disciplined programming. In practice, type systems form the backbone of languages and allow more abstract style of design, completely independent from the implementation.
- **Language Safety** If designed correctly, language safety can be achieved through static type checking.
- **Documentation** Types are useful when studying a computer program. Strict types give hints about the program behaviour and constitute a form of documentation for a programmer that cannot become outdated since type checking occurs for every compilation.
- **Efficiency** Early type systems were introduced to improve the efficiency of programming languages. For example, Fortran improved its efficiency by differentiating integers from floating point arithmetics.

By defining interfaces between different parts of a computer program, and then checking that the parts have been connected in a consistent way, security bugs can be eliminated. This checking can happen either at compile time or at run time. For the most part, such extensions aim to overcome the problems of integrating different programming languages. For instance, the integration of SQL with the Java programming language is typically realised with the Java Database Connectivity (JDBC) application library [80]. By using it, the programmer has to pass the SQL query to the database as a string. Thought this process, the Java compiler is completely unaware of the SQL language contained within the Java code. Consider the case of a Java application that needs to access a Database. A typical implementation for the Java programming language is presented in the listing below:

```
public class SQLExample {
    public void executeSQL(String id) {
        try {
            Statement stmt = connection.createStatement();
            ResultSet rs = stmt.executeQuery(
                "SELECT * FROM t where id = " + id
            [...]
```

```
        } catch (SQLException e) { /* error */ }
    }
}
```

The class `SQLSimpleExample` contains a method called `execSQL` that accepts a String formal parameter, labeled `id`. The value passed by the parameter is concatenated with the SQL query string. The above implementation is vulnerable to SQL injection attacks since the parameter `id` is blindly concatenated to the SQL query and is never checked in any way.

Type-safe programming interfaces like *SQL "Domain Object Model"* [144] and the *Safe Query Objects* [57] were two of the first attempts to detect DSL-driven injection attacks via type extension. Both of the above mechanisms act as preprocessors and translate an SQL database schema into the host general purpose language. The generated collection of objects is used as an application library for the main application, thus ensuring type safety and syntax checking at compile-time. SQLJ [72] is a language extension of Java that supports SQL. It offers type and syntax checking for both languages at compile-time. JDBC Checker [90] analyzes Java code and searches for JDBC calls and SQL statements, then detects possible errors. Notably, the SQL statements are checked against the database schema. *SugarJ* [75] provides a method through which languages can be extended with specific syntax, in order to embed DSLs. The major contribution of this framework is that can be applied on many languages as host languages. Currently it supports Java, Haskell and Prolog. All the aforementioned mechanisms, eliminate the incestuous relationship between untyped Java strings and SQL statements, but don't address legacy code, while also requiring programmers to learn a radically new API.

*Web*SSARI is used to verify web applications [237] written in PHP. It is based on Denning's lattice model for analyzing secure information flow in imperative programming languages [63] and uses type qualifiers as a means for explicitly associating security classes with variables and functions that can lead to SQL injection defects. Wassermann and Su [231] propose a sound and precise approach, which depends on specifying the semantics of all PHP string functions. (The implementation described contains specifications for 243 functions.) The same authors have also proposed an approach that deals with static analysis and coding practices together [230, 231]. Specifically, they automatically analyze the application's source code to locate SQL statement invocations that are considered unsafe. To analyze the code they utilize context free grammars and language transducers [155]. Syntax embeddings have also been proposed to detect code that is susceptible to various kinds of code injections [37]. This approach embeds the grammar of a DSL language into that of a host language and automatically reconstructs code statements by adding functions that provide security layers. Such an approach is quite interesting since it introduces security features at a very early stage of software development.

Type extensions is a formal way to wipe out code injection defects, but they have two distinct disadvantages. First, programmers need to learn new constructs and modify their code in multiple places, and, second, they are not tested against the various forms of attacks so they cannot be considered as efficient tools for security purposes.

2.2 Dynamic Detection

Dynamic detection involves the development of methods and tools to fortify such applications without actually removing the defects from the application's code. A great number of methods that belong to this category involves some kind of *dynamic program analysis* [33]. Dynamic analysis requires a running system and involves sufficient test inputs to examine the behavior of a system.

2.2.1 Runtime Tainting

Runtime tainting is based on data-flow analysis (see Subsection 2.1.3). In practice, it enforces security policies by marking untrusted ("tainted") data and tracing its flow through the program. Runtime

tainting may be viewed as an approximation of the verification of non-interference [223] or the more general concept of secure information flow. Since information flow in a system cannot be verified by examining a single execution trace of the system, the results of taint analysis will necessarily reflect approximate information regarding the information flow characteristics of the system to which it is applied.

Runtime tainting is a feature in some programming languages, such as Perl[5] and Ruby. The following Perl code is vulnerable to SQL injection since it does not check the value of the `$foo` variable, which is instantiated by user input:

```
#!/usr/bin/perl
my $name = $cgi->param("foo");
...
$dbh->TaintIn = 1;
$dbh->execute("SELECT * FROM users WHERE name = '$foo';");
```

If taint mode is turned on, Perl would refuse to run the command and exit with an error message, because a tainted variable is being used in a query. Without such checking, a user could enter *foo'; DROP TABLE users - -*, thereby running a command that deletes the entire database table.

SigFree is a tool that follows this approach to block buffer overflow attacks by detecting the presence of malicious binary code [228]. This is motivated by the fact that buffer overflow attacks typically contain executable code while legitimate client requests never contain executable code in most services. Still, this is not always true and because of this the tool suffers from false positives. LIFT [178] also counters binary code injection attacks in a similar manner. The system by Haldar et al. [238] provides runtime tainting for applications written in Java, while the work by Xu et al. [94] covers applications written in C. These mechanisms use a context sensitive analysis to detect and reject queries if untrusted input has been used to create certain types of SQL tokens. *SecuriFly* [141] is a similar mechanism based on Program Query Language (PQL),[6] which is a language for expressing patterns of events on objects.

A dynamic checking compiler called WASC [163] includes runtime tainting to prevent SQL and JavaScript injection attacks. To counter similar attacks, PHP *Aspis* [170] applies partial taint tracking at the language level to augment values with taint meta-data in order to track their origin. The authors of reference [222], use runtime tainting to prevent XSS attacks. This is done by tracking the flow of information inside the browser. If sensitive information is about to be transferred to a third party, the user can decide if this should be permitted or not. Runtime Tainting has been partially or fully used in other similar approaches [161, 163, 164, 195]. Such approaches generally require significant changes to the compiler or the runtime system.

2.2.2 Instruction Set Randomization

Another approach that has been previously proposed as a generic methodology to counter code injection attacks is instruction Set Randomization (ISR) [122, 123]. This technique employs the notion of encrypted software and it is partially based on address space layout randomization[7] (ASLR), a technique used to counter binary code injection attacks (see Subsection 1.1.1). The main idea behind ISR is to create an execution environment that is unique to the running process. This environment is created by using a randomization algorithm. Hence, an attack against this system will fail as the attacker cannot guess the key of this algorithm. The main issue with this approach is that it uses a secret key in order

[5] http://search.cpan.org/~rhandom/Taint-Runtime-0.03/lib/Taint/Runtime.pm
[6] http://pql.sourceforge.net/
[7] http://pax.grsecurity.net/docs/aslr.txt

to match the execution environment. As a result, security is dependent on attackers not being able to discover the key.

In the case of JavaScript, consider a XOR function that encodes all JavaScript source of a web page on the server-side and then, on the client-side, the web browser decodes the source by applying the same function again. Variations of this approach include: *Noncespaces* [93] and *x*JS [15]. Even if ISR is theoretically a sound approach for countering code injection, these implementations have flaws. Noncespaces does not protect from persistent data injection. XJS does not have such problems and covers a wide variety of JavaScript injection attacks. Still, it does not counter phishing-driven JavaScript injection attacks and XCS attacks.

Solutions based on ISR have been also applied to native code and to SQL injections. For instance, SQL*rand* [34] implements ISR as follows: the modified query is either reconstructed at runtime using a cryptographic key that is inaccessible to the attacker, or the user input is tagged with delimiters that allow an augmented SQL grammar to detect SQL injection attacks [40, 176, 208]. Even if SQLrand imposes a low computational overhead, it imposes an infrastructure overhead since it requires the integration of a proxy for the database system. All ISR implementations require significant source code modifications.

2.2.3 Policy Enforcement

Policy enforcement is mainly associated with database security [52, 165, 202, 215] and operating system strict access controls [104, 110, 233]. In such contexts, policies expressed in specific languages [8], usually limit information dissemination to authorized entities only. Currently, policy enforcement is one of the most common approaches to detect JS injection attacks. In this approach, web developers define specific security policies on the server side. Then the policies are enforced either in the user's browser at runtime or in a server-side proxy that intercepts all HTML responses.

All modern browsers include a JavaScript (JS) engine to support the execution of JavaScript. Most JS engines employ restrictions like the *same origin policy* [210] and a *sandbox mechanism* [65, 177]. Still, such schemes cannot stop malicious users from injecting scripts into the user's browser. Consider a legitimate web page that does not validate the input posted by its users. By exploiting this vulnerability, an attacker can post data that will inject JavaScript into a dynamically generated page. Thus the attacker can trick a legitimate user into downloading a well-hidden script from this host in order to steal the user's cookies. This injected script is confined by a sandboxing mechanism and conforms to the same origin policy, but it still violates the security of the browser [62, 189].

Other implementations of this approach include mechanisms like *BrowserShield* [184] and *CoreScript* [240]. Both mechanisms intercept JavaScript code on a page as it executes and rewrite it in order to check if it is subject to server-provided, vulnerability descriptions. Such implementations impose a significant overhead due to the JavaScript rewriting. DSI [160], MET [77], *Blueprint* [137] and BEEP [112] require significant source modifications by the web developers in order to introduce their policies. Specifically, in MET the security policies are specified as JavaScript functions and they are included at the top of every web page while in BEEP web developers need to write security hooks for every embedded script of the application. Moreover, in Blueprint the developer needs to learn and use a new API in order to correctly escape dynamic content. *Google Caja*[8] is another policy enforcement approach provided by Google. It is based on the object-capability security model [145] and it aims to control what embedded third party code can do with user data.

A policy enforcement technique developed by Mozilla introduced the notion of signed scripts.[9] Signed scripts provide a mechanism for elevating the privileges of a script in order to allow it to interact with the browser internals and the end user's machine. This technique never became popular as other

[8] http://code.google.com/p/google-caja/

[9] http://www.mozilla.org/projects/security/components/signed-scripts.html

browser vendors did not support it, it requires web application modifications and its focus is very limited. Along this trend, a security layer called *Content Security Policy* (*CSP*) was added to the latest version of Firefox to detect various types of attacks, including cross-site scripting.[10] To eliminate such attacks, web site administrators can specify which domains the browser should treat as valid sources of script and which not. Then, the browser will only execute scripts that exist in source files from white-listed domains. Still, this layer has limitations. Embedded JavaScript code is unsupported and developers are not allowed to use the `eval` function, the `Function` constructor and others, since CSP disables all features that allow code in strings to be executed. Therefore, existing applications that make use of such constructs have to be rewritten. Finally, to the best of our knowledge, CSP has not been extensively validated yet and consequently there is no indication of its effectiveness.

Apart from JavaScript injection attacks, policy enforcement has been also used to detect binary code injection attacks. Specifically, Kiriansky et al. [127] have proposed *program shepherding* which monitors control flow transfers in order to restrict execution privileges based on code origins and ensure that program sandboxing will not be breached. In a similar manner, Control-Flow Integrity (CFI) [2], follows a predetermined control-flow graph that serves as a specification of control transfers allowed in the program. Then, at runtime, specific checks enforce this specification.

2.2.4 Training

Training techniques are based on the ideas of Denning's original intrusion detection framework [64]. In the CIA context, a training mechanism, registers all valid legitimate code statements during a training phase. This can be done in various ways according to the implementation. Then, only those will be accepted, approved or recognized during production.

JS injection training approaches record and store valid JavaScript code statements in various forms, and thereby detect attacks as outliers from the set of valid statements. SWAP [236] encodes all the legitimate scripts that exist in the original application. Then, a JavaScript detection component placed in a web proxy searches for injected scripts in the server's responses. If no injected scripts are found, the proxy decodes the legitimate scripts and sends them to the client. This approach is relatively inflexible since it does not support dynamic scripts. Similar limitations exist in XSS-GUARD [30], which maps legitimate scripts to HTTP responses. To support dynamic scripts during the creation of the legitimate identifiers the authors of XSSDS [114] substitute string-tokens with specified identifiers. Still, XSSDS suffers from a false positive alarms.

In the case of DSL-driven injection attacks, methods record and store valid DSL code statements and thereby detect DSL-driven injection attacks as outliers from the set of valid statements. An early approach, DIDAFIT [133] detects SQL injection attacks by recording all database transactions. Subsequent refinements by Valeur et al. [218] tagged each transaction with the corresponding application as an extension of their anomaly detection framework called *libAnomaly*[11]. Furthermore, AMNESIA [97, 98], a tool that also detects SQL injection attacks, associates a query model with the location of each query in the application and then monitors the application to detect when queries diverge from the expected model during runtime. SQLG*uard* [40] is another mechanism that detects SQL injection attacks based on parse tree validation. In particular, the mechanism compares at run time, the parse tree of the query before inclusion of user input with that resulting after inclusion of input. If the trees diverge, the application is probably under attack. SM*ask* [113] identifies malicious code by automatically separating user input from legitimate code. This is done by introducing specific syntactic constructs that handle server-side languages used for data management separately. To achieve this, the mechanism uses a pre-processor and a post-processor to mask and unmask legitimate code. In essence, for every string constant the pre-processor enforces a syntactic separation between code and data by masking certain

[10] https://developer.mozilla.org/en/Introducing_Content_Security_Policy
[11] http://seclab.cs.ucsb.edu/academic/projects/projects/libanomaly/

parts of the string. After the processing of an HTTP request, the post-processor detects and neutralizes malicious code.

Laranjeiro et al. [13, 131, 132] have proposed a similar mechanism to detect both SQL and XPath injection attacks in Web services. When it is not possible to run a complete learning phase, a set of heuristics is used by the mechanism to accept or discard doubtful cases. Finally, Mattos et al. [143] developed an signature-based attack detection engine that utilizes ontologies to counter XML and XPath injection attacks. By using ontologies to model data provides explicit and formal semantic relationships between data and possible attacks.

2.3 Analysis

Tables 2.1 and 2.2 illustrate the advantages and the disadvantages of the various CIA countermeasures. In particular, we compare all mechanisms based on specific non-functional requirements (see Subsection 1.3).

Flexibility Flexibility indicates if an approach can be adjusted in order to detect different CIA categories. Typically, all approaches, except for *lexical analysis* have been used to detect various CIA defects. As we described earlier, *lexical analysis* is a simplistic approach that cannot be used to identify source code-driven injection attacks. Even if a corresponding tool existed, the false alarms would be far too many. This is because source code-driven injection attacks are language independent and *lexical analysis* can only search for specific keywords or sequences of keywords. As a result, it is only used to detect code constructs that can lead to binary code injection attacks.

Ease of Use A security mechanism should be easy to use. In the *static analysis* context, a mechanism should require minimum effort from the security auditor. Observe, that *lexical analysis* and *data-flow analysis* mechanisms are easy to use since the only thing that is needed to perform their analysis is the source code. *Model checking* and *type system extensions* on the other hand, require a lot of effort from the side of the auditor, either to write specifications, modify source code or learn new constructs (see Subsections 2.1.4 and 2.1.6).

In the case of *dynamic detection*, *ease of use* involves the deployment of the mechanism. To determine the effort and infrastructure required to use the mechanism, we examined the author's description of the mechanism and its current implementation. One of our basic criteria was if developers are required to modify their code and if they do, to what extent. As an example, consider the mechanisms coming from the *policy enforcement* category. In most cases developers should modify multiple application components to enable each mechanism. Note also that mechanisms like *BrowserShield*, MET and BEEP require modifications both on the server and the client side. Thus, it would be difficult for them to be adopted by browser vendors because of the required modifications on the web user's browser. In the same manner SQLrand imposes a major infrastructure overhead because it requires the integration of a proxy for the database system to detect SQL injection attacks. In addition, the *training* mechanisms that detect DSL-driven injection attacks, require multiple source code modifications. In particular, to use AMNESIA, developers should modify every code fragment that involves the execution of a query.

False Positives and Negatives The accuracy of security mechanisms can be judged by the existence of incorrect data, namely: False Positives (FP) and False Negatives (FN) (also known in statistics as *type* I and *type* II errors [172]). Specifically, a FP is a result that indicates that an attack is taking place, when it actually has not. A FN occurs when an attack actually takes place, and the mechanism fails to detect it. For every mechanism we list its FP and FN rates as registered in the corresponding publication. If the publication mentions that the mechanism actually produces false positives or negatives but it does not explicitly states any rates we use an X mark (✘). If it is accurate, we use a tick mark (✔). If the mechanism was not tested in terms of accuracy at all, we add a question mark (**?**).

Note that there are mechanisms that even if they seem accurate, their testing might be really poor contrary to other schemes that may have false alarms, but have been tested thoroughly. For example,

Blueprint appears to be an effective solution to detect JavaScript injection attacks, but the corresponding publication includes only two test cases. On the other hand, DSI appears to have false positives and negatives, but it was evaluated on a dataset of 5.318 web sites with known vulnerabilities.

An interesting observation involves the mechanisms that detect JavaScript injection attacks. Unfortunately, most countermeasures, even if the corresponding publications state that they are accurate, are actually vulnerable to a novel way of attacking (except for [15]). This technique involves non-HTML elements [23] (see the PostScript file example in Subsection 1.1.2). Since most mechanisms require the presence of a Document Object Model (DOM) tree to detect an attack, in this case they will fail.

Implementation Independence In the case of *static analysis* mechanisms, implementation independence indicates if a mechanism is developed based upon a specific programming language. For instance, all *lexical analysis* tools except for RATS, only analyze applications written in the C programming language. Still, RATS, which can be used on other languages, doesn't find code injection vulnerabilities in any other language except for C. In the same manner, *type system extension*, SQLJ can only be used by Java developers.

In the *dynamic detection* context, *implementation independence* shows if the mechanism depends either on the characteristics of the programming language that was used to develop it or on the implementation details of the protecting entity. For instance, CSP depends on the features provided by the C++ programming language and it can only protect web users that use *Mozilla Firefox* as their browser. For every implementation dependent mechanism, we list the corresponding language. Also, PHP Aspis can detect various forms of CIAs that target applications written in PHP only.

User Experience The user's experience is affected if a mechanism suffers from runtime overhead. Take for example a mechanism from the *dynamic detection* category. If this mechanism adds significant overhead to the applications functionality, the application's owner would consider it useless. In the *static analysis* context, this can be measured in the case of the *type system extension approach* since their use affects the application overall. In the table we list the overhead for every mechanism as stated in the original publication. If the publication mentions that the mechanism suffers from a runtime overheard but does not explicitly state the occurring overhead we use the X mark (✘). If the authors did not measure the overhead we use a question mark (**?**).

Discussion Approaches can be interdependent. An approach can borrow heavily from another. For instance, *runtime tainting* is actually based on *data-flow analysis*, while *model checking* and *symbolic execution* share the state explosion issue. Also, we observe that approaches from the *static analysis* category, are more formal than the ones that belong to the *dynamic detection* category. *Training*, ISR and *policy enforcement* are more conceptual and practical approaches rather than formal. In their background though can hide different schemes. Take for instance the mechanisms coming from the *training* category that use parse-tree validation, signatures and others. These are sub-approaches which is the main reason why they are language independent. For example, the parse tree validation scheme can be implemented for different concepts and with different programming languages.

Interestingly, even if there are so many countermeasures that detect CIA vulnerabilities, few of them are used by other researchers. One of the few mechanisms that have been extensively used to produce results in other publications is FindBugs.

Furthermore, there are publications that question the effectiveness of some mechanisms. For instance, Sovarel et al. [203] have investigated thoroughly the effectiveness of ISR and showed that an attacker may be able to circumvent ISR by determining the randomization key and their results indicate that doing ISR in a way that provides a certain degree of security against a motivated attacker is more difficult than previously thought. In the same manner, Zitser et al. [244] and Wilander et al. [1], have extensively tested and questioned some of the aforementioned tools that detect binary code injection attacks.

An interesting observation is that huge amount of work was done so far to prevent SQL and JavaScript injection attacks. This is reasonable since these two categories have been topping the vulnerability

lists of numerous bulletin providers for several years (see Subsection 1.3). Note though, that there are no type system extension mechanisms developed to detect dynamic language-driven CIAs. In all cases, the mechanisms are used to detect DSL-driven CIAs. Still, approaches that can directly protect applications from XPath injection attacks are yet to be proposed. Note that the mechanism proposed by Laranjeiro et al. [13, 131, 132] protects only web services from XPath injection attacks. In the same manner, there is only one mechanisms designed to detect PHP injection attacks, PHP Aspis.

In this dissertation we propose a novel and generic approach of preventing code injection attacks. During the design of the approach we took into account all the aforementioned requirements. Our approach can be seen as an improvement of the *training approach*. To specify if an application is under attack we use a blend of features that is unique for every vulnerable code statement. The key property that differentiates our scheme is that these features do not depend entirely on the code statement, but also take into account elements from its execution context. At the end of the training phase, a model of all legitimate statements is produced. This entails almost zero false positive and false negative rates making our approach robust and effective: at runtime, our scheme checks all code statements for compliance with this model and can thus block the statements that contain additional injected elements. Another distinct advantage of our approach is that it can be easily retrofitted to any system and it does not depend on the entity that is protected. We have implemented our approach to protect server-side applications from SQL and XPath injection attacks, and web users from JavaScript injection attacks on the client-side.

Approach	Flexibility[1]	Mechanism	Requirements				Attack Vector
			Ease Of Use[2]	FP,FN	Implementation Independence[3]	Computational Overhead[4]	
Lexical Analysis	✘	ITS4 [220, 221]	✔	52%,9%	✘(C)	¬	*binary code*
		PScan [101]	✔	✘,✘	✘(C)	¬	*binary code*
		Flawfinder [1]	✔	71%,4%	✘(C)	¬	*binary code*
		RATS [50]	✔	67%,17%	✔	¬	*binary code*
		BOON [225]	✔	31%,73%	✘(C)	¬	*binary code*
Data-Flow Analysis	✔	*CodeSurfer* [9, 162]	✔	20%,?	✔	¬	*binary code*
		Splint [78]	✔	19%,70%	✘(C)	¬	*binary code*
		Livshits et al. [136]	✔	✘,?	✘(Java)	¬	SQL, *JavaScript*
		FindBugs [19, 107, 204]	✔	36%,✘	✘(Java)	¬	SQL, *JavaScript*
		Pixy [118]	✔	50%,?	✔	¬	SQL, *JavaScript*
		XSSdetect	✔	✘,?	✔	¬	*JavaScript*
Model Checking	✔	QED [140]	✘	✔,✘	✘(Java)	¬	SQL, *JavaScript*
Symbolic Execution	✔	SAFELI [87]	✘	✘,✘	✔	¬	SQL
		Kudzu [190]	✔	✔,✘	✔	¬	*JavaScript*
Type System Extensions	✔	*SQL* DOM [144]	✘	?,?	✔	20%	SQL
		Safe Query Objects [57]	✘	?,?	✘(Java)	?	SQL
		SQLJ [72]	✘	✔,?	✘(Java)	?	SQL
		SugarJ [75, 76]	✘	?,?	✔	?	SQL, XML
		Wasserman et al. [231]	✘	20.8%,?	✔	✘	SQL
		*Web*SSARI [237]	✘	✔,?	✘(PHP)	98.4%	SQL

[1] *Flexibility* indicates if the approach can be adjusted in order to detect different CIA categories.
[2] In order to describe a static analysis mechanism as "*easy to use*", the mechanism should require minimum effort from the security auditor.
[3] *Implementation Independence* shows if the static analysis mechanism is tailored to a specific programming language.
[4] *User Experience* is affected if the mechanism adds runtime overhead to the application. In the context of *static analysis* this can be measured in the case of the *type system extension approach*.

Table 2.1: *Static Analysis*: Comparison summary of tools designed to detect CIA vulnerabilities.

Approach	Flexibility[1]	Mechanism	Requirements				Attack Vector
			Ease Of Use[2]	FP,FN	Implementation Independence[3]	Computational Overhead[4]	
Runtime Tainting	✔	*SigFree* [228]	✔	✘,✘	✘(C)	10%	*binary code*
		LIFT [178]	✔	✔,✔	✔	6.2%	*binary code*
		Haldar et al. [94]	✘	?,?	✘(Java)	✘	SQL
		SecuriFly [141]	✔	✘,✔	✘(Java)	9–125%	SQL, *JavaScript*
		Xu et al. [238]	✘	✔,✘	✔	76%	SQL, *JavaScript*
		WASC [163]	✔	✘,✘	✔	30%	SQL, *JavaScript*
		PHP *Aspis* [170]	✔	✘,✘	✘(PHP)	2.2×	SQL, *JavaScript*, PHP
		Vogt et al. [222]	✔	✘,?	✔	?	*JavaScript*
ISR	✔	*SQLrand* [34]	✔	?,?	✔	6.5*ms*	SQL
		Noncespaces [93]	✘	✔,✔	✔	10.3%	*JavaScript*
		xJS [15]	✔	✔,✔	✔	1.6–40*ms*	*JavaScript*
Policy Enforcement	✔	DSI [160]	✔	✘,✘	✔	1.85%	*JavaScript*
		BrowserShield [184]	✘	✔,✔	✔	8%	*JavaScript*
		Blueprint [137]	✘	✔,✔	✔	13.6%	*JavaScript*
		CoreScript [240]	✔	?,?	✔	?	*JavaScript*
		MET [77]	✘	?,?	✔	?	*JavaScript*
		BEEP [112]	✘	✔,✔	✔	13.6%	*JavaScript*
		CSP	✘	?,?	✘(C++)	?	*JavaScript*
		Google Caja	✔	?,?	✔	?	*JavaScript*
		Kiriansky et al. [127]	✔	?,?	✘	∼1%	*binary code*
		CFI [2]	✔	?,?	✘	0.09–26.78%	*binary code*
Training	✔	AMNESIA [96, 97, 98]	✘	✔,✔	✔	?	SQL
		DIDAFIT [133]	✘	✘,✘	✔	?	SQL
		Valeur et al. [218]	✔	0.37%,?	✔	1*ms*	SQL
		SQLGuard [40]	✔	?,?	✔	3%	SQL
		Laranjeiro et al. [13, 131, 132]	✘	✘,✘	✔	✘	SQL, XPath
		Mattos et al. [143]	✘	✔,?	✔	?	XML, XPath
		SMask [113]	✘	✘,✘	✔	?	SQL, *JavaScript*
		SWAP [236]	✔	✔,✔	✔	∼180%	*JavaScript*
		XSSDS [114]	✔	✘,✘	✔	?	*JavaScript*
		XSS-GUARD [30]	✔	?,✘	✔	5—24%	*JavaScript*

[1] *Flexibility* indicates if the approach can be adjusted in order to detect different CIA categories.
[2] In the context of *dynamic detection* a mechanism should be easily deployed.
[3] *Implementation Independence* shows if the mechanism depends either on the characteristics of the programming language that was used to develop it or on the implementation details of the protecting entity.
[4] *User Experience* is affected if the mechanism suffers from runtime overhead.

Table 2.2: *Dynamic Detection*: Comparison summary of mechanisms developed to counter CIAs.

Chapter 3

Design

Following our classification in Section 1.1, we present an approach that protects against executable source code injection attacks. First, we examine the behaviour of security bugs that involve input-validation issues and can lead to CIAs [191]. Our main goal was to understand the fundamental nature of such vulnerabilities in order to design a stable approach. Our findings (presented in the upcoming subsection) indicate that the suggestion of a novel approach to contain CIAs is still prominent. Then, we analyze the three basic entities involved in the code injection problem by using a functional programming language. Based on this background and with respect to the requirements presented in Section 2.3 we present an *implementation independent* and *flexible* approach to protect entities against CIAs that utilize SQL, XPath and JavaScript as attack vectors.

3.1 The Evolution of Security Bugs

Studying security bugs has already been an area of academic research. Ozment and Schechter [169] examined the code base of the OpenBSD Operating System (OS) to determine whether its security is increasing over time. In particular, they measured the rate at which new code has been introduced and the rate at which defects have been reported over a 7.5 year period and fifteen releases. Even though the authors present statistical evidence of a decrease in the rate at which vulnerabilities are being reported, defects seem to appear persistent for a period of at least 2.6 years. Massacci et al. [142] observed the evolution of software defects by examining six major versions of Firefox. To achieve this they created a database schema that contained information coming from the Mozilla Firefox-related Security Advisories (MFSA) list,[1] Bugzilla entries and others. Their findings indicated that security bugs are persistent over time. They also showed that there are many web users that use old versions of Firefox, meaning that old attacks will continue to work. Zaman et al. [242] focused again on Firefox to study the relation of security bugs with performance bugs. This was also done by analysing the project's Bugzilla. Their research presented evidence that security bugs require more experienced developers to be fixed. In addition, they suggested that security bug fixes are more complex than the fixes of performance and other bugs. Shahzad et al. [197] analysed large sets of vulnerability data-sets to observe various features of the vulnerabilities that they considered critical. Such features were the functionality and the criticality of the defects. Their analysis included the observation of vulnerability disclosures, the behavior of hackers in releasing exploits for vulnerabilities, patching and others. In their findings they highlighted the most exploited defects and showed that the percentage of remotely exploitable vulnerabilities has gradually increased over the years. Finally, Edwards et al. [70] have recently conducted a study similar to ours in which they have considered only four projects. Their results demonstrate that the number of exploitable bugs does not always improve with each new release

[1] http://www.mozilla.org/projects/security/known-vulnerabilities.html

and that the rate of discovery of exploitable bugs begins to drop three to five years after the initial release.

For our research, we further analyzed the results of the experiment presented in Subsection 1.2 and observed the evolution and the behaviour of software bugs that can actually lead to a code injection attack. To achieve this, we kept the categorization that we introduced in Subsection 1.2 that included the *Security High* bug categories that involves the bugs that interest us since they lead to CIAs. Contrary to the previous research presented in the above paragraph, our analysis involved thousands of projects. This gave us the opportunity to derive more robust conclusions on the subject. In our research we:

- Analyzed how such bugs evolve over time.
- Examined their persistence across software releases.
- Studied the relation between such bugs and a project release's size.
- Examined their correlation with other bug categories.

The observation of the above characteristics provided a stable background to develop our approach and implement corresponding prototypes.

Category	Spearman Correlation	p-value
Security High	0.08	$\ll 0.05$
Security Low	0.02	$\ll 0.05$
Style	0.03	$\ll 0.05$
Correctness	0.04	$\ll 0.05$
Bad Practice	0.03	$\ll 0.05$
MT Correctness	0.09	$\ll 0.05$
i18n	0.06	$\ll 0.05$
Performance	*(0.01)*	0.07
Experimental	0.09	$\ll 0.05$

Table 3.1: Correlations between version and defects count.

How Security Bugs Evolve Over Time The relation between bugs and time can be traced from the number of bugs per category in each project version. We can then calculate the Spearman correlations between the defects count and the ordinal version number across all projects to see if bigger versions relate to higher or lower defect counts. The results are shown in Table 3.1. Although the tendency is for defect counts to increase, this tendency is extremely slight.

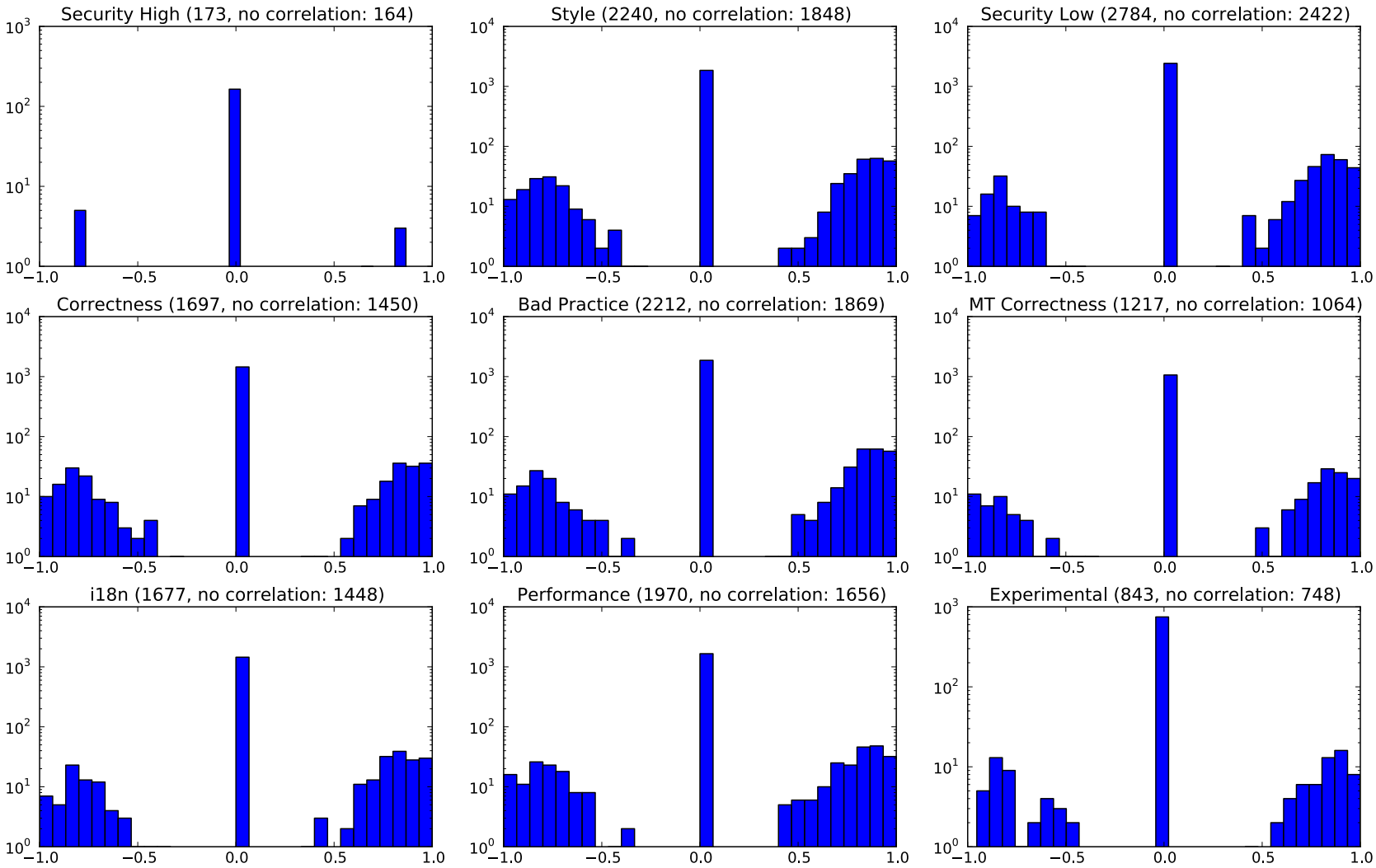


Figure 3.1: Histograms of correlations between bug counts and version ordinals per project. In brackets the total population size and the number of no correlation instances.

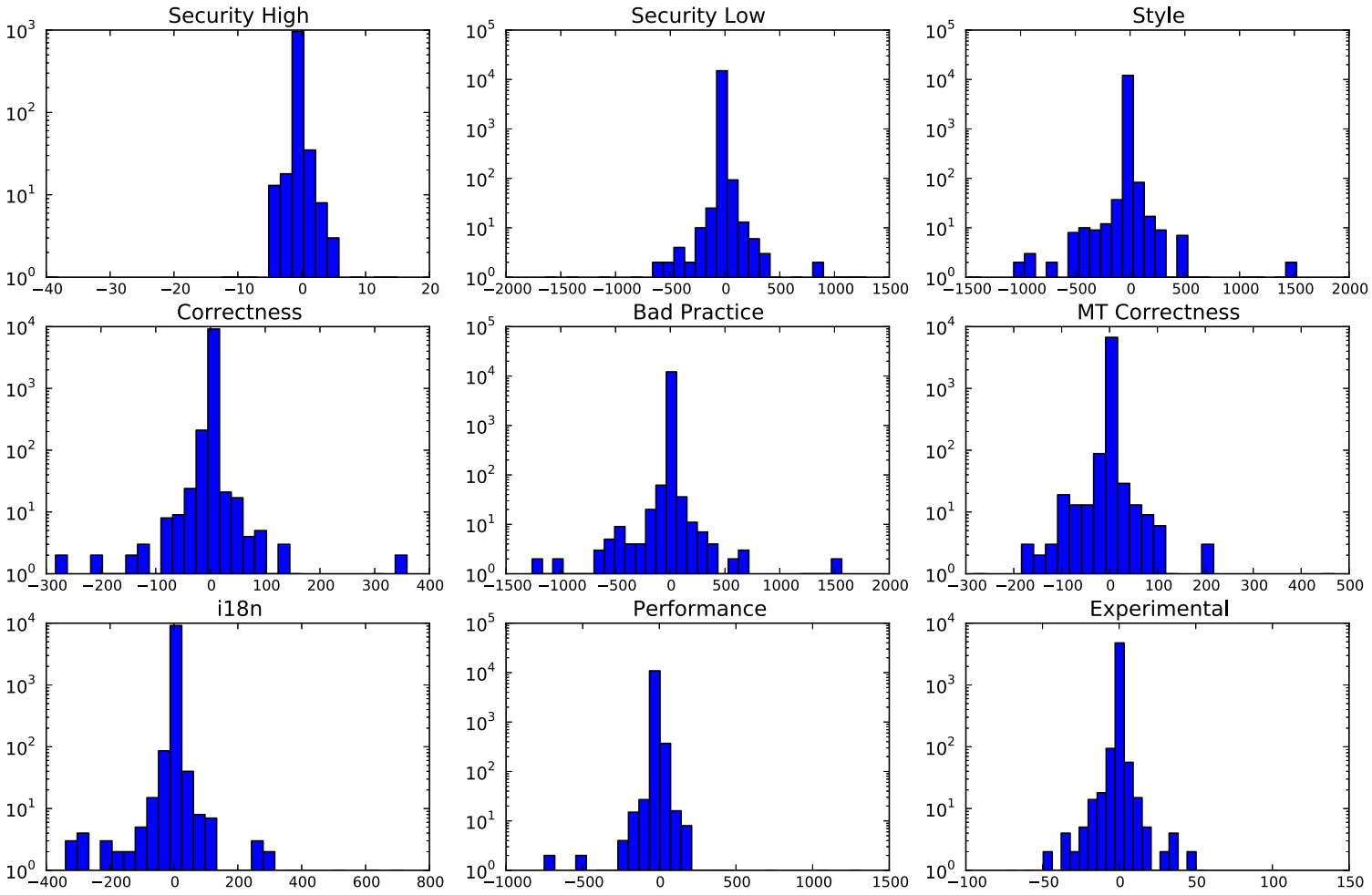


Figure 3.2: Changes in bug counts between versions.

The zero tendency applies to all versions of all projects together. The situation might be different in individual projects. We therefore performed Spearman correlations between bug counts and version ordinals in all projects we examined. These paint a different picture from the above table, shown in Figure 3.1. The spike in point zero is explained by the large number of projects for which no correlation could be established—note that the scale is logarithmic. Still, we can see that there were projects where a correlation could be established, either positive or negative. The *Security High* category is particularly bimodal, but this is explained by the small number of correlations that could be established, nine in total.

Overall, Table 3.1 and Figure 3.1 suggest that *we cannot say that across projects defect counts increase or decrease significantly across time*. In individual projects, however, defect counts can have a strong upwards or downwards tendency. There may be no such thing as a "project" in general, only particular projects with their own idiosyncrasies, quality features, and coding practices. Another take on this theme is shown in Figure 3.2, which presents a histogram of the changes of different bug counts in project versions. In most cases, a bug count does not change between versions; but when it does change, it may change upwards or downwards. Note also the spectacular changes of introducing or removing thousands of defects; this may be the result of doing and undoing a pervasive code change that runs foul of some bug identification rule.

Persistence of Security Bugs To examine the relation between the persistence of different kinds of bugs, and of security bugs in particular, we used as a persistence indicator the number of versions a bug remains open in a project. To "tag" a bug we created a bug identifier by using the type of the bug, the method name and the class name in which the bug was found in. We chose not to use the line number of the location of the bug since it could change from version to version and after a possible code refactoring. We grouped the persistence numbers by bug categories and then performed a Mann-Whitney U [103] test among all bug category pairs. The results are presented in Table 3.4. Cells in brackets show pairs where no statistically significant difference was found. In general, although the average number of versions bugs in different bug categories that remained open was statistically different in many cases, the difference is not spectacular. *In all cases a bug persists on average between two and three versions*, with the difference being in the decimal digits.

The Relation of Defects with the Size of a JAR We explored the relation between defects with the size of a project version, measured by the size of its JAR file by carrying out correlation tests between the size and the defect counts for each project and version. The results, all statistically significant ($p \ll 0.05$) can be seen in Table 3.2. *The Security High category stands out by having a remarkably lower effect than the other categories*, even *Security Low* that nearly tops the list. As we mentioned earlier, bugs that belong to the *Security High* category are related to user-input validation issues. Hence, even if a programmer adds code to a new version, if this code does not require user input, the possibility of such bug is minimal. Given the above finding, we could also say that it would be productive to search for and fix security bugs even if a project grows bigger.

Security Bugs VS Other Bug Categories To see whether bugs flock together we performed pair-wise correlations between all bug categories. We calculated the correlations between the number of distinct bugs that appeared in a project throughout its lifetime, see Figure 3.3 and Table 3.3. We found significant, but not always strong, correlations between all pairs. In general, the *Security High* category showed the weakest correlations with the other categories. Our results show that in general *bugs do flock together*. We do not find projects with only a certain kind of bug; bugs come upon projects in swarms of different kinds. Bugs of the *Security High* category, though, are different: they are either not associated with other bugs, or only weakly so. Perhaps it takes a special kind of blunder to make it a security hazard. Thus, to find such defects, code reviewers with experience in software security issues might be needed.

Discussion Compared to other bug categories, *Security High* bugs have two distinct features: they can severely affect an organization's infrastructure [196], and they can cause significant financial dam-

Category	Spearman Correlation	p-value
Security High	0.19	$\ll 0.05$
Security Low	0.65	$\ll 0.05$
Style	0.68	$\ll 0.05$
Correctness	0.51	$\ll 0.05$
Bad Practice	0.67	$\ll 0.05$
MT Correctness	0.51	$\ll 0.05$
i18n	0.53	$\ll 0.05$
Performance	0.63	$\ll 0.05$
Experimental	0.36	$\ll 0.05$

Table 3.2: Correlations between JAR size and defects count.

age to an organization [21, 213]. Specifically, whereas a software bug can cause a software artifact to fail, a security bug can allow a malicious user to alter the execution of the entire application for his or her own gain. In this case, such bugs could give rise to a wide range of security and privacy issues, like the access of sensitive information, the destruction or modification of data, and denial of service. Moreover, security bug disclosures lead to a negative and significant change in market value for a software vendor [212]. Hence, one of the basic pursuits in every new software release should be to mitigate such bugs. Contrary to this, we found that, although bugs do close over time in particular projects, we do not have an indication that across projects they decrease as projects mature. This indicates that critical programming errors that can lead to code injection attacks are not fixed in a constant manner, an issue that has been also cited by other researchers [70, 169]. Keep in mind that the versions examined in this experiment were actual releases of the software artifacts. In a complementary experiment that we performed (see Appendix A) we examined multiple revisions of four open source projects (a revision is the new state that a Version Control System (VCS) system goes to, for every new contribution). In this particular experiment we found that security bugs were actually increasing as these projects evolve.

In addition, we showed that critical security bugs are not eliminated in a way that is particularly different from the other bugs. Also, having an average of two to three versions persistence in a sample where 60% of the projects have three versions, is not a positive result especially in the case of the *Security High* bugs. Concerning the relation between severe security bugs and a project's size we showed that they are not proportionally related. Furthermore, the pairwise correlations between all categories indicated that even though all the other categories are related, severe bugs do not appear together with the other bugs. Note though that by selecting an large ecosystem that includes applications written only in Java, we excluded by default measurements that involve vulnerabilities like the infamous buffer overflow defects [124].

Appendixes C and B present two more experiments made within our research in order to observe security issues related to attacks like CIAs in other contexts. In Appendix B we explored the security risks associated with virtual images from the public catalogs of a cloud service provider. Surprisingly, we found that many of them were vulnerable to CIAs. In Appendix C we tried to measure the diversity that exists in today's software. Geer et al. [119] claim that cyber-diverse computer systems and applications could prove to be more resistant to potential attacks than systems that tend to monoculture, which is the exact opposite of diversity Our results showed that cyberdiversity exists but exists in minimal extent. Hence, we cannot say that are there are populations that are actually resistant to attacks let alone to code injection attacks. The above results indicate that CIA vulnerabilities exist in different contexts and have their own different behaviour.

Security High	1.00	0.21	0.22	0.25	0.18	0.34	0.21	0.31	0.33
Security Low	0.21	1.00	0.63	0.63	0.56	0.56	0.62	0.60	0.47
Style	0.22	0.63	1.00	0.68	0.62	0.63	0.66	0.58	0.46
Bad Practice	0.25	0.63	0.68	1.00	0.54	0.58	0.64	0.63	0.51
Correctness	0.18	0.56	0.62	0.54	1.00	0.56	0.55	0.52	0.52
MT Correctness	0.34	0.56	0.63	0.58	0.56	1.00	0.56	0.56	0.49
Performance	0.21	0.62	0.66	0.64	0.55	0.56	1.00	0.60	0.52
i18n	0.31	0.60	0.58	0.63	0.52	0.56	0.60	1.00	0.60
Experimental	0.33	0.47	0.46	0.51	0.52	0.49	0.52	0.60	1.00

Table 3.3: Pairwise correlations between different bug categories.

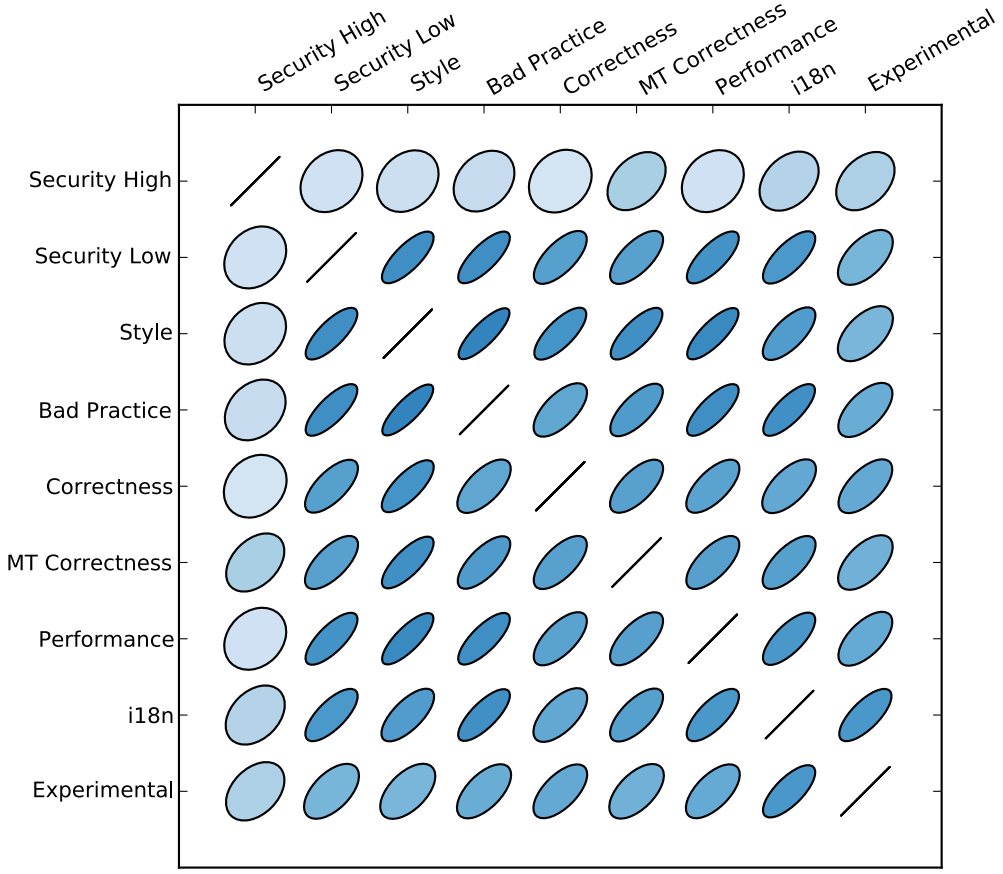


Figure 3.3: Correlation matrix plot for bug categories.

Table 3.4: Bug persistence comparison.

Security High	*(0.04, $p = 0.97$ 2.72, 2.36 243, 35048)*	2.22, $p < 0.05$ 2.72, 2.12 243, 49043	*(−0.51, $p = 0.61$ 2.72, 2.50 243, 12905)*	2.77, $p < 0.01$ 2.72, 2.11 243, 49324	*(1.02, $p = 0.31$ 2.72, 2.48 243, 10227)*	*(−1.19, $p = 0.23$ 2.72, 2.74 243, 10718)*	*(−1.00, $p = 0.32$ 2.72, 2.65 243, 23598)*	*(−0.33, $p = 0.74$ 2.72, 2.85 243, 2686)*
Security Low		20.27, $p \ll 0.05$ 2.36, 2.12 35048, 49043	−3.59, $p \ll 0.05$ 2.36, 2.50 35048, 12905	25.17, $p \ll 0.05$ 2.36, 2.11 35048, 49324	5.59, $p \ll 0.05$ 2.36, 2.48 35048, 10227	−7.55, $p \ll 0.05$ 2.36, 2.74 35048, 10718	−8.19, $p \ll 0.05$ 2.36, 2.65 35048, 23598	*(−1.39, $p = 0.17$ 2.36, 2.85 35048, 2686)*
Style			−17.96, $p \ll 0.05$ 2.12, 2.50 49043, 12905	5.66, $p \ll 0.05$ 2.12, 2.11 49043, 49324	−6.84, $p \ll 0.05$ 2.12, 2.48 49043, 10227	−20.61, $p \ll 0.05$ 2.12, 2.74 49043, 10718	−26.18, $p \ll 0.05$ 2.12, 2.65 49043, 23598	−8.30, $p \ll 0.05$ 2.12, 2.85 49043, 2686
Correctness				21.38, $p \ll 0.05$ 2.50, 2.11 12905, 49324	7.44, $p \ll 0.05$ 2.50, 2.48 12905, 10227	−3.57, $p \ll 0.05$ 2.50, 2.74 12905, 10718	−2.91, $p < 0.01$ 2.50, 2.65 12905, 23598	*(0.40, $p = 0.69$ 2.50, 2.85 12905, 2686)*
Bad Practice					−10.02, $p \ll 0.05$ 2.11, 2.48 49324, 10227	−23.63, $p \ll 0.05$ 2.11, 2.74 49324, 10718	−30.32, $p \ll 0.05$ 2.11, 2.65 49324, 23598	−9.98, $p \ll 0.05$ 2.11, 2.85 49324, 2686
MT Correctness						−10.17, $p \ll 0.05$ 2.48, 2.74 10227, 10718	−10.83, $p \ll 0.05$ 2.48, 2.65 10227, 23598	−4.03, $p \ll 0.05$ 2.48, 2.85 10227, 2686
i18n							*(1.29, $p = 0.20$ 2.74, 2.65 10718, 23598)*	2.46, $p < 0.05$ 2.74, 2.85 10718, 2686
Performance								*(1.92, $p = 0.05$ 2.65, 2.85 23598, 2686)*

Security Low
Style
Correctness
Bad Practice
MT Correctness
i18n
Performance
Experimental

The matrix presents pairwise Mann-Whitney U test results between the different bug categories. Each cell contains the test result (the value of U), the p-value, the average for each category and the sample size for each category. Cells in brackets show pairs where no statistically significant difference was found.

3.2 Theory: The Basic Entities of the Code Injection Problem

To adduce the essence of our method, we have to identify the three basic entities of the code injection problem (see Figure 3.4). The *application* is the entity that contains the software vulnerability that actually leads to a code injection attack in the way we presented in Section 1.1. Typically, it is software that causes a device (a computer, a mobile phone etc.) to perform useful tasks and helps users to perform specific activities. *Data* involve the input to be entered into a computer for processing. Actually, the malicious code will be incorporated in the user-input data. The *runtime environment* accepts code as input and causes it to be executed. Specifically, it is software designed to support the execution of computer programs written in some programming language. This runtime system contains implementations of basic low-level commands and may also implement higher-level commands and may support type checking, debugging, and even code generation and optimization. It is the entity that will execute the injected code too.

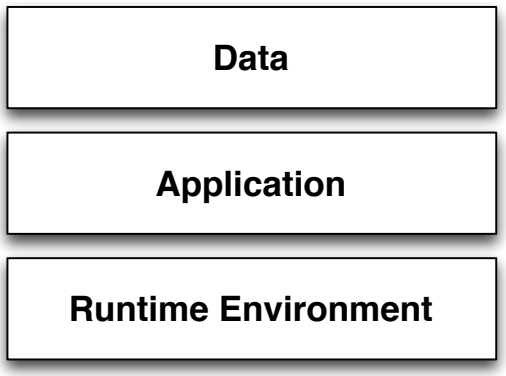


Figure 3.4: Basic entities involved in the code injection problem.

Figure 3.5, shows a simple *Scheme* interpreter for a tiny subset of the *Scheme* programming language [69]. The interpreter was built upon the standard textbook by Abelson et al. [3], has reasonable error handling and represents the runtime environment in our case. The following code represents a function, written also in the Scheme programming language, which has been given the name `vulnerable_square`. The function, represents the operation of multiplying something by itself. The thing to be multiplied is given a local name, `x`. Note that the interpreter is called explicitly to perform the multiplication within the function.

```
(define (vulnerable_square x)
    (interpret (* x x)))
```

Now consider the function below:

```
(define (endless_loop x)
    (cond ((> x 0) (endless_loop x))
        ((= x 0) (endless_loop x))
        ((< x 0) (endless_loop x))))
```

Regardless the input, this function will lead to an infinite loop. If this code is injected as an input argument to the `vulnerable_square` function, it will crash the interpreter and lead to a denial of service.

```
(define primitive-environment
       `((apply . ,apply) (assq . ,assq)
              (car . ,car) (cadr . ,cadr) (caddr . ,caddr)
              (cadddr . ,cadddr) (cddr . ,cddr) (cdr . ,cdr)
              (cons . ,cons) (eq? . ,eq?) (list . ,list) (map . ,map)
              (memv . ,memv) (null? . ,null?) (pair? . ,pair?)
              (read . ,read) (set-car! . ,set-car!)
              (set-cdr! . ,set-cdr!) (symbol? . ,symbol?)))


(define new-env
       (lambda (formals actuals env)
              (cond
                     ((null? formals) env)
                     ((symbol? formals) (cons (cons formals actuals) env))
                     (else
                        (cons
                           (cons (car formals) (car actuals))
                           (new-env(cdr formals) (cdr actuals) env))))))

(define lookup
       (lambda (var env)
              (cdr (assq var env))))


(define assign
       (lambda (var val env)
              (set-cdr! (assq var env) val)))

(define exec
       (lambda (expr env)
              (cond
                     ((symbol? expr) (lookup expr env))
                     ((pair? expr)
                      (case (car expr)
                             ((quote) (cadr expr))
                             ((lambda)
                              (lambda vals
                                 (let ((env (new-env (cadr expr) vals env)))
                                    (let loop ((exprs (cddr expr)))
                                       (if (null? (cdr exprs))
                                          (exec (car exprs) env)
                                          (begin
                                             (exec (car exprs) env)
                                             (loop (cdr exprs))))))))
                             ((if)
                              (if (exec (cadr expr) env)
                                  (exec (caddr expr) env)
                                  (exec (cadddr expr) env)))
                             ((set!) (assign (cadr expr) (exec (caddr expr) env) env))
                             (else
                                    (apply
                                       (exec (car expr) env)
                                       (map (lambda (x) (exec x env)) (cdr expr))))))
                     (else expr))))


(define interpret
              (lambda (expr)
                     (exec expr primitive-environment)))
```

Figure 3.5: A simple interpreter in the Scheme programming language.

An isomorphic situation involving an SQL injection attack is the following: Consider a three-tiered application [182] written in the Java programming language, with a *presentation* tier that receives user input which, in turn, is used by the *application* tier to create an SQL query. Then, the query is sent to the database of the *data* tier to be executed.

```
Class.forName ("com.microsoft.jdbc.sqlserver.SQLServerDriver");
String url = "jdbc:microsoft:" +
    + "sqlserver://localhost:1433;databasename = MyDB";
Connection conn =
    DriverManager.getConnection(url, "username", "password");
Statement stmt = con.createStatement();
String sql = "SELECT table1.field1 FROM table1 " +
    + "WHERE table1.field2 = " + user_input_1 +""
    + "AND table1.field3 > " + user_input_2 + ";
ResultSet rs = stmt.executeQuery(sql);
```

In the above code fragment, the application initially invokes a connectivity driver used to to connect to the Microsoft SQL Server. The main function of such a driver is to provide a portability layer by obtaining SQL statements from the application and forwarding them to the database. Then, the application concatenates strings that contain data coming from a web user (`user_input_1` and `user_input_2`) with a query that is sent directly to the database. If the input is not validated, then injected code can reach the SQL runtime environment in the same manner as the injected function written in Scheme can reach the interpreter (also see Subsection 1.1). In our research we developed a security layer that stands between the application entity and the runtime environment entity and it depends neither on the former, nor on the latter.

Ideally, the application in the example above should have been implemented in a secure manner and as a result prevent such malicious behaviours. For instance, most SQL injection attacks can be prevented by passing user data as parameters of previously prepared SQL statements instead of intermingling user data directly in the SQL statement. For example, the statement we examined previously, could be passed to the database with a question mark used as a placeholder for the parameter, and a separate type-checked API call could be used for setting the first parameter of the SQL statement to the desired value. For instance, recollect the "incorrect type handling" attack example, mentioned in Subsection 1.1.2. In this example, the URL for accessing the site's fifth press release was the following:

```
http://www.website.com/pressRelease.jsp?RelID=5
```

And the statement executed within the application's DBMS:

```
SELECT description, issuedate, body FROM pressRel WHERE RelID = 5
```

By changing the URL into something like this:

```
http://www.website.com/pressRelease.jsp?RelID=5%20AND%201=1
```

the attackers can manipulate the statement as they choose. To counter this kind of an attack in Java, programmers could utilize the `PreparedStatement` interface to set the first parameter of the statement to the desired value:

```
PreparedStatement pstmt = con.prepareStatement(
    "SELECT description, issuedate, body FROM pressRel" +
    "WHERE RelID = ?");
pstmt.setInt(1, 5);
```

In the case of C#, the code should look like this:

```
cmd = new MySqlCommand("SELECT description, issuedate, body
    FROM pressRel WHERE RelID=@value",MySqlConn.conn);
cmd.Parameters.AddWithValue("@value", SqlDbType.Int);
cmd.Prepare();
```

These practices can indeed increase the robustness of applications. However, experience has shown us that the expectation for them to be embraced to the extent of completely eliminating security vulnerabilities is just wishful thinking [156]. Interestingly, even when using such practices, there are possibilities of injection attacks through their inappropriate usage. The example below explains such a scenario where the input variables are passed directly into the prepared statement and thereby paving way for an SQL injection attack:

```
PreparedStatement pstmt = con.prepareStatement(
    "SELECT description, issuedate, body FROM pressRel" +
    "WHERE RelID = '+ user_input +'");
```

3.3 Approach

Consider the successor function written in the lambda calculus notation [3, 31, 130]:

$$SUCC := \lambda n.\lambda f.\lambda x.f(nfx) \tag{3.1}$$

This function takes a number `n`, and returns `n + 1` by adding an additional application of `f`. Since the `m-th` composition of `f`, composed with the `n-th` composition of `f` gives the `m+n-th` composition of `f`, addition can be defined as follows:

$$PLUS := \lambda m.\lambda n.\lambda f.\lambda x.mf(nfx) \tag{3.2}$$

The code above, typically expects natural numbers as arguments to perform the addition and return another natural number. The problem is that a user could provide as an argument a function instead of a number, and change the execution flow of the function (this is where the nature of CIA comes into play). This is because, lambda calculus, like Scheme, has *first-class functions* [3]. Nevertheless, if the application that used this function "*had been trained*" in some way to only expect numerals as arguments we would not have such a problem.

Algorithm 1 illustrates our proposed approach. As we mentioned earlier our approach involves a *security layer* accepts the request to forward code from the application to the runtime environment to be executed. The code is examined and if it contains injected elements the layer issues an alarm. The layer operates in two modes, *training* and *production*. During training, every vulnerable code statement is associated with a *signature*. This signature is a unique identifier that during production mode will determine if a CIA is taking place. Before generating and storing a signature our layer analyzes the code. Code analysis involves the complete removal of what is expected as user input, i.e., string literals and numbers, so that the signature is a template, and representative, of a class of user inputs, omitting only the actual input received—in which case it would be useless as a predictor. Then specific features related to the execution context are combined to create the signature identifier, which is registered as valid in an auxiliary table where all known valid signatures are stored. After the signature generation, the application's normal execution flow continues.

If the security layer is in production mode, the first two steps are the same with the training mode. The code is analyzed in the same manner and a signature is generated again. After the generation of the signature, the layer validates it by checking if it exists in the table of valid signatures. If it does not,

it means that an injection attack is taking place. The layer prevents the execution of the injected code and specific details regarding the invalid call are logged.

Algorithm 1 Algorithm of the Proposed Approach

```
1:  function SECUREFUNCTION(code, training_mode)
2:      strippedCode ← AnalyzeCode(code)                          // remove user input
3:      executionContextElements ← AnalyzeExecutionEnvironment()
4:                                          // get elements related to the execution context
5:      s ← GenerateSignature(strippedCode, executionContextElements)
6:      if training_mode = true then                              // We are in training mode
7:          RegisterSignature(s)
8:      else                                                      // We are in production mode
9:          if ValidateSignature(s) then                          // Signature is valid
10:             r ← ExecuteCode(code)                             // Execute the code
11:             return r
12:         else                                                  // Signature is not valid
13:             LogPossibleAttack(code)
14:             return
15:         end if
16:     end if
17: end function
```

A key element of our approach is the efficient generation of signatures. The legitimate signatures produced in the training mode are based on features that when combined provide a unique identifier. Some of these features depend entirely on the code statement that is about to be executed. These include SQL keywords in the case of an SQL statement, XML attributes in the case of an XML code fragment, etc. The above features do not represent a code statement uniquely. For instance, you can have two identical, legitimate SQL queries, coming from two different methods of an application. In this case, these statements have the same functionality (*"behaviour"*). Other features are independent of the code statements, but depend on the execution flow and environment: these include the caller method, its class name, the connection between the application and the database, the line number of the file that triggers an execution, and others. By combining such elements we can allow to have specific behaviours from specific places within an application.

The various features must be selected in such a way that every legitimate "behaviour" of a vulnerable code statement at execution time, is associated with one signature in an injective relation; literally, if C is the set of all legitimate code statement "behaviours" of an application, S is the set of the legitimate signatures and C and S are disjoint sets, the following expression must stand:

$$f : J \rightarrow S \text{ is an injection} \tag{3.3}$$

If it does, when a malicious user attempts an attack the injected code will lead to a signature that does not exist in the table and the attack will be intercepted. In the same manner, if the function presented in 3.2 had a signature that represents its behaviour, providing a function as argument as described earlier should lead to an alarm.

To achieve the above, the removal of actual user input from the signature generating function is required. As presented in Algorithm 1, this is done by the `AnalyzeCode` function. For this removal to take place the `AnalyzeCode` function follows the steps listed in Algorithm 2.

Signature creation also requires the retrieval of the elements that are related to the execution context. As illustrated in Algorithm 1 this is done by the `AnalyzeExecutionEnvironment` function. Depending on the attack, the elements extracted from this function may vary. Hence it is implemented differently depending on the type of the attack and the execution context. Based upon the above, a valid

Algorithm 2 User-Input Removal

```
1: function AnalyzeCode(code)
2:     stringsFreeCode ← removeQuotedStrings(code)          // remove quoted strings
3:     numbersFreeCode ← removeNumbers(stringsFreeCode)     // remove numbers given as input
4:     strippedCode ← removeComments(numbersFreeCode)       // remove comments
5:     return strippedCode
6: end function
```

signature is defined by the following, where + stands for string concatenation:

$$S = \texttt{AnalyzeCode(code)} + \texttt{AnalyzeExecutionEnvironment()} \tag{3.4}$$

To facilitate the handling of the signatures and ensure that they are not manipulated in any way, a hash function is applied to the combined elements before they are stored in the signature storage table.

3.4 Preventing DSL-driven CIAs

In this section, we demonstrate the validity of our approach for guarding against injection attacks on DSLs. Specifically, we show how it can guard applications against two of the most common DSL-driven injection attacks, namely: SQL and XPath. The general architecture behind both mechanisms is presented as a UML communication diagram in Figure 3.6. In each case, the signatures created during the training mode involve the stripped down statement (as presented in Algorithm 2) and a critical element of the execution context of the application, the method invocation stack trace. The stack trace is retrieved following Algorithm 3 (which is practically the implementation of the `AnalyzeExecutionEnvironment` function in the DSL context) and it includes the details of all methods and call location.

In practice, our scheme can be applied to all DSLs that are integrated into a General-Purpose Language (GPL) using the pattern *Implementation:Embedding* as proposed by Mernik et al [150]. This is because this pattern includes all DSLs that are using an application library as their implementation scheme.

Algorithm 3 Traversing the Call Stack

```
1: function GETSTACKTRACE(frame)
2:     while frame ≠ bottomFrame do     // check if we are in the bottom of the stack
3:         frame ← frame.caller         // obtain the caller frame
4:         s ← getFrameDetails(frame)   // retrieve class name, method name etc.
5:         stackTrace ← stackTrace + s  // append the details to form the stack trace
6:     end while
7:     return stackTrace
8: end function
```

3.4.1 SDriver/SQL: A Signature-Based Proxy Database Driver

The architecture of typical tiered web applications consists of at least an application running on a web server and a back-end database [230]. Between these two tiers, there is in most cases a database connectivity driver based on protocols like Open Database Connectivity (ODBC) or JDBC. The main function of such a driver is to provide a portability layer by obtaining SQL statements from the application and forwarding them to the database. Our proposed driver, which we call SD*river*/SQL, is also a connectivity driver that operates however as a shim or proxy standing between the application and the

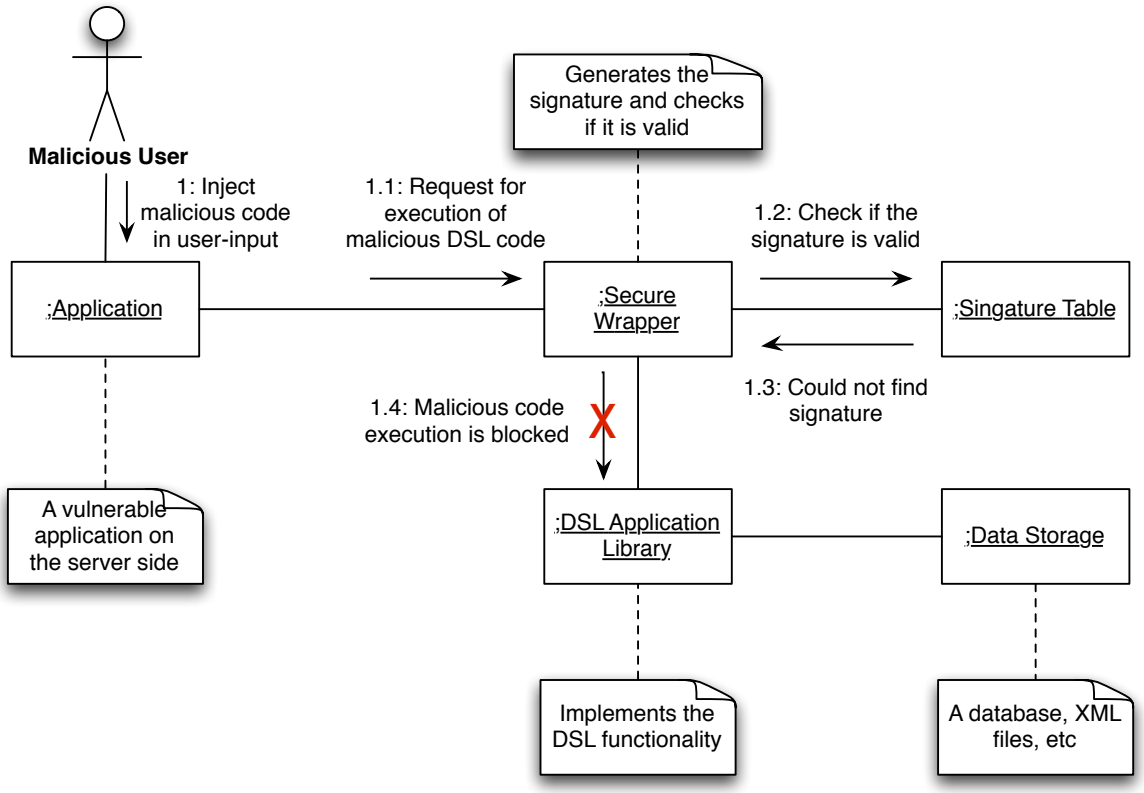


Figure 3.6: DSL-driven injection attack interception scenario.

database interface driver (see Figure 3.7). SD*river*/*SQL* is transparent: its only role is to prevent SQL injection attacks (see Section 1.1.2), and it depends neither on the application, nor on the underlying connectivity driver. To work as a connectivity driver, our driver implements the complete interface of the connectivity protocol. However, most of the driver's methods simply forward the request to the underlying connectivity driver. Only a few methods capture and process requests in order to prevent SQL injection attacks. In this respect our driver acts as a proxy for the underlying driver working as a firewall between the original driver and the application.
Following our approach, to secure the application from SQL injection attacks SD*river*/*SQL* must go through a training phase. This involves executing all the SQL queries of the application so that the driver can identify them in a way we will show below. Then, the driver's operation can shift into production mode, where the driver takes into account all the trained legitimate queries to prevent SQL injection attacks by detecting and blocking them.

Training Mode Every SQL query of an application is identified through a signature created by combining three of its characteristics.

1. The method invocation stack trace. This includes the details of all methods and call location, from the method of the application where the query is executed down to the target method of the connectivity driver.
2. The SQL keywords.
3. The tables and the fields that the query uses in order to retrieve its results.

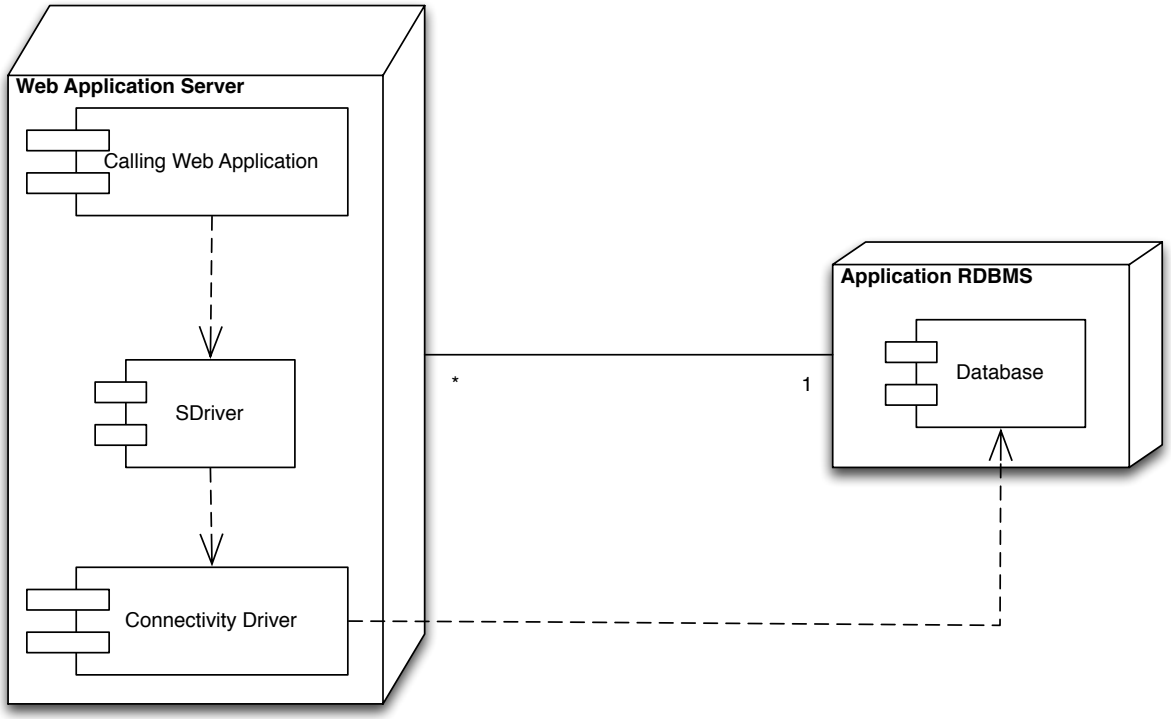


Figure 3.7: The architecture of SD*river*/SQL

A formal representation of the above is the following: If K is the set of the stack traces; L is the set of the SQL keywords and M the set of the application tables, the set of the query signatures S will be defined as follows:

$$S = \{\omega : \omega = k(l^* m^*)^+, k \in K, l \in L, m \in M\} \tag{3.5}$$

When the system operates in the training mode, each query signature Q is added to S. In production mode a query with a signature Q is considered legal *iff* $Q \in S$. Obviously, a query cannot be unambiguously identified by using one of the above characteristics alone. To combine these characteristics, when a query is being sent to the database SD*river*/SQL carries out two actions. First, it strips down the query, removing all numbers and string literals. So if the following statement is being executed

```
SELECT table1.field1 FROM table1 WHERE table1.field2 = 'foo' AND
  table1.field3 > 3
```

the driver removes 'foo' and 3 from the query string.

SD*river*/SQL also traverses down the call stack, saving the details of each method invocation, until it reaches the statement's origins. To achieve this, it follows the steps shown earlier in Algorithm 3. The association of stack frame data with each SQL query is an important defense against maliciously crafted attacks that try to masquerade as legitimate queries. As an example, consider an application that will send the password for a forgetful user, Alice, via email by executing

```
SELECT password from userdata WHERE id = 'Alice'
```

This same application could allow users to lock their terminal, but allow the unlocking either with the user's password or with the administrator password (the 4.3 BSD *lock* command behaved in this peculiar

way). The corresponding query to verify the password on the locked Alice's workstation would be as follows.

```
SELECT password from userdata WHERE id = 'Alice' OR id = 'admin'
```

It is now easy to see that a malicious user, Bob, could obtain the administrator's password by email by entering on the password retrieval form as his user identifier

```
nosuchuser' OR id = 'admin
```

Without the differentiating factor of a stack trace, the preceding query would have the same signature as the one used for unlocking the terminal, and would therefore escape a traditional signature-based SQL injection attack protection system.

Our initial design had SD*river*/*SQL* storing each query's keywords, table names, and stack trace into separate tables of an auxiliary database. During implementation we realized that, because the only operations we were interested in were adding a query Q to the set of known queries S and testing whether $Q \in S$, we could substitute the full signature S with its hash. This substitution is valid, because SD*river*/*SQL* operates under the premise of best effort rather than absolute correctness [102]. Therefore, the stack trace and the stripped down query are concatenated and the driver applies a hash function on them to form the stored form of the query signature. When the system is operating in training mode, all the signatures are saved in an auxiliary database table, so that when the system operates in production mode the driver can check whether a query is legal or not.

Production Mode The driver's functionality during the production mode does not differ significantly from the one in the training mode. The steps are the same until the driver derives the query's signature. At that point, the driver consults the database table of saved query signatures to verify that the query is legal. If the driver identifies it as a legitimate one then the query passes through. If it does not, then the application is probably under attack. In such a case the driver can halt the application with an exception, it can log an error message, or it can forward an alarm to an enterprise-wide intrusion detection system. As an example, consider an attack that takes advantage of incorrectly filtered quotation characters. The additional keywords that the malicious user injects will definitely lead to an unknown signature. In this case the driver becomes aware of the attack and prevents it.

3.4.2 SDriver/XPath: A Secure XPath Application Library

In order to secure an application from XPath injection attacks (see Section 1.1.2), we designed a mechanism based on the basic principles our approach. The mechanism is very similar to its SQL counterpart and we call it SD*river*/XP*ath*. SD*river*/XP*ath* is also designed as a *proxy* library that contains the default implementation of XPath and has the security features of our approach. During its training phase all the XPath queries of the application must be executed so that we can can identify them in a way we showed in Section 3.3. Then the operation of the mechanism shifts into production mode, where it takes into account all the legal queries and can thereby prevent XPath injection attacks.

All XPath queries of an application can be identified combining three of its characteristics, namely: its *call stack trace*, its *nodes*, their corresponding *axes* and the various *operators*. As it did before, the stack trace involves the stack of all methods from the method of the application where the query is executed down to the target method of our mechanism. The nodes of the statement, is a set that includes various kinds of keywords like: attributes, elements, text etc. The XPpath axes are particular keywords that define a node-set, relative to a specified node. Finally, the operators are the functions that operate on values or other functions.

A formal representation of the application's identifiers (quite similar to 3.5 as someone should expect) that should be accepted as legitimate ones, is the following: If during an application's normal

run, K is the set of method stack traces at the point where an XPath statement is executed; N is the set of the nodes; A the set of the corresponding axes, and O the set of the operators, the set of the legitimate query identifiers I is defined as follows:

$$I = \{\omega : \omega = (k, a_1, a_2, ...), k \in K, a_i \in (N \cup A \cup O)\} \tag{3.6}$$

When the system operates in the training mode, each query identifier Q is added to I. In production mode a query with a identifier Q is considered legal *iff* $Q \in I$. To combine these characteristics, when a query is about to be executed our mechanism carries out two actions. First, it strips down the query, removing all numbers and string literals. So if the statement presented in Section 1.1.2 is being executed the corresponding values for `cId` and `pFilter` will be removed from the query string.The mechanism also traverses down the call stack, in the same manner as its SQL counterpart.

3.5 nSign: Protecting Web Users from JavaScript Injection Attacks

Most web sites consist of HTML documents.[2] A web page, besides the textual content and markup, may reference other external documents or resources such as images and multimedia files. A web page can be dynamic or static. Static pages are delivered to the user exactly as stored on the server-side, contrary to dynamic web pages which provide different content for each individual browser request. In order to provide a rich user experience or allow for dynamically updating parts of a page without requesting its entire content from the web server, the majority of dynamic pages support *client-side scripting* [180]. Client-side scripting refers to computer programs that are executed on the client-side, by the user's web browser. One of most common scripting languages used is JavaScript.

JavaScript, which is an ECMAScript[3] dialect, is a dynamic, weakly typed, prototype-based, object-oriented scripting language. It supports a structured programming syntax and treats functions as first-class objects [81]. Most modern web browsers are accompanied by a JS engine that interprets and executes JavaScript. Every script included in a web page that is displayed within a browser will be processed by the browser's JS engine. There are two ways to include a script within a web page, either in the HTML markup of the page or in external documents referenced by the page. In the first case such a script is referred to as an *embedded script*, whereas in the later it is referred to as an *external script* [81]. The following code snippet includes the method `alert`, which is used to display pop-up boxes, and demonstrates the declaration of an embedded script in the HTML code of a web page:

```
<script type="text/javascript">
    alert('Message');
</script>
```

In order to reference an external script (for example a script named *external.js*), a developer should use the following format:

<script type="text/javascript" src="external.js" />

The primary use of JavaScript is to interact with the DOM of the page. The DOM is a cross-platform and language-independent API for valid HTML and well-formed XML documents. It defines the logical structure of such documents and the way a document is accessed and manipulated.[4] Even if a web browser does not have to use the DOM to render an HTML document, the DOM is required by scripts that wish to inspect or modify a web page dynamically. Some simple examples of the interaction between JavaScript and the DOM include: the validation of input values of a web form in

[2] http://www.w3.org/TR/html4/struct/global.html
[3] http://www.ecmascript.org/
[4] http://www.w3.org/TR/DOM-Level-2-Core/introduction.html

order to make sure that they are acceptable before being submitted to the server and the toggling of images as the mouse cursor moves over them. Furthermore, the execution of JavaScript code within the context of the web browser allows it to programmatically access critical objects of this context like cookies or browser history.

The above features provide the potential for malicious authors to access critical information regarding a web user. To contain such risk, browser authors use two restrictions. In particular, scripts run in a sandbox where they can only perform web-related actions and not general-purpose programming tasks (e.g. creating files) [65]. Also, scripts are constrained by the same origin policy. This policy permits scripts running on pages originating from the same site to access each other's methods and properties with no specific restrictions, but prevents access to most methods and properties across pages on different sites [210].

To demonstrate that our scheme is applicable in the dynamic language-driven injection context we chose to apply our approach to counter JavaScript injection attacks (see Section 1.1.2). Note here, that even if we are still protecting against code injection attacks, we are now guarding the user's web browser and not a server-side application as was the case in the previous two examples. To meet the requirements of this context we have significantly extended our approach in a way that we present below.

The basic weakness that the attackers exploit in the countermeasures we described in Section 2 is the fact that most web sites fail to properly encode data that comes from an untrusted source. As a result, the JavaScript engines of the web browsers used to access these sites are unaware of the fact that the contained JavaScript may be malicious. The key idea of our scheme in this context focuses exactly on this fact. In particular, our validation layer wraps the JS engine of the browser to differentiate legitimate scripts from ones that have been tampered with.

An important objective of our scheme here, is to associate legitimate scripts with their origins and all the external URLs they reference. Every script included in a web page that is displayed within a browser will be processed by the browser's JS engine. This engine is responsible for the interpretation and/or compilation, and ultimately the execution of the client-side scripts. There are many ways to include a script within a web page. This can be done either in the HTML markup of the page or in external documents referenced by the page [81]. The later also involves the direct usage of hosted JS libraries like jQuery, AngularJS, and others.[5] In addition, scripts may contain references to external resources like images, web pages and others, thus making the JS engine instruct the browser to interact with numerous entities coming from different sources across the Internet.

Figure 3.8 illustrates the architecture of our scheme. Its key component is a modified JS engine, which we call *nSign*. *nSign* includes our validation layer which has been extended to fit this context — see Algorithm 4. To secure a browser from JavaScript injection attacks, *nSign* first goes through a signature generation phase. This is done by the site developers or administrators during testing of deployment. During this phase, every legitimate script is mapped to an identifier that we call a "*script signature*". This signature is then associated with a set of "*URL signatures*" that are generated from the domain part of URLs extracted from the arguments passed to the script during execution. All signatures are stored in an auxiliary table.

During production mode, the steps of *nSign* are exactly the same as during signature generation mode until *nSign* derives the signature for the script to be executed. At that point, *nSign* checks if a corresponding script signature exists in the aforementioned table. If there is, the script initially passes as legitimate. Then, while executing it, it checks if the script arguments reference any unexpected URLs. If no corresponding signature is found or an unregistered URL signature is encountered, the browser is probably under attack. In such a case *nSign* can halt the execution, log an error message, or forward an alarm to the web site's administrator.

[5] https://developers.google.com/speed/libraries/

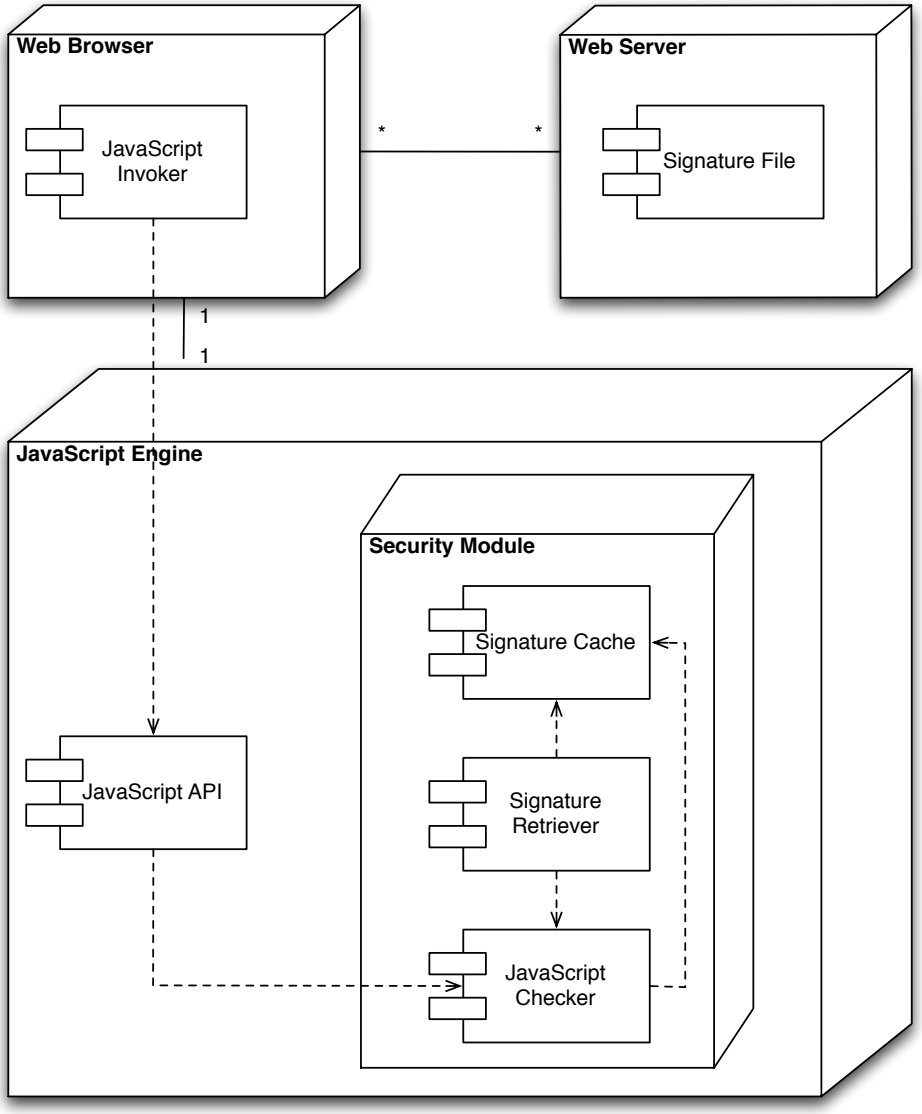


Figure 3.8: The architecture of *n*S*ign*.

Algorithm 4 Validation Layer Algorithm

```
1:  function SECURITYLAYER(script)
2:      strippedScript ← AnalyzeScript(script)          // remove user input, dynamically changing elements
3:      domainName ← GetDomainName()
4:      scriptType ← GetScriptType()
5:      if scriptType = evalFeed then                   // the current script is a script fed to eval
6:          stackTrace ← RetrieveJavaScriptStackTrace(script)
7:          executionContextElements ← domainName + scriptType + stackTrace
8:      else                                            // we are dealing with an embedded or an external script
9:          executionContextElements ← domainName + scriptType
10:     end if
11:     s ← GenerateScriptSignature(strippedScript, executionContextElements)
12:     u[ ] ← GetReferencedURLSignatures(script)
13:     if signature_generation_mode = true then
14:         RegisterScriptSignature(s)
15:         RegisterReferencedURLSignatures(s, u)
16:         r ← ExecuteScript(script)
17:     else
18:         if ValidateScriptSignature(s) then // script signature exists in the signature table - legitimate script
19:             if CheckReferencedURLSignatures(s, u) then
20:                 r ← ExecuteScript(script)
21:             else                                    // unexpected URL reference
22:                 r ← LogPossibleAttack(script)
23:             end if
24:         else                                        // script signature is not valid
25:             r ←LogPossibleAttack(script)
26:         end if
27:     end if
28:     return r
29: end function
```

3.5.1 Signature Generation

Script signatures are created by combining features that depend on the JavaScript code that is going to be executed and features that depend on the origins and the type of the script. Then, they are associated to URL signatures that represent the domain part of external resources referenced by the script during its execution.

Script Features Before generating a script signature *nSign* analyzes the script's code. This step entails stripping down the script from potential user input (namely string literals etc.) and isolating code fragments that could change for every execution. Consequently, the signature does not include variable names and class names for classes defined by the JavaScript code as elements due to the dynamic nature of JavaScript. Such elements could depend on a user's credential (i.e. username) and change during an application run. Key elements that we use to generate a script signature, include the script's block structure, as expressed by block delimiters, and aggregated counts of selected JavaScript keywords (see Figure 3.9). By JavaScript keywords, we mean all the reserved words (i.e. `while, for`) and common function names (i.e. `eval`) of the language.[6]

Apart from the aforementioned elements, for every script, we retrieve the domain name part of any URL that is referenced in string literals within the script. For example, if a script contains a static reference to a remote image, the domain name part of this reference will be included in the script signature. The script analysis and strip down is performed using a simple state machine that collects the structural information regarding the script and parses string literals for references to external resources

[6] https://developer.mozilla.org/en/JavaScript/Reference

```
function(qoptions) {
    var e = (typeof(enc) == 'function')?"n":"s";
    var r = qc.qcrnd();
    var sr = '',qo = '', ref = '', je = 'u', ns = '1';
    var qocount = 0;qc.qad = 0;
    if(typeof qc.qpixelsent == "undefined"){
        qc.qpixelsent = new Array();
    }
}
```

```
(){(())()(){()}}
  function:  1
    var:  4
  typeof:  2
    if:  1
  Array:  1
```

Figure 3.9: A real-world JavaScript function and its aggregated keywords and block structure. Script source: BBC.com.

in a single pass.

In order to normalize the input provided to the state machine for analysis, we use the internal Abstract Syntax Tree (AST) representation generated by the engine after parsing the script to format the script code. This eliminates features of the original script that are not useful for our analysis (such as comments and superfluous whitespace), thus helping to improve its performance. The above features do not represent a script uniquely. In essence, they partially represent the "behaviour" of a script.

Execution Context Features Such features include the type of the script and the domain name part of the URL that triggered the script execution. The type can be classified in the three following categories:

- *Embedded*, refers to script code contained within HTML `<script>` ... `</script>` tags.
- *External*, refers to external scripts referenced by a web page using the `src` attribute of the `<script/>` tag.
- *Eval*, refers to code that has been passed as an argument to the `eval` method for evaluation and execution.

The codebase of a script may be either the URL of the web page containing the script (in case it is embedded) or the URL of the external file referenced by the web page. The domain name is the only usable part of the script's codebase. This is because many web sites reference dynamic pages that produce JavaScript code or generate embedded script code dynamically. Thus the part of the URL following the domain name is not necessarily static.

In the case of `eval` we also include the invocation stack trace that led to the function by traversing the JavaScript call stack and retrieving the calling entity (location and function name) of each JavaScript stack frame (as presented in Algorithm 3). After retrieving and combining all the above features, we can identify the paths from where we can expect specific script "behaviours". After the combination, *nSign* applies a hash function to them to generate the final script signature.

Runtime Features and Association to Script Signatures As we mentioned earlier, our objective is to associate a script with all the URLs it references. Until now, this is partially done by extracting the domain name part of any URL statically referenced within a script. However, referenced URLs are not always static parts of the script code; they can be assembled by simple string operations like

concatenations of variables or can be retrieved through the DOM.[7] They might even be formed by processing data retrieved via Asynchronous JavaScript and XML (AJAX) calls [115], (possibly containing objects in JSON format). Finally, they can be contained in JSON-with-padding (JSONP)[8] responses. Eventually, these URLs may be used as string arguments to a defined function.

Since an attack could be orchestrated by modifying the data retrieved by the browser in such a manner [67], it is important that the string arguments passed to JS functions are scanned for external URLs. This allows us to capture any hosts dynamically referenced by a script and associate them with the signature that has been generated for the script being executed. Thus a script signature is associated to a set of URL signatures that represent the domains that the script may dynamically access at runtime. During runtime we can monitor the values of string arguments passed to JS functions and prevent the script from interacting with non-legitimate domains. Contrary to the other features used in the script signature generation, these references do not need to be taken into account for generating the script signature, since they are not dependent on the code's structure and execution flow. They represent domains that the script may access during its execution, therefore their nature is purely dynamic. These references go through the same hashing algorithm used for signature generation to create the final URL signatures.

3.5.2 Signature Propagation and Retrieval

When all signatures for a web site have been generated, they are stored locally on the server. Specifically, all signatures are stored in a database that serves two distinct purposes. The first involves the export of valid signatures to a plain text file, along with a timestamp. This process must take place once the web site is ready for deployment in the production environment, and after all its functionality has been exercised in the signature generation mode, so that all legitimate signatures have been generated. This file, hereinafter called the "*signature file*", should be placed on the root folder of the web site, and made publicly available so that *nSign* can retrieve it while in production mode — see Figure 3.8.

In production mode, this database serves as a local cache of valid signatures on the client side, in order to avoid requesting the signature file from a web site every time the user visits it. This file is requested and transferred over a secure channel (Secure Sockets Layer (SSL)) in order to avoid tampering and Man-in-the-Middle (MITM) attacks [10]. The request for the signature file occurs only once per browser session and per visited web site. The signature retrieval is illustrated in Figure 3.10. After retrieving the signature file, the timestamp that accompanies it is checked to make sure that the local database contains the latest version of valid signatures for the given web site. If the timestamp of the local signature file is older than the one that exists on the server, it means that a new set of signatures has been generated as a result of an application modification. In this case, the new signature file is downloaded and the local cache is updated. While the browser is running, the set of valid signatures is cached in a hashed set in memory in order to allow for efficient signature validation.

However, the fact that the signature file is requested once per session could lead to a false positive alert. Specifically, if the signature file is updated at the server-side, the clients that are currently visiting the site will have an outdated signature file. Hence, *nSign* may generate a signature that will not pass as legitimate. To avoid this, our scheme takes one more step before raising an alarm: it checks again whether there is an updated signature file on the server. If there is, *nSign* checks again if the signature exists in the updated signature file before alerting the user (see Figure 3.10).

[7] http://www.w3.org/DOM/
[8] http://www.json-p.org/

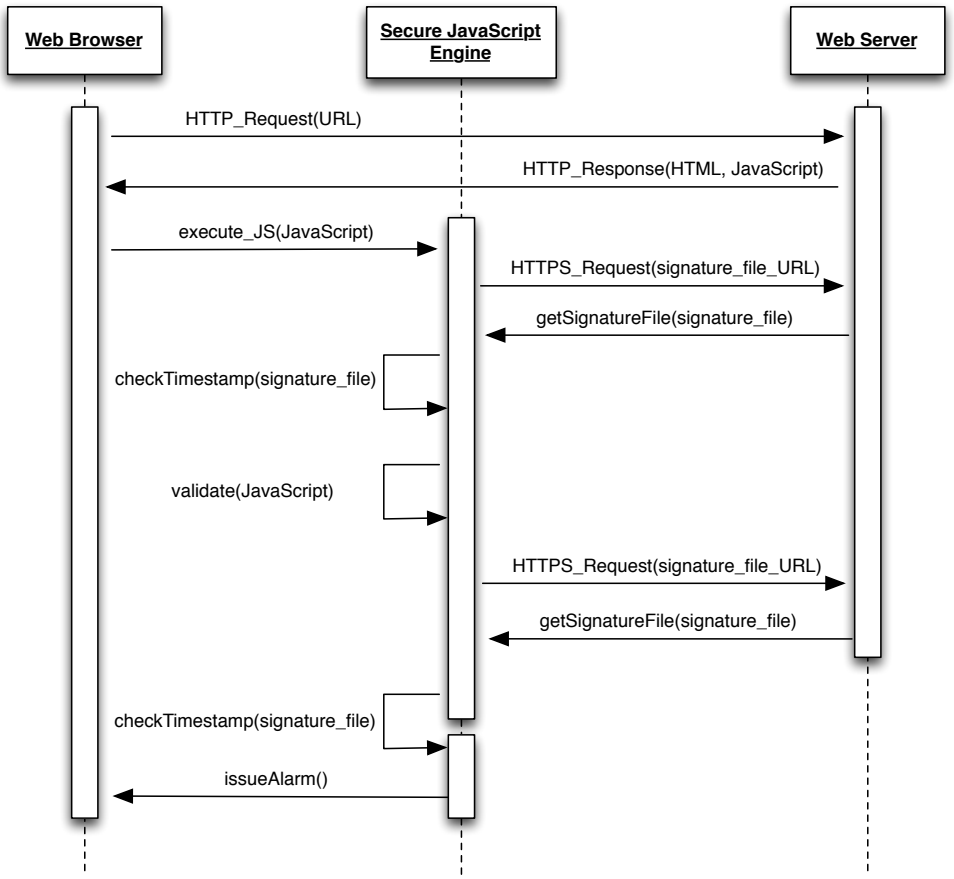


Figure 3.10: Signature check and retrieval during a new session.

Chapter 4

Implementation

In this chapter we describe the implementations of the mechanisms presented in the previous chapter. The implementations of these mechanisms were done in order to ensure the validity of our approach. Note that the ones presented in Section 4.1 were done in the Java programming language, while the mechanism presented in Section 4.2 was implemented in C++. This highlights the fact that our approach does not require the characteristics of a specific programming language, thus conforming with *implementation independence* requirement mentioned in Section 2.3.

4.1 DSL Support

We have implemented a mechanism that counters SQL injection attacks and one that protects applications from XPapth injection attacks based on the design presented in Subsections 3.4.1 and 3.4.2 respectively. Both mechanisms depend neither on the application nor on the storage entities (database, XML documents) and can be easily retrofitted to any system.

4.1.1 SDriver/SQL

We have implemented SD*river*/*SQL* (see Section 3.4.1) in the Java platform, but implementations in other operating environments are certainly feasible. The secure database driver acts as a JDBC driver wrapped around other drivers that implement a database's JDBC protocol (see Figure 4.1). JDBC drivers known as "native-protocol drivers"[1] (or type 4 JDBC drivers) convert JDBC calls directly into the vendor-specific database protocol. At the client's (application) side, a separate driver is needed for each database. SD*river*/*SQL* does not depend on the application or the native driver and it is placed between them. To accomplish that, the application must be modified in a single position: in the location where the application establishes a connection with a driver. For the application to be secured, the SD*river*/*SQL* must establish a connection with the driver that the application is meant to use. To achieve that, we pass the driver's name through the URL of the original connection (see Figure 4.1). For example, if the application is meant to connect to the Microsoft SQL Server 2000 the source code would look like this:

```
Class.forName ("com.microsoft.jdbc.sqlserver.SQLServerDriver");
Connection conn = DriverManager.getConnection(
    "jdbc:microsoft:sqlserver://localhost:1433;databasename = MyDB",
    "username", "password");
```

The modified code for using the SD*river*/*SQL* would be:

[1] http://java.sun.com/products/jdbc/driverdesc.html

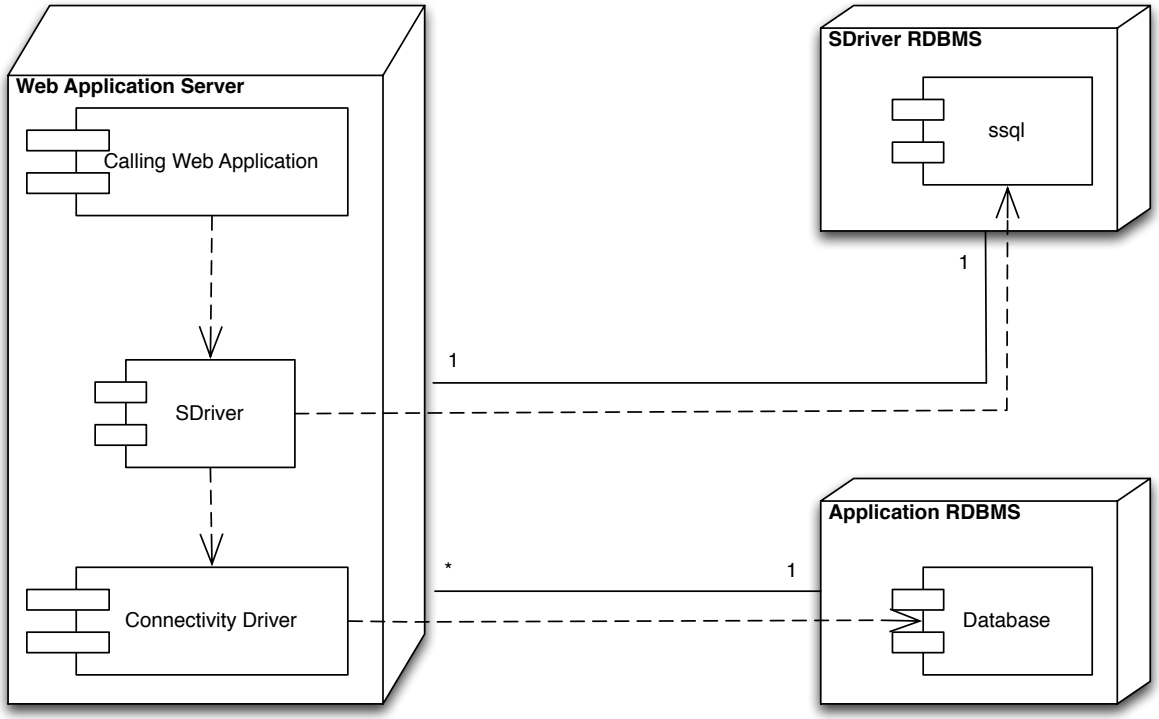


Figure 4.1: SD*river*/*SQL*'s architecture.

```
Class.forName ("org.SDriver");
Connection conn = DriverManager.getConnection(
  "jdbc:com.microsoft.jdbc.sqlserver.SQLServerDriver:" +
  "microsoft:sqlserver://localhost:1433;databasename = MyDB",
  "username", "password");
```

In essence, contrary to many other countermeasures that require multiple modifications on the applications' source code, our mechanism minimizes such modifications down to one line of code.

SD*river*/*SQL* is not a classic native-protocol Relational Database Management System (RDBMS) driver. The implementation of most of the driver's methods simply involves calling the corresponding methods of the underlying driver. Figure 4.2 illustrates the implementation of SD*river*/*SQL*'s main class. Note that by using the `connect` method created a connection interface with the the driver that the application was supposed to use thus, providing transparency. This is why the rest of the methods (`getMajorVersion`, `getMinorVersion`, `jdbcCompliant`) simply return the result of the corresponding driver. Only a few methods from those that a native-protocol driver implements pass SQL code through them, and can therefore be used to launch an SQL injection attack. These methods are the various forms of `addBatch`, `execute`, `executeQuery`, and `executeUpdate`. To secure applications against SQL injection attacks, SD*river*/*SQL* interposes itself in these methods examining the query string that is about to be executed. For this examination to take place, SD*river*/*SQL* follows the steps of Algorithm 1. Figure 4.1 also indicates that SD*river*/*SQL* depends on another database component called *ssql*. This works as the signature data store mentioned in Section 3.3. The pseudocode listed in Figure 4.3 (typically a combination of Algorithms 1, 3 and 2) sums up the steps of SD*river*/*SQL*. To determine in which mode will work (training or production), SD*river*/*SQL* opens an external `.txt` file.

One of the tricky parts of the SD*river*/*SQL* implementation is the code that traverses the applic-

```
public class SDriver implements java.sql.Driver {
        private static final String URL_PREFIX = "jdbc:SDriver:";
        private Driver nextDriver;
        static {
                try {
                        SDriver SDriverInstance = new SDriver();
                        java.sql.DriverManager.registerDriver( SDriverInstance );
                } catch (SQLException e) {
                        System.err.println ("Could not register SDriver");
                        e.printStackTrace ();
                }

        }
        public int getMajorVersion() {
                return nextDriver.getMajorVersion();
        }
        public int getMinorVersion() {
                return nextDriver.getMinorVersion();
        }
        public boolean jdbcCompliant() {
                return nextDriver.jdbcCompliant();
        }
        public boolean acceptsURL(String url) throws SQLException {
                return url.startsWith(URL_PREFIX);
        }
        public Connection connect( String url, Properties prop)
                        throws SQLException {
                SConnection localConInstance = null;
                if (acceptsURL(url)) {
                        try {
                                localConInstance = new SConnection(url, prop);
                                nextDriver = localConInstance.nextDriver;
                        } catch (ClassNotFoundException e) {
                                e.printStackTrace ();
                        } catch (SQLException e) {
                                e.printStackTrace ();
                        }
                }
                return (Connection) localConInstance;
        }
        public DriverPropertyInfo [] getPropertyInfo ( String url, Properties prop)
                        throws SQLException {
                return nextDriver.getPropertyInfo ( url, prop);
        }
}
```

Figure 4.2: The main class of SD*river*/*SQL* in Java.

```
manageQuery(String query) {
    signature = getQuerySignature( stripQuery (query ));
    if ( inSignatureTable ( signature ))
        return;
    if (inTrainingMode)
        // insert signature into the signature table
    else {
        // issue an alarm or raise an exception
        // write query to a log file
    }
}

stripQuery ( String query) {
    query = removeQuotedStrings(query);
    query = removeNumbers(query);
    query = removeComments(query);
}

getQuerySignature( String query) {
    for (StackTraceElement ste : stackTrace)
        signature .append(ste );
    signature .append(query);
    return MD5(signature);
}
```

Figure 4.3: The operation of SD*river*/*SQL*.

ation's stack (see Algorithm 3). Perversely, in Java the way to access the stack frames is to create an object of type `Throwable`. The class `Throwable` is the superclass of all errors and exceptions in Java. To aid the display and debugging of exceptions `Throwable` objects support a method called `getStackTrace`, which returns an array of stack frames. Each stack frame provides methods for obtaining the corresponding file name, method name, and line number. The actual implementation of the above are depicted in Figure 4.4. This figure contains the implementation of the `getQuerySignature` method which is the one that concatenates the stripped down query with the stack trace and produces the signature after applying an MD5 algorithm on the concatenated strings. The following listing shows the contents of a stored stack trace:

```
com.SStatement.getQuerySignature(SStatement.java:556)
com.SStatement.manageQuery(SStatement.java:489)
com.SStatement.executeQuery(SStatement.java:430)
beans.querybean.selection1(querybean.java:20)...
```

Every stack element contains information about a method invocation including the method name, the package, the file, and the line number. The first method will always be `getQuerySignature` because it is the one that traverses the call stack. The element that participates more in the diversity of a signature is typically the fourth one: the application's method that directs the connectivity driver to pass the query to the database.

Although we have implemented SD*river*/*SQL* as a JDBC proxy, the same approach could also be used for applications written in other languages, like C and C++. Furthermore, the association of queries with their stack trace can be used to minimize the extent of source code modification in other approaches, like AMNESIA [97, 98].

```
public String getQuerySignature(String strippedDownQuery) {
        String ID, complete, frameToString, s = null;
        String completeTrace = "";
        int counter = 0;
        byte b[] = null;
        byte d[] = null;
        Throwable t = new Throwable();
        StackTraceElement stack[] = t.getStackTrace();
        if (stack == null || stack.length <= 1) return null;
        StringBuffer sb = new StringBuffer();
        while (counter != stack.length) {
                frameToString = stack[counter].toString();
                completeTrace = completeTrace.concat(frameToString);
                counter++;
        }
        complete = completeTrace.concat(strippedDownQuery);
        b = complete.getBytes();
        int secondCounter = 0;
        try {
                MessageDigest algorithm =
                        MessageDigest.getInstance("MD5");
                algorithm.reset();
                algorithm.update(b);
                d = algorithm.digest();
                for(counter = 0; counter < d.length; counter++) {
                        s = Integer.toHexString(d[counter] & 0xff);
                        if(d.length == 1) sb.append('0');
                            sb.append(s);
                }
        } catch (NoSuchAlgorithmException e) {
                e.printStackTrace();
        }
        ID = sb.toString();
        return ID;
}
```

Figure 4.4: The implementation of `getQuerySignature` method in Java.

4.1.2 SDriver/XPath

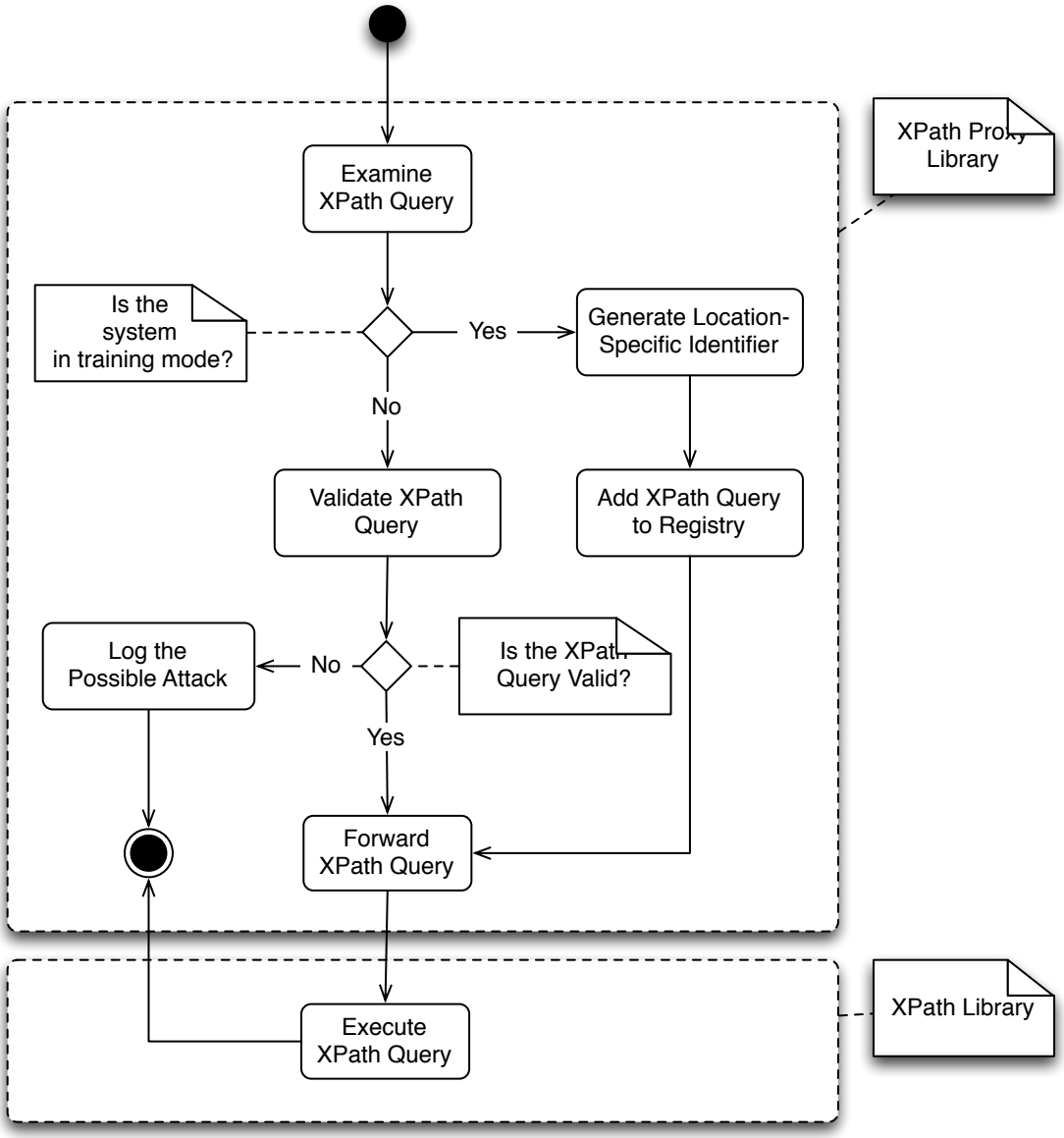


Figure 4.5: SD*river*/XP*ath* activity diagram.

SD*river*/XP*ath* is practically a proxy library that contains the default implementation of XPath [26, 125] and has the security features of our approach. SD*river*/XP*ath*'s implementation is very similar to its SQL counterpart. Note that major code fragments from the implementation of SD*river*/*SQL* were also used to build this module. In essence, the implementations of the methods (see Equation 3.4) that derive the signatures in SD*river*/*SQL* are essentially the same in this context. For instance, the `getQuerySignature` depicted in Figure 4.4 was used unaltered.

Figure 4.5 illustrates the proposed injection detection scheme as a UML activity diagram. The XPath proxy library which we call SD*river*/XP*ath* accepts the request to execute XPath code from the application. If the training mode is activated, then the XPath code is analyzed and the signature is generated and registered as valid. After the training phase the application's flow is followed as designed.

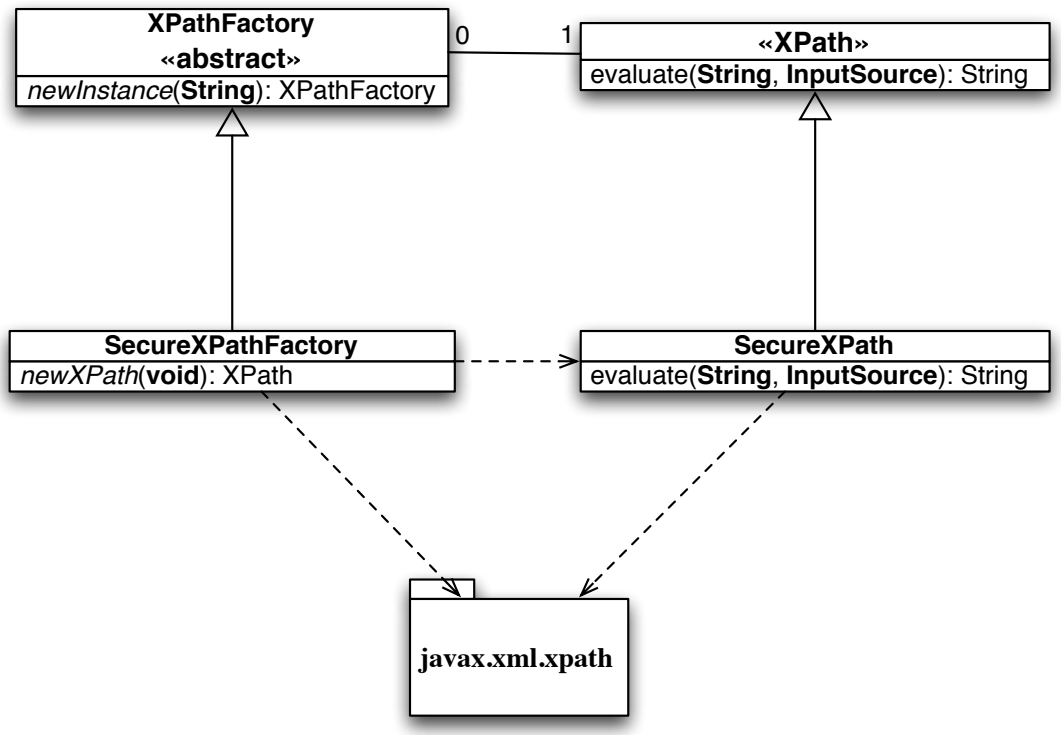


Figure 4.6: Overall System Design

If the library is in production mode, the XPath code is analyzed again and the system checks if the XPath query is valid. Actually, the location-specific identifier is generated again and is compared against all signatures in the registry for validity. If not, specific details regarding the invalid call is logged. Upon validation, the XPath code is forwarded to the standard XPpath library and is executed and the results are returned to the application.

Figure 4.6 depicts our design as a UML class diagram. Our library acts as a plug-in in JDK's XML implementation. It implements the `abstract` class `XPathFactory` and the interface `XPath`. Specifically, the class `SecureXPathFactory` is the entry point for our library, and the `SecureXPath` is the actual implementation of the XPath application library. Both classes depend heavily on `javax.xml.xpath` package, that contains the XPath implementation. The class `SecureXPath` implements two functionalities. First, following the steps of Algorithm 1, each XPath query is associated with a signature. Then, all signatures are stored in the *registry*. The registry can use various back-ends to store its data, like databases, flat files etc. Before the deployment of the application, the development and the testing teams should execute the application many times in order for our secure layer to record all the valid queries. In our case, the registry is a in-memory `TreeSet` structure. Its implementation can be easily modified though, to support any store and retrieval mechanism, according to the application needs.

As we mentioned above, the SD*river*/XP*ath* library depends on `javax.xml.xpath`[2] package. A typical usage of XPath library in a Java application is exhibited in the following code fragment.

```
XPathFactory xpf = XPathFactory.newInstance();
XPath xpath = xpf.newXPath();
xpath.compile("//orders/customer");
```

[2] http://java.sun.com/j2se/6/docs/api/javax/xml/xpath/package-summary.html

The XPath implementation parser is initialised by the appropriate XPathFactory. Then the compile method is used to instantiate the XPath query. The XPathFactory is the implementation of the *Factory* Pattern [88], which permits with an elegant way to support many XPath library implementations. Our implementation uses this facility to introduce the security library without any significant changes in the source code of the original application. Practically, a developer needs to address two things, in order to fortify his application:

1. Add into his classpath the JAR file with the secure library.

2. Invoke the appropriate XPathFactory class.

For example, the previous code fragment should be modified as follows to include our approach:

```
XPathFactory xpf =
      XPathFactory.newInstance(XPathFactory.DEFAULT_OBJECT_MODEL_URI,
                               "org.sdriver.xpath.SecureXPathFactory",
                               classloader);
XPath xpath = xpf.newXPath();
xpath.compile("//orders/customer");
```

The rest of the code in the application should remain the same. The *training mode* of the library is activated through the XPathFactory.setFeature() method, which is a standard call of the XPathFactory abstract class. In our previous example, the *training mode* can be set using the following code:

```
XPathFactory xpf =
    XPathFactory.newInstance(XPathFactory.DEFAULT_OBJECT_MODEL_URI,
        "org.sdriver.xpath.SecureXPathFactory", classloader);
xpf.setFeature("TrainingMode", true);
XPath xpath = xpf.newXPath();
```

Hence, adding the library as a security layer is as easy as it was with its SQL counterpart.

4.2 nSign: JavaScript Language Support

In order to evaluate our proposed scheme against JavaScript injection attacks (see Section 3.5, we modified the *Mozilla SpiderMonkey* open source JS engine.[4] SpiderMonkey is the JS engine used by numerous open source software projects, most namely the Firefox web browser. It is written in the C++ programming language. We instrumented the SpiderMonkey API methods that are used by the browser as entry points to the JS engine, by wrapping the implementation of each one so that it goes through our validation layer. This validation layer distinguishes legitimate scripts from injected ones in the way we presented in the previous section. More advanced mechanisms employed by modern JS engines such as Just In Time (JIT) compilation [12] or caching of compiled scripts are neither affected by nor affect our mechanism, since it operates on an intermediate level between the API and core of the JS engine. This also means that no changes are required on the code of the browser itself.

[4] http://www.mozilla.org/js/spidermonkey/

4.2.1 Intercepting JavaScript Code

The entry points of a JS engine are the API calls used by the browser to execute the scripts contained in or referenced by a web page. In the case of SpiderMonkey this API is provided via the `jsapi.h` file.[3] In order to improve performance, applications that use a JS engine try to minimize script compilation time by holding references to compiled scripts and invoking them multiple times with different argument values. We therefore needed to instrument API methods that either take as input a script as a string or execute a script that has already been compiled by the engine. JavaScript is also used internally by Firefox in order to implement various functions mostly related to the user interface. To avoid processing such scripts we devised a mechanism for filtering out "internal" calls to SpiderMonkey, by examining the security principals and the script origin associated with each API call.

Although we managed to identify the entry point of every JavaScript execution, there is a case that must be handled with special care: the code fed as an argument to the `eval` function may be stripped out during code analysis. Therefore, we also wrapped the `eval` function implementation (located within the `jsobj.cpp` file) in order to perform code analysis on its input.

A potential source of false positives could occur if we did not consider the JSON data included in scripts used by various web applications. JSON is an open standard format that uses human-readable text to transmit data objects consisting of attributevalue pairs. It is used primarily to transmit data between a server and web application, as an alternative to XML. To define the structure of JSON data for validation, documentation, and interaction control a JSON Schema must be determined (see Figure 4.7). The content and the size of such data varies over time (see Figure 4.8). In addition, to convert JSON data into an object, the `eval` function can also be used. Then `eval` invokes the JavaScript compiler. Since JSON is a proper subset of JavaScript, the compiler will correctly parse the text and produce an object structure. As a result, JSON data could affect the block structure of a script and lead to false alarms. SpiderMonkey distinguishes internally all JSON data coming from JavaScript code. This distinction came in as a handy shortcut because our scheme does not intend to generate signatures for this kind of data. Even if an attacker manages to pass a script through JSON data, the script will finally reach the JavaScript engine and a script signature will be generated. As this signature will not exist in the signature file, the malicious script execution will be blocked.

Although the word JavaScript may bring to mind features such as DOM objects, AJAX [152] calls (`XMLHttpRequest` objects) and event handlers like `onclick`, these features are not part of SpiderMonkey or most JS engines. They are typically provided by the applications using the JS engine, by exposing some of their objects and functions to JavaScript code.[4] For instance, the implementation of `XMLHttpRequest` in Mozilla Firefox is located in class `nsXMLHttpRequest`. However, we were able to intercept `XMLHttpRequests` initiated by JavaScript code, because SpiderMonkey uses an API method to call back to a `nsXMLHttpRequest` object, by wrapping references to the corresponding native functions provided by Firefox in `JSObject` structures. In a similar manner we are able to intercept the execution of scripts that are triggered by event handlers.

By identifying all JavaScript entry points we are able to analyze all possible scripts in the way we described in Section 3.5.1. For our code analysis, we have implemented a parser and used regular expressions in order to retrieve the code structure and the keywords respectively (see Figure 4.9).

4.2.2 JavaScript Feature Extraction

The processing of document and script locations is performed using a custom URL parser that is able to extract the various parts it consists of, such as the protocol (e.g. HTTP), authority (domain name and port), path, and query. The same parser is used to extract the static references to external resources contained in a script. The line in which a script is located within a document is passed by the browser

[3] https://developer.mozilla.org/en-US/docs/SpiderMonkey/JSAPI_Reference
[4] https://developer.mozilla.org/en-US/docs/SpiderMonkey/JSAPI_User_Guide

```
{
    "name": "Product",
    "properties": {
        "id": {
            "type": "number",
            "description": "Identifier",
            "required": true
        },
        "name": {
            "type": "string",
            "description": "Name of the product",
            "required": true
        },
        "price": {
            "type": "number",
            "minimum": 0,
            "required": true
        },
        "tags": {
            "type": "array",
            "items": {
                "type": "string"
            }
        },
        "stock": {
            "type": "object",
            "properties": {
                "warehouse": {
                    "type": "number"
                },
                "retail": {
                    "type": "number"
                }
            }
        }
    }
}
```

Figure 4.7: A JSON Schema for products and their attributes.

```
{
    "id": 4,
    "name": "Test",
    "price": 301,
    "tags": [ "Pool", "Bar" ],
    "stock": {
        "warehouse": 400,
        "retail": 15
    }
}
```

Figure 4.8: JSON data that can be validated with the Schema presented in Figure 4.7.

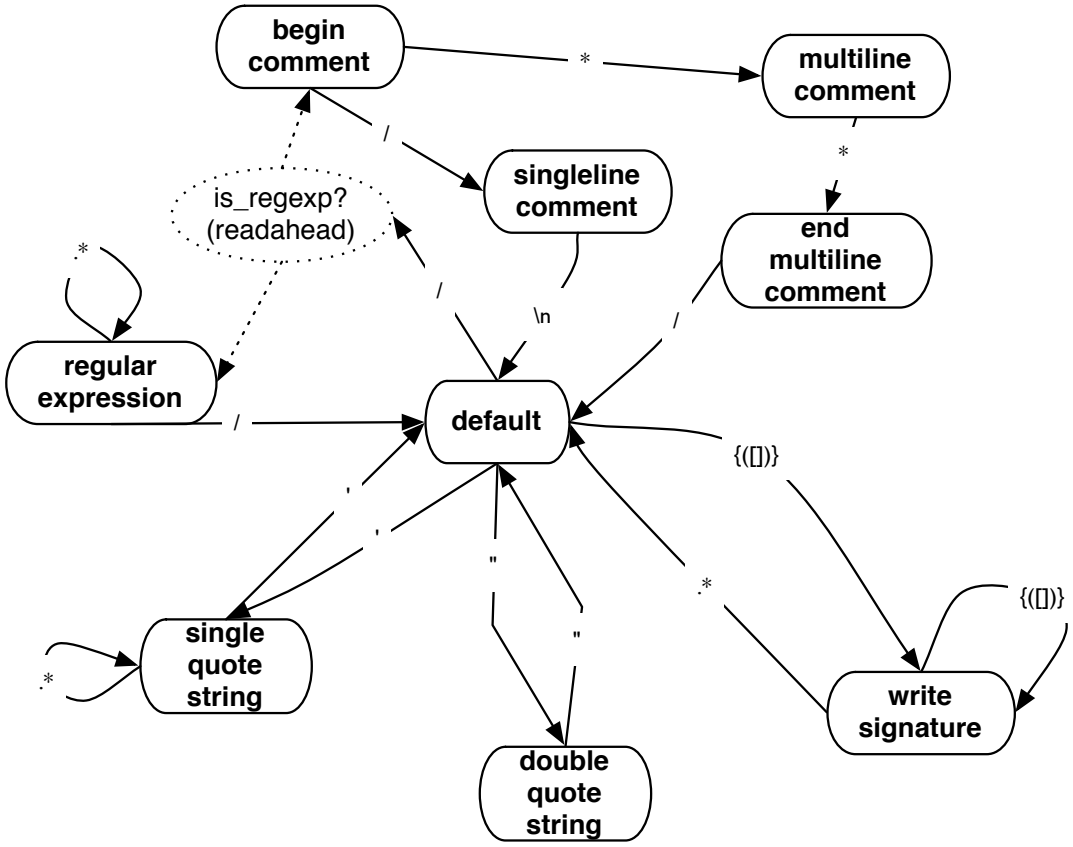


Figure 4.9: A state diagram illustrating the functionality of our JS parser.

to the API methods as a parameter, so that by examining the location of a script (codebase and line number) we can distinguish between *internal* and *external* scripts.

If the JavaScript code is a script fed to `eval`, we also retrieve the JavaScript stack trace that led to the `eval` function call. To accomplish that, we utilize the data structures that SpiderMonkey uses to monitor the execution of a script, most notably the `JSContext` and `JSStackFrame` (see Figure 4.11). The former contains the execution stack, while the later is a structure representing a frame of the JS virtual machine stack. A frame contains information regarding the method being executed, the program counter, the formal and actual arguments, the local store associated with each call, and the script being executed (function name if available, source location, etc.). It also contains a pointer to the next frame on the stack, which allows us to traverse the execution stack by iterating through the frames and collecting the information we need for each one. This information includes the location of the script being executed (domain name part of the URL of the document containing the script) and the name of the JavaScript function if available or "anonymous function" otherwise. This information is appended to the stripped down form of the source code being passed to the `eval` method and is used to generate the final script signature. The structure of the stack and the stack frames are illustrated in Figure 4.10.

The final signature for a given script occurs by combining the above features and applying a hash method on them. We used the MD5 hash algorithm since it is fast and reliable for our needs, and the chance of collisions is minimal in the given scope. However it takes minimal effort to use a more robust hash algorithm such as SHA1.

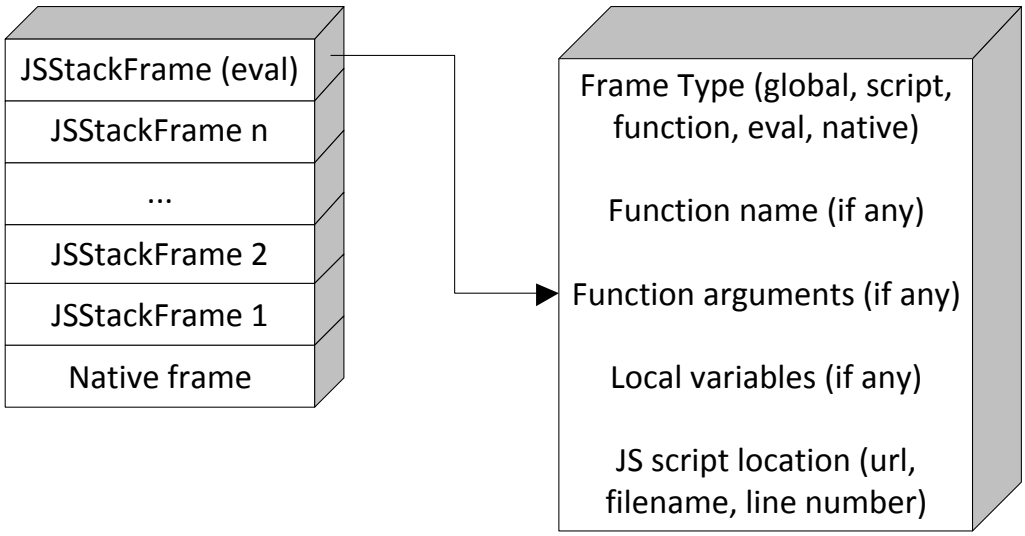


Figure 4.10: JavaScript execution stack and stack frame elements.

As we described in Section 3.5.1, we needed to associate the domain part of URLs that can be referenced by a script at runtime with the corresponding script signature. The execution of any compiled script eventually goes through the function `RunScript`. Within this function, we have access to all the argument values that are passed to any compiled JavaScript function. `RunScript` performs some sanity checks and then either invokes the JS interpreter or the JIT compiler (if JIT compilation is enabled). At this point, we extract URLs contained in string arguments. If any URLs are found, we retrieve the domain part of each one and apply the same hash function to it, thus generating a URL signature. Note that this process is only performed when a compiled script is executed, and it is not applicable in the case of `eval`.

4.2.3 Signature Management

To collect all valid signatures automatically, the administrator of the protected web site can utilize a web testing framework that should execute all possible scripts and as a result generate all their corresponding script signatures and the associated URL signatures. This is a common issue in web application testing, but luckily a number of approaches have been created to overcome it [14].

An interesting option is using the *Crawljax*[5] testing framework. Despite its name, Crawljax is not a web crawler, but the implementation of an approach developed to test all AJAX interfaces of a web application [153] and consequently execute all JavaScript code fragments of the web site. To accomplish that, Crawljax provides an interface that allows the tester to configure the depth of the testing (practically the depth that the web site crawler will reach during the test) and the browser to be used. By using the browser with our modified JS engine, the administrator can automatically retrieve all signatures without having to manually browse the whole site and execute its functionality (see Section 5.3.2). In addition, the generation of all signatures can be ensured by employing a JS

[5] `http://crawljax.com/`

```
static string GetEvalStackTrace(JSContext *cx, JSStackFrame * start )
{
    stringstream ss( std :: stringstream :: in | std :: stringstream :: out );

    /* navigate to the first frame of interest */
    if (! start )
        start = cx->maybefp();
    FrameRegsIter i(cx);
    while (! i.done() && i.fp() != start )
        ++i;

    if (i.done())
    {
        ss << "Unable to traverse stack, invalid frame?\n";
    }
    else
    {
        for (; !i.done(); ++i)
        {
            JSStackFrame *const fp = i.fp();

            if (fp->isEvalFrame())
                ss << "eval -> ";
            if (fp->isScriptFrame())
            {
                JSUri scriptUri (fp->script()->filename);
                ss << "(" << scriptUri. getAuthority () << ") ";
            }
            if (fp->isFunctionFrame())
            {
                const Value &v = ObjectValue(fp->callee());
                if (v.isNull ())
                    ss << " null";
                else if (v.isUndefined ())
                    ss << " undefined";
                else if (v. isString ())
                    ss << " " << (void *)v. toString ()->chars();
                else if (v. isObject () && v.toObject (). isFunction ())
                {
                    JSObject *funobj = &v.toObject ();
                    JSFunction *fun = GET_FUNCTION_PRIVATE(cx, funobj);
                    if (!fun->atom)
                        ss << " unnamed function";
                    else
                        ss << " " <<
                            JS_GetStringBytes(ATOM_TO_STRING(fun->atom));
                }
                return ss. str ();
}
```

Figure 4.11: Retrieving the `eval` stack trace in C++.

code coverage tool like JSCover.[8] This is particularly convenient not only for small web sites but also for rapidly evolving web applications with dynamically generated pages and scripts, but may become tedious for applications containing scripts that reference numerous external URLs.

As a back-end for storing the signatures generated by *nSign* we employed an SQLite database.[6] SQLite is a relational database management system contained in a small C programming library. In contrast to other database management systems, SQLite is not a separate process that is accessed from the client application, but an integral part of it. During signature generation the database is used to store the signatures that will then be exported into the signature file. In production mode the database serves as a local cache of signatures for the web sites that the user has visited in the current session. This helps us avoid multiple requests for the signature file on each visited web site.

To aid evaluation in our prototype implementation we did not fully implement the mechanism for the retrieval of the signatures via a secure channel from within *nSign*, but rather used a mockup proxy that retrieves the signature files from either local store or using an external application to fetch signature files over a secure channel. The space requirements for the signature database is minimal since most web sites are typically associated with a few dozens or hundreds of signatures, so our scheme has a small overall impact on the user's machine — see Section 5.3.2.

[8] http://tntim96.github.io/JSCover/
[6] http://www.sqlite.org

Chapter 5

Evaluation

A mechanism that secures another entity should be evaluated in terms of detection accuracy [10]. As we mentioned in Section 2.3, the accuracy of a mechanism can be judged by the existence of FP and FN. Specifically, accuracy is defined by the following equation [172]:

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN} \tag{5.1}$$

The equation above, also involves True Positives (TP) and True Negatives (TN). TP (*sensitivity*) measures the proportion of actual attacks which are correctly identified as such. TN (*specificity*) measures the proportion of negatives which are correctly identified as such. In essence, the percentage of legitimate usages who are correctly identified as not being attacks.

In addition, security mechanisms should impose a low computational overhead in order to be adapted. In this chapter we present the evaluation of our tools based on the aforementioned criteria.

5.1 SDriver/SQL

We have extensively evaluated SD*river*'s accuracy and operation cost. In our tests we found out no false positives or false negatives. Also its operation cost indicates that it is efficient and it can be used to secure real-world applications.

5.1.1 Accuracy

We evaluated the accuracy of SD*river*/*SQL* through three experiments: a synthetic benchmark, a notoriously insecure application, and a bundle of previously evaluated real-world applications. Our synthetic benchmark was a JSP application with the same technical characteristics as those described in reference [34]. This application allowed a user to inject SQL into a "where" clause with no input validation, and retrieve information concerning application data. After placing SD*river*/*SQL* between the application and the database, the attack was successfully prevented.

We then searched for a real-world web application that had a record of being vulnerable to SQL injection attacks. According to the common vulnerability database CVE[1] and the security bulletin providers US-CERT,[2] Secunia,[3] and Armorize Technologies,[4] a notoriously vulnerable application is Daffodil Customer Relationship Management (CRM) 1.5.[5] In Daffodil 1.5, remote attackers could

[1] http://cve.mitre.org/
[2] http://www.us-cert.gov/
[3] http://secunia.com
[4] http://www.armorize.com
[5] Daffodil can be obtained from http://www.daffodildb.com/crm/

execute arbitrary SQL commands via unspecified parameters in a login action. In particular, users wanting to access Daffodil had to fill-in a simple username and password form. By using a SQL injection attack similar to the one we presented in Section 1.1.2, an unauthorized user could access administrator facilities. SD*river*/*SQL* recognized and blocked the attack, without otherwise interfering with Daffodil's operation.

Application	Signatures	Unsuccessful	Successful	Prevented
bookstore	168	288	39	39 (100%)
classifieds	122	270	37	37 (100%)
employee directory	61	207	31	31 (100%)
events	65	115	29	29 (100%)
portal	156	312	49	49 (100%)

Table 5.1: SD*river*/*SQL*'s precision.

Finally, we selected five real-world web applications that have been used in the literature for previous evaluations[6] [95, 208]. We attempted a wide variety of attacks based on incorrectly filtered quotation characters, incorrectly passed parameters, untyped parameters, tautologies, and others [11, 99].

Table 5.1 shows, for each web application, the number of the signatures stored in the SD*river*/*SQL* database after training, the number of unsuccessful attacks (attacks that did not get past the application's defenses), the number of successful attacks (attacks that could potentially compromise the application), and the number of attacks prevented by SD*river*/*SQL*, in absolute terms and as a percentage over the total number of successful attacks. The table's columns follow the labeling introduced in reference [95].

While testing, we realized that the five applications shared a common feature; In a misguided attempt to avoid SQL injection attacks, they scanned user input for single quotation marks and replaced them with double quotation marks. This technique masked the SQL injection attack problem, but introduced a data leakage vulnerability. For instance, a user input parameter consisting of the string *any*\' in the application "portal" would result in the execution of the following query:

```
SELECT e.date_start AS e_date_start, e.event_desc AS e_event_desc,
  e.event_name AS e_event_name, e.location AS e_location,
  e.presenter AS e_presenter FROM events e WHERE
  (e.event_desc LIKE '%any\"%' OR  e.event_name LIKE '%any\"%' OR
  e.presenter LIKE '%any\"%') ORDER BY e.date_start DESC
```

The preceding statement would raise an exception revealing information about the underlying database and its schema.

Given that our driver works as a wrapper around other connectivity drivers, we could also instrument it to handle exceptions when running in production mode. As a result, critical information like the above would not be revealed. However, because well-written applications have their own sophisticated exception handling, we made secure exception handling an optional configurable feature. With its secure exception handling activated, our tool successfully prevented all attacks in this last test without suffering from false negatives. Furthermore, we did not encounter false positives (legitimate queries misreported as an attack) in any of the three experiment classes we performed. Hence, based on Equation 5.1 SD*river*/*SQL* is 100% accurate.

Application	Execution time (ns)		
Database	Original	SD*river*/*SQL*	Overhead (%)
SQL Server	175	183	4.7
MySQL	121	126	3.7

Table 5.2: Proxy driver baseline cost.

Application	Execution time (μs)			Overhead (%)	
Database	Original	Training	Production	Training	Production
SQL Server	605	1221	841	102	39
MySQL	401	1009	613	60	35

Table 5.3: The cost of SQL query processing under SD*river*/*SQL*.

5.1.2 Operation Cost

Deploying SD*river*/*SQL* is relatively straightforward: the only requirement is the ability to modify the database's connection string. This can be achieved by specifying an appropriate application-specific parameter (Java property), by modifying the source code, or (in extreme cases) by patching the application's binary. Furthermore, one must then execute the application in training mode. An automated test suite that will exercise (ideally all) the application's calls to methods containing SQL strings with user-input data, would make this exercise trivial. Otherwise appropriate scenarios must be devised and executed each time a new version of the application is installed.

The driver's architecture allowed us to test its performance on two RDBMSs: SQL Server 2000 and MySQL (version 5.0.24). All tests were performed on a Pentium 4 Central Processing Unit (CPU) clocked at 2.6 GHz on a machine with 512MB RAM running Java 1.6.0 under Windows XP Professional. We first measured the baseline overhead of SD*river* by executing a JDBC method — `getAutoCommit()` — that is passed through directly to the underlying database driver without further processing. The results, appearing in Table 5.2 indicate that the cost of interposing SD*river*/*SQL* is negligible. In order to obtain accurate time values we used the method `System.currentTimeMillis()`, which according to literature is the most efficient and correct method to use in similar cases [199]. Subsequently, we measured the overhead of the SQL injection attack detection code by executing the following moderately complex SQL statement, with and without SD*river*:

```
SELECT d_name, d_SorL, d_year,d_genre, d_cover FROM artists,
  disks, recorded WHERE a_name = '"+ selectedartist +"' AND
  d_name = rec_d_name AND rec_a_name = a_name
```

Application	Execution time (ns)		
Database	Original	SD*river*/*SQL*	Overhead (%)
SQL Server	78	79	1
MySQL	16	17	6

Table 5.4: Early prototype baseline cost.

[6]The applications can be obtained from `http://www.gotocode.com/`

Application	Execution time (μs)			Overhead (%)	
Database	Original	Training	Production	Training	Production
SQL Server	813	3016	3078	271	279
MySQL	797	2796	2703	251	239

Table 5.5: The cost of SQL query processing under the early SD*river*/*SQL* prototype.

The performance overhead for the two RDBMSs was similar (see Table 5.3). In training mode the queries take twice as long to execute. However, this cost is not unreasonable, because this is an execution mode that will be rarely exercised. In production mode, the operation cost is significantly lower: below 50% for both RDBMSs. Early versions of our tool, incurred a significant overhead (with a range from 239% to 279% in both training and production mode—see Table 5.5). We optimized away this overhead by streamlining the regular expressions used for stripping the SQL queries, and by caching the signatures into a static `Hashtable` when a connection between the application and the SD*river*/*SQL* is first set up in production mode. We also considered limiting the depth of the stack frame processing during production mode, by calculating in training mode a stack frame prefix tree (trie) [128], but the performance improvements were negligible.

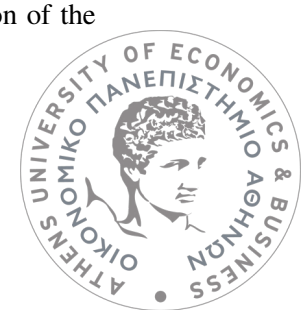
5.2 SDriver/XPath

In this case, our prototype is build in order to counter many forms of XPath injection attacks including taking advantage of:

1. incorrectly passed parameters,
2. incorrect type handling, and
3. incorrectly filtered quotation characters.

XPath language is usually combined with Extensible Stylesheet Language Transformations (XSLT) applications, thus we were unable to find a simple yet realistic example to test our library in terms of accuracy. In addition, XPath injection attacks constitute a quite recent[7], form of attacks, hence there aren't any real-world web applications that have a record of being vulnerable to this kind of attacks. The prototype implementation of the proposed XPath library introduces functionality that repels code injections, but also adds an overhead at runtime. The secure XPath library was tested in laboratory for performance against the standard XPath library shipped with the JDK. We have to note that the secure XPath library implementation depends on the JDKs' library. Consequently, the time difference between was clearly introduce by the security layer. It is logical, that the performance of the secure XPath library can be increased with careful optimisation, and the results of our experiments are indicative. The benchmark was executed on a `Core 2 Duo 2.4Ghz` CPU on a Mac OS X (v10.5). The Java version was 1.6 and the library was compiled with aggressive compile-time optimisations. The experiment process was simple, we populated the *registry* with 10 XPath expressions and the invoked the `XPath.compile()` method (standard library call). The iteration number was 1,000,000. The XPath query that we used follows: The benchmark results showed that the library performed 128% slower than the standard XPath library (21,311 milliseconds against 48,761 milliseconds). The benchmark was executed five (5) times and the statistical mean was used to produce the above results. In addition, each benchmark iteration had a warm-up period for the virtual machine. The implementation of the benchmark is depicted in Figure 5.1.

[7] `http://www.ibm.com/developerworks/xml/library/x-xpathinjection.html`



```
public class OverheadXPathTest extends XPathTest {
        public OverheadXPathTest(boolean secure) {
                super(secure);
        }
        public void execute() {
                String [] queries =
                        { "/orders/customer[@id='foo']/order/item[price >= 5]",
                          "author[last-name [position()=1]= 'Bob']",
                          "//EXAMPLE/CUSTOMER[@id='1' and (@type='B' or @type='C')]",
                          "/bookstore/book[price>35]/title",
                          "//EXAMPLE/CUSTOMER[@id='1' and @type='B']",
                          "//EXAMPLE/CUSTOMER[@id='2' or @type='C']",
                          "//EXAMPLE/CUSTOMER[substring(@type,1,2) ='DE']",
                          "//EXAMPLE/CUSTOMER[contains(@type,'DECEA')]",
                          "//EXAMPLE/CUSTOMER[contains(.,'Smith')]" };

                final int iterations = 1000000;
                System.out.println("Measuring compile overhead ... "
                                + ((secure ? "SDriver/XPath" : "JAXP")));
                System.out.println("Number of Iterations : " + iterations);
                System.out.println("Queries tested : " + queries.length);
                if (secure) {
                        try {
                                xpathfactory.setFeature("TrainingMode", true);
                        } catch (XPathFactoryConfigurationException e) {
                                System.out.println("Could not set the Training Mode (true)");
                        }

                        XPath xpathtr = xpathfactory.newXPath();
                        for ( String query : queries ) {
                                try {
                                        xpathtr.compile(query);
                                } catch (XPathExpressionException e) {
                                        System.out.println("Compilation of "
                                        + query + " failed ... ");
                                }
                        }
                }
                if (secure) {
                        try {
                                xpathfactory.setFeature("TrainingMode", false);
                        } catch (XPathFactoryConfigurationException e) {
                                System.out.println("Could not set the Training Mode (true)");
                        }
                }
                XPath xpath = xpathfactory.newXPath();
                for ( String query : queries ) {
                        for (int i = 0; i < iterations; i++) {
                                try {
                                        xpath.compile(query);
                                } catch (XPathExpressionException e) {
                                        System.out.println("Could not compile ... "
                                        + query + " (" + e.toString() + ")");
                                        break;
                                }
                        }
                }
                for ( String query : queries ) {
                        long start = System.currentTimeMillis();
                        for (int i = 0; i < iterations; i++) {
                                try {
                                        xpath.compile(query);
                                } catch (XPathExpressionException e) {
                                        System.out.println("Could not compile ... "
                                        + query + " (" + e.toString() + ")");
                                        break;
                                }
                        }
                        long end = System.currentTimeMillis();
                        System.out.println("Input: "
                                + query + "---" + (end - start) + " msecs");
                }
        }
}
```

Figure 5.1: The testing benchmark for measuring the overhead of SD*river/*XP*ath*.

Application	Version	Signatures	Attack Type	Prevented
Joomla	1.5.20	9	Cookie Stealing	✔
phpMyFAQ	2.6.8	4	Cookie Stealing	✔
Pluck[1]	4.6.3	6	Redirect	✔
JCart[2]	1.1	4	Redirect	✔
TikiWiki[3]	1.9.8.1	5	Cookie Stealing	✔

[1] Each signature corresponds to one script of the application.
[2] http://www.pluck-cms.org/
[3] http://conceptlogic.com/jcart/
[4] http://info.tiki.org/tiki-index.php

Table 5.6: Accuracy of *nSign*.

5.3 nSign

nSign is a security module that wraps the JS engine of a browser. A browser can interact with hundreds of websites every day the could contain numerous scripts. Hence, the corresponding stored signatures, in the case of *nSign*, would be much more than in the above cases. Thus, apart from the detection accuracy and the computational overhead of this mechanism, we have also evaluated its *maintenance cost*.

5.3.1 Accuracy

To evaluate the effectiveness of *nSign*, we searched for web applications that had a record of being vulnerable to JavaScript injection attacks. Our experiment involved vulnerable applications that were installed and attacked locally, and real-world JavaScript-based XSS exploits hosted by XSSed.com.[1] With the laboratory tests, we attempt to test and improve the effectiveness of our approach while with the real-world tests we checked how our scheme responds to normal conditions. At the end of this section, we discuss about a certain type of attacks that could bypass our mechanism and how we deal with them.

Laboratory Tests To find applications to set up and test in laboratory conditions we consulted *Secunia*,[8] a security bulletin provider that includes thousands of cases of vulnerable applications, including version details, vulnerability type and others. Then, we downloaded the vulnerable versions of five well-known OSS projects including Joomla,[9] and phpMyFAQ.[10] First, we applied our signature generation mechanism to the applications, focusing on the pages that were vulnerable. Then, we switched to production mode and performed various attacks based on the vulnerability type of every application. Defects like the absence of proper input sanitization and the improper verification of HTTP requests allowed us to perform redirect and cookie stealing attacks [211]. For example, in phpMyFAQ, URLs are not sanitized correctly, making it possible to inject JavaScript code into the user's browser and steal cookies. All attacks were successfully detected without encountering any false positives or negatives. Table 5.6 shows, for each web application, its vulnerable version, the number of the signatures stored in the database after the signature generation mode, the attack type that we performed, and if the detection was successful.

Prevention of Real-world Attacks We selected fifty real-world vulnerable applications taken directly from the vulnerability archive hosted by XSSed.com. This archive has been previously used for

[1] http://www.xssed.com/
[8] http://secunia.com/
[9] http://www.joomla.org/
[10] http://www.phpmyfaq.de/

evaluation in other papers [15, 160]. Initially, we searched for XSS attacks that utilize JavaScript. Then we checked if their status is *unfixed*, which means that the site administrators have not fixed the existing defect yet and the site is vulnerable to attacks even if it is online. The vulnerable web pages included top-ranking sites like eBay.com, nydailynews.com and others. Based on the existing examples that XSSed.com provides, we attempted a variety of attacks. Our scheme recognized and blocked all of the attacks, without producing any false alarms. In addition, it did not interfere with the browser's operation in any way. Tables 5.7 and 5.8 contain the complete list of the vulnerable sites that we tested. In particular, they contain for every vulnerable domain name, the URL pointing at the corresponding entry at the XSSed.com archive, the current status (meaning, during the writing of this thesis) of the vulnerability and the Alexa[11] pagerank of the domain. Our laboratory and real-world tests indicate that *nSign* 100% accurate.

[11] http://www.alexa.com/

Vulnerable Domain	URL entry at XSSed.com	Status	Pagerank
search.nate.com	http://www.xssed.com/mirror/70707/	*unfixed*	684
mg269.imageshack.us	http://www.xssed.com/mirror/70704/	*unfixed*	85
www.aolsvc.merriam-webster.aol.com	http://www.xssed.com/mirror/70719/	*unfixed*	49900
cookbooks.adobe.com	http://www.xssed.com/mirror/70721/	*unfixed*	54
wap.ebay.ie	http://www.xssed.com/mirror/70726/	*unfixed*	5362
hotels.qantas.com.au	http://www.xssed.com/mirror/70735/	*unfixed*	5209
www.attijaribank.com.tn	http://www.xssed.com/mirror/70696/	*unfixed*	358651
www.topgear.com	http://www.xssed.com/mirror/70346/	*unfixed*	3946
webforms.ey.com	http://www.xssed.com/mirror/70345/	*unfixed*	18732
cib.ibank.ge	http://www.xssed.com/mirror/69548/	*unfixed*	471958
ie.jrc.ec.europa.eu	http://www.xssed.com/mirror/69254/	*unfixed*	1004
esupport.trendmicro.com	http://www.xssed.com/mirror/68756/	*fixed*	3170
mail.kmr.gov.ua	http://www.xssed.com/mirror/68689/	*unfixed*	2689249
sge.corumba.ms.gov.br	http://www.xssed.com/mirror/68688/	*unfixed*	51922
zones.computerworld.com	http://www.xssed.com/mirror/70652/	*unfixed*	2775
www.nydailynews.com	http://www.xssed.com/mirror/67620/	*unfixed*	629
software.gsfc.nasa.gov	http://www.xssed.com/mirror/67530/	*unfixed*	897
seguridad.terra.es	http://www.xssed.com/mirror/67296/	*unfixed*	1783
www.jinx.com	http://www.xssed.com/mirror/70589/	*unfixed*	35644
www.nsbank.com	http://www.xssed.com/mirror/70528/	*unfixed*	77402
blackhat2010.sched.org	http://www.xssed.com/mirror/70534/	*unfixed*	82348
www.ccbill.com	http://www.xssed.com/mirror/70537/	*unfixed*	1652
w2.eff.org	http://www.xssed.com/mirror/70353/	*unfixed*	23171
www.adobe.com	http://www.xssed.com/mirror/70463/	*unfixed*	54
www.antenna.gr	http://www.xssed.com/mirror/70495/	*fixed*	18468

Table 5.7: Real-world, vulnerable websites where *nSign* was tested in terms of accuracy — Part 1.

Vulnerable Domain	URL entry at XSSed.com	Status	Pagerank
pub.kathimerini.gr	http://www.xssed.com/mirror/70510/	*fixed*	6605
vod.grnet.gr	http://www.xssed.com/mirror/70523/	*fixed*	237986
panorama.ert.gr	http://www.xssed.com/mirror/70506/	*fixed*	17110
www.cia.gov	http://www.xssed.com/mirror/70465/	*fixed*	13123
www.eff.org	http://www.xssed.com/mirror/70354/	*unfixed*	23234
roma.corriere.it	http://www.xssed.com/mirror/71241/	*unfixed*	391
www.slideshare.net	http://www.xssed.com/mirror/71127/	*unfixed*	286
www.truste.com	http://www.xssed.com/mirror/71015/	*unfixed*	17036
www.millenniumbank.ro	http://www.xssed.com/mirror/71460/	*unfixed*	242771
quicktrip.olympicair.com	http://www.xssed.com/mirror/71369/	*unfixed*	39957
blogs.oreilly.com	http://www.xssed.com/mirror/71319/	*unfixed*	3464
search.indiatimes.co	http://www.xssed.com/mirror/71201/	*unfixed*	166
www.mysql.com	http://www.xssed.com/mirror/71496/	*fixed*	630
www.accionpyme.mecon.gov.ar	http://www.xssed.com/mirror/68499/	*unfixed*	114898
reg.sun.com	http://www.xssed.com/mirror/71687/	*unfixed*	1741
www.clickbank.com	http://www.xssed.com/mirror/71689/	*unfixed*	187
support.ts.fujitsu.com	http://www.xssed.com/mirror/69286/	*unfixed*	4615
jobs.orange.com	http://www.xssed.com/mirror/70165/	*unfixed*	32146
www.wi-fi.org	http://www.xssed.com/mirror/64296/	*unfixed*	289023
www.ema.europa.eu	http://www.xssed.com/mirror/70197/	*unfixed*	1035
sandiego.bbb.org	http://www.xssed.com/mirror/70320/	*unfixed*	1783
www.energyrating.gov.au	http://www.xssed.com/mirror/67540/	*unfixed*	467403
help.comodo.com	http://www.xssed.com/mirror/71276/	*unfixed*	3922
leibniz.stanford.edu	http://www.xssed.com/mirror/71178/	*unfixed*	1308
west.stanford.edu	http://www.xssed.com/mirror/71175/	*unfixed*	1308

Table 5.8: Real-world, vulnerable websites where *nSign* was tested in terms of accuracy — Part 2.

Mimicry Attacks A possible false negative result could occur if an adversary performed a mimicry attack [209, 224]. Our scheme involves two key features to prevent this kind of attacks, namely: the static and the dynamic references to remote domains.

Consider a vulnerable page that does not check the user data stored on the server and contains a simple inline script that displays an alert message to the user. In this case, an attacker could inject a similar script with a malicious alert message, but with the same block structure. Such an attack could affect the browsing experience of the web user but it could not pose a serious threat to the user's privacy. A significant threat will occur if the attacker includes in the script an element stored on another server. But in this case, *nSign* will detect the attack. This is because a script signature includes all the static references to remote elements (see Section 3.5).

Take for instance, the JavaScript code presented in Figure 5.2, that implements a banner rotator used by various web sites.[12] Every time that the banner runs, it creates a value ("`core`") that depends on the current date and the length of the array that contains the references of the various images to be displayed. Then, based on this value it shows a specific image to a user by using a jQuery function.[13] In a vulnerable web site that allows users to post data and contains this banner rotator, a malicious user could create and store a script that has the same code structure, with the same JavaScript keywords like the above. In this script the attacker could also include references to tiny images hosted on a web server that is maintained by him in order to retrieve the Internet Protocol (IP) addresses of the users that visit the vulnerable site. Our scheme would prevent such an attack since the references to the remote images would lead to a different script signature.

However, as we mentioned in Section 3.5.1, URLs used by the attacker do not have to be contained statically in the script. Instead they could be obtained from the document DOM API or they could be dynamically assembled. Consider the following legitimate code:

```
var message = "Thank you for your visit!";
var base = "http://www.example.com/";
var path = "thepathto/mini.gif";
var url = base + path;
```

Upon script execution, variable `url` contains a statically referenced URL that will be included in the script signature. Interestingly, by performing a mimicry attack like the following:

```
var message = "Thank you for your visit!";
var base = "http:";
var path = "//bot.net/xss?s=http://www.example.com";
var url = base + path;
```

the attacker's script could lead to a legitimate script signature, as no brackets or block delimiters have been used, the script's block structure is unaffected and the statically referenced URL is also included in the script. However, the malicious URL will eventually be assembled and passed as an argument to a JavaScript function during runtime. Hence, such an attack would also be prevented since there will be no matching URL signature for the injected URL.

5.3.2 Performance and Signature Maintenance

To evaluate the runtime performance of our prototype implementation and provide details on the automation of the signature collection, we performed two experiments. The first one evaluates the real world effectiveness of the proposed signature collection scheme on the server side, while the second

[12] `http://www.hotscripts.com/listing/banner-rotator/`

[13] jQuery is a cross-browser JavaScript library designed to simplify the client-side scripting of HTML [85].

```
<script type="text/javascript">
var currentdate = 0;
var core = 0;

function initArray() {
  this.length = initArray.arguments.length;
  for (var i = 0; i < this.length; i++) {
     this[i] = initArray.arguments[i];
  }
}

var link = new initArray(
       "http://example.com",
       "http://foo.com"
);

var image = new initArray(
       "http://example.com/mini.gif",
       "http://foo.com/small.gif"
);

var currentdate = new Date();
var core = currentdate.getSeconds() % image.length;
var ranlink = link[core];
var ranimage = image[core];

src = '<a href="' + ralink + '"><img src="'
       + ranimage + '" border="0"></a><br>';
$("#container").html(src);
</script>
```

Figure 5.2: Example: Banner rotator script.

one evaluates the performance of our solution on the client side. The measurements in the first case were performed with the assistance of the Crawljax tool, mentioned in Section 4.2.3. For the second experiment, we used a build of Firefox (Mercurial repository revision identifier `56491:d253c44465ae`). We ran our experiments against nine high profile web sites, all of which make heavy use of JavaScript. Both experiments were run on a client with a quad core Intel i5 processor with 4GB of Random-Access Memory (RAM) running Linux (64-bit Ubuntu 10.10).

Signature Generation with Crawljax For the first experiment, we configured Crawljax to automatically follow the links from the site's homepage up to three levels deep in the site's page hierarchy (see Figure 5.3). We disabled following links to other sites, but we allowed random input to be sent to HTML forms. To capture `eval` method invocations, we also configured Crawljax to simulate DOM events, in order to exercise as many execution paths as possible. The number of signatures captured for each site, along with the total time required by Crawljax to crawl the site (including network latency) is reported in Table 5.9. The results show that the total number of signatures is small even for very large web sites containing thousands of pages; the optimization of JavaScript delivery to the client on such sites (all scripts in a single file) might have played a role on that front. Even if a full web site

```
public final class testNSign {
        private static String URL = "";
        private static String DOMAIN = "";
        private static final String ALL_ANCHORS = "a";
        private static int MAX_CRAWL_DEPTH = 3;
        private static int MAX_STATES = 200;

        private SiteSimpleExample() {
                // Utility class
        }
        private static CrawlSpecification getCrawlSpecification () {
            URI url = null;
            try {
                url = new URI(URL);
            } catch (URISyntaxException e) {
                e.printStackTrace ();
            }
                CrawlSpecification crawler = new CrawlSpecification(URL);
                crawler.click(ALL_ANCHORS);
                // limit the crawling scope
                crawler.setMaximumStates(MAX_STATES);
                crawler.setDepth(MAX_CRAWL_DEPTH);
                crawler.setInputSpecification(getInputSpecification());
                // Make sure we only crawl Google and no external web site
                crawler.addCrawlCondition("Only crawl " + DOMAIN, new UrlCondition(DOMAIN));
                return crawler;
        }
        private static InputSpecification getInputSpecification () {
                InputSpecification input = new InputSpecification ();
                // enter "Crawljax" in the search field
                input.field("q").setValue("Crawljax");
                return input;
        }
        private static CrawljaxConfiguration getConfig() {
                CrawljaxConfiguration crawljaxConfiguration = new CrawljaxConfiguration();
                crawljaxConfiguration.setBrowser(BrowserType.firefox);
                crawljaxConfiguration.setCrawlSpecification(getCrawlSpecification());
                // Generate a crawl report
                // crawljaxConfiguration.addPlugin(new CrawlOverview());
                return crawljaxConfiguration;
        }
        public static void main(String[] args) {
                String[] urls = {
                        "http://www.wikipedia.org",
                        "http://www.in.gr",
                        "http://www.cnn.com",
                        "http://www.bbc.co.uk",
                        "http://www.ebay.com",
                        "http://www.yahoo.com",
                        "http://www.twitter.com",
                        "http://www.facebook.com",
                        "http://www.gmail.com",
                        "http://www.amazon.com"
                };
                URL = args[0];
                DOMAIN = args[1];
                MAX_CRAWL_DEPTH = Integer.parseInt(args[2]);
                long ts = System.currentTimeMillis();
                try {
                    CrawljaxController crawljax = new CrawljaxController(getConfig());
                    crawljax.run();
                } catch (Exception e) {
                    e.printStackTrace();
                } finally {
                System.out.println(URL + " time: "
                    + ((System.currentTimeMillis() - ts) / 1000) + " sec");
                }
        }
}
```

Figure 5.3: The testing benchmark for measuring the *Crawljax* execution statistics.

Domain	Signatures	Crawljax Time[1]
google.com	158	38:27
ebay.com	84	33:16
amazon.com	160	60:08
twitter.com	11	2:04
facebook.com	60	4:00
wikipedia.org	19	66:50
cnn.com	123	52:23
bbc.co.uk	234	47:00
in.gr	114	50:42

[1] Time is measured in minutes and seconds.

Table 5.9: *Crawljax* execution statistics.

scan is required, our measurements indicate that the signature collection is feasible and it can be easily automated via a testing suite like Crawljax. We mention Crawljax as an example of a testing framework that can be used to generate signatures efficiently. It is not the end-all solution; each developer may use a different testing approach or tool taking into account of the specific characteristics of his application.

Domain	Script Type	Signatures	nSign Time				Total	SpiderMonkey[5]
			Analysis[1]	Retrieval[2]	Stack[3]	Args[4]		
google.com	source	22	35676.1	178.9	0	0	35855.0	43325.4
	compiled	105	2962.4	0.7	0	1.9	2965.1	2119.1
ebay.com	source	10	50612.1	536.4	0	0	51148.5	69470.6
	compiled	97	1588.0	0.7	0	2.8	1591.7	832.7
amazon.com	source	45	7786.3	144.1	0	0	7930.5	24632.5
	compiled	173	1109.5	0.7	0	2.8	1113.1	8863.9
twitter.com	source	15	91187.5	3586.4	0	0	94774.0	164051.0
	compiled	800	885.7	0.6	0	2.9	889.4	11071.7
facebook.com	eval	1	256.4	0.6	20	0	277.0	1575.4
	source	65	40515.5	704.7	0	0	41220.3	123993.9
	compiled	1168	1384.4	0.8	0	3.9	138929.2	7716.3
wikipedia.org	source	18	33360.6	330.9	0	0	33691.6	59238.7
	compiled	106	2366.0	0.7	0	3.0	2369.8	2079.8
cnn.com	eval	8	217.7	0.7	23.6	0	237.1	1530.8
	source	27	36300.4	293.3	0	0	36593.8	110151.1
	compiled	267	940.8	0.7	0	4.2	945.8	1932.9
bbc.co.uk	source	58	14016.8	60.7	0	0	14077.6	22079.9
	compiled	169	1609.8	1.494	0	2.8	1614.1	1263.2
in.gr	source	37	5563.8	65.0	0	0	5628.9	18832.101
	compiled	167	707.0	0.9	0	2.3	710.2	1607.2

[1] Code stripping and feature extraction.
[2] Signature retrieval from the signatures file.
[3] Stack traversal.
[4] URL extraction from string arguments.
[5] SpiderMonkey's script execution time without our functionality.

Table 5.10: Performance of *nSign* prototype. Time is measured in μs.

Computational Overhead To measure the overhead of signature generation we instrumented both SpiderMonkey's JavaScript parser and interpreter and our signature generation code with high precision time counters. For SpiderMonkey we only measured the time required to parse the input code, not including the execution time. The total time required to execute our code was broken down into four operations:

- Code formatting, stripping and basic feature extraction.
- Time to lookup a signature from the persistent store.
- Time to traverse the execution stack (only applicable in the case of `eval` calls).
- Time to scan function arguments for URLs.

Using the instrumented version of SpiderMonkey, we used Firefox to visit the aforementioned list of web sites in order to trigger signature generation. For web sites whose home page requires authentication (e.g. Gmail), we logged in with the benchmarker's account before benchmarking; the login information persisted across benchmark runs. Each benchmark run consisted of starting a saved Firefox session with all web sites opened as tabs. Measuring performance at the microsecond level in complex systems in non-isolated environments can introduce a high degree of variance in the obtained results [89]. In our case, sources that added to variance are background OS services and system interrupts required for processing user and network events. Consequently, and in order to obtain statistically significant results, we run the experiments more than thirty times.

As can be seen in Table 5.10, the bulk time is spent on the generation of script signatures; upon further investigation, we concluded that the majority of time is spent on running the library's regular expression engine that analyzes the code, rather than the block structure signature extractor. This means that our implementation might be improved by utilizing the SpiderMonkey's abstract syntax tree in order to identify the JavaScript keywords required for the signature generation. Also, since most sites do not use the `eval` facility to execute code, the time required to generate stack traces is negligible and in most cases non existent. Similarly, most compiled scripts invoked either had no string arguments or these arguments did not contain URLs, making the argument parsing overhead minimal in most cases. Depending on the web site design, the number of produced signatures can be quite large; even so, since our system only stores an MD5 hash for each signature, the space requirements are trivial.

Additionally, for a fixed script fed to the `eval` function, we measured the execution time for different stack depths, from a direct call to `eval` up to a stack depth of 20. As a result we had twenty one measurements for twenty one stack depths. The execution time is linear to the JavaScript stack ($t = 17.68 + 4d$, $p \ll 0.05$, $r^2 = 0.9939$, where d is the stack depth — see Figure 5.4).

Compared to the time required by SpiderMonkey to parse each page's scripts, our scheme appears to impose a high overhead; in practical terms however, this overhead is not perceivable by end users, as their experience is largely dominated by network latency and page rendering. On our test machine, and for all tested sites, the roundtrip time required for downloading a non-embedded script is in the range of seconds, which makes our scheme's overhead negligible (less than 0.05%). The results indicate that the described solution can be of practical value as the overhead it imposes is minimal both in time (less than forty milliseconds) and in space (a few kilobytes) terms, and as a result it does not affect the user's experience.

5.3.3 Trade-offs

As we mentioned in Section 3.5, we designed our scheme in order to provide flexibility regarding the addition or removal of features from script signatures. For instance, the semicolons contained in the script could be easily added as a feature to the script signature. This renders mimicry attacks more

difficult to perform, since the number of script commands must remain unaltered, thus imposing a strict limit on the attacker and providing extra robustness. This though, comes with a trade-off between accuracy and computational overhead. Similarly, by enforcing the checking of associations referenced by a script the computational overhead grows, but without this feature our scheme would become susceptible to an attack like the one presented in Subsection 5.3.1.

In order to simplify the signature management and propagation, the client-side processing could be simplified by removing the signature caching mechanism. The signatures that are valid for a given page could be sent over to the client along the response using standard HTTP headers. The communication overhead should be minimal, since a page is associated to a limited number of signatures. As a result, the size of the headers would not increase significantly. However such an approach would require modifications on the server side, so that the valid signatures will be sent to the client embedded in the response headers.

Over the recent years the web application development practice has displayed an increasing trend towards the use of common JavaScript libraries like jQuery. This provides an opportunity to minimize the number of signatures that need to be generated for a website, by white listing the signatures of popular libraries. In essence, this technique not only leads to a smaller signature set, but also minimizes the need for re-generating signatures for a web application if a library it depends on is updated. Web applications are typically built and tested against a specific version of a third party library, thus no re-syncing is required for every new release of a library. If a developer chooses to use a new version of a library the full set of signatures needs to be generated anyway, but most popular JS libraries have a release cycle of six months or more.

This type of white listing is becoming increasingly meaningful given that web developers nowadays link directly to the latest version of libraries hosted on a Content Distribution Network (CDN) [207], instead of bundling local copies with their application. Direct linking provides the advantage that no manual updating of libraries is required, and may possibly protect against vulnerabilities contained in outdated library versions. In addition, CDNs typically offer higher performance and availability for delivering libraries to the end user.

Such a whitelisting approach could allow for false positives in the following scenario: consider a web application that links to the latest version of a JS library that contains calls to the `eval` method. The web application itself does not use `eval` directly but relies on some API method of the library that does. If the library is updated and the aforementioned method is changed so that `eval` is called through a different path, a different stack trace will be generated, causing a false positive. However the effect of this shortcoming can be minified by employing the white listing technique only for libraries that are not directly or indirectly dependent on `eval`.

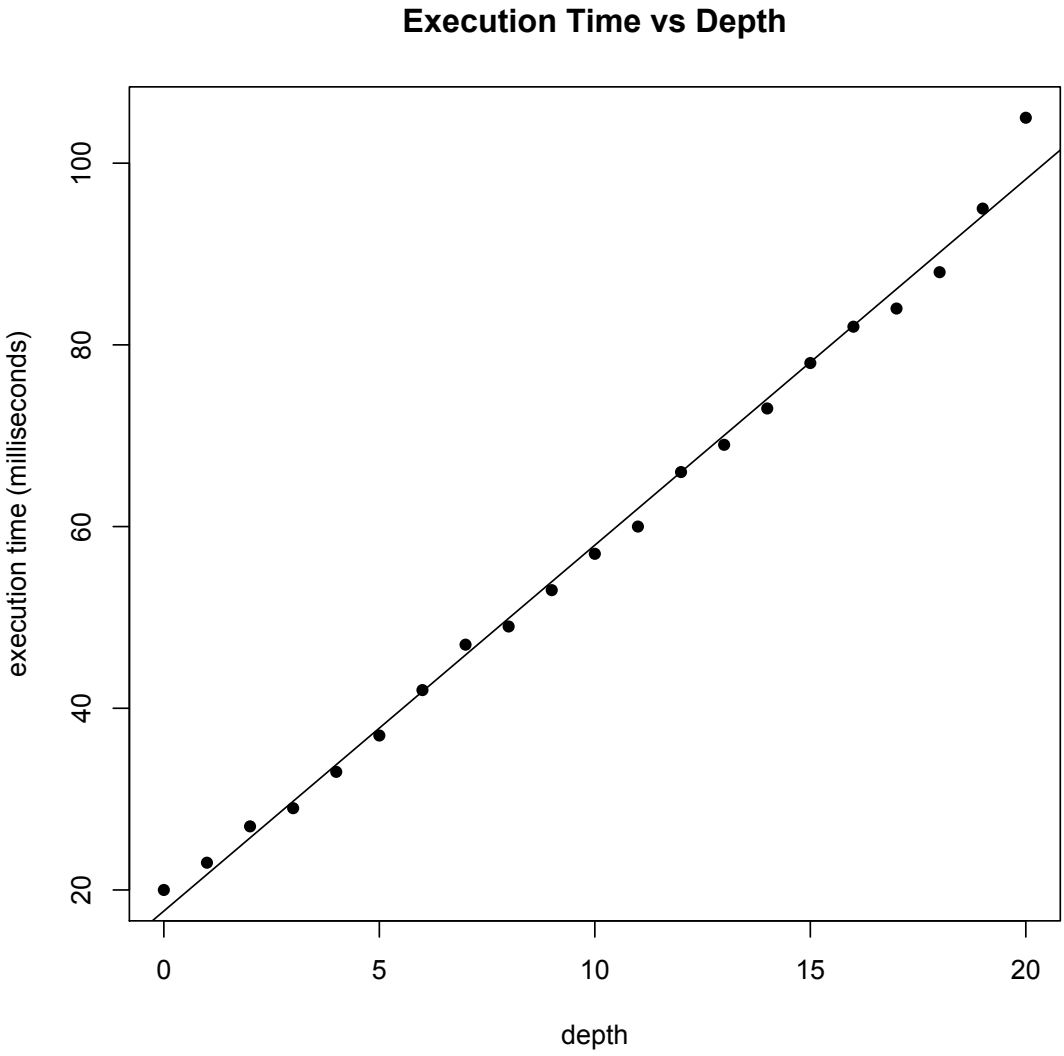


Figure 5.4: The execution time of stack traversal in relation to the depth of the stack in the case of the `eval` function.


ATHENS UNIVERSITY OF ECONOMICS & BUSINESS
ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

Chapter 6

Conclusions

With a code injection attack a malicious user can introduce code into a computer program or system by taking advantage of unchecked assumptions it makes about its inputs [183]. Such attacks are currently considered as one of the most damaging classes of application attacks and have been topping the vulnerability lists of numerous bulletin providers for several years [84, 208]. CIAs can severely affect an organization's infrastructure, and can cause significant financial damage to it [21]. In this thesis we have presented an approach that counters a specific class of CIAs in a novel way.

6.1 Contributions

To develop our approach we studied the existing approaches that counter CIAs and identified the positive and negative aspects of other countermeasures for the various CIA categories. Then, we analysed more than 260 GB of interdependent project versions to see how security bugs evolve over time, their persistence, their relation with other bug categories, and their relationship with size in terms of bytecode. We found that, although bugs do close over time in particular projects, we do not have an indication that across projects they decrease as projects mature. Moreover, defect counts may increase, as well as decrease in time. We also found that security bugs are not eliminated in a way that is particularly different from the other bugs. Also, having an average of two to three versions persistence in a sample where 60% of the projects have three versions, is not a positive result especially in the case of the bugs that lead to code injection attacks. Concerning the relation between severe security bugs and a project's size we showed that they are not proportionally related. Given that, we could say that it would be productive to search for and fix security bugs even if a project grows bigger. Furthermore, the pairwise correlations between all categories indicated that even though all the other categories are related, severe bugs do not appear together with the other bugs. Also, it is interesting to see that security bugs were one of the top two bug categories existing in a large ecosystem. Finally, we highlighted the domino effect, and showed evidence that indicates that Linus' Law does not apply in the case of the security bugs. These observations set the basis for the development of our approach.

Unless an application is severely flawed, its vulnerabilities are likely to be located in a few places, and attackers wishing to exploit them are likely to try to "play around the rules". Such opportunities are rare, and their exploitation entails forcing an application to do something outside the normal course of events. Exactly because such abnormal behavior stands out from the application's normal conduct, it is possible to detect it and take protective action when it occurs. Our approach, a *dynamic detection, training* method, takes advantage of this in order to prevent a broad class of code injection attacks. To distinguish between normal and abnormal events, we identify and register vulnerable code statements using unique signatures that we generate during a training phase. Then, at runtime, our framework checks all statements for compliance with the trained model and can thus block code statements containing additional maliciously injected elements. The training phase can take place during regression

and user acceptance testing prior to release, so that developers do not need to alter their working processes significantly. The approach introduces a runtime overhead, but the overhead compares favorably when related to the full execution cost of the protected statements; with complex statements, it will be negligible. To validate our approach, we have implemented three mechanisms that protect applications from SQL, XPath and JavaScript injection attacks. The applicability of the method to any executable source code injection attack, hints at a possible generalisation to any CIA.

SD*river*/*SQL* is a mechanism and a prototype application that prevents SQL injection attacks against web applications. If such an attack happens, the structure of the query, and therefore its signature will be altered, and SD*river*/*SQL* will be able to detect it. By associating a complete stack trace with the root of each query, the mechanism can correlate queries with their call sites. This increases the specificity of the stored query signatures and avoids false negative results. The increased specificity of the signatures also allows us to discard a large number of the query's elements, thereby also reducing false positive results. Although we have implemented SD*river*/*SQL* as a JDBC proxy, the same approach could also be used for applications written in other languages, like C and C++. To counter XPath injection attacks, SD*river*/XP*ath* has a very similar functionality with its SQL counterpart.

*n*S*ign* is a signature-based scheme to counter the threat of JavaScript injection attacks. The scheme's key property is that it represents every legitimate script of a web site with a unique identifier, a "script signature". This signature comprise features that depend on the script itself and features that depend either on the origins of the script, or its type. For the scripts that are fed to the `eval` function, the signature also includes the JavaScript stack trace that led to the function. This is done because `eval` is typically used in JavaScript injection attack methods and it is thus important to know all the possible execution paths of the calling application that reach the method and make sure that it is not called from a non-recorded location. The creation of a signature takes place in the JS engine of the web browser. To guard against code mimicry attacks signatures are also associated with "URL signatures". These signatures represent the domain part of URLs that are dynamically referenced by the corresponding script during execution. All signatures are generated during testing and stored on the server side. Then, the JS engine of the browser on the client-side retrieves the signature file in a secure way. Finally, for every script that is about to be executed on the engine a script signature is created. If this signature is matched with one of the downloaded signatures then the engine executes the corresponding script. While executing it, it checks if there are any unexpected URL references. If so, the browser is considered to be under attack and the engine blocks the script's execution. An advantage of *n*S*ign* is that the check for injected scripts takes place within the JS engine, which, by default, does not provide any validation checks. As a result, we intercept all scripts coming from every web location and in any possible way. In our tests we found out no false positives or false negatives. Also, contrary to other schemes like CSP and BEEP [137], web administrators are not required to modify the code of a web site to support our scheme. Furthermore, our scheme unlike CSP allows developers to use the `eval` function. Another advantage of our approach is that the user experience is not affected by our scheme. Users never interact with our mechanism in any way and their browsing experience is not noticeably affected by the overhead that the mechanism imposes. Although we have implemented our mechanism in SpiderMonkey — the JS engine used by Firefox — the same module could be retrofitted in other JS engines like Chrome's V8.[1]

A disadvantage of our approach is that when a signature feature is altered, a new training phase is necessary. However, with the increased adoption of test-driven development [111], and use of automated testing frameworks like JUnit [24], this training phase can be easily repeated. Another disadvantage in the case of *n*S*ign* is that since it involves modifications both in the server and the client side. This means that it would be difficult to be adopted by browser vendors.

[1] `http://code.google.com/p/v8/`

6.2 Future Work

Future work on our approach involves the following issues:

Further Analysis of Other Ecosystems Our approach could benefit from the analysis of security bugs found on other ecosystems that serve different languages, such as, Python's PyPY (Python Package Index), Perl's CPAN (Comprehensive Perl Archive Network), and Ruby's RubyGems. In addition, it would be interesting to perform similar experiments on proprietary software [54].

Expansion Despite warnings and advice for many years now, insecure software is still released. One reason is that developers are wary of incorporating into their practice cumbersome methods and unyielding tools. Countering that, our approach is easy to use, requiring minimal changes in existing code. Thus, it can be extended to more domains and languages than the three shown here. For instance, we could apply our method to counter a new emerging threat: Lightweight Directory Access Protocol (LDAP) [109] injection attacks. LDAP injection is an attack used to exploit applications that construct LDAP statements based on user input. If an application fails to properly validate user input, it is possible to modify LDAP statements using a local proxy. This could result in the execution of arbitrary commands like the modification of the content of the LDAP tree. The exploitation techniques of this attack are very similar to SQL injection. For example, in a page with a user search form, the code below could be responsible to get input values to generate an LDAP query.

```
<input type="text" size=20 name="uName">Insert Your Username Here.</input>
```

Then, the query is narrowed down for performance and the underlying code for this function could be the following:

```
String ldapQuery = "(cn = " + $uName + ")";
System.out.println(ldapQuery);
```

If the variable `$userName` is not validated, an LDAP injection attack could be performed as follows: If a user provides "*" as input, the system may return all the usernames on the LDAP database. Also, if a user inserts as input the string: "jim) (|(password = *))", the string concatenation will lead to the reveal of Jim's password.

Integration We intend to package our mechanisms in a way that will allow straightforward deployment, and experimentation with different approaches for handling the reported attacks. For instance, the association of queries with their stack trace in the case of SD*river*/*SQL*, can be used to minimize the extent of source code modification in other approaches, like AMNESIA [97, 98]. Also in the case of *n*S*ign*, we are also planning to implement the signature file retrieval from the server.

Testing The generation of all valid signatures is critical for our approach. Thus, we must show that using existing automated testing frameworks to create signatures is sufficient enough for our mechanisms to work properly. We did this with Crawljax in the case of *n*S*ign*. Even in this case thought, we configured Crawljax to automatically follow the links from the site's homepage up to three levels deep in the site's page hierarchy. In the future we need to examine how signature generation will work for the whole website. Finally, we plan to further evaluate our mechanisms in terms of accuracy. This is because attackers seem to continuously find new ways to perform CIAs by using new techniques. Hence, we need to keep up with new attacks and test our tools accordingly to see if they can detect these attacks successfully.

6.3 Emerging Challenges

With the increasing use of mobile applications and the adoption of modern ways that web applications are set up, CIAs should be examined in a different way. Even though mobile applications may require

a submission approval process to be featured in application stores, there are numerous mobile applications out in the wild, that are vulnerable. Developers seem to treat mobile applications as toy-like applications assuming that it is quite difficult for an attacker to perform an attack on them. Nevertheless, apart from other security defects, the code injection problem has also reached mobile applications[2] [73].

Also, architectures that include modern technologies like MongoDB[3] could be vulnerable to complex CIAs that could involve more than one CIA subcategory. In particular, a JavaScript injection attack could be performed to alter an SQL-like MongoDB query that is build dynamically based on user input. Specifically, when using JavaScript, developers have to make sure that any variables that cross the PHP-to-JavaScript boundary are passed in the scope field of the `MongoCode` class,[4] that is not interpolated into the JavaScript string. This can come up when using the `MongoDB::execute()` method and clauses like `$where` and `group-by`. For example, suppose that JavaScript is used to greet a user in the database logs:

```
<?php
$username = $_POST['username'];
$db->execute("print('Hello, $username!');");
?>
```

If a malicious user passes `'); db.users.drop(); print('` as a username, he or she could actually delete the entire database.

A decade ago, large computing infrastructures were held and managed by few organizations. To ensure the security of such infrastructures, organizations either recruited experts or formed internal groups to handle this task. Today, within the concept of cloud computing every organization can easily set up an infrastructure for hosting data, deploying software, services and others. In this context, responsible for setting up the infrastructure, is the same person that will define its security requirements. However, this person is not necessarily a security expert. Therefore, a bad configuration of the security properties could threat either the data of the organization or the operations. Indeed, as we observed in Appendix B, VMs hosted on the Amazon Elastic Compute Cloud (EC2) suffer from defects that can lead to CIAs.

Eventually, apart from the various attack vectors, there is another important factor that emerges: *modern application contexts*. Thus, researchers that develop new approaches to tame code injection attacks should also take into considaration such domains too.

As we mentioned in Chapter 1, researchers are currently trying to examine the problem of code injection in a more formal manner. According to the theoretical work of Ray and Ligatti [183]: *"Precisely detecting code injection attacks requires white-box, runtime-monitoring mechanisms. Under reasonable assumptions, such mechanisms can detect code injection attacks in output programs of size n in O(n) time and space."* This is an indication that efficient defense mechanisms should be based both on the internal structures of an application and on extracting information from a running system and use them to detect malicious behaviors. The authors also mention that CIA countermeasures should be based on *taint tracking* and *data-flow analysis*.

The 2012 security threat report published by the SANS[5] institute aptly mentions that: *"the technologies of tomorrow are just emerging, the requirements and circumstances are changing rapidly and so are the threats and risks. Cloud and mobile application development will require much more collaboration and closer communications between information security, development, the business lines and operations."* Indeed, the research and academic community knows little about how to stop an attacker

[2] https://www.owasp.org/index.php/OWASP_Mobile_Security_Project
[3] http://www.mongodb.org/
[4] http://www.php.net/manual/en/class.mongocode.php
[5] http://www.sans.org/

from harming either web infrastructures developed in modern ways or mobile applications. Hence, synthesizing empirical findings after examining the aforementioned contexts, together with existing pieces of theoretical could possibly lead to approaches that could cover all types of CIAs in different problem domains.


ATHENS UNIVERSITY OF ECONOMICS & BUSINESS
ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

Appendix A

The Evolution of Security Bugs: Complementary Experiment

In this section we present another framework that examines how security-related bugs evolve into a software repository, through time. To achieve this we automatically analyzed every revision of the project from its early revisions to the latest commits. Our framework combined *FindBugs*[1] [18, 107], and *Alitheia Core*, an extensible platform designed for performing large-scale software quality evaluation studies [91]. To show how the number of bugs change through time, we applied our framework to four different open source projects. Our initial observations set the basis for discussing issues that may improve vulnerability discovery models [229, 234] and identify recurring vulnerabilities [35, 117]. In this way we can ensure the reliability and flexibility of a system, which are the main objectives of software evolution [157]. Finally, we highlight security-related issues like the *domino effect* [213].

Framework Description Our framework included FindBugs (see Subsection 1.2) as a bug detector, and Alitheia Core which is a platform that provided us with an efficient way to access different projects and their repositories. Specifically, Alitheia Core [91] is a platform designed for facilitating large scale quantitative software engineering studies. To do so, it preprocesses software repository data (both source code and also process artifacts, such as emails and bug reports) into an intermediate format that allows researchers to provide custom analysis tools. Alitheia Core automatically distributes the processing load on multiple processors while enabling both programmatic and Representational State Transfer (REST) API based access to the raw data, the metadata, and the analysis results. Alitheia Core is extensible through plug-ins, in both the analysis tool front and also the raw data access from. A wealth of services, notably a metadata schema and automated tool invocation, is offered to analysis tool writers by the platform. To analyse a project, Alitheia Core needs a local mirror of the project's source code, mailing list and bug repository. The analysis itself is split in pre-defined phases (e.g. data extraction, data inference, metric extraction etc), during which Alitheia Core automatically applies a set of pre-defined data extraction and analysis plug-ins. At the end of the process, the researcher can either query the results database directly or browse the results using a simple web based interface.

To integrate FindBugs with Alitheia Core we have created a new Alitheia Core metric plug-in that works in the following steps (Figure A.1 depicts these steps as a UML state diagram): for every project and every revision of this project, the metric creates a build. Then it invokes FindBugs to examine this build and create an analysis report. A user can select whether to examine the project alone or the project together with its dependencies. Finally, from this report, it retrieves the security-related bugs and updates the database. Figure A.2 presents how the two components are integrated.

Building a software project is a multistep process that involves discovering and downloading the project dependencies, invoking the project's build script and retrieving the build artifacts. To auto-

[1] http://findbugs.sourceforge.net/

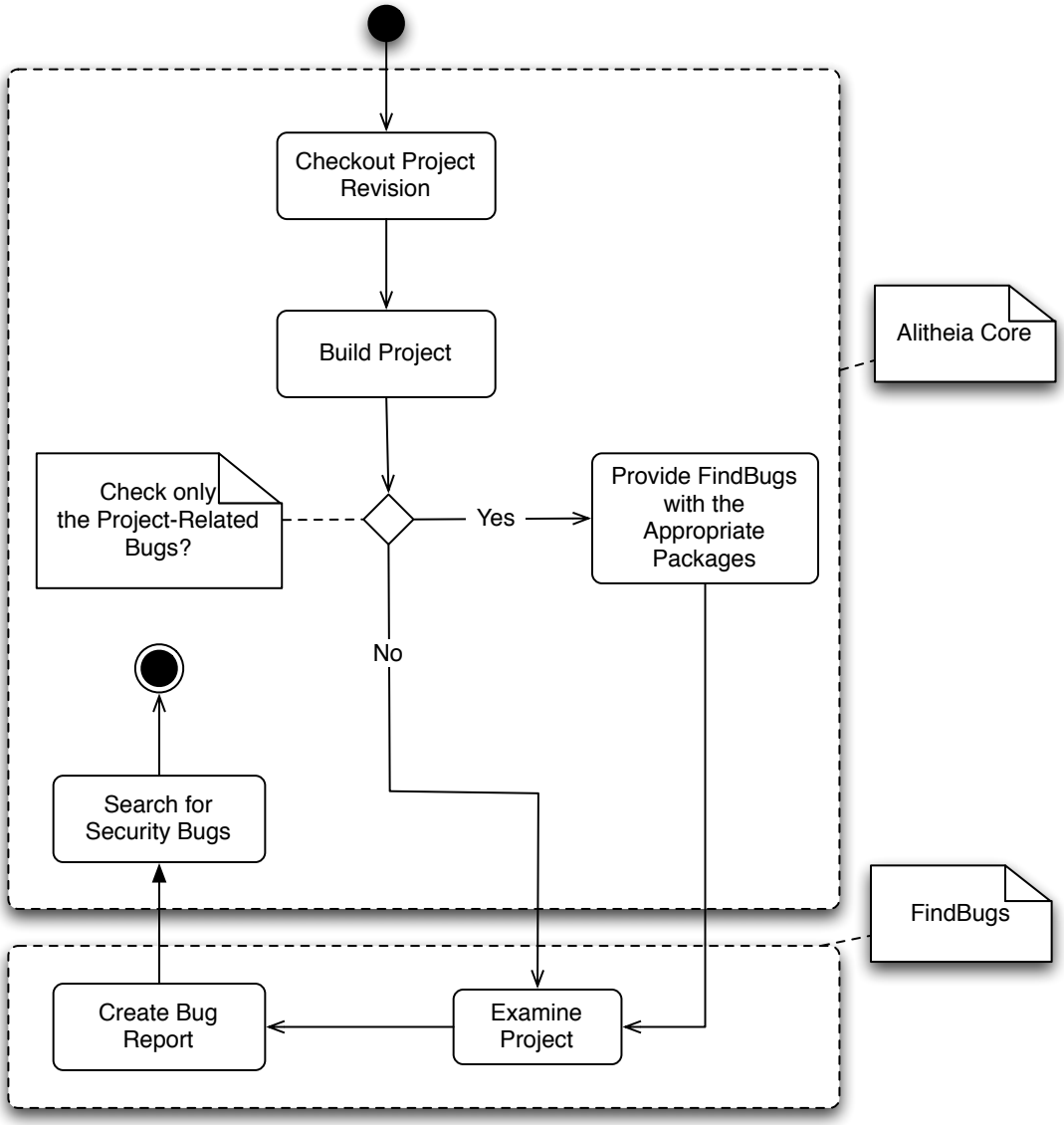


Figure A.1: A state diagram indicating the steps taken by our framework.

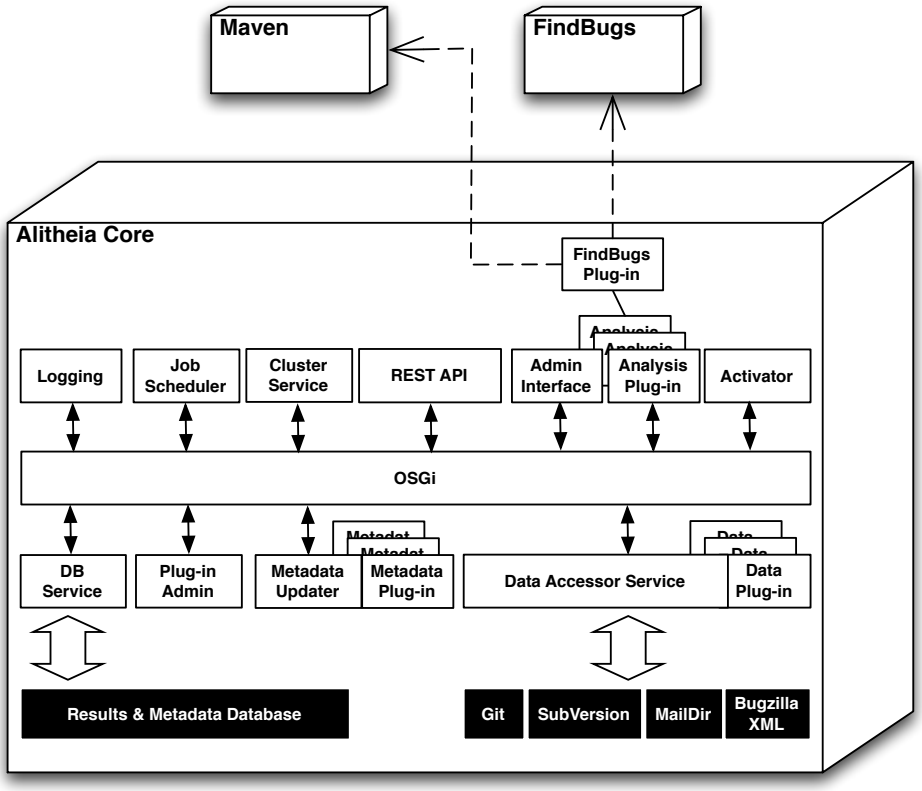
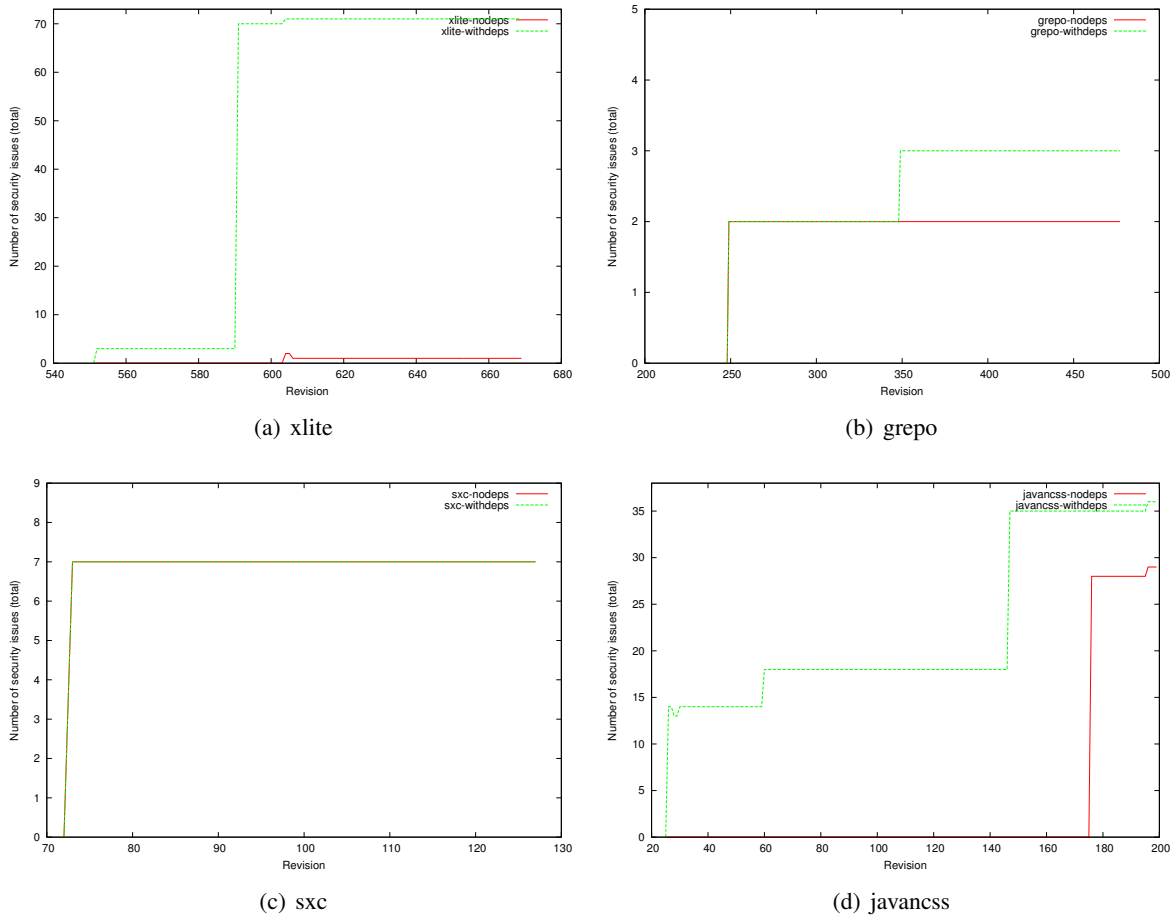


Figure A.2: Alitheia Core and FindBugs integration.



(a) xlite (b) grepo

(c) sxc (d) javancss

Figure A.3: Bug frequency for all four projects.

matate some of these tasks, modern build systems such as Maven[2] include support for resolving and downloading dependencies declared in the project's build file, while they also follow a standard directory structure for code and build artifacts (see Subsection 1.2). The Findbugs plug-in exploited the conventions supported by Maven to automatically build each project and retrieve the generated bytecode archives. For example, it knew that source code is placed into the `src/main/java` directory, while build artifacts are placed under `target/`. It was therefore sufficient to walk the directory structure and find the bytecode archives (JAR files) in order to retrieve the project's (or any sub-project's) package structure and compiled code, respectively.

After a build the Findbugs binary was invoked. In order to examine the bytecode that was created by the sources that belong to the specified project and not by its dependencies, we collected all the corresponding project packages and then we used the `-onlyAnalyze` option of FindBugs to pass them as one parameter. By using the `-xml` option the report that is made contains all the bug descriptions in an XML format. As a result, we could easily parse this report in order to collect the bugs that we are interested in. The bugs were then associated with file revision information that Alitheia Core stored in its database, through path name matching and thus results can be stored with respect to each file version. To speed up searches, the Findbugs plug-in also stores summaries of number of incidents found per project version.

Results We have examined four open source projects that are based on the Maven build system

[2] `http://maven.apache.org/`

namely: *xlite*,[3] *sxc*,[4] *javancss*,[5] and *grepo*.[6] Our experiment included two measurements. First, for every revision, we applied FindBugs only to the bytecode of a specified project. Then for our second measurement, we also included the dependencies of this project. Figure A depicts the results of both measurements for every project. We have selected maven-based projects to automatically build every revision of a project and examine it with FindBugs on the spot. Some of these projects may not look interesting from a security point of view. For instance, *javancss* counts lines of code. Still, all of them deal with untrusted input, thus they could become targets for exploits.

The most interesting observation that we can make is that the security bugs are increasing as projects evolve. This is particularly noteworthy and shows that bugs should be fixed in time to decrease the effort and cost of the security audits after the end of the development process. Another observation is the existance of the *domino effect*. The usage of external libraries introduces new bugs. As we can see in all cases the sum of the security related bugs in the second measurement is bigger or equal than the first one. A mathematical representation of the this could be the following: If *bop* is the variable that represents the sum of the security related bugs of every project for every revision, *boa* the sum of the bugs that also concern the dependencies of the project for every revision and i is the number of a project revision, the following expression stands for every project:

$$\sum_{i=0}^{n} boa \geq \sum_{i=0}^{n} bop \tag{A.1}$$

There are also cases where there is no security bug in the majority of the revisions of a project but there are bugs in the libraries that it includes i.e. in the *javancss* project. Still, there is a case (the *sxc* project) where there are no bugs in the libraries that the project depends on.

The Alitheia Core framework provides ways to check what changes have been made after a commit. By taking advantage of this feature we observe that there are situations where the total bugs are increased after the addition of a new library. For instance, in the 591st revision of the *xlite* project, the 349th revision of the *grepo* project and the 30th revision of the *javancss* project, developers have added new libraries in their project. On the other hand, the number of bugs decreases when developers update a library (for example in the 28th revision of the *javancss* project). Thus, project libraries should always be updated not only because of the additional functionality that they provide, but also for security reasons. In general we can observe how the changes made in third-party software can affect the evolution of a project while its developers are unaware of that.

Another interesting issue regards the bugs themselves. Table A.1, shows for the last revision of every project the security bugs that have been found. As we can see, even after hundreds of revisions there are trivial bugs but there are also bugs that could be severe for the application. For instance, the last revision of *javancss* includes a code fragment that creates an SQL prepared statement from a non-constant string. If this string is not checked properly, an SQL injection attack is prominent. By determining the occurance of a bug for a large number of projects, and by examining all revisions, we could generate the frequency of the appearance of this bug. Such an estimation could be crucial for vulnerability discovery models. The most important observation of the above esperiment was the increase of the bugs as the project evolves, which is not an encouraging observation. Other ovservations included the existance of the domino effect and the dependence of a software project from its libraries.

[3] http://xircles.codehaus.org/projects/xlite
[4] http://xircles.codehaus.org/projects/sxc
[5] http://xircles.codehaus.org/projects/javancss
[6] http://xircles.codehaus.org/projects/grepo

Project Name	**Bug Description** (taken from the FindBugs website)	**Occurance**
javancss	Dm: Hardcoded constant database password	2
javancss	EI: May expose internal representation by returning reference to mutable object	8
javancss	MS: Field isn't final and can't be protected from malicious code	3
javancss	MS: Field should be moved out of an interface and made package protected	4
javancss	MS: Field should be package protected	14
javancss	MS: Field isn't final but should be	4
javancss	SQL: A prepared statement is generated from a nonconstant String	1
sxc	EI: May expose internal representation by returning reference to mutable object	7
xlite	MS: Field should be both final and package protected	1
xlite	EI: May expose internal representation by returning reference to mutable object	8
xlite	MS: Public static method may expose internal representation by returning array	1
xlite	MS: Field should be package protected	1
xlite	MS: Field isn't final but should be	60
grepo	EI: May expose internal representation by returning reference to mutable object	5

Table A.1: Occurences of security bugs in the last revision of every project.

Appendix B

Code Injection Defects in the Cloud Environment

In this experiment, we attempted to explore the security risks associated with virtual images from the public catalogs of a cloud service provider. Our results, together with the results of similar studies [22, 41] indicate that both users and providers may be vulnerable to risks like data loss, unauthorized access and others. In addition, many VMs are susceptible to various kinds of CIAs.

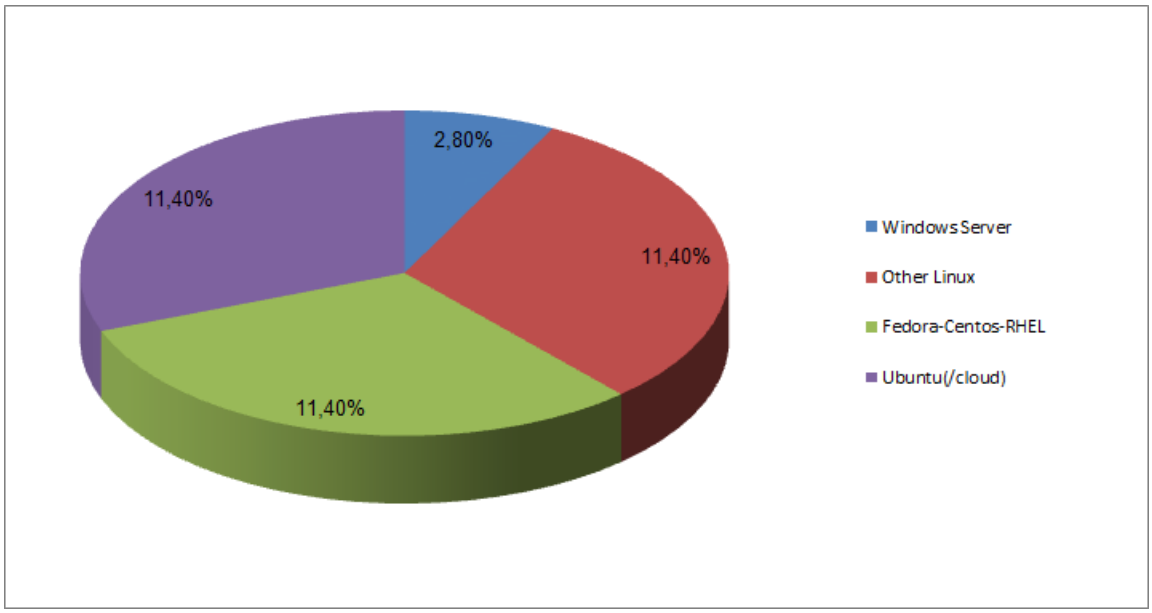


Figure B.1: VM distribution in our experiment.

In particular, we performed a series of penetration tests on a number of virtual machines running different operating systems. All VMs were hosted on the Amazon EC2, which is a part of the Amazon Web Services (AWS) platform. To perform the penetration tests, we used the Tenable Nessus vulnerability scanner. Our methodology included the following steps; first, we retrieved a list of available Amazon machine images. Then we picked a random image, launched it on the cloud and retrieved its IP address. After that, we invoked the Nessus scanner and passed the IP address as a parameter to it. When the test was over, we terminated the image.

In total, we examined 70 VMs (see Figure B.1. The operating systems OS running on these images can be distinguished in four basic categories, namely: Windows Server (14 images), Ubuntu (26 im-

ages), CentOS (9 images) and other Linux OSs (21 images, including Slackware, Arch Linux). Keep in mind that Amazon does not use vanilla distributions of these operating systems but modified distributions that match the requirements of virtual machines. Our first observation was that 22 VMs (10 Ubuntu images, 8 other Linux images, 3 Windows Server images and 1 CentOS) were vulnerable through HTTP methods. These VMs had minimum three vulnerabilities that exploit the HTTP protocol. In addition, the virtual machines of the Windows family present many serious problems with the Microsoft Remote Desktop Protocol (RDP) protocol. Specifically, all images running Windows, except for one, were vulnerable to attacks targeting this protocol. These images had minimum four defects coming from this protocol. Another observation, involved virtual machines with obsolete versions of the Apache Server. Regardless of the operation system, such images were vulnerable to numerous attacks like MITM, XSS and SQL injection (see Figure B.2). This indicates that installing the latest version of the Apache Server software could solve the above problems. In general, as you can see in the figure below, the VMs were vulnerable to different types of critical attacks. Most defects found on the VMs, could lead mostly to MITM and Denial-of-Service (DoS) attacks. Such attacks could be avoided by configuring SSL protocol settings properly. For example, in many cases there were mistakes in the computers name and some certificates had expired. From the 70 images only 26 turned out to be secure, namely: 8 CentOS VMs, 8 Ubuntu, 8 VMs with other Linux OSs and 2 VMs with the Windows Server OS.

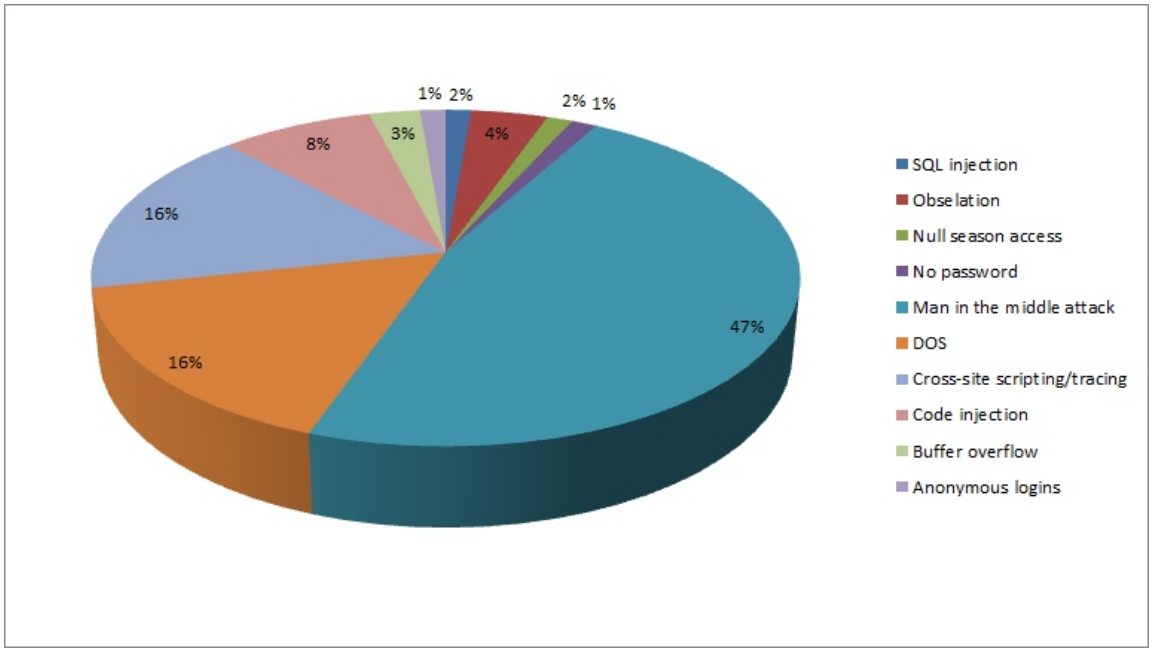


Figure B.2: Defect distribution among the analyzed VMs.

Appendix C

Cyberdiversity: Seeking for CIA-Persistent Monocultures

Monocultures can be seen as a population that consists of identical members that belong to the same organism. Even if monocultures are very rare in nature, in cyberspace they seem to flourish. A collection of identical computer platforms is easier, therefore cheaper to manage, because, for example, they will share the same configuration while maintaining minimum user training costs. In addition, interoperability and standardization is easier to be achieved and maintained in a monoculture [29]. However, these advantages can become at the same time disadvantages. In a monoculture, when a piece of malware manage to intrude in one member of the monoculture, in a similar way it can affect the rest of them because all share the same vulnerabilities [119]. Currently, there is a great controversy whether the benefits of cyberdiversity could be overshadowed by its side-effects [16, 29, 119, 171].

In this experiment we aimed to measure the diversity that exists in the software that is used today. For the collection of the needed data, we developed a system based on the client-server model. A clientside application will run at users' computers and send specific data back to our server. This data is stored in a database that is used during the measurements.

Type of the needed data Our primary goal was to measure the diversity of software. So we tried to find a way to compare different instances of the same files. By different instances of the same files, we mean different instances of a particular file (e.g. "`foo.exe`") that reside in different computers. For example, the file "`foo.exe`" that resides in computer A and the file "`foo.exe`" that resides in computer B are two different instances of the same file. Cyberdiversity introduction techniques apply transformations to software that lead to the production of files, which have differences between them but share the same functionality. Thus, the files that were collected were the binary files. Furthermore, we also collected the files that have external dependencies with the binaries. These files are the static and dynamic libraries. Specifically, for Windows our client-side application collected executable and library files (`.exe` and `.dll`), for UNIX-like systems apart from the executable files, it collected the static libraries (`.a` files) and the dynamic libraries (`.so` files). Additionally, for the Mac OS X operating system we also collected the Mac-oriented library files (`.dylib` libraries). In this case though, a specific problem arised. It was impossible to collect and send back these files "as they were". In this case we would need enormous storage space to store the files and excessive bandwidth to send them back to the server. The solution was to send back the MD5 hash of these files instead. Hence, no useful information was lost because we wanted to examine if the files were exactly the same. Given that the hash produced from the MD5 algorithm was unique for each input with extremely high possibility, we could make the desirable comparison.

System Description Our client-side application ran only with the users permission and it had three main tasks to complete. The first was to collect information about the operating system of each computer that it was running at. This information refered to the version of the operating system. Secondly,

it had to scan each computer in order to collect the data, which was used for the extraction of our results. Its last task was to assure that they will run only one time at every computer. This was necessary because it ensured that the results would not be falsified. To accomplish that, our client-side application retrieved a signature from each computer. This signature was the machine's Security Identifier (SID) for Windows and the Media Access Control (MAC) address for UNIX-like systems. Our server-side scheme had also three tasks to accomplish. Its first task was to notify the client-side applications if they have already ran at a computer in order not to run again. This was achieved by comparing the signatures that were saved in our servers database together with the corresponding signature of each computer. Its second task was to store the data that was collected from the client-side applications to the server-side database. The final task of the scheme was to calculate the similarity percentage of every computer with all the computers that have already been visited by the client application.

Facebook application To collect as many data as possible, a Facebook application was developed. This application consisted of two parts: A web application running through the Facebook website and modified client-side applications that could connect with Facebook servers and consequently with the web application. Users could ran the Facebook client-side application at their computers and then they could visit the web application in order to compare their results with their Facebook friends that have also ran the client.

Cyberdiversity Metrics Before the metric definitions, we need to define the term "variant of a file". Cyberdiversity introduction techniques produce different variations of a given file but these variations share exactly the same functionality. For example, assume that the correspondent executable file for a program "`foo`" is "`foo.exe`". If "`foo.exe`" is produced with the use of cyberdiversity introduction techniques, different variations of this file will be produced. These variations will have internal differences between them but all of them will share the same functionality under the same name ("`foo.exe`"). We call these variations: "variants of a file". We also defined previously the term "instance of a file". The first metric calculates the probability of a successful targeted attack, if the attack targets the most frequent variant of a file. Hence we calculate the percentage of the computers that are affected in the worst case, where the attack affects the most frequent variant of a file. The smaller the probability, the more cyberdiverse the file is and thus, the rate of the propagation of the attack will be slower. This probability is calculated as follows:

$$\rho = \nu/\tau \tag{C.1}$$

Where ρ is the probability, ν stands for the number of instances of the most frequent variant of a given file and τ represents the total number of instances of that file. The second metric is the ratio of the number of variants to the total number of instances of all the variants of a file. This ratio shows if a file has enough variants. This in turn, indicates that this file is diverse. The bigger the ratio is for a file, the more variants this file has and as a result more attacks are needed to compromise all the instances of this file. The smaller the ratio is for a file, the more instances are accumulated in every variant of a file, thus the bigger will be the number of instances that could be compromised by a single attack. This ratio is calculated as follows:

$$ratio = \beta/\tau \tag{C.2}$$

Where β represents the number of the variants a given file has and τ represents the total number of instances of that file. The third metric is the coefficient of variation (CV) of the variants of each file. CV is the ratio of the standard deviation to the mean. In our case, CV shows how the instances of a file are distributed amongst the variants of a file. If the CV is small enough, this means that the instances are distributed uniformly. In this case, the probability of a single attack that compromises a large number of instances of a file is significant.

Measurements The sample collected for our study consists of data retrieved from 214 computers. 176 of these computers ran Windows as their operating system, 27 ran Linux, 10 ran Mac OS X and

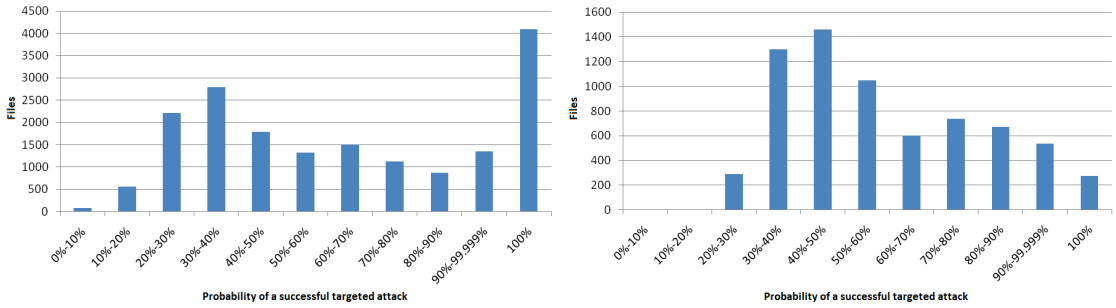


Figure C.1: Results based on the probability of a successful targeted attack for Windows (left) and Linux (right).

one ran FreeBSD. From these computers we collected 205,221 files, which altogether have 1,309,834 instances. 111636 of these files have only one instance, hence they cant take part into the process of the results. For the extraction of the results, we take into consideration only the files that have 10 instances or more, in order to have more accurate results. We present our results in Figures C.1, C.2 and C.3. The column of every diagram depicts the number of the processed files while every row depends on the corresponding metric.

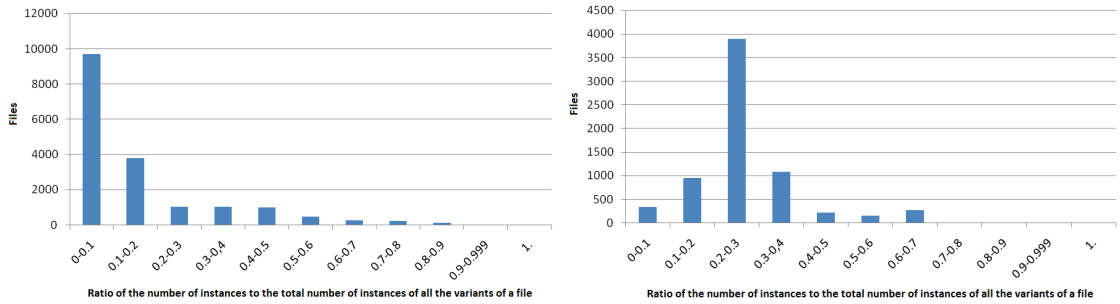


Figure C.2: Results based on the ratio of the number of instances of all variants of a file for Windows (left) and Linux (right).

The results based on C.1 are shown in Figure C.1. The diagrams show that the probability for the biggest part (58% for Windows and 55.7% for Linux) of the files is over 50% and for a minimal part (3.7% for Windows and 0% for Linux) of the files the possibility is under 20%. This shows that there is not enough cyberdiversity. If cyberdiversity was sufficient, the percentage of files with under 20% probability should be bigger and there should be a little amount of files with over 50% probability. Even if there are differences between the Windows-diagrams and the Linux-diagrams, the main conclusion for both is that the diversity is not sufficient. The most important difference is that there are fewer files with exactly 100% probability in Linux. These files are not cyberdiverse and thus there are more Windows files that have no diversity. In Figure C.2 the results based on C.2 are presented. The diagrams show that for the biggest part of the files the ratio is small (under 0.3). This indicates that with a single attack a malicious user could compromise a great number of instances of a file. Here there is a distinct difference between Windows and Linux. For Linux there is congestion in the range of 0.2 to 0.3 while for Windows there is congestion in the range of 0 to 0.1. This can be explained by the fact that there are fewer files that have no diversity. However, this difference is not so important and this does not change the general conclusion that there cyberdiversity is not sufficient.

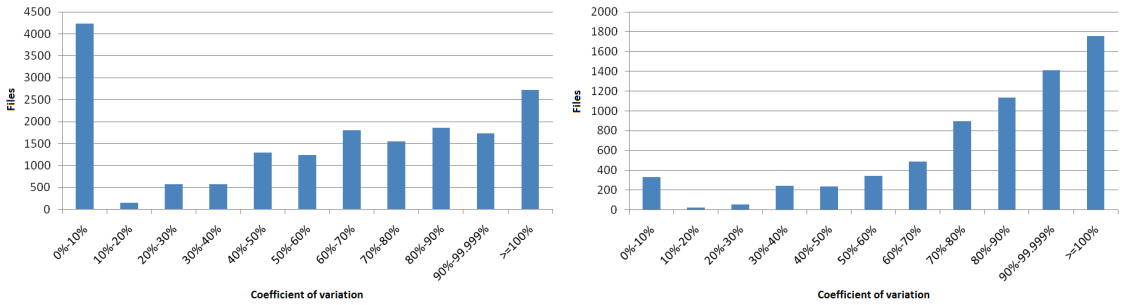


Figure C.3: Results based on the coefficient of variation for Windows (left) and Linux (right).

The results based on CV are shown in Figure C.3. Given that the smaller the CV is, the more uniformly the instances are distributed. Here we observe that most of the files arent distributed uniformly and most of them are concentrated in the biggest percentages. The concentration noticed in the first column is almost exclusively the result of the fact that the files that correspond in this column have no diversity and this column should not be considered in the extraction of the results because here we evaluate how uniform the diversity is and the files that correspond to the first column have no diversity. Finally, to emphasize the lack of diversion and its extend, we have also included the files that have no diversity at all.

Appendix D

Publication List

D.1 Journal Articles

1. Dimitris Mitropoulos and Diomidis Spinellis. Fatal injection: A survey of modern code injection attack countermeasures. *PeerJ Computer Science*. pages e136, November 2017.

2. Dimitris Mitropoulos, Kostantinos Stroggylos, Diomidis Spinellis, and Angelos D. Keromytis. How to train your browser: Preventing XSS attacks using contextual script fingerprints. ACM *Transactions on Privacy and Security*, 19(1):2:1—2:31, July 2016.

3. Dimitris Mitropoulos, Vassilios Karakoidas, Panagiotis Louridas, and Diomidis Spinellis. Countering Code Injection Attacks: A Unified Approach. *Information Management and Computer Security*, 19(3):177—194, 2011. **Note**: "*Highly Commended Paper*" distinction by the Emerald Publishers. The award was given by the journal's editorial board to three papers as part of the "*Literati Network Awards for Excellence 2012"*.

4. Dimitris Mitropoulos and Diomidis Spinellis. SDriver: Location-Specific Signatures Prevent SQL Injection Attacks. *Computers & Security*, 28:121—129, May 2009.

D.2 Conference Articles

1. Dimitrios Mitropoulos, Georgios Gousios, Panagiotis Papadopoulos, Vasilios Karakoidas, Panos Louridas, and Diomidis Spinellis. The vulnerability dataset of a large software ecosystem. In *Proceedings of the 3rd International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security* (BADGERS *2014*). IEEE Computer Society, September 2014.

2. Dimitris Mitropoulos, Vassilios Karakoidas, Panos Louridas, Georgios Gousios, and Diomidis Spinellis. The Bug Catalog of the Maven Ecosystem. In MSR *'14: Proceedings of the 2014 International Working Conference on Mining Software Repositories, Data Track*, pages 372365. ACM, May 2014.

3. Dimitris Mitropoulos, Vassilios Karakoidas, Panos Louridas, Georgios Gousios, and Diomidis Spinellis. Dismal Code: Studying the Evolution of Security Bugs. In *Proceedings of the* LASER *Workshop 2013, Learning from Authoritative Security Experiment Results*, pages 37–48. Usenix Association, October 2013. **Note**: Awarded the LASER *Workshop Scholarship*. The award was given by the organizing committee of the LASER Workshop.

4. Dimitris Mitropoulos, Georgios Gousios, and Diomidis Spinellis. Measuring the Occurrence of Security-Related Bugs through Software Evolution. *In* PCI *2012: Proceedings of 16th Panhellenic Conference on Informatics (*PCI *2012)*, pages 117–122. IEEE Computer Society, 2012.

5. Konstantinos Kravvaritis, Dimitris Mitropoulos, and Diomidis Spinellis. Cyberdiversity: Measures and Initial Results. In Costas Vassilakis and Nikolaos Tselikas, editors, PCI *2010: Proceedings of 14th Panhellenic Conference on Informatics (*PCI *2010)*, pages 135–140. IEEE Computer Society, September 2010.

6. Dimitris Mitropoulos, Vassilios Karakoidas, and Diomidis Spinellis. Fortifying Applications Against XPath Injection Attacks. *In A. Poulymenakou, N. Pouloudi, and K. Pramatari, editors, 4th Mediterranean Conference on Information Systems*, pages 1169–1179, September 2009.

D.3 Magazine Articles

1. Dimitris Mitropoulos. Security Bugs in Large Software Ecosystems. XRDS*: Crossroads, The* ACM *Magazine for Students*, 20(2):15–16, 2013.

2. Dimitris Mitropoulos. Data Security in the Cloud Environment. XRDS*: Crossroads, The* ACM *Magazine for Students*, 19(3):11–11, 2013.

3. Dimitris Mitropoulos. Fatal Injection: the Server's Side. XRDS*: Crossroads, The* ACM *Magazine for Students*, 19(2):12–14, 2012

4. Dimitris Mitropoulos. How Secure is Your Software? XRDS*: Crossroads, The* ACM *Magazine for Students*, 19(1):11–13, 2012.

Acronyms

ACM Association for Computing Machinery

AJAX Asynchronous JavaScript and XML

AST Abstract Syntax Tree

API Application Programming Interface

ARP Address Resolution Protocol

AWS Amazon Web Services

BCEL Bytecode Engineering Library

CAPTCHA Completely Automated Public Turing test to tell Computers and Humans Apart

CDN Content Distribution Network

CFG Control-Flow Graph

CFI Control-Flow Integrity

CIA Code Injection Attack

CGI Common Gateway Interface

CRM Customer Relationship Management

CPU Central Processing Unit

CVS Concurrent Version System

CSP Content Security Policy

CSS Cascading Style Sheets

DBMS Database Management System

DFS Depth First Search

DISA Defense Information Systems Agency

DNS Domain Name System

DOM Document Object Model

DoS Denial-of-Service

DRE Defect Removal Effectiveness

DSL Domain Specific Language

ECT Electronic Communication Trail

EC2 Elastic Compute Cloud

FP False Positives

FN False Negatives

FSF Free Software Foundation

GB Gigabytes

GPL General-Purpose Language

GQM Goal Question Metric

GUI Graphical User Interface

HV Halstead Volume

HTML HyperText Markup Language

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

ICSE International Conference on Software Engineering

IEEE Institute of Electical and Elecronic Engineering

IEC International Electrotechnical Commission

IF Information Flow

IL Intermediate Language

IP Internet Protocol

IM Instant Messaging

IRC Instant Relay Chat

ISR instruction Set Randomization

ISO International Organization for Standardization

JAR Java Archive

JDBC Java Database Connectivity

JDK Java Development Kit

JIT Just In Time

JS JavaScript

JSON JavaScript Object Notation

JSONP JSON-with-padding

JSP JavaServer Pages

LDAP Lightweight Directory Access Protocol

LOC Lines of Code

MAC Media Access Control

MIME Multipurpose Internet Mail Extensions

MITM Man-in-the-Middle

MCC McCabe Cyclomatic Complexity

MFSA Mozilla Firefox-related Security Advisories

MSR Mining Software Repositories

MTTR Mean Time To Repear

MT Multi-Threaded

MARC Mail Archive

NASA National Aeronautics and Space Administration

NAS Network-Attached Storage

NFRs Non-Functional Requirements

NOC Number of Children

ODBC Open Database Connectivity

OOP Object-Oriented Programming

ORM Object Relational Mapping

OSS Open Source Software

OS Operating System

OWASP The Open Web Application Security Project

PCI DSS Payment Card Industry Security Standards Council

POV Point of View

PQL Program Query Language

RAM Random-Access Memory

RDBMS Relational Database Management System

RDP Remote Desktop Protocol

REST Representational State Transfer

RFC Response For Class

RHDB Release History DataBase

RCS Revision Control System

SCM Software Configuration Management

SCCS Source Code Control System

SF SourceForge

SID Security Identifier

SMS Short Message Service

SSL Secure Sockets Layer

SNA Social Network Analysis

SNMP Simple Network Management Protocol

SOHO Small Office/Home Office

SQL Structured Query Language

SQO-OSS Software Quality Observatory for Open Source Software

SVN Subversion

TAN Transaction Authentication Number

TAR Tape Archive

TLS Transport Layer Security

TOCTOU Time Of Check, Time Of Use

TP True Positives

TN True Negatives

UML Unified Modeling Language

UMTS Universal Mobile Telecommunication Standard

URL Universal Resource Locator

VM Virtual Machine

VCS Version Control System

XCS Cross-Channel Scripting

XML eXtensible Markcup Language

XSS Cross-Site Scripting

XSLT Extensible Stylesheet Language Transformations

Bibliography

[1] *A Comparison of Publicly Available Tools for Static Intrusion Prevention*, Karlstad, Sweden, November 2002.

[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, pages 340–353, New York, NY, USA, 2005. ACM. ISBN 1-59593-226-7.

[3] Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA, 2nd edition, 1996. ISBN 0262011530.

[4] Marwan Abi-Antoun, Daniel Wang, and Peter Torr. Checking threat modeling data flow diagrams for implementation conformance and security. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 393–396, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-882-4.

[5] Ashish Aggarwal and Pankaj Jalote. Integrating static and dynamic analysis for detecting vulnerabilities. In *COMPSAC '06: Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, pages 343–350, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2655-1.

[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. ISBN 0321486811.

[7] Vasilios Almaliotis, Alexandros Loizidis, Panagiotis Katsaros, Panagiotis Louridas, and Diomidis Spinellis. Static program analysis for java card applets. In *Proceedings of the 8th IFIP WG 8.8/11.2 International Conference on Smart Card Research and Advanced Applications*, CARDIS '08, pages 17–31, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-85892-8.

[8] Anne Anderson. A comparison of two privacy policy languages: Epal and xacml. Technical report, Mountain View, CA, USA, 2005.

[9] Paul Anderson and Mark Zarins. The CodeSurfer software understanding platform. In *Proceedings of the 13th International Workshop on Program Comprehension*, IWPC '05, pages 147–148, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2254-8.

[10] Ross J. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 2001. ISBN 0471389226.

[11] C. Anley. *Advanced SQL Injection in SQL Server Applications*. Next Generation Security Software Ltd., 2002.

[12] Jason Ansel, Petr Marchenko, Úlfar Erlingsson, Elijah Taylor, Brad Chen, Derek L. Schuff, David Sehr, Cliff L. Biffle, and Bennet Yee. Language-independent sandboxing of just-in-time compilation and self-modifying code. *SIGPLAN Not.*, 46(6):355–366, June 2011. ISSN 0362-1340.

[13] Nuno Antunes, Nuno Laranjeiro, Marco Vieira, and Henrique Madeira. Effective detection of SQL/XPath injection vulnerabilities in web services. In *Proceedings of the 2009 IEEE International Conference on Services Computing*, SCC '09, pages 260–267, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3811-2.

[14] Shay Artzi, Julian Dolby, Simon Holm Jensen, Anders Møller, and Frank Tip. A framework for automated testing of javascript web applications. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 571–580, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0445-0.

[15] Elias Athanasopoulos, Vasilis Pappas, Antonis Krithinakis, Spyros Ligouras, Evangelos P. Markatos, and Thomas 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, Berkeley, CA, USA, 2010. USENIX Association.

[16] D. Aucsmith. Monocultures are hard to find in practice. *IEEE Security and Privacy*, 1(6):15–16, November 2006. ISSN 1540-7993.

[17] Andrea Avancini and Mariano Ceccato. Comparison and integration of genetic algorithms and dynamic symbolic execution for security testing of cross-site scripting vulnerabilities. *Inf. Softw. Technol.*, 55(12):2209–2222, December 2013. ISSN 0950-5849.

[18] Nathaniel Ayewah and William Pugh. A report on a survey and study of static analysis users. In *Proceedings of the 2008 workshop on Defects in large software systems*, DEFECTS '08, pages 1–5, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-051-7.

[19] Nathaniel Ayewah and William Pugh. The Google FindBugs fixit. In *Proceedings of the 19th international symposium on Software testing and analysis*, ISSTA '10, pages 241–252, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-823-0.

[20] Nathaniel Ayewah, William Pugh, J. David Morgenthaler, John Penix, and YuQian Zhou. Evaluating static analysis defect warnings on production software. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '07, pages 1–8, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-595-3.

[21] Dejan Baca, Bengt Carlsson, and Lars Lundberg. Evaluating the cost reduction of static code analysis for software security. In *Proceedings of the third ACM SIGPLAN workshop on Programming languages and analysis for security*, PLAS '08, pages 79–88, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-936-4.

[22] Marco Balduzzi, Jonas Zaddach, Davide Balzarotti, Engin Kirda, and Sergio Loureiro. A security analysis of amazon's elastic compute cloud service. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 1427–1434, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-0857-1.

[23] Adam Barth, Juan Caballero, and Dawn Song. Secure content sniffing for web browsers, or how to stop papers from reviewing themselves. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, pages 360–371, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3633-0.

[24] Kent Beck and Erich Gamma. Test infected: Programmers love writing tests. *Java Report*, 3(7): 37–50, July 1998.

[25] Shoham Ben-David, Cindy Eisner, Daniel Geist, and Yaron Wolfsthal. Model checking at IBM. *Form. Methods Syst. Des.*, 22(2):101–108, March 2003. ISSN 0925-9856.

[26] Michael Benedikt and Christoph Koch. XPath leashed. *ACM Comput. Surv.*, 41(1):1–54, 2008. ISSN 0360-0300.

[27] Michael Benedikt, Wenfei Fan, and Gabriel Kuper. Structural properties of XPath fragments. *Theor. Comput. Sci.*, 336(1):3–31, 2005. ISSN 0304-3975.

[28] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker blast: Applications to software engineering. *Int. J. Softw. Tools Technol. Transf.*, 9(5): 505–525, October 2007. ISSN 1433-2779.

[29] Kenneth P. Birman and Fred B. Schneider. The monoculture risk put into context. *IEEE Security and Privacy*, 7(1):14–17, 2009.

[30] Prithvi 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, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-70541-3.

[31] William Blum and C.-H. Luke Ong. The safe lambda calculus. In *Proceedings of the 8th International Conference on Typed Lambda Calculi and Applications*, TLCA'07, pages 39–53, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 978-3-540-73227-3.

[32] Hristo Bojinov, Elie Bursztein, and Dan Boneh. XCS: cross channel scripting and its impact on web applications. In *CCS '09: Proceedings of the 16th ACM conference on Computer and communications security*, pages 420–431, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-894-0.

[33] A. S. Boujarwah, K. Saleh, and J. Al-Dallal. Testing Java programs using dynamic data flow analysis. In *Proceedings of the 2000 ACM Symposium on Applied Computing - Volume 2*, SAC '00, pages 725–727, New York, NY, USA, 2000. ACM. ISBN 1-58113-240-9.

[34] S. Boyd and A. Keromytis. SQLrand: Preventing SQL injection attacks. In M. Jakobsson, M. Yung, and J. Zhou, editors, *Proceedings of the 2nd Applied Cryptography and Network Security (ACNS) Conference*, pages 292–304. Springer-Verlag, 2004. Lecture Notes in Computer Science Volume 3089.

[35] Mehran Bozorgi, Lawrence K. Saul, Stefan Savage, and Geoffrey M. Voelker. Beyond heuristics: learning to classify vulnerabilities and predict exploits. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '10, pages 105–114, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0055-1.

[36] Sergey Bratus, Michael E. Locasto, Len Sassaman Meredith L. Patterson, and Anna Shubina. Exploit programming: From buffer overflows to "Weird Machines" and theory of computation. 36(6):13–21, December 2011. ISSN 1044-6397.

[37] Martin Bravenboer, Eelco Dolstra, and Eelco Visser. Preventing injection attacks with syntax embeddings. In *GPCE '07: Proceedings of the 6th international conference on Generative programming and component engineering*, pages 3–12, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-855-8.

[38] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Comput. Netw. ISDN Syst.*, 30(1-7):107–117, April 1998. ISSN 0169-7552.

[39] Mason Brown and Alan Paller. Secure software development: Why the development world awoke to the challenge. *Inf. Secur. Tech. Rep.*, 13(1):40–43, 2008. ISSN 1363-4127.

[40] Gregory Buehrer, Bruce W. Weide, and Paolo A. G. Sivilotti. Using parse tree validation to prevent sql injection attacks. In *Proceedings of the 5th International Workshop on Software Engineering and Middleware*, SEM '05, pages 106–113, New York, NY, USA, 2005. ACM. ISBN 1-59593-205-4.

[41] Sven Bugiel, Stefan Nürnberger, Thomas Pöppelmann, Ahmad-Reza Sadeghi, and Thomas Schneider. AmazonIA: when elasticity snaps back. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 389–400, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0948-6.

[42] Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S. Păsăreanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. Symbolic execution for software testing in practice: Preliminary assessment. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 1066–1071, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0445-0.

[43] Brendon Cahoon and Kathryn S. McKinley. Data flow analysis for software prefetching linked data structures in java. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, PACT '01, pages 280–291, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7695-1363-8.

[44] Rich Cannings, Himanshu Dwivedi, and Zane Lackey. *Hacking Exposed Web 2.0: Web 2.0 Security Secrets and Solutions (Hacking Exposed)*. McGraw-Hill Osborne Media, 2007. ISBN 0071494618, 9780071494618.

[45] CERT. CERT vulnerability note VU282403. Online `http://www.kb.cert.org/vuls/id/282403`, 2002. Accessed, January 7th, 2007.

[46] Walter Chang, Brandon Streiff, and Calvin Lin. Efficient and extensible security enforcement using dynamic data flow analysis. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, pages 39–50, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-810-7.

[47] Hao Chen and David Wagner. MOPS: An infrastructure for examining security properties of software. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, CCS '02, pages 235–244, New York, NY, USA, 2002. ACM. ISBN 1-58113-612-9.

[48] Hao Chen, Drew Dean, and David Wagner. Model checking one million lines of C code. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium (NDSS)*, pages 171–185. The Internet Society, 2004.

[49] Karl Chen and David Wagner. Large-scale analysis of format string vulnerabilities in Debian Linux. In *Proceedings of the 2007 Workshop on Programming Languages and Analysis for Security*, PLAS '07, pages 75–84, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-711-7.

[50] Brian Chess and Gary McGraw. Static analysis for security. *IEEE Security and Privacy*, 2(6): 76–79, 2004. ISSN 1540-7993.

[51] Brian Chess and Jacob West. *Secure programming with static analysis*. Addison-Wesley Professional, 2007. ISBN 9780321424778.

[52] Adam Chlipala. Static checking of dynamically-varying security policies in database-backed applications. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–, Berkeley, CA, USA, 2010. USENIX Association.

[53] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 258–269, New York, NY, USA, 2002. ACM. ISBN 1-58113-463-0.

[54] Sandy Clark, Stefan Frei, Matt Blaze, and Jonathan Smith. Familiarity breeds contempt: The honeymoon effect and the role of legacy code in zero-day vulnerabilities. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 251–260, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0133-6.

[55] Edmund Clarke, Anubhav Gupta, Himanshu Jain, and Helmut Veith. Verified software: Theories, tools, experiments. chapter Model Checking: Back and Forth Between Hardware and Software, pages 251–255. Springer-Verlag, Berlin, Heidelberg, 2008. ISBN 978-3-540-69147-1.

[56] Edmund M. Clarke, E. Allen Emerson, and Joseph Sifakis. Model checking: Algorithmic verification and debugging. *Commun. ACM*, 52(11):74–84, November 2009. ISSN 0001-0782.

[57] W.R. Cook and S. Rai. Safe query objects: statically typed objects as remotely executable queries. In *ICSE 2005: 27th International Conference on Software Engineering*, pages 97–106, 2005. doi: 10.1109/ICSE.2005.1553552.

[58] Ricardo Corin and Felipe Andrés Manzano. Taint analysis of security code in the KLEE symbolic execution engine. In *Proceedings of the 14th International Conference on Information and Communications Security*, ICICS'12, pages 264–275, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-34128-1.

[59] Crispin Cowan. Software security for open-source systems. *IEEE Security and Privacy*, 1(1): 38–45, 2003. ISSN 1540-7993.

[60] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th conference on USENIX Security Symposium - Volume 7*, SSYM'98, pages 5–5, Berkeley, CA, USA, 1998. USENIX Association.

[61] Sajal K. Das, Krishna Kant, and Nan Zhang. *Handbook on Securing Cyber-Physical Critical Infrastructure*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012. ISBN 0124158153, 9780124158153.

[62] Willem De Groef, Dominique Devriese, Nick Nikiforakis, and Frank Piessens. Flowfox: a web browser with flexible and precise information flow control. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 748–759, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1651-4.

[63] Dorothy E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, May 1976. ISSN 0001-0782.

[64] Dorothy Elizabeth Robling Denning. An intrusion detection model. *IEEE Transactions on Software Engineering*, 13(2):222–232, February 1987.

[65] Mohan Dhawan and Vinod Ganapathy. Analyzing information flow in JavaScript-based browser extensions. In *Proceedings of the 2009 Annual Computer Security Applications Conference*, ACSAC '09, pages 382–391, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3919-5.

[66] Mohan Dhawan, Chung-chieh Shan, and Vinod 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, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-827-8.

[67] S. Di Paola and G. Fedon. Subverting Ajax. In *Proceedings of the 23rd CCC Conference*, 2006.

[68] Jing Dong, Tu Peng, and Yajing Zhao. Model checking security pattern compositions. In *Proceedings of the Seventh International Conference on Quality Software*, QSIC '07, pages 80–89, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-3035-4.

[69] R. Kent Dybvig. *The Scheme Programming Language*. MIT Press, fourth edition, 2009.

[70] Nigel Edwards and Liqun Chen. An historical examination of open source releases and their vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 183–194, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1651-4.

[71] Manuel Egele, Peter Wurzinger, Christopher Kruegel, and Engin Kirda. Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. In *DIMVA '09: Proceedings of the 6th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 88–106, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-02917-2.

[72] Andrew Eisenberg and Jim Melton. SQLJ Part 1: SQL routines using the Java programming language. *SIGMOD Rec.*, 28(4):58–63, 1999. ISSN 0163-5808.

[73] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. A study of android application security. In *Proceedings of the 20th USENIX conference on Security*, SEC'11, pages 21–21, Berkeley, CA, USA, 2011. USENIX Association.

[74] Dawson Engler and Ken Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 237–252, New York, NY, USA, 2003. ACM. ISBN 1-58113-757-5.

[75] Sebastian Erdweg. *Extensible Languages for Flexible and Principled Domain Abstraction*. PhD thesis, Philipps-Universitat Marburg, 2013.

[76] Sebastian Erdweg, Lennart C.L. Kats, Tillmann Rendel, Christian Kästner, Klaus Ostermann, and Eelco Visser. Library-based model-driven software development with SugarJ. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, SPLASH '11, pages 17–18, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0942-4.

[77] Úlfar Erlingsson, Benjamin Livshits, and Yinglian 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, Berkeley, CA, USA, 2007. USENIX Association.

[78] David Evans and David Larochelle. Improving security using extensible lightweight static analysis. *IEEE Softw.*, 19(1):42–51, 2002. ISSN 0740-7459.

[79] Michael Fagan. Design and code inspections to reduce errors in program development. pages 575–607, 2002.

[80] Maydene Fisher, Jon Ellis, and Jonathan Bruce. *JDBC API Tutorial and Reference*. Addison Wesley, 3^rd edition, 2003.

[81] David Flanagan. *JavaScript: The Definitive Guide*. O'Reilly Media, Inc., 2006. ISBN 0596101996.

[82] Stephanie Forrest, Steven A. Hofmeyr, and Anil Somayaji. Computer immunology. *Commun. ACM*, 40(10):88–96, October 1997. ISSN 0001-0782.

[83] Lloyd D. Fosdick and Leon J. Osterweil. Data flow analysis in software reliability. *ACM Comput. Surv.*, 8(3):305–330, September 1976. ISSN 0360-0300.

[84] Aurélien Francillon and Claude 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, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-810-7.

[85] Jack Franklin. *Beginning jQuery*. Apress, Berkely, CA, USA, 1st edition, 2013. ISBN 1430249323, 9781430249320.

[86] Douglas G. Fritz and Robert G. Sargent. An overview of hierarchical control flow graph models. In *Proceedings of the 27th Conference on Winter Simulation*, WSC '95, pages 1347–1355, Washington, DC, USA, 1995. IEEE Computer Society. ISBN 0-7803-3018-8.

[87] Xiang Fu and Kai Qian. SAFELI: SQL injection scanner using symbolic execution. In *Proceedings of the 2008 Workshop on Testing, Analysis, and Verification of Web Services and Applications*, TAV-WEB '08, pages 34–39, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-053-1.

[88] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Indianapolis, IN 46920, 1994. ISBN 0201633612.

[89] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Adding rigorous statistics to the Java benchmarker's toolbox. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, OOPSLA '07, pages 793–794, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-865-7.

[90] Carl Gould, Zhendong Su, and Premkumar Devanbu. Static checking of dynamically generated queries in database applications. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 645–654, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2163-0.

[91] Georgios Gousios and Diomidis Spinellis. Alitheia Core: An extensible software quality monitoring platform. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 579–582, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-3453-4.

[92] Johan Gregoire, Koen Buyens, Bart De Win, Riccardo Scandariato, and Wouter Joosen. On the secure software development process: CLASP and SDL compared. In *SESS '07: Proceedings of the Third International Workshop on Software Engineering for Secure Systems*, page 1, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2952-6.

[93] Matthew Van Gundy and Hao 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)*, San Diego, CA, 2009.

[94] Vivek Haldar, Deepak Chandra, and Michael Franz. Dynamic taint propagation for Java. In *ACSAC '05: Proceedings of the 21st Annual Computer Security Applications Conference*, pages 303–311, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2461-3. doi: 10.1109/CSAC.2005.21.

[95] W. G. 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*, pages 174–183. ACM Press, November 2005.

[96] W. G. 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*, pages 174–183. ACM Press, November 2005.

[97] W. G. Halfond and A. Orso. Preventing SQL injection attacks using AMNESIA. In *ICSE 2006: Proceedings of the 28th International Conference on Software Engineering*, pages 795–798. ACM Press, May 2006.

[98] William G. J. Halfond and Alessandro Orso. Combining static analysis and runtime monitoring to counter SQL-injection attacks. In *WODA '05: Proceedings of the Third International Workshop on Dynamic Analysis*, pages 1–7, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-126-0.

[99] William G.J. Halfond, Jeremy Viegas, and Alessandro Orso. A classification of SQL-injection attacks and countermeasures. In *Proceedings of the International Symposium on Secure Software Engineering*, March 2006.

[100] Youssef Hanna, Hridesh Rajan, and Wensheng Zhang. Slede: A domain-specific verification framework for sensor network security protocol implementations. In *Proceedings of the First ACM Conference on Wireless Network Security*, WiSec '08, pages 109–118, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-814-5.

[101] Jon Heffley and Pascal Meunier. Can source code auditing software identify common vulnerabilities and be used to evaluate software security? In *Proceedings of the Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04) - Track 9 - Volume 9*, HICSS '04, pages 90277–, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2056-1.

[102] Val Henson. An analysis of compare-by-hash. In *Proceedings of HotOS IX: The 9th Workshop on Hot Topics in Operating Systems*, pages 13–18, Berkeley, CA, May 2003. USENIX Association.

[103] T. P. Hettmansperger and J. W. McKean. *Robust nonparametric statistical methods*. Kendall's Library of Statistics, 1998. ISBN 0-340-54937-8.

[104] Boniface Hicks, Sandra Rueda, Luke St.Clair, Trent Jaeger, and Patrick McDaniel. A logical specification and analysis for selinux mls policy. *ACM Trans. Inf. Syst. Secur.*, 13(3):26:1–26:31, July 2010. ISSN 1094-9224.

[105] Gerard J. Holzmann. The model checker SPIN. *IEEE Trans. Softw. Eng.*, 23(5):279–295, May 1997. ISSN 0098-5589.

[106] David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, December 2004. ISSN 0362-1340.

[107] David Hovemeyer and William Pugh. Finding more null pointer bugs, but not too many. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '07, pages 9–14, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-595-3.

[108] Michael Howard and David LeBlanc. *Writing Secure Code*. Microsoft Press, Redmond, WA, second edition, 2003. ISBN 0-7356-1722-8.

[109] T. Howes and M. Smith. The LDAP URL format, 1997.

[110] Trent Jaeger, Xiaolan Zhang, and Antony Edwards. Policy management using access control spaces. *ACM Trans. Inf. Syst. Secur.*, 6(3):327–364, August 2003. ISSN 1094-9224.

[111] Ron Jeffries and Grigori Melnik. Guest editors' introduction: TDD–the art of fearless programming. *IEEE Software*, 24(3):24–30, May 2007. doi: 10.1109/MS.2007.75.

[112] Trevor Jim, Nikhil Swamy, and Michael 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, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-654-7.

[113] Martin Johns and Christian Beyerlein. SMask: preventing injection attacks in web applications by approximating automatic data/code separation. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, pages 284–291, New York, NY, USA, 2007. ACM. ISBN 1-59593-480-4.

[114] Martin Johns, Björn Engelmann, and Joachim 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, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3447-3.

[115] David Johnson, Alexei White, and Andre Charland. *Enterprise AJAX: Strategies for Building High Performance Web Applications*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2007. ISBN 0132242060.

[116] Robert Timothy Johnson. *Verifying Security Properties Using Type-qualifier Inference*. PhD thesis, Berkeley, CA, USA, 2006. AAI3253911.

[117] Ashlesha Joshi, Samuel T. King, George W. Dunlap, and Peter M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, SOSP '05, pages 91–104, New York, NY, USA, 2005. ACM. ISBN 1-59593-079-5.

[118] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, SP '06, pages 258–263, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2574-1.

[119] Daniel E. Geer Jr., David Aucsmith, and James A. Whittaker. Monoculture. *IEEE Security and Privacy*, 1(6):14–19, 2003.

[120] Isaiah Pinchas Kantorovitz. Lexical analysis tool. *SIGPLAN Not.*, 39(5):66–74, May 2004. ISSN 0362-1340.

[121] Chris Karlof, Umesh Shankar, J. D. Tygar, and David Wagner. Dynamic pharming attacks and locked same-origin policies for web browsers. In *Proceedings of the 14th ACM conference on Computer and communications security*, CCS '07, pages 58–71, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-703-2.

[122] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis 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, New York, NY, USA, 2003. ACM. ISBN 1-58113-738-9.

[123] Angelos D. Keromytis. Randomized instruction sets and runtime environments past research and future directions. *IEEE Security and Privacy*, 7(1):18–25, January 2009. ISSN 1540-7993.

[124] Angelos D. Keromytis. Buffer overflow attacks. In *Encyclopedia of Cryptography and Security (2nd Ed.)*, pages 174–177. 2011.

[125] Benny Kimelfeld and Yehoshua Sagiv. Revisiting redundancy and minimization in an XPath fragment. In *EDBT '08: Proceedings of the 11th international conference on Extending database technology*, pages 61–72, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-926-5.

[126] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976. ISSN 0001-0782.

[127] Vladimir Kiriansky, Derek Bruening, and Saman P. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, pages 191–206, Berkeley, CA, USA, 2002. USENIX Association. ISBN 1-931971-00-5.

[128] Donald E. Knuth. *The Art of Computer Programming*, volume 3: Sorting and Searching. Addison-Wesley, Reading, MA, 1973. ISBN 0-201-03803-X.

[129] Deguang Kong, Quan Zheng, Chao Chen, Jianmei Shuai, and Ming Zhu. ISA: A source code static vulnerability detection system based on data fusion. In *Proceedings of the 2Nd International Conference on Scalable Information Systems*, InfoScale '07, pages 55:1–55:7, ICST, Brussels, Belgium, Belgium, 2007. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering). ISBN 978-1-59593-757-5.

[130] Jean-Louis Krivine. A call-by-name lambda-calculus machine. *Higher Order Symbol. Comput.*, 20(3):199–207, September 2007. ISSN 1388-3690.

[131] Nuno Laranjeiro, Marco Vieira, and Henrique Madeira. Protecting database centric web services against SQL/XPath injection attacks. In *Proceedings of the 20th International Conference on Database and Expert Systems Applications*, DEXA '09, pages 271–278, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-03572-2.

[132] Nuno Laranjeiro, Marco Vieira, and Henrique Madeira. A learning-based approach to secure web services from SQL/XPath injection attacks. In *Proceedings of the 2010 IEEE 16th Pacific Rim International Symposium on Dependable Computing*, PRDC '10, pages 191–198, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4289-8.

[133] Sin Yeung Lee, Wai Lup Low, and Pei Yuen Wong. Learning fingerprints for a database intrusion detection system. In Dieter Gollmann, Günter Karjoth, and Michael Waidner, editors, *ESORICS '02: Proceedings of the 7th European Symposium on Research in Computer Security*, pages 264–280, London, UK, 2002. Springer-Verlag. ISBN 3-540-44345-2. Lecture Notes In Computer Science 2502.

[134] Kyung-Suk Lhee and Steve J. Chapin. Buffer overflow and format string overflow vulnerabilities. *Software: Practice and Experience*, 33(5):423–460, 2003. ISSN 0038-0644.

[135] Zhiqiang Lin, Xuxian Jiang, Dongyan Xu, Bing Mao, and Li Xie. AutoPaG: Towards automated software patch generation with source code root cause identification and repair. In *Proceedings of the 2Nd ACM Symposium on Information, Computer and Communications Security*, ASIACCS '07, pages 329–340, New York, NY, USA, 2007. ACM. ISBN 1-59593-574-6.

[136] V. Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in java applications with static analysis. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, SSYM'05, pages 18–18, Berkeley, CA, USA, 2005. USENIX Association.

[137] Mike Ter Louw and V. N. Venkatakrishnan. Blueprint: Robust prevention of cross-site scripting attacks for existing browsers. In *SP '09: Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, pages 331–346, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3633-0.

[138] Mircea Lungu and Michele Lanza. The small project observatory: A tool for reverse engineering software ecosystems. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 289–292, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-719-6.

[139] Nikolai Mansourov and Djenana Campara. *System Assurance: Beyond Detecting Vulnerabilities*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2010. ISBN 9780123814142.

[140] Michael Martin and Monica S. Lam. Automatic generation of XSS and SQL injection attacks with goal-directed model checking. In *Proceedings of the 17th Conference on Security Symposium*, SS'08, pages 31–43, Berkeley, CA, USA, 2008. USENIX Association.

[141] Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using PQL: a program query language. In *OOPSLA '05: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 365–383, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-031-0.

[142] Fabio Massacci, Stephan Neuhaus, and Viet Hung Nguyen. After-life vulnerabilities: a study on firefox evolution, its vulnerabilities, and fixes. In *Proceedings of the Third international conference on Engineering secure software and systems*, ESSoS'11, pages 195–208, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-19124-4.

[143] Thiago Mattos, Altair Santin, and Andreia Malucelli. Mitigating xml injection 0-day attacks through strategy-based detection systems. *IEEE Security and Privacy*, 11(4):46–53, July 2013. ISSN 1540-7993.

[144] Russell A. McClure and Ingolf H. Krüger. SQL DOM: Compile time checking of dynamic SQL statements. In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*, pages 88–96, 2005. ISBN 1-59593-963-2. doi: 10.1145/1062455.1062487.

[145] Gary McGraw. *Software Security: Building Security In*. Addison-Wesley Professional, 2006. ISBN 0321356705.

[146] Gary McGraw. Automated code review tools for security. *IEEE Computer*, 41(12):108–111, 2008.

[147] Kenneth Lauchlin McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. PhD thesis, Pittsburgh, PA, USA, 1992. UMI Order No. GAX92-24209.

[148] Fadi Meawad, Gregor Richards, Floréal Morandat, and Jan Vitek. Eval begone!: semi-automated removal of eval from javascript programs. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '12, pages 607–620, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1561-6.

[149] Daniel Mellado, Eduardo Fernández-Medina, and Mario Piattini. Security requirements engineering framework for software product lines. *Inf. Softw. Technol.*, 52(10):1094–1117, October 2010. ISSN 0950-5849.

[150] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344, 2005. ISSN 0360-0300.

[151] Stephan Merz. Model checking: A tutorial overview. In *Proceedings of the 4th Summer School on Modeling and Verification of Parallel Processes*, MOVEP '00, pages 3–38, London, UK, UK, 2001. Springer-Verlag. ISBN 3-540-42787-2.

[152] Ali Mesbah and Arie van Deursen. A component-and push-based architectural style for Ajax applications. *J. Syst. Softw.*, 81(12):2194–2209, December 2008. ISSN 0164-1212.

[153] Ali Mesbah and Arie van Deursen. Invariant-based automatic testing of AJAX user interfaces. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 210–220, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-3453-4.

[154] Alice Miller, Alastair Donaldson, and Muffy Calder. Symmetry in temporal logic model checking. *ACM Comput. Surv.*, 38(3), September 2006. ISSN 0360-0300.

[155] Yasuhiko Minamide. Static approximation of dynamically generated web pages. In *WWW '05: Proceedings of the 14th international conference on World Wide Web*, pages 432–441, New York, NY, USA, 2005. ACM. ISBN 1-59593-046-9.

[156] Dimitris Mitropoulos and Diomidis Spinellis. SDriver: Location-specific signatures prevent SQL injection attacks. *Computers and Security*, 28:121–129, May/June 2009. ISSN 0167-4048.

[157] Roland T. Mittermeir. Software evolution: let's sharpen the terminology before sharpening (out-of-scope) tools. In *Proceedings of the 4th International Workshop on Principles of Software Evolution*, IWPSE '01, pages 114–121, New York, NY, USA, 2001. ACM. ISBN 1-58113-508-4.

[158] Leon Moonen. A generic architecture for data flow analysis to support reverse engineering. In *Proceedings of the 2Nd International Conference on Theory and Practice of Algebraic Specifications*, Algebraic'97, pages 10–10, Swinton, UK, UK, 1997. British Computer Society. ISBN 3-540-76228-0.

[159] Madanlal Musuvathi and Dawson R. Engler. Model checking large network protocol implementations. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation - Volume 1*, NSDI'04, pages 12–12, Berkeley, CA, USA, 2004. USENIX Association.

[160] Yacin Nadji, Prateek Saxena, and Dawn Song. 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, Washington, DC, USA, 2006. IEEE Computer Society.

[161] Yacin Nadji, Prateek Saxena, and Dawn Song. Document structure integrity: A robust basis for cross-site scripting defense. In *Proceeding of the Network and Distributed System Security Symposium (NDSS)*, 2009.

[162] Csaba Nagy and Spiros Mancoridis. Static security analysis based on input-related software faults. In *Proceedings of the 2009 European Conference on Software Maintenance and Reengineering*, CSMR '09, pages 37–46, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3589-0.

[163] Susanta Nanda, Lap-Chung Lam, and Tzi-cker Chiueh. Dynamic multi-process information flow tracking for web application security. In *Proceedings of the 2007 ACM/IFIP/USENIX International Conference on Middleware Companion*, MC '07, pages 19:1–19:20, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-935-7.

[164] Anh Nguyen-Tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. In Ryichi Sasaki, Sihan Qing, Eiji Okamoto, and Hiroshi Yoshiura, editors, *SEC*, pages 295–308. ISBN 0-387-25658-X.

[165] Linda M. Null and Johnny Wong. The diamond security policy for object-oriented databases. In *Proceedings of the 1992 ACM Annual Conference on Communications*, CSC '92, pages 49–56, New York, NY, USA, 1992. ACM. ISBN 0-89791-472-4.

[166] Kurt M. Olender and Leon J. Osterweil. Interprocedural static analysis of sequencing constraints. *ACM Trans. Softw. Eng. Methodol.*, 1(1):21–52, January 1992. ISSN 1049-331X.

[167] Bruno C.d.S. Oliveira, Meng Wang, and Jeremy Gibbons. The visitor pattern as a reusable, generic, type-safe component. *SIGPLAN Not.*, 43(10):439–456, October 2008. ISSN 0362-1340.

[168] Leon J. Osterweil and Lloyd D. Fosdick. Some experience with DAVE: A fortran program analyzer. In *Proceedings of the June 7-10, 1976, National Computer Conference and Exposition*, AFIPS '76, pages 909–915, New York, NY, USA, 1976. ACM.

[169] Andy Ozment and Stuart E. Schechter. Milk or wine: does software security improve with age? In *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, Berkeley, CA, USA, 2006. USENIX Association.

[170] Ioannis Papagiannis, Matteo Migliavacca, and Peter Pietzuch. Php aspis: Using partial taint tracking to protect against injection attacks. In *Proceedings of the 2Nd USENIX Conference on Web Application Development*, WebApps'11, pages 2–2, Berkeley, CA, USA, 2011. USENIX Association.

[171] David Lorge Parnas. Which is riskier: OS diversity or OS monopoly? *Commun. ACM*, 50(8): 112–, August 2007. ISSN 0001-0782.

[172] R. Peck and J.L. Devore. *Statistics: The Exploration & Analysis of Data*. Available Titles Aplia Series. Brooks/Cole, Cengage Learning, 2010. ISBN 9780840058010.

[173] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002. ISBN 0-262-16209-1.

[174] Jonathan Pincus and Brandon Baker. Beyond stack smashing: Recent advances in exploiting buffer overruns. *IEEE Security and Privacy*, 2(4):20–27, July 2004. ISSN 1540-7993.

[175] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

[176] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 85–100, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0977-6.

[177] Vassilis Prevelakis and Diomidis Spinellis. Sandboxing applications. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 119–126, Berkeley, CA, USA, 2001. USENIX Association. ISBN 1-880446-10-3.

[178] Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou, and Youfeng Wu. LIFT: A low-overhead practical information flow tracking system for detecting security attacks. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 135–148, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2732-9.

[179] Steven Raemaekers, Arie van Deursen, and Joost Visser. The maven repository dataset of metrics, changes, and dependencies. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 221–224, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4673-2936-1.

[180] David Raggett. Client-side scripting and HTML. *World Wide Web J.*, 2:29–37, April 1997. ISSN 1085-2301.

[181] Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, Scott A. Smolka, Terrance Swift, and David Scott Warren. Efficient model checking using tabled resolution. In *Proceedings of the 9th International Conference on Computer Aided Verification*, CAV '97, pages 143–154, London, UK, UK, 1997. Springer-Verlag. ISBN 3-540-63166-6.

[182] Ariel Ortiz Ramirez. Three-tier architecture. *Linux J.*, 2000(75es), July 2000. ISSN 1075-3583.

[183] Donald Ray and Jay Ligatti. Defining code-injection attacks. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '12, pages 179–190, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1083-3.

[184] Charles Reis, John Dunagan, Helen J. Wang, Opher Dubrovsky, and Saher Esmeir. BrowserShield: Vulnerability-driven filtering of dynamic HTML. *ACM Trans. Web*, 1, September 2007. ISSN 1559-1131.

[185] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 49–61, New York, NY, USA, 1995. ACM. ISBN 0-89791-692-1.

[186] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. The eval that men do: A large-scale study of the use of eval in javascript applications. In *Proceedings of the 25th European conference on Object-oriented programming*, ECOOP'11, pages 52–78, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-22654-0.

[187] Jose Romero-Mariona, Hadar Ziv, Debra J. Richardson, and Dennis Bystritsky. Towards usable cyber security requirements. In *Proceedings of the 5th Annual Workshop on Cyber Security and Information Intelligence Research: Cyber Security and Information Intelligence Challenges and Strategies*, CSIIRW '09, pages 64:1–64:4, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-518-5.

[188] Michelle Ruse, Tanmoy Sarkar, and Samik Basu. Analysis & detection of SQL injection vulnerabilities via automatic test case generation of programs. In *Proceedings of the 2010 10th IEEE/IPSJ International Symposium on Applications and the Internet*, SAINT '10, pages 31–37, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4107-5.

[189] Hossein Saiedian and Dan Broyle. Security vulnerabilities in the same-origin policy: Implications and alternatives. *Computer*, 44(9):29–36, September 2011. ISSN 0018-9162.

[190] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for javascript. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 513–528, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4035-1.

[191] Theodoor Scholte, Davide Balzarotti, and Engin Kirda. Quo Vadis? a study of the evolution of input validation vulnerabilities in web applications. In *Proceedings of the 15th International Conference on Financial Cryptography and Data Security*, FC'11, pages 284–298, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-27575-3.

[192] Benjamin Schwarz, Hao Chen, David Wagner, Jeremy Lin, Wei Tu, Geoff Morrison, and Jacob West. Model checking an entire linux distribution for security violations. In *Proceedings of the 21st Annual Computer Security Applications Conference*, ACSAC '05, pages 13–22, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2461-3.

[193] Robert Seacord. Secure coding in C and C++: Of strings and integers. *IEEE Security and Privacy*, 4(1):74, 2006. ISSN 1540-7993.

[194] Nuno Seixas, José Fonseca, Marco Vieira, and Henrique Madeira. Looking at web security vulnerabilities from the programming language perspective: A field study. In *ISSRE '09: Proceedings of the 2009 20th International Symposium on Software Reliability Engineering*, pages 129–135, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3878-5.

[195] R. Sekar. An efficient black-box technique for defeating web application attacks. In *Proceeding of the Network and Distributed System Security Symposium (NDSS)*, 2009.

[196] Hossain Shahriar and Mohammad Zulkernine. Mitigating program security vulnerabilities: Approaches and challenges. *ACM Comput. Surv.*, 44(3):11:1–11:46, June 2012. ISSN 0360-0300.

[197] Muhammad Shahzad, Muhammad Zubair Shafiq, and Alex X. Liu. A large scale exploratory analysis of software vulnerability life cycles. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 771–781, Piscataway, NJ, USA, 2012. IEEE Press. ISBN 978-1-4673-1067-3.

[198] Haihao Shen, Sai Zhang, Jianjun Zhao, Jianhong Fang, and Shiyuan Yao. XFindBugs: extended findbugs for AspectJ. In *Proceedings of the 8th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '08, pages 70–76, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-382-2.

[199] Jack Shirazi. *Java Performance Tuning*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2nd edition, 2002. ISBN 0596003773.

[200] K. Sivakumar and K. Garg. Constructing a "common cross site scripting vulnerabilities enumeration (CXE)" using CWE and CVE. In *Proceedings of the 3rd international conference on Information systems security*, ICISS'07, pages 277–291, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 3-540-77085-2, 978-3-540-77085-5.

[201] Ian Sommerville. *Software Engineering (5th Ed.)*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1995. ISBN 0-201-42765-6.

[202] Sang H. Son, Craig Chaney, and Norris P. Thomlinson. Partial security policies to support timeliness in secure real-time databases. Technical report, Charlottesville, VA, USA, 1998.

[203] Ana Nora Sovarel, David Evans, and Nathanael Paul. Where's the feeb? the effectiveness of instruction set randomization. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, SSYM'05, pages 10–10, Berkeley, CA, USA, 2005. USENIX Association.

[204] Jaime Spacco, David Hovemeyer, and William Pugh. Tracking defect warnings across versions. In *Proceedings of the 2006 international workshop on Mining software repositories*, MSR '06, pages 133–136, New York, NY, USA, 2006. ACM. ISBN 1-59593-397-2.

[205] Diomidis Spinellis and Panagiotis Louridas. A framework for the static verification of API calls. *J. Syst. Softw.*, 80:1156–1168, July 2007. ISSN 0164-1212.

[206] Michael L. Stamat and Jeffrey W. Humphries. Training $\neq$ education: putting secure software engineering back in the classroom. In *Proceedings of the 14th Western Canadian Conference on Computing Education*, WCCCE '09, pages 116–123, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-415-7.

[207] Konstantinos Stamos, George Pallis, Athena Vakali, Dimitrios Katsaros, Antonis Sidiropoulos, and Yannis Manolopoulos. CDNsim: A simulation tool for content distribution networks. *ACM Trans. Model. Comput. Simul.*, 20(2):10:1–10:40, May 2010. ISSN 1049-3301.

[208] Zhendong Su and Gary Wassermann. 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 372–382. ACM Press, January 2006. doi: 10.1145/1111037.1111070.

[209] Sufatrio and Roland H. C. Yap. Improving host-based IDS with argument abstraction to prevent mimicry attacks. In *Proceedings of the 8th International Conference on Recent Advances in Intrusion Detection*, RAID'05, pages 146–164, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-31778-3, 978-3-540-31778-4.

[210] Masaru 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, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3329-2.

[211] Shuo Tang, Nathan Dautenhahn, and Samuel T. King. Fortifying web-based applications automatically. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 615–626, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0948-6.

[212] Rahul Telang and Sunil Wattal. Impact of software vulnerability announcements on the market value of software vendors - an empirical investigation. In *Workshop on the Economics of Information Security*, page 677427, 2007.

[213] Jay-Evan J. Tevis and John A. Hamilton. Methods for the prevention, detection and removal of software security vulnerabilities. In *Proceedings of the 42nd annual Southeast regional conference*, ACM-SE 42, pages 197–202, New York, NY, USA, 2004. ACM. ISBN 1-58113-870-9.

[214] Marianthi Theoharidou and Dimitris Gritzalis. Common body of knowledge for information security. *IEEE Security & Privacy*, 5(2):64–67, 2007.

[215] Bhavani Thuraisingham and William Ford. Security constraint processing in a multilevel secure distributed database management system. *IEEE Trans. on Knowl. and Data Eng.*, 7(2):274–293, April 1995. ISSN 1041-4347.

[216] Katrina Tsipenyuk, Brian Chess, and Gary McGraw. Seven pernicious kingdoms: A taxonomy of software security errors. *IEEE Security & Privacy*, 3(6):81–84, 2005.

[217] Aliaksei Tsitovich. Detection of security vulnerabilities using guided model checking. In Maria Garcia de la Banda and Enrico Pontelli, editors, *ICLP*, volume 5366 of *Lecture Notes in Computer Science*, pages 822–823. Springer, 2008. ISBN 978-3-540-89981-5.

[218] Fredrik Valeur, Darren Mutz, and Giovanni Vigna. A learning-based approach to the detection of SQL attacks. In Klaus Julisch and Christopher Kruegel, editors, *Intrusion and Malware Detection and Vulnerability Assessment: Second International Conference, DIMVA 2005*, pages 123–140, July 2005. doi: 10.1007/11506881_8. Lecture Notes in Computer Science 3548.

[219] John Viega and Gary McGraw. *Building Secure Software: How to Avoid Security Problems the Right Way*. Addison-Wesley, Boston, MA, 2001. ISBN 0-201-72152-X.

[220] John Viega, J. T. Bloch, Y. Kohno, and Gary McGraw. ITS4: A static vulnerability scanner for C and C++ code. In *ACSAC*, pages 257–259. IEEE Computer Society, 2000. ISBN 0-7695-0859-6.

[221] John Viega, J. T. Bloch, Tadayoshi Kohno, and Gary McGraw. Token-based scanning of source code for security problems. *ACM Trans. Inf. Syst. Secur.*, 5(3):238–261, 2002. ISSN 1094-9224.

[222] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-site scripting prevention with dynamic data tainting and static analysis. In *Proceeding of the Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2007.

[223] David Von Oheimb. Information flow control revisited: Noninfluence = noninterference + nonleakage. In Pierangela Samarati, Peter Y. A. Ryan, Dieter Gollmann, and Refik Molva, editors, *ESORICS*, volume 3193 of *Lecture Notes in Computer Science*, pages 225–243. Springer, 2004. ISBN 3-540-22987-6.

[224] David Wagner and Paolo Soto. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the 9th ACM conference on Computer and communications security*, CCS '02, pages 255–264, New York, NY, USA, 2002. ACM. ISBN 1-58113-612-9.

[225] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceeding of the Network and Distributed System Security Symposium (NDSS)*, pages 3–17, 2000.

[226] Helen Wang. Attacks target web server logic and prey on XCS weaknesses: technical persepctive. *Commun. ACM*, 53(8):104–104, 2010. ISSN 0001-0782.

[227] Huaiqing Wang and Chen Wang. Taxonomy of security considerations and software quality. *Commun. ACM*, 46(6):75–78, June 2003. ISSN 0001-0782.

[228] Xinran Wang, Chi-Chun Pan, Peng Liu, and Sencun Zhu. SigFree: A signature-free buffer overflow attack blocker. *IEEE Trans. Dependable Secur. Comput.*, 7(1):65–79, 2010. ISSN 1545-5971.

[229] Y. Wang, W. M. Lively, and D. B. Simmons. Software security analysis and assessment model for the web-based applications. *J. Comp. Methods in Sci. and Eng.*, 9:179–189, April 2009. ISSN 1472-7978.

[230] Gary Wassermann and Zhendong 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.

[231] Gary Wassermann and Zhendong Su. Sound and precise analysis of web applications for injection vulnerabilities. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 32–41, New York, NY, USA, 2007. ACM Press. ISBN 978-1-59593-633-2. doi: 10.1145/1250734.1250739.

[232] Cindy Wilson and Leon J. Osterweil. Omega a data flow analysis tool for the C programming language. *IEEE Trans. Softw. Eng.*, 11(9):832–838, September 1985. ISSN 0098-5589.

[233] Janice Winsor. *Solaris System Administrator's Guide*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 3rd edition, 2000. ISBN 0130277029.

[234] Yan Wu, Robin A. Gandhi, and Harvey Siy. Using semantic templates to study vulnerabilities recorded in large software repositories. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Secure Systems*, SESS '10, pages 22–28, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-965-7.

[235] Glenn Wurster and P. C. van Oorschot. The developer is the enemy. In *NSPW '08: Proceedings of the 2008 workshop on New security paradigms*, pages 89–97, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-341-9.

[236] 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, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-3725-2.

[237] Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, Berkeley, CA, USA, 2006. USENIX Association.

[238] Wei Xu, Sandeep Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *Security '06: Proceedings of the 15th USENIX Security Symposium*, pages 121–136, Berkeley, CA, August 2006. USENIX Association.

[239] Yves Younan, Wouter Joosen, and Frank Piessens. Runtime countermeasures for code injection attacks against C and C++ programs. *ACM Comput. Surv.*, 44(3):17:1–17:28, June 2012. ISSN 0360-0300.

[240] Dachuan Yu, Ajay Chander, Nayeem Islam, and Igor Serikov. Javascript instrumentation for browser security. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT Symposium on Principles of programming languages*, pages 237–249, New York, NY, USA, 2007. ACM. ISBN 1-59593-575-4.

[241] Chuan Yue and Haining Wang. Characterizing insecure javascript practices on the web. In *WWW '09: Proceedings of the 18th International Conference on World wide web*, pages 961–970, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-487-4.

[242] Shahed Zaman, Bram Adams, and Ahmed E. Hassan. Security versus performance bugs: a case study on Firefox. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 93–102, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0574-7.

[243] Misha Zitser, D. E. Shaw Group, and Tim Leek. Testing static analysis tools using exploitable buffer overflows from open source code. *SIGSOFT Softw. Eng. Notes*, 2004.

[244] Misha Zitser, Richard Lippmann, and Tim Leek. Testing static analysis tools using exploitable buffer overflows from open source code. *SIGSOFT Softw. Eng. Notes*, 29(6):97–106, October 2004. ISSN 0163-5948.